CS152 HW4

1. See bang-bang.scm

2. (6.32) Is it possible to write a tail-recursive version of the classic quicksort algorithm? Why or why not?

It is possible to write a tail-recursive version of quicksort because the last 2 procedures in quicksort are recursive calls on each half divided by the pivot, making it tail recursive.

3.

6.41 *Loop unrolling* (described in Exercise C-5.21 and Section C-17.7.1) is a code transformation that replicates the body of a loop and reduces the number of iterations, thereby decreasing loop overhead and increasing opportunities to improve the performance of the processor pipeline by reordering instructions. Unrolling is traditionally implemented by the code improvement phase of a compiler. It can be implemented at source level, however, if we are faced with the prospect of "hand optimizing" time-critical code on a system whose compiler is not up to the task. Unfortunately, if we replicate the body of a loop k times, we must deal with the possibility that the original number of loop iterations, n, may not be a multiple of k. Writing in C, and letting $k = 4$, we might transform the main loop of Exercise C-5.21 from

```
i = 0;
do {
    sum += A[i]; squares += A[i] * A[i]; i++;
} while (i < N);
```

to

```
i = 0;  j = N/4;
do {
    sum += A[i]; squares += A[i] * A[i]; i++;
    sum += A[i]; squares += A[i] * A[i]; i++;
    sum += A[i]; squares += A[i] * A[i]; i++;
    sum += A[i]; squares += A[i] * A[i]; i++;
} while (--j > 0);
do {
    sum += A[i]; squares += A[i] * A[i]; i++;
} while (i < N);
```

In 1983, Tom Duff of Lucasfilm realized that code of this sort can be "simplified" in C by interleaving a `switch` statement and a loop. The result is rather startling, but perfectly valid C. It's known in programming folklore as "Duff's device":

```
i = 0; j = (N+3)/4;
switch (N%4) {
    case 0: do{ sum += A[i]; squares += A[i] * A[i]; i++;
    case 3:     sum += A[i]; squares += A[i] * A[i]; i++;
    case 2:     sum += A[i]; squares += A[i] * A[i]; i++;
    case 1:     sum += A[i]; squares += A[i] * A[i]; i++;
            } while (--j > 0);
}
```

Duff announced his discovery with "a combination of pride and revulsion." He noted that "Many people… have said that the worst feature of C is that `switch`es don't `break` automatically before each `case` label. This code forms some sort of argument in that debate, but I'm not sure whether it's for or against." What do you think? Is it reasonable to interleave a loop and a `switch` in this way? Should a programming language permit it? Is automatic fall-through ever a good idea?

This example shows that switch cases can be used to simplify a loop by taking advantage of fall through. Although programming languages should permit this, It is not intuitive to interleave a loop and switch this way. Automatic fall through is a good idea when certain scenarios only need to ignore the cases before the target case. Since fall through can be stopped with a break in the code, it should be up to the programmer to decide if fall through is needed.

4.

7.2 In the following code, which of the variables will a compiler consider to have compatible types under structural equivalence? Under strict name equivalence? Under loose name equivalence?

```
type  T = array [1..10] of integer
      S = T
A  :  T
B  :  T
C  :  S
D  :  array [1..10] of integer
```

Under structural equivalence, the compiler will consider A, B, C and D to have compatible types. Under strict name equivalence, the compiler will consider A and B to have compatible types. Under loose name equivalence, the compiler will consider A, B, and C to have compatible types.

5.

7.3 Consider the following declarations:

```
1.  type cell              -- a forward declaration
2.  type cell_ptr = pointer to cell
3.  x : cell
4.  type cell = record
5.        val : integer
6.        next : cell_ptr
7.  y : cell
```

Should the declaration at line 4 be said to introduce an alias type? Under strict name equivalence, should x and y have the same type? Explain.

No, it does not create an alias type, rather it defines the type cell from the declaration in line 1. Even under strict name equivalence, x and y have the same type since they have the same type definition: cell.

6. See queue.rs

7.

8.14 Suppose A is a 10×10 array of (4-byte) integers, indexed from [0][0] through [9][9]. Suppose further that the address of A is currently in register r1, the value of integer i is currently in register r2, and the value of integer j is currently in register r3.

Give pseudo-assembly language for a code sequence that will load the value of A[i][j] into register r1 (a) assuming that A is implemented using (row-major) contiguous allocation; (b) assuming that A is implemented using row pointers. Each line of your pseudocode should correspond to a single instruction on a typical modern machine. You may use as many registers as you need. You need not preserve the values in r1, r2, and r3. You may assume that i and j are in bounds, and that addresses are 4 bytes long.

Which code sequence is likely to be faster? Why?

a)
```
mult (r2, r2, 40)      // r2 *= 40
sll (r3, r3, 2)        // r3 <<= 2
add (r1, r1, r3)       // r1+=r3
add (r1, r1, r2)       // r1+=r2
r1 := *r1
```

b)
```
sll (r2, r2, 2)        // r2 <<=2
add (r1, r1, r2)       // r1+=r2
r1:= *r1

sll (r3, r3, 2)        // r3 <<=2
add (r1, r1, r3)       // r1+=r3
r1:= *r1
```

The first code sequence (row-major contiguous allocation) will be faster because it only loads once instead of twice since loading is a very costly operation.

8. See log-min.scm

9. See rotation-filter.scm

10. See permutation.scm