Pedram Safaei
CS 326
HW 4

1.

Postfix: b b b * 4 a * c * - sqrt + 2 a * /

Prefix: / + b sqrt - * b b * * 4 a c * 2 a

2.

a) A and B

```
if A
      then B
else
      false
```

[if A then B else false]

b) A or B

```
if A
      true
else
      B
```

[if A then true else B]

3.

a) Using while loop

```
line = read_line();
while(!all_blanks(line))
{
      process_line(line);
      line = read_line();
}
```

b) Using a do loop

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

or

```
_Bool done = false;
```

```
while (!done) {
      line = read_line();

      if (all_blanks(line))
            done = true;
      else
            process_line(line);
}

_Bool done = false;
do {

      line = read_line();

      if (all_blanks(line))
            done = true;
      else
            process_line(line);
} while (!done);
```

4.

```
(define (factorial n)
    (let loop ((n n)
               (accumulator 1))
      (if (= n 0) accumulator
          (loop (- n 1)(* n accumulator)))))
```

5.

Example where inline may be significantly faster

```
#define COMPUTE(n) n * 3 + n – n

inline int COMPUTE(n) {return n * 3 + n – n;}
```

If we assume that “n” here is an expression, applicative evaluation will be used in the inline function’s argument and would not need to recompute it for each instance of our “n” in the function, which would be faster than using the macro version, since the macro version uses normal-order evaluation of our “n” and have to recompute it for each instances.

Example where Macro may be significantly faster

```
#define KRYPTONITE 420

inline int KRYPTONITE() {return 420;}
```

In this case our macro would be faster because it would just substitute the the number for our KRYPTONITE instead of calling the function and executing the body of that function every time.