1.1

```
10
```
Outputs **10**.
```
(+ 5 3 4)
```
Outputs **12**.
```
(- 9 1)
```
Outputs **8**.
```
(/ 6 2)
```
Outputs **3**.
```
(+ (* 2 4) (- 4 6))
```
Outputs **6**.
```
(define a 3)
```
No output.
```
(define b (+ a 1))
```
No output.
```
(+ a b (* a b))
```
Outputs **19**.
```
(= a b)
```
Outputs **#f**.
```
(if (and (> b a) (< b (* a b)))
    b
    a)
```
Outputs the value of `b`, which is **4**.
```
(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))
```
Outputs the result of `(+ 6 7 a)`, which is **16**.
```
(+ 2 (if (> b a) b a))
```
Outputs the result of `(+ 2 b)`, which is **6**.
```
(* (cond ((> a b) a)
         ((< a b) b)
         (else -1))
         (+ a 1))
```
Outputs the result of `(* b (+ a 1))`, which is **16**.

1.2

$$\frac{5+4+(2-(3-(6+\frac{4}{5})))}{3(6-2)(2-7)}$$

Would become (/ (+ 5 4 (- 2 (- 3 (+ 6 (/ 4 5))))) (* 3 (- 6 2) (- 2 7))) in prefix notation.

1.3

Procedure must take three numbers as arguments and return the sum of the squares of the two larger numbers.
A solution would be:
```
(define (sum-of-squares x y z)
        (let ((lowest (min x y z)))
             (cond ((= x lowest) (+ (* y y) (* z z)))
                   ((= y lowest) (+ (* x x) (* z z)))
                   ((= z lowest) (+ (* x x) (* y y))))))
```

1.4

The procedure is
```
(define (a-plus-abs-b a b)
        ((if (> b 0) + -) a b))
```
The procedure defines a function called `a-plus-abs-b` which takes two arguments, `a` and `b`. If `b > 0`, it selects the + operator and applies `a` and `b` to it. Otherwise, it selects the − operator to apply `a` and `b` to. In other words, if `b` is positive (and thus equal to its absolute value), it is added to `a` and the result is "returned" (for lack of a better word). If `b` is negative (or zero), it is subtracted from `a` and "returned", taking advantage of the fact that subtracting a negative number from another number is effectively the same as adding the absolute value of that negative number to the other number.

1.5

The function returns 0 if the first argument is 0, and it returns the second argument otherwise. Bill tests the function, passing in 0 as the first argument. In an applicative-order interpreter, the expression will cause the program to hang indefinitely. Since `(p)` is defined as itself, and applicative-order interpreters would evaluate the arguments of a function before taking any further action, the interpreter will be stuck attempting to evaluate `(p)` before evaluating the rest of the function.
In a normal-order interpreter, the expression will simply cause the interpreter to output `0`. Normal-order interpreters do not evaluate arguments to a function until absolutely necessary. As it turns out, the second argument to `test` does not need to be evaluated in the event that the first argument is 0, which it is in this particular case. The interpreter will thus output `0`, since there is no need to even attempt to evaluate `(p)`.
Incidentally, testing these expressions in *DrRacket* seems to reveal it to be an applicative-order interpreter.