CS 61A, Summer 2006 July 5, 2006

HOMEWORK ASSIGNMENT 2B
DUE WEDNESDAY JULY 12, 2006 AT 11:59 PM

1. The following is from page 33 of SICP. Describe the invariant of `iter`. You don't have to explain why it is the invariant, but at least make sure that you've convinced yourself. Explain how you can use the invariant to prove that `factorial` always returns the correct result.

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

2. Here is a different iterative factorial program. Describe the invariant of `helper`. You don't have to explain why it is the invariant, but at least make sure that you've convinced yourself. Explain how you can use the invariant to prove that `factorial` always returns the correct result.

```
(define (factorial n)
  (define (helper result counter)
    (if (= counter 0)
        result
        (helper (* counter result)
                (- counter 1))))
  (helper 1 n))
```

3. What is the domain of the `pigl` procedure that we have seen several times in lecture and in lab? Be precise; exactly which arguments successfully return a correct result?

```
(define (pigl wd)
  (if (vowel? (first wd))
      (word wd 'ay)
      (pigl (word (butfirst wd) (first wd)))))
```

4. Prove (or give an informal explanation) that `pigl` must reach its base case, given an argument in its domain.

5. Here is an excerpt from `twenty-one.scm`:

```
(define (play-customer customer-hand-so-far dealer-up-card rest-of-deck)
  (cond ((> (best-total customer-hand-so-far) 21) -1)
        ((strategy customer-hand-so-far dealer-up-card)
        (play-customer (se customer-hand-so-far (first rest-of-deck))
                       dealer-up-card
                       (bf rest-of-deck)))
        (else
          (play-dealer customer-hand-so-far
                       (se dealer-up-card (first rest-of-deck))
                       (bf rest-of-deck)))))
```

This procedure has three formal parameters. `Customer-hand-so-far` is of type `hand` (where a `hand`, as described in lecture, is a sentence of `cards`); `dealer-up-card` is of type `card`. What about `rest-of-deck`? What type is it? How should this be documented?

6. Consider the following `sort` procedure, which takes a sentence of numbers and returns a sentence consisting of those numbers in sorted order. Prove the correctness of `sort` **using induction on the length of** `sent`. Try to be as logically rigorous as possible. Note: You will need to prove that `insert` is correct before you can prove that `sort` is correct.

```
(define (sort sent)
  (if (empty? sent)
      '()
      (insert (first sent) (sort (bf sent)))))

(define (insert num sent) ; the domain of insert is numbers and *sorted* sentences
  (cond ((empty? sent) (se num sent))
        ((< num (first sent)) (se num sent))
        (else (se (first sent) (insert num (bf sent))))))
```

Don't worry if you don't understand induction too well. I promise that I won't put induction on any midterms or on the final.

Extra For Experts (optional).

The following program is in `~cs61a/lib/bubsort.scm`. `Sort` takes as its argument a sentence of numbers, and returns a sentence with the same numbers in increasing order. Figure out how it works, and document it properly, with data types, invariants, or other assertions as appropriate.

```
(define (bubble sent)
  (cond ((empty? (butfirst sent)) sent)
        ((<= (first sent) (first (bf sent)))
          (se (first sent) (bubble (bf sent))))
        (else (se (first (bf sent))
          (bubble (se (first sent) (bf (bf sent))))))))

(define (sort sent)
  (if (empty? sent)
      '()
      (let ((bub (bubble sent)))
        (se (sort (butlast bub))
            (last bub)))))
```