Comp 210: Test 3

Matthias Felleisen

December 10, 1996

Name: ______________________________

Problem 1 [10pts] Gianna Nannini loves 'amerika, even though she rocks in 'italy. So she wrote a program that looks for her favorite place, in a maze of places:

```
(define-structure (place name next-index))

(define v
  (vector (make-place 'italy 1)
          (make-place 'atlantic 5)
          (make-place 'amerika 2)
          (make-place 'socks 4)
          (make-place 'jeans 3)
          (make-place 'pocket 2)))

(define (Visit vec start)
  (begin
    (printf "visiting node ~s~n" start)
    (if (eq? (place-name (vector-ref vec start)) 'amerika)
        'found-it
        (Visit vec (place-next-index (vector-ref vec start))))))
```

Here is how (*Visit* *v* 0) evaluates:

```
  (Visit v 0)
= (begin
    (printf "visiting node ~s~n" 0)
    (printf "visiting node ~s~n" 1)
    (printf "visiting node ~s~n" 5)
    (printf "visiting node ~s~n" 2)
    'found-it)
```

In contrast, (*Visit* *v* 4) prints many more lines and never stops. What does it print and why doesn't it stop?

```
The function Visit vists places 3 and 4 repeatedly because it
doesn't remember and check whether it has been to a place before.
```

Modify *Visit* so that it always stops. If a place with 'amerika is reachable, the modified version should still print 'found-it; if not, it should print 'sorry. Like the original, the modified version of *Visit* should still take two arguments. Indicate which programming method you choose for the modification.

```
(define been-there null)

(define (Visit vec start)
  (begin
    (printf "visiting node ~s~n" start)
    (if (eq? (place-name (vector-ref vec start)) 'Amerika)
        'found-it
        (if (member start been-there) ;; member is allowed
            'sorry
            (begin
              (set! been-there (cons start been-there))
              (Visit vec (place-next-index (vector-ref vec start))))))))
```

or shorter:

```
(define (Visit vec start)
  (begin
    (printf "visiting node ~s~n" start)
    (cond
      ((eq? (place-name (vector-ref vec start)) 'Amerika) 'found-it)
      ((member start been-there) 'sorry)
      (else (set! been-there (cons start been-there))
            (Visit vec (place-next-index (vector-ref vec start)))))))
```

Problem 2. [10pts] Franz Schubert wants to write things in C, everything. So when someone presents him with Scheme code, he immediately asks for translations. Help him!

Part 2:a. Here is a recursive summation function for vectors (whose length is a power of 2):[1]

```
(define V (vector 12 13 14 15))

(define (Sum-para left right)
  (if (= (+ left 2) right)
      (+ (vector-ref V left) (vector-ref V (+ left 1)))
      (let ((mid (quotient (+ left right) 2)))
        (+ (Sum-para left mid) (Sum-para mid right)))))

(Sum-para 0 (vector-length V))
```

Continue the following evaluation:

```
  (Sum-para 0 (vector-length V))
= (Sum-para 0 4)
= ...
```

Show the recursive steps only.

```
  (Sum-para 0 4)
= (+ (Sum-para 0 2) (Sum-para 2 4))
= (+ 25 (Sum-para 2 4))
= (+ 25 29)
= 54
```

[1] This definition shows how vectors can be summed up in parallel.

Complete the following translation of the program into C:

```
int V[4] = {12,13,14,15};

...

The function is recursive as the above trace shows, so it must be
translated into a recursive procedure:

int sum_para(int V[], int left, int right){
  if (left+2==right)
    return V[left]+V[left+1];
  else
    {
      int mid = (left+right)/2;
      return sum_para(V,left,mid) + sum_para(V,mid,right);
    };
}
```

Part 2:b. Here is a recursive function that creates rolling averages of vectors:

```
;; Sample input:
(define V (vector 12 13 14 15))

;; Non-changing parts:
(define VL (vector-length V))
(define U (build-vector (- VL 2) (lambda (i) 0)))

;; Auxiliary function for computing averages of three vector elements:
(define (average V i)
  (/ (+ (vector-ref V (- i 1)) (vector-ref V i) (vector-ref V (+ i 1))) 3))

;; The function proper:
(define (Sum-ave i)
  (if (= i (- VL 1))
      (void)
      (begin (vector-set! U (sub1 i) (average V i))
             (Sum-ave (+ i 1)))))

(Sum-ave 1)
```

Continue the following evaluation:

```
  (define V (vector 12 13 14 15))
  (define VL (vector-length V))
  (define U (build-vector (- VL 2) (lambda (i) 0)))
  (Sum-ave 1)

= (define V^ (vector 12 13 14 15))
  (define V V^)
  (define U^ (vector 0 0))
  (define U U^)
  (define VL 4)
  (Sum-ave 1)

= ...
```

Show the recursive steps and the effects on the vectors only. (Yes, the space provided suffices!)

```
;; next step shown for completeness only!
= (begin (vector-set! U^ 1 13) (Sum-ave 2))

= (Sum-ave 2) ;; plus (define U^ (vector 13 0))

= (Sum-ave 3) ;; plus (define U^ (vector 13 14))

= (void)
```

Complete the following translation of the program into C:

```
int V[4] = {12,13,14,15};
int VL = 4;

double U[2] = {0,0};

double average(int V[], int i){
  return (V[i-1]+V[i]+V[i+1])/3.0;
}
...

// Since Sum-ave is a tail-recursive function as the above evaluation
// shows, we use a loop plus a wrapper:

int sum_ave(int j){
  for(int i=j;i!=VL;i=i+1)
    {
      U[i-1] = average(V,i);
    };
}
```

Problem 3. [10pts] Neil Young discovered how to Jam a long time ago. Now he has heard truly new and interesting things about our Jam and wants to try out this new language.

Part 3:a. Translate the following arithmetic expression into the Jam assembly language for him:

$(- \ (* \ x \ 3) \ 15)$

where x is stored at location 1000.

```
(ldi 0 1000) ;;  load x's address
(ld 0 0)     ;;  load x's value
(ldi 1 3)    ;;  load the constant 3
(mul 0 0 3)  ;;  multiply x by 3, into register x
(lid 1 15)   ;;  load the constant 15
(sub 0 0 1)  ;;  subtract 15 from x, into register x
```

Part 3:b. Also translate the following expression into the Jam notation for him:

(**if** (= x 5) (**set!** x (+ x 5)))

The placeholder x located at 1008. Use symbolic labels instead of absolute addresses for branch points.

```
(ldi 0 1008) ;;  load x's address into r0
(ld 1 0)     ;;  load x's value into r1
(ldi 2 5)    ;;  load the constant 5
(sub 3 1 2)  ;;  subtract 5 from x's value, into register r3
(bnez 3 DONE);;  test and branch to DONE point
(add 1 2 1)  ;;  add 5 to x's value, into x's register r1
(st 1 0)     ;;  store x's value in x's address
DONE:
```

Problem 4. [15pts] Guy Steele just loves tinkering with songs, coffee (especially the one from *the* island), and abstract machines. He recently discovered the Jam 2000 and thought that it was a great machine to run Java on. He wants to add Java's `new_array` instruction to the Jam. The instruction is a special instruction and has the following format:

Encoding	*Fmt*	*Assembly*
? ? ? y x 7 0 0	S	`(new_array x y)`

If possible, the instruction `(new_array x y)` creates a vector of length Ry, starting at location Rx.

The Jam cpu is supposed to use locations Rx and $Rx+1$ to hold information about the vector and locations $Rx+2, \ldots, Rx+Ry+1$ for the values of the vector elements. To this end, it first determines whether there is enough space left in memory, that is, whether $MAX\text{-}ADDR - Rx + 1 \geq Ry + 2$. If not, the cpu signals an error:

```
(machine-error "new-array: out of memory")
```

Otherwise, the cpu writes

1. 4, the tag for arrays, into the location Rx,
2. Ry into the location $Rx + 1$, and
3. 0 into locations $Rx + 2, \ldots, Rx + Ry + 1$.

Add the appropriate function to the table of special instructions (see Appendix B).

```
;; In slot 7:
(lambda (sign digits)
  (local ((define x (vector-ref digits 3))
          (define y (vector-ref digits 4)))
    (if (>= ;; number of slots available:
            (+ (- MAX-ADDR (get-reg x)) 1)
            ;; number of slots needed:
            (+ (get-reg y) 2))
      (begin
        (set-mem! (get-reg x) 4)
        (set-mem! (+ (get-reg x) 1) (get-reg y))
        (for_i= (+ (get-reg x) 2)
                (lambda (i) (= i (+ (get-reg x) (get-reg y) 2)))
                add1
                (lambda (i) (set-mem! i 0))))
      (machine-error "new-array: out of memory"))))
```

Problem 5. [5pts] Briefly answer the following questions:

1. Who claimed that all models of computation possess the same computational power?

   `Mentioned in class:  Church; Turing okay`

2. Who proved that mechanical computers have absolute limits?

   `Mentioned in class:  Goedel; Turing okay`

3. What is the difference between the C expression `V[5]` and the Scheme expression (*vector-ref* *V* `5`)?

```
Both expressions extract the sixth element from a vector,
but the Scheme expression implies that
  * V is checked for "vectorness"
  * the index (5) is checked for "integerness"
  * the index is checked against the range of the vector.
```

Appendix A

```
;; Perform action p for the values index,
;; (step index), (step (step index)), etc.,
;; until end? is satisfied.
;;
;; index: int
;; step: int → int
;; end: int → boolean
;; p: int → void
;; result: void
;; effect: those of (p index), etc.
(define (for_i= index end? step p)
  (if (end? index)
      (void)
      (begin (p index)
             (for_i= (step index) end? step p))))
```

Appendix B

```
(define SpecialInstructions
  (local ((define (bad sign digits) (machine-error "bad machine instruction")))
    (vector
      (lambda (sign digits) (abort (shell))) ; 0
      (lambda (sign digits) (display (get-reg (vector-ref digits 3)))) ; 1
      (lambda (sign digits) (newline)) ; 2
      bad ; 3
      bad ; 4
      bad ; 5
      bad ; 6
      bad ; 7
      bad ; 8
      bad ; 9
      )))
```