Quiz 2 Solutions

Problem 1 [25 points]. The Scheme special form **or** behaves as follows: if ev-val(A) = `#f`, then ev-val(`(or` A B`)`) = ev-val(B). If ev-val(A) is not `#f`, then ev-val(`(or` A B`)`) = ev-val(A). (We restrict `or` to having exactly *two* components, viz., no `(or` A B C`)`.)

1(a) [7 points]. Give rewrite rules which completely determine the behavior of `or` according to the above specification.

Solution.

$$\texttt{(or \#f } M\texttt{)} \longrightarrow M$$
$$\texttt{(or } V \; M\texttt{)} \longrightarrow V$$

if V is an immediate value not equal to `#f`*.*

1(b) [6 points]. Give a simple example expression, A, which shows why `(or` A `'dummy)` cannot be regarded as sugar for `(if` A A `'dummy)`.

Solution. *Let A be* `(counter)`*. Then the context*

```
(define counter
   (let ((init 0))
      (lambda () (set! init (+ init 1))
                 init)))
[ ]
```

distinguishes `(or` A `'dummy)`*, which returns 1 in this context, from* `(if` A A `'dummy)`*, which returns 2.*

1(c) [6 points]. Show that, nevertheless, `or` can be regarded as sugar.

Solution. `(or` A B`)` *desugars to*

```
(let ((val-a A))
    (if val-a val-a B))
```

where `val-a` *is a "fresh" variable, specifically a variable which does not occur in B.*[1]

[1] The need for a fresh variable can be avoided by the slightly trickier desugaring

`(let ((val-a` A`) (thunk-b (lambda ()` B`))) (if val-a val-a (thunk-b)))`

1(d) [6 points]. Modify the definition of *control context* (Handout 14, p. 6) to handle the `or` special form, and carefully define an `or`-evaluation redex (cf. Handout 14, p. 6).

Solution.

$$\langle\text{control context}\rangle ::= \ldots \mid \texttt{(or } \langle\text{control context}\rangle\ \langle\text{expression}\rangle\texttt{)}$$

An `or`*-evaluation redex is $E[T]$ where E is an evaluation context and T is an* `or`*-trigger, viz., an expression matching the lefthand sides of the rules in 1(a).*

Problem 2 [25 points]. Let

$$\mathcal{H}_0 ::= \{\, M \mid M \text{ is a Scheme expression and } \texttt{(}M\ \texttt{'}M\texttt{)} \text{ halts} \,\}.$$

2(a) [8 points]. Explain why Scott's Rice's Theorems do *not* directly imply that $\mathcal{H}_0$ is undecidable.

Solution. *$\mathcal{H}_0$ is not evaluation invariant, e.g., let M be*

```
(begin (lambda (qn)
          (if (eq? (car qn) 'begin)
              #t
              <spin>)))
```

$\texttt{(}M\ \texttt{'}M\texttt{)}$ halts so $M \in \mathcal{H}_0$, but ev-val$(M) = \texttt{(}\lambda\texttt{(qn)}\ \cdots\texttt{)}$*, call this P, so $\texttt{(}P\ \texttt{'}P\texttt{)}$ does not halt, and therefore $P \notin \mathcal{H}_0$.*

2(b) [10 points]. Prove that $\overline{\mathcal{H}_0}$ is not half-decidable (that is, not r.e.).

Solution. *Suppose H_c was a half-decider for $\overline{\mathcal{H}_0}$, so for all Scheme expressions, M,*

$$\texttt{(}H_c\ \texttt{'}M\texttt{)} \textit{ halts iff } M \notin \mathcal{H}_0 \textit{ iff } \texttt{(}M\ \texttt{'}M\texttt{)} \textit{ does not halt.}$$

Letting M be H_c yields an immediate contradiction.

2(c) [7 points]. Explain why $\mathcal{H}_0$ is half-decidable.

Solution.

```
(λ(qn)
    (smeval
        (list qn (list 'quote qn))))
```

is a half-decider for $\mathcal{H}_0$.

Problem 3 [25 points]. Let $\mathcal{H}_0$ be as in Problem 2, and let

$$\mathcal{N} ::= \{ \texttt{(0 } S\texttt{)} \mid S \in \mathcal{H}_0 \} \cup \{ \texttt{(1 } S\texttt{)} \mid S \notin \mathcal{H}_0 \}$$

where S ranges over S-expressions.

3(a) [15 points]. Suppose N was a Scheme expression which was a half-decider for $\mathcal{N}$. Fill in the expressions ①, ② below so that the following expression is a decider for $\mathcal{H}_0$:

```
(lambda (qs)
    (define (try n)
        (cond ( (<step-smeval> (list 'N ①) n) #t)
              ( (<step-smeval>       ②       n) #f)
              (else (try (+ n 1)))))
    (try 0))
```

Solution. *For the expression ①, suppose*

$$\text{ev-val}(\texttt{qs}) = \text{ev-val}(\texttt{'}S).$$

We want to apply the metacircular interpreter to a quoted expression for the application of N to `'(0` S`)`. *Remembering that* `'`L *abbreviates* `(quote` L`)`, *we want to apply the metacircular interpreter to*

$$\texttt{'(}N\texttt{ (quote (0 }S\texttt{)))}.$$

Desugaring the outer quote we get

$$\texttt{(list '}N\texttt{ (list 'quote (list 0 '}S\texttt{))}.$$

So expression ① is

```
(list 'quote (list 0 qs)).
```

Similarly, expression ② is

$$\texttt{(list '}N\texttt{ (list 'quote (list 1 qs))}.$$

3(b) [10 points]. Explain why neither $\mathcal{N}$ nor $\overline{\mathcal{N}}$ is half-decidable.

Solution. *If there were a half-decider for $\mathcal{N}$, part (a) implies $\mathcal{H}_0$ would be decidable, which it isn't. Therefore $\mathcal{N}$ is not half-decidable. A symmetric argument shows that a half-decider of $\overline{\mathcal{N}}_0$ would yield a decider for $\overline{\mathcal{H}}_0$, but $\overline{\mathcal{H}}_0$ is not decidable either.*

Problem 4 [25 points]. We consider Scheme extended with

$$\langle\text{primitive procedure id}\rangle ::= \ldots \mid \texttt{int->sym}$$

where `int->sym` computes a one-to-one total function

$$\text{int-to-sym} : \mathbb{N} \to \langle\text{symbol}\rangle\, .n$$

4(a) [20 points]. Let M be a closed Scheme expression which computes a partial function $f : \mathbb{Z} \to \mathbb{Z}$. Sketch how to transform M into an expression M' such that

1. M' also computes f,
2. M' does not contain any symbols,
3. M' does not contain any occurrence of `int->sym`.

Hint: Represent int-to-sym(n) by (the numeral for) pair$(0, n)$, where pair is an integer pairing function. You may assume pair : $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ is total, computable, and one-to-one.

Solution. *M' will use the number* pair$(0, n)$ *to represent int-to-sym(n). To avoid confusing numbers that represent symbols with numbers that are numbers, it must represent numbers in some distinguishable way, say, using the numeral for* pair$(1, n)$ *to represent the number n. (Saying just this much was worth 16/20 points.)*

Now construct M' as follows: replace all numerals, $\hat{n}$ in M, by $\widehat{\text{pair}(1, n)}$ *and symbols* '$\langle\text{name}\rangle$ *by* $\widehat{\text{pair}(0, n)}$ *where int-to-sym* (n) = '$\langle\text{name}\rangle$.

We also must replace occurrences of procedure identifiers operating on numbers or symbols so they work instead on representations of numbers and symbols. For example, the procedure identifiers `symbol?` *and* `integer?` *are replaced by variables (which we presume are fresh)* `rep-symbol?` *and* `rep-integer?`*, which we define as follows;*

```
(define (rep-symbol? n)
  (and (integer? n)
       (= (first n) 0)))

(define (rep-integer? n)
  (and (integer? n)
       (= (first n) 1)))
```

The only other procedure identifiers specifically for symbols are `eq?` *and* `int->sym`*. These we replace by variables* `rep-eq?` *and* `rep-int->sym` *with definitions*

```
(define (make-nth-symbol n) (pair 0 n))

(define (make-int-rep n) (pair 1 n))

(define (unrep-int intrep) (second intrep))

(define (number-of-symbol symrep) (second symrep))

(define (rep-eq? symrep1 symrep2)
  (if (and (rep-symbol? symrep1)
           (rep-symbol? symrep2))
      (= (number-of-symbol symrep1)
         (number-of-symbol symrep2))
      (+ +)))                              ;error


(define (rep-int->sym intrep)
  (if (rep-integer? intrep)
      (make-nth-symbol (unrep-int intrep))
      (+ +)))                              ;error
```

Also, all integer primitive procedure identifiers must be replaced by procedures that work on integer representations. For example, we replace `=` *and* `*` *by* `rep-=` *and* `rep-*` *with definitions*

```
(define (rep-= intrep1 intrep2)
   (if (and (rep-integer? intrep1)
            (rep-integer? intrep2))
       (= (unrep-int intrep1)
          (unrep-int intrep2))
       (+ +)))                 ;error

(define (rep-* intrep1 intrep2)
   (if (and (rep-integer? intrep1)
            (rep-integer? intrep2))
       (make-int-rep (* (unrep-int intrep1)
                        (unrep-int intrep2)))
       (+ +)))                 ;error
```

Quibble*: We assume here that the arithmetic operators like* `*` *are binary, i.e., take only two arguments.*

Finally, let M_0 be the result of all the above replacements, and define M' to be:

```
(lambda (n) (unrep-int (M_0 (make-int-rep n))))
```

4(b) [5 points]. Conclude that the set of $f : \mathbb{Z} \to \mathbb{Z}$ computable in Scheme extended with a one-to-one `int->sym` is the same as the set of $f : \mathbb{Z} \to \mathbb{Z}$ computable in Scheme *without* `int->sym`.

Solution. *Any f computed by M with* `int->sym` *is computed by M' without* `int->sym`, *so having* `int->sym` *leads to no new computable functions from $\mathbb{Z}$ to $\mathbb{Z}$.*