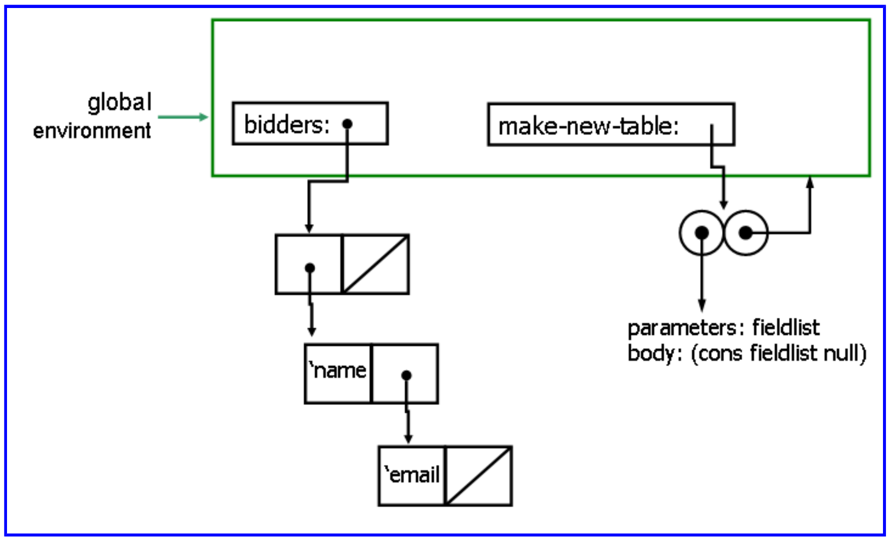
University of Virginia, Department of Computer Science
cs150: Computer Science — Spring 2007

Problem Set 5: Wahoo! Auctions — Comments

Question 1: Draw the global environment after `(define bidders (make-new-table (list 'name 'email)))` is evaluated. You only need to show the environment relevant to the value of `bidders`.

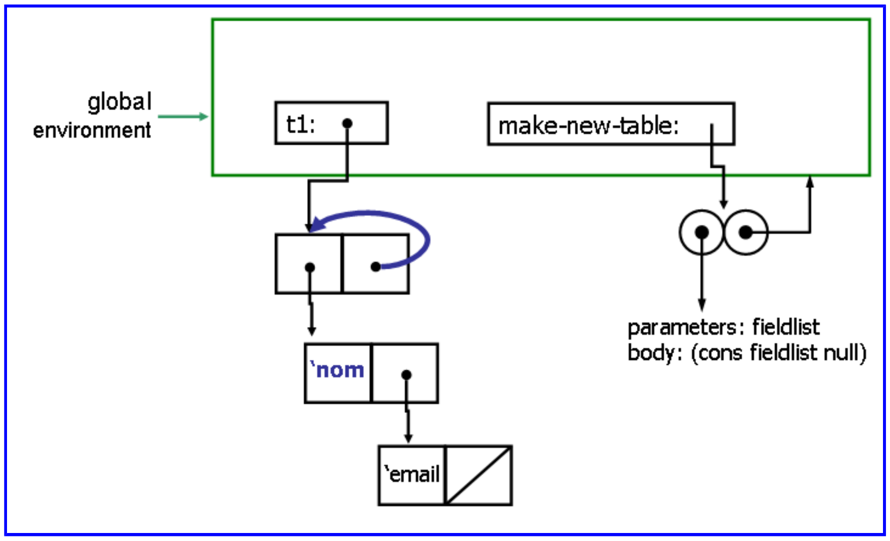
Answer: The global environment contains all the Scheme primitives, but we only show the new definitions. The only important ones for this question are `make-new-table` which is a procedure, amd `bidders` which is defined as `(make-new-table (list 'name 'email))`. To evaluate `(make-new-table (list 'name 'email))` we evaluate the body of the procedure that `make-new-table` evaluates to in an environment pointing to a frame where its parameter (`fieldlist`) is bound to the value of `(list 'name 'email)`. The body of `make-new-table` is `(cons fieldlist null)`. Evaluating `fieldlist` in the application environment produces the value of `(list 'name 'email)`, hence the value of `bidders` is a `cons` cell where the `car` is `(list 'name 'email)` and the `cdr` is `null`.



Question 2:
a. Draw the global environment after the following expressions are evaluated (starting in a clean global environment where `make-new-table`, `table-fields` and `table-entries` are defined as above):

```
(define t1 (make-new-table (list 'name 'email)))
(set-car! (table-fields t1) 'nom)
(set-cdr! t1 t1)
```

Answer: The state after `(define t1 (make-new-table (list 'name 'email)))` is the same as in question 1, except `t1` instead of `bidders`. The `(set-car! (table-fields t1) 'nom)` expression replaces the value of the `car` part of `(table-fields t1)` with `'nom`. The value of `(table-fields t1)` is just `(car t1)`. So, `(set-car! (table-fields t1) 'nom)` replaces the `'name` in `(car (car t1))` with `'nom`. Then, `(set-cdr! t1 t1)` replaces the value in the `cdr` part of `t1` with the value of `t1`. The value of `t1` is the cons cell `t1` points to. Hence, the global environment looks like:



b. Suppose we then evaluate:

```
(define (length lst) (if (null? lst) 0 (+ 1 (length (cdr lst)))))
(length t1)
```

Explain why the evaluation of `(length t1)` never terminates.

Answer: Looking at the diagram, we see that the `cdr` of `t1` is `t1`. So, everytime we evaluate `(cdr t1)` we get `t1`. We can keep evaluating `cdr` forever, but never get to `null`.

Note that the primitive `length` procedure behaves differently:

```
> (length t1)
length: expects argument of type ; given #0=(1 . #0#)
```

The `length` primitive expects a list, but t1 is not a list since it does not end with `null`.

Question 3:
a. Why does the definition of `table-insert!` use `append!` instead of `append`? (Your answer should clearly explain what would go wrong if we used `append`.)

Answer: We use `append!` since we want `table-insert!` to modify the table. If we used `append` it would create a new list instead of modifying the old one. We could define `table-insert!` using `append` like this:

```
(define (table-insert! table entry)
  (assert (= (length entry) (length (table-fields table))))
  (set-car! table (append (table-entries table) (list entry))))
```

This would be simpler, but less efficient since we have to create a new list and copy all of the table entries into it every time we add an entry to the table.

b. Why does the definition of `table-insert!` need to do something different in the case where `(table-entries table)` is null? Hint: try evaluating

```
(define lst null)
(append! lst 3)
lst
```

Answer: To mutate a list, we need a `cons` cell to modify. But, `null` is not a `cons` cell. There is no way to mutate `null` — if we could, it would change the value of `null` everywhere! For `append!` to work, the list passed in must not be `null`, so `append!` has a `cons` cell to mutate using `set-cdr!`.

Question 4:
a. Define `post-item!` and `insert-bid!`.

```
(define (post-item! name description)
  (table-insert! items (list name description)))

(define (insert-bid! bidder item amount)
  (table-insert! bids (list bidder item amount)))
```

b. Describe the running time of your `insert-bid!` procedure.

Answer: Our `insert-bid!` procedure applies `table-insert!` to the `bids` table and the new bid. Creating the new bid using `list` and the three operands is Θ(1). The `table-insert!` procedure includes `(append! (table-entries table) (list entry)))`. Since `append!` is built-in, we have to guess at its running time. If we assume a definition like in SICP Section 3.3:

```
(define (last-pair x)
  (if (null? (cdr x)) x (last-pair (cdr x))))

(define (append! x y)
  (set-cdr! (last-pair x) y)
  x)
```

the running time of `append!` is in Θ(*n*) where *n* is the number of elements in the first list, since it evaluates `(last-pair x)` and `last-pair` cdr's down the list. Hence, `insert-bid!` has running time in Θ(*b*) where *b* is the number of entries in the `bids` table.

Question 5: Define the `table-select` procedure. You may find the `find-element-number`, `get-nth` and `filter` procedures defined in *listprocs.scm* useful (but are not required to use them).

```
(define (find-field-number table field)
  (find-element-number (table-fields table) field))

(define (table-select table field proc)
  (make-table
   (table-fields table)
   (let ((fieldno (find-field-number table field)))
     (filter
      (lambda (x) (proc (get-nth x fieldno)))
      (table-entries table)))))
```

Question 6: Describe the amout of work your `table-select` procedure is using Θ notation. Be careful to specify carefully what any variables you use in your answer mean.

If the number of fields in the table is constant, the `table-select` procedure we defined has running time in Θ(*n*) where *n* is the number of entries in the `table` argument. The `make-table`, `table-fields` and `table-entries` are constant time. The `filter` procedure needs to look at every element in its input list, so its running time is in Θ(*n*) where *n* is the number of elemnts in the input list. This assumes the filter input procedure is constant time. If the number of fields in the table is constant, then that is the case. Thus, the total work is in Θ(*n*) where *n* is the number of entires in the input table.

If the number of fields is not constant, then we need to consider how the work required by `find-field-number` and `get-nth` scales with the selected field. Both procedures need to `cdr` down a list until the matching entry is found. Hence, they are Θ(*m*) where *m* is the number of elements in the input list. As it is used in `table-select` it is the number of fields in the table in both cases. Hence, the total work is in Θ(*nf*) where *n* is the number of entries in the `table` argument and *f* is the number of fields in the `table` argument.

Question 7: Define a `get-highest-bid` procedure that takes an item name as a parameter and evaluates to the bid entry that is the highest bid on that item. You shouldn't assume that the last entry that is a bid on this item is the highest bid for that item.

First, we have to decide what to do if there are no bids on the item. One approach would be to produce an error if there are no bids. This would be the safest approach, but it would make it more difficult to use `get-highest-bid` in other procedures. Instead, we decide that if there are no bids, the highest bid should be `0`.

One way to define `get-highest-bid` is to use `quicksort` to order all the bids by bit amount, and then select the first one:

```
(define (get-highest-bid item)
  (let ((sortbids
         (quicksort
          (get-bids item)
          (lambda (entry1 entry2)
            (> (get-nth entry1 (find-field-number bids 'amount))
               (get-nth entry2 (find-field-number bids 'amount)))))))
    (if (> (length sortbids) 0) (car sortbids) 0)))
```

Note how the definition uses `let` to avoid needing to evalute the expensive bid finding code twice.

Another approach would be to add a zero-bid to the list so we don't need the special case:

```
(define (get-highest-bid item)
  (car
   (quicksort
    (cons (list 'dummy-bidder item 0) (get-bids item))
    (lambda (entry1 entry2)
      (> (get-nth entry1 (find-field-number bids 'amount))
         (get-nth entry2 (find-field-number bids 'amount)))))))
```

The sorting approach was used by most students, but it is quite inefficient since we only care about finding the highest bid. We could do this using `find-best`:

```
(define (get-highest-bid item)
  (let ((fieldno (find-field-number bids 'amount)))
    (find-best
     (lambda (bid1 bid2)
       (if (> (get-nth bid1 fieldno) (get-nth bid2 fieldno)) bid1 bid2))
     (get-bids item))))
```

Question 8: Define a new `place-bid` procedure that satisfies the description above. Don't attempt to do everything at once! Start by satisfying one of the properties first and testing your procedure before trying to satisfy the other property.

```
(define (place-bid bidder item amount)
  (let ((bidder-entry (table-entries
                       (table-select bidders 'name
                                     (make-string-selector bidder))))
        (item-entry (table-entries
                     (table-select items 'item-name
                                   (make-string-selector item)))))
    (if (= (length bidder-entry) 0)
        (printf "~a is not a legitimate bidder!" bidder)
        (if (> (length bidder-entry) 1)
            (printf "Multiple matching bidders: ~a" bidder-entry)
            (if (= (length item-entry) 0)
                (printf "The ~a is not for sale!" item)
                (if (> (length item-entry) 1)
                    (printf "Multiple matching items: ~a" item-entry)
                    (let ((bidder-ent (car bidder-entry))
                          (item-ent (car item-entry)))
                      (let ((highest-bid (get-highest-bid item)))
                        (if (or (null? highest-bid)
                                (> amount
                                   (get-nth highest-bid
                                            (find-field-number bids 'amount))))
                            (begin
                              (printf "~a is now the high bidder for ~a: ~a"
                                      bidder item amount)
                              (insert-bid! bidder item amount))
                            (printf "Bid amount does not exceed previous highest bid: ~a
                                    highest-bid)))))))))
```

Question 9: Define a procedure `end-auction!` that completes the auction. It should go through every item in the `items` table. If there are any bids, the item is sold to the highest bidder and a message displays the winning bid. If there are no bids on the item, the item remains unsold.

```
(define (end-auction!)
   (map
    (lambda (item-entry)
      (let ((item-name (get-nth item-entry (find-field-number items 'item-name))))
        (let ((high-bid (get-highest-bid item-name)))
          (if (zero? high-bid)
              (printf "No bids on ~a.~n"
                      (get-nth item-entry
                               (find-field-number items 'item-name)))
              (printf "Congratulations ~a!  You have won the ~a for $~a.~n"
                      (get-nth high-bid (find-field-number bids 'bidder-name))
                      item-name
                      (get-nth high-bid (find-field-number bids 'amount)))))))
    (table-entries items))
   (void) ;; This makes end-auction! evaluate to no value, instead of the
          ;; list of #void's the map would evaluate to
```

Question 10: How much work is your `end-auction!` procedure? (You answer should use Θ notation, and should clearly define the meaning of all variables you use in your answer.)

The work required by `end-auction!` depends on both the number of items and the number of bids. We use these two variables:

- *m* — the total number of items in the auction
- *b* — the total number of bids in the auction

Our `end-auction!` procedure uses `map` with the input list `(table-entries items)`. The `map` procedure has running time in $\Theta(n)$ work where *n* is the number of elements in the list. **But**, this assumes the amount of work done for each item is constant. In `end-auction!` it is not constant though! For each item, the map procedure evaluates `(get-highest-bid item-name)`, as well as `get-nth` twice. The `get-nth` procedure is $\Theta(n)$ where *n* is the value of its second parameter, since it needs to `cdr` down the list *n* times to find the n^{th} element. In both cases, *n* is constant though, since it is the result of `(find-field-number bids 'bidder-name)` or `(find-field-number bids 'amount)`. Hence, we only need to worry about how much work `(get-highest-bid item-name)` requires.

This depends on the implementation we use. If we use the first definition from Question 7 the running time is in $\Theta(b \log b)$:

```
(define (get-highest-bid item)
  (let ((sortbids
         (quicksort
          (get-bids item)
          (lambda (entry1 entry2)
            (> (get-nth entry1 (find-field-number bids 'amount))
               (get-nth entry2 (find-field-number bids 'amount)))))))
    (if (> (length sortbids) 0)
        (car sortbids)
        0)))
```

The `get-highest-bid` evaluates `(get-bids item)`. This selects matching bids from the bids table. Since `table-select` has running time in $\Theta(n)$ where *n* is the number of entries in the table (question 6), this requires work in $\Theta(b)$. The length of the resulting list is on average `(/ b m)` since all bids are for items in the table. Then, `get-highest-bid` evaluates `(quicksort proc (get-bids item))`. The `quicksort` procedure has running time in $\Theta(n \log n)$ where *n* is the number of elements in the input list (assuming the comparison function is constant time, which it is in this case). Since the length of the input list is b/m, the work required is in $\Theta(b/m \log b/m)$. But, `get-highest-bid` has to do both the work of `get-bids` and then the `quicksort`, so its total work is in $\Theta(b + b/m \log b/m)$.

Evaluating `end-auction!` evaluates `get-highest-bid` once for each item, so the total work is in: $\Theta(mb + b\log b/m)$ On the other hand, if the more efficient `get-highest-bid` procedure were used, it requires work in $\Theta(b + b/m)$ since there is no sorting done. Then, the total running time is in: $\Theta(mb + b) = \Theta(mb)$.