- #nocond: -1pt for not using cond
- #tailrecursive: -1pt for non-tail-recursive functions

Question 1. [3] #tailrecursive

```
let rec find list cmp key = match list with
  | [] -> None
  | (k,v)::_ when cmp key k -> Some v
  | _::xs -> find xs cmp key
```

Question 2. [3] #tailrecursive #nocond

```
(define (find list eq? key)
  (cond ((null? list) #f)
        ((eq? key (caar list)) (cadar list))
        (else (find (cdr list) eq? key))))
```

Question 3. [5]

```
Array extend [
  find: key with: eq [
    1 to: self size do: [:i|
      |tuple|
      tuple := self at: i.
      (key perform: eq with: (tuple at: 1))
        ifTrue: [ ^ tuple at: 2 ].
    ].
    ^ nil.
  ]
].
```

- -1pt for subclass: instead of extend
- -1pt for not using perform:with:

Question 4. [4+3]

```
while ($line = <>) {
   next unless $line =~ m/(\S+)\s*->\s*(\S+)/;
   my ($key, $val) = ($1, $2);
   push @{$graph{$key}}, $val;
}
```

- ok to check for extra \s* at beginning and end
- ok to not use my to declare variables

```
for $key (sort keys %graph) {
   print "$key ->";
   print " $_" for sort @{$graph{$key}};
   print "\n";
}
```

Question 5. [1.5+1.5]

```
let sum = List.fold_left (+.) 0.0
```

- -0.5pt if floats are missing on `+.` and `0.`

```
(define (sum list) (foldl + 0 list))
```

Question 6. [1]

```
Object subclass: Expr [].
```

Question 7. [4]

```
Expr subclass: NumExpr [
   |value|
   NumExpr class >> new: val [ ^ super new init: val ]
   init: val [ value := val ]
   eval [ ^ value ]
   printOn: stream [ ^ stream << value ]
].
```

Question 8. [6]

```
Expr subclass: AddExpr [
   |left right|
   AddExpr class >> new: lf with: rt [
     ^ super new init: lf with: rt
   ]
   init: lf with: rt [ left := lf. right := rt. ]
   eval [ ^ left eval + right eval ]
   printOn: stream [
      ^ stream << '(' << left << '+' << right << ')'
   ]
].
```

Question 9. [4]

```
let rec merge less ls1 ls2 = match ls1, ls2 with
  | [], ls2 -> ls2
  | ls1, [] -> ls1
  | x::xs, y::ys ->
    if less x y then x :: merge less xs ls2
                else y :: merge less ls1 ys;;
```

—or—

```
let rec merge less ls1 ls2 = match ls1, ls2 with
  | [], ls2 -> ls2
  | ls1, [] -> ls1
  | x::xs, y::ys when less x y -> x :: merge less xs ls2
  | x::xs, y::ys -> y :: merge less ls1 ys;;
```

Question 10. [4] #nocond

```
(define (merge less ls1 ls2)
   (cond ((null? ls1) ls2)
         ((null? ls2) ls1)
         ((less (car ls1) (car ls2))
                (cons (car ls1) (merge less (cdr ls1) ls2)))
         (else (cons (car ls2) (merge less ls1 (cdr ls2))))))
```

Question 11. [3]

```
while ($line = <>) {
   $count += $& while $line =~ s/\d+//;
}
print $count, "\n";
```

- ok to use (\d+) and $1 (correctly)

Question 12. [3+1]

```
let rec grep p list = match list with
  | [] -> []
  | x::xs when p x -> x :: grep p xs
  | _::xs -> grep p xs
```

- ok to use `x::xs ->` with an `if then else`

```
let oddpos = grep (fun x -> x > 0 && x mod 2 = 1);;
```

—or—

```
let oddpos = let f x = x > 0 && x mod 2 = 1 in grep f
```

Question 13. [3] #tailrecursive #nocond

```
(define (gcd x y)
    (cond ((> x y) (gcd y (- x y)))
          ((< x y) (gcd x (- y x)))
          (else x)))
```

- ok for tests to be done in any order

Question 14. [1] [grading timelimit=5s]

```
cat $* | tr A-Z a-z | tr -c a-z '\n' | sort | uniq | fmt -65
```

- 1 pt if answer even vaguely resembles this. else 0