Mr S and Mr P

John McCarthy, artificial intelligence expert and inventor of Lisp, proposes this puzzle:

> We pick two numbers a and b so that $a \geq b$ and both numbers are within the range [2,99]. We give Mr P the product $a \times b$ and give Mr S the sum $a+b$. Then the following dialog takes place:
>
> Mr P: I don't know the numbers.
> Mr S: I knew you didn't know.
> I don't know either.
> Mr P: Now I know the numbers.
> Mr S: Now I know them too.
>
> Find the numbers a and b.

McCarthy's mathematical solution is complex and hard to read. Oleg Kiselyov followed McCarthy's solution to create a programming solution in Haskell. This note recreates Oleg's solution in Scheme.

We must augment Scheme's standard library: `upto` returns a list of integers, `filter` removes invalid items from a list, `all` is true only if all items in a list are true, and `list-of` provides list comprehensions:

```
(define (upto m n)
  (if (< n m)
      '()
      (cons m (upto (+ m 1) n))))

(define (filter pred? xs)
  (cond ((null? xs) '())
        ((pred? (car xs))
          (cons (car xs)
                (filter pred? (cdr xs))))
        (else (filter pred? (cdr xs)))))

(define (all pred? xs)
  (cond ((null? xs) #t)
        ((pred? (car xs))
          (all pred? (cdr xs)))
        (else #f)))

(define-syntax list-of
  (syntax-rules (in is)
    ((_ 'aux expr base) (cons expr base))
    ((_ 'aux expr base (x in xs) more ...)
      (let loop ((z xs))
        (if (null? z)
            base
            (let ((x (car z)))
              (list-of 'aux expr
                (loop (cdr z)) more ...)))))
    ((_ 'aux expr base (x is y) more ...)
      (let ((x y))
        (list-of 'aux expr base more ...)))
    ((_ 'aux expr base p? more ...)
      (if p?
          (list-of 'aux expr base more ...)
          base))
    ((_ expr more ...)
      (list-of 'aux expr '() more ...))))
```

Now we turn attention to the puzzle. The Haskell and Scheme solutions both appear below, with Haskell on lines starting with >:

```
> module MrSP where
```

First, we define the list of good numbers that can participate in the solution:

```
> good_nums = [2..99]::[Int]
```

```
(define good-nums (upto 2 99))
```

Given a number p, we want to find all its good factors a and b, with $a \geq b$, and return them (the pairs of them) in a list. We use the obvious and straight forward memoization, pre-computing the factors in a list:

```
> good_factors_table = map gf [0..]
>  where gf p = [ (a,b) | a<-good_nums,
>              b<-good_nums, a >= b, a*b==p ]
```

```
(define good-factors-table
  (let ((gf (lambda (p)
              (list-of (list a b)
                (a in good-nums)
                (b in good-nums)
                (>= a b)
                (= p (* a b))))))
    (map gf (upto 0 10000))))
```

```
> good_factors p = good_factors_table !! p
```

```
(define (good-factors p)
  (list-ref good-factors-table p))
```

The upper limit of $99 \times 99 \approx 10000$ must be specified because Scheme's eager lists are finite but Haskell's lazy lists are infinite. We apply the same memoization to find all the good summands a and b, with $a \geq b$, and return the pairs of summands in a list:

```
> good_summands_table = map gs [0..]
>  where gs p = [ (a,b) | a<-good_nums,
>              b<-good_nums, a >= b, a+b==p ]
```

```
(define good-summands-table
  (let ((gs (lambda (s)
              (list-of (list a b)
                (a in good-nums)
                (b in good-nums)
                (>= a b)
                (= s (+ a b))))))
    (map gs (upto 0 10000))))
```

```
> good_summands p = good_summands_table !! p
```

```
(define (good-summands s)
  (list-ref good-summands-table s))
```

We need to test if a list has only one item:

```
> singleton_p [_] = True
> singleton_p _   = False
```

```
(define (singleton? xs)
  (and (pair? xs) (null? (cdr xs))))
```

We can now encode the dialog between Mr P and Mr S, which provides five facts. The first fact is that Mr P doesn't know the numbers. But Mr P would have known the numbers if the product had had a unique good factorization, so we say:

```
> fact1 (a,b) =
>   not (singleton_p $ good_factors $ a*b)
```

```
(define (fact1? ab)
  (not (singleton?
    (good-factors (apply * ab)))))
```

The second fact is similar; Mr S doesn't know the numbers either:

```
> fact2 (a,b) =
> not (singleton_p $ good_summands $ a+b)
```

```
(define (fact2? ab)
  (not (singleton?
    (good-summands (apply + ab)))))
```

The third fact is that Mr S knows that Mr P doesn't know the numbers. In other words, for all possible summands that make *a*+*b*, Mr P cannot be certain of the numbers:

```
> fact3 (a,b) =
> all fact1 (good_summands $ a+b)
```

```
(define (fact3? ab)
  (all fact1? (good-summands (apply + ab))))
```

Mr S now knows that Mr P doesn't know the numbers. Thus, the fourth fact is that of all factorizations of *a*×*b* there exists only one that makes the third fact true:

```
> fact4 (a,b) =  singleton_p $
>   filter fact3 (good_factors $ a*b)
```

```
(define (fact4? ab)
  (singleton?
    (filter fact3?
      (good-factors (apply * ab)))))
```

The fifth fact is that Mr S knows that Mr P found the numbers. Thus, only one decomposition of *a*+*b* makes the fourth fact true:

```
> fact5 (a,b) = singleton_p $
>   filter fact4 (good_summands $ a+b)
```

```
(define (fact5? ab)
  (singleton?
    (filter fact4?
      (good-summands (apply + ab)))))
```

Finally, we define the list of all pairs of numbers that satisfy all five facts:

```
> result = [(a,b) | a<-good_nums,
>   b<-good_nums, a >= b, all ($ (a,b))
>   [fact1,fact2,fact3,fact4,fact5] ]
```

```
(define result
  (list-of (list a b)
    (a in good-nums)
    (b in good-nums)
    (>= a b)
    (all (lambda (pred?) (pred? (list a b)))
         (list fact1? fact2? fact3?
               fact4? fact5?))))
```

To compute the answer, evaluate `result` in Haskell or Scheme. Both return the singleton list containing the pair of numbers 13 and 4, which is the solution to the puzzle.

For product 52 and sum 17, `fact1?` and `fact2?` are obviously true. The good summands of 17 produce 30, 42, 52, 60, 66, 70 and 72 when multiplied, all of which have multiple factor-pairs, so `fact3?` is true. The good factors of 52 produce 17 and 28 when summed, which both have multiple summand-pairs, so `fact4?` is true. And `fact5?` is true, since the good-summands list of 17 and the good-factors list of 52 have only the pair 13 and 4 in common. The calculation of `result` performs brute-force search through all pairings of *a* and *b*, performing an analysis similar to the one given above; it finds only one pairing that makes all five facts true, which is the unique solution.

Oleg rightly praises the clarity and brevity of the Haskell code, and the literal translation to Scheme shares those attributes. Our exercise shows the translation is simple and concise. The elegance of the code derives from the mathematical nature of the problem, Oleg's good software engineering, and the easy malleability of Scheme, and shows how expressive Haskell and Scheme can be in capable hands; the result is nearly poetic.

References: McCarthy introduced the Mr S and Mr P puzzle in his 1987 paper "Formalization of Two Puzzles Involving Knowledge" available at `http://www-formal.stanford.edu/jmc/puzzles.html`. Oleg's note is available at `http://okmij.org/ftp/Haskell/Mr-S-P.lhs`. The author's email address is `pbewig@gmail.com`.