

Computer Science I

Professor Tom Ellman
Lecture 25

SET-MAP-RELATION

Write a definition of a Scheme procedure called “*set-map-relation*”. This procedure takes a *set* and a *relation* as arguments. It returns a new set in which each member *e* of *set* has been replaced by the set of all values *v* such that the pair (*make-op e v*) is in *relation*.

SET-MAP-RELATION

```
(define *couples* (make-set (make-op 'adam 'eve)
                            (make-op 'romeo 'juliet)
                            (make-op 'bill 'hilary)
                            (make-op 'bill 'monica)))

(define *set* ("set" adam romeo bill))

(set-map-relation *set* *couples*) ==> ("set" ("set" eve)
                                       ("set" juliet)
                                       ("set" monica hilary))

(define set-map-relation (lambda (set relation) ...?...))
```

SET-MAP-RELATION

```
(define set-map-relation
  (lambda (set relation)
    (map-set (lambda (e) (image e relation)) set)))
```

SET-SUM

Write a definition of a Scheme procedure called “*set-sum*”. This procedure takes one argument called “*set*”. The argument *set* is a set of numbers. The procedure *set-sum* returns the sum of the numbers in the set. If the set is empty, *set-sum* returns zero.

```
(set-sum (make-set 1 3 5)) ==> 9
(set-sum (make-set 7)) ==> 7
(set-sum the-empty-set) ==> 0

(define set-sum (lambda (set) ...?...))
```

SET-SUM

```
(define set-sum
  (lambda (set)
    (if (empty-set? set)
        0
        (let ((e (pick set)))
          (+ e (set-sum ((residue e) set)))))))
```

Maximal Argument

Write a definition of a Scheme procedure called “*maximal-argument*”. This procedure takes two arguments called “*set*” and “*fn*”. The argument *set* is a non-empty set. The argument *fn* is a procedure of one parameter. It takes a member of *set* as a parameter and returns a number. The procedure *maximal-argument* returns a member *e* of *set* such that that (*fn e*) has the highest possible value.

```
(maximal-argument (make-set 1 3 5) (lambda (x) x))    ==> 5
(maximal-argument (make-set 1 3 5) (lambda (x) (- x))) ==> 1
(maximal-argument (make-set -5 0 5) (lambda (x) (- (* x x)))) ==> 0

(define maximal-argument (lambda (set fn) ...?...))
```

Maximal Argument

```
(define maximal-argument
  (lambda (set fn)
    (cond ((empty-set? set) 'error)
          ((empty-set? ((residue (pick set)) set)) (pick set))
          (else (let ((e (pick set)))
                  (let ((r (maximal-argument ((residue e) set) fn)))
                    (if (>= (fn e) (fn r)) e r)))))))
```

Maximal Argument

```
(define maximal-argument
  (lambda (fn set)
    (letrec ((helper (lambda (set arg val)
                      (if (empty-set? set)
                          arg
                          (let ((new-arg (pick set)))
                            (let ((rest ((residue new-arg) set))
                                (new-val (fn new-arg)))
                              (if (> new-val val)
                                  (helper rest new-arg new-val)
                                  (helper rest arg val))))))))
      (if (empty-set? set)
          'error
          (let ((arg (pick set)))
            (let ((rest ((residue arg) set))
                  (val (fn arg)))
              (helper rest arg val)))))))
```

Knapsack Problem

A CS101 student purchases every Scheme book in print in order to study for the final exam. Since the exam is open book, he resolves to bring all the books with him. Unfortunately, his knapsack only holds a maximum of *limit* pounds before falling apart. The student forms a set *weights* of numbers that represents the weights of the books. He proceeds to find a subset of these weights that will fit in his knapsack without breaking it. He wants the subset to have as much total weight as possible, while still fitting in the knapsack. Write a procedure called “*pack*” that takes *weights* and *limit* as parameters, and returns a set of numbers indicating which books the student should take to the exam.

Knapsack Problem

```
(pack (make-set 2 4 5) 6) ==> ("set" 2 4)
(pack (make-set 2 4 5) 7) ==> ("set" 2 5)
(pack (make-set 2 4 5) 8) ==> ("set" 2 5)

(define pack (lambda (weights limit) ...?...))
```

Knapsack Problem

```
(define pack
  (lambda (limit weights)
    (maximal-argument (set-comprehension (power-set weights)
                                         (lambda (s)
                                           (<= (set-sum s) limit)))
                     set-sum)))
```

Nerdus Americanus

Suppose we want to simulate the behavior of a simple-minded creature called "Nerdus Americanus". This creature engages in five activities: *sleeping*, *eating*, *programming*, and *debugging*. Its behavior is cyclic: after it sleeps it always eats, after it eats it programs, etc., until after debugging it goes back to sleep.

Define a relation called "**nerd-cycle**" that can be used to represent the behavior of Nerdus Americanus. The relation should be usable with the value procedure in the following way:

```
(define next-activity
  (lambda (activity) ((value activity) *nerd-cycle*)))

(next-activity 'sleeping)    ==> eating
(next-activity 'eating)     ==> programming
(next-activity 'programming) ==> debugging
(next-activity 'debugging)  ==> sleeping
```

Nerdus Americanus

```
(define *nerd-cycle*
  (make-set (make-op 'sleeping 'eating)
            (make-op 'eating 'programming)
            (make-op 'programming 'debugging)
            (make-op 'debugging 'sleeping)))
```

Predict Activity

Write a definition of a Scheme procedure called "*predict-activity*". This procedure takes an argument called "*current-activity*" and an argument called "*n*". The argument *current-activity* is one of the Nerdus Americanus activities. The argument *n* is a number. The procedure *predict-activity* returns a symbol indicating the activity in which Nerdus Americanus will be engaged exactly *n* hours after he begins *current-activity*. You may assume the existence of a procedure called *activity-duration* that takes an activity as a parameter, and returns the number of hours that Nerdus Americanus spends on the activity, before moving on to the next one.

Predict Activity

```
(define *activity-duration*
  (make-set (make-op 'sleeping 4)
            (make-op 'eating 1)
            (make-op 'programming 3)
            (make-op 'debugging 5)))

(define activity-duration
  (lambda (activity) ((value activity) *activity-duration*)))

(activity-duration 'programming) ==> 3

(predict-activity 'sleeping 2) ==> sleeping
(predict-activity 'sleeping 4) ==> eating
(predict-activity 'sleeping 16) ==> sleeping

(define predict-activity (lambda (current-activity n) ...?..))
```

Predict Activity

```
(define predict-activity
  (lambda (current-activity n)
    (if (<= n (activity-duration current-activity))
        current-activity
        (predict-activity (next-activity current-activity)
                          (- n (activity-duration current-activity))))))
```