

Computer Science I

Professor Tom Ellman
Lecture 21

Party Time!

- Whom shall you invite?
- We will write the “Party Planner” program.
- It will select SETS of guests.
- By reasoning with RELATIONS.
- Before we party, we need some tools for manipulating sets and relations.

The Signature of a Procedure

- An expression that describes the data types of the inputs and output of the procedure.
- Something we use to talk about procedures.
- Not part of the Scheme language.

Signature Examples

CAR :	PAIR	OBJ	
CDR :	PAIR	OBJ	
CONS :	OBJ × OBJ	PAIR	
NOT :	BOOL	BOOL	
NULL? :	OBJ	BOOL	
SUM :	LIST	NUMBER	
APPLY :	((OBJ × ... × OBJ)	OBJ) × LIST) OBJ	
MAP :	((OBJ OBJ) × LIST)	LIST	
FOR-ALL? :	(LIST × (OBJ	BOOL))	BOOL
COMPREHENSION :	(LIST × (OBJ	BOOL))	LIST
COUNT-SATISFYING :	(LIST × (OBJ	BOOL))	NUMBER

Building a Set

```
Welcome to Dr. Scheme.  
> (define *guests* the-empty-set)  
> (define *guests* (adjoin 'larry *guests*))  
> (define *guests* (adjoin 'curly *guests*))  
> (define *guests* (adjoin 'moe *guests*))  
> *guests*  
("set" moe curly larry)
```

Testing for Membership in a Set

```
Welcome to Dr. Scheme.  
> ((element 'curly) *guests*)  
#t  
> ((element 'larry) *guests*)  
#t  
> ((element 'moe) *guests*)  
#t  
> ((element 'clinton) *guests*)  
#f  
> ((element 'starr) *guests*)  
#f
```

Testing for Membership in a Set

```
Welcome to Dr. Scheme.  
> ((contains *guests*) 'curly)  
#t  
> ((contains *guests*) 'larry)  
#t  
> ((contains *guests*) 'moe)  
#f  
> ((contains *guests*) 'clinton)  
#f  
> ((contains *guests*) 'starr)  
#f
```

Dismantling a Set

```
Welcome to Dr. Scheme.  
> (define *unlucky* (pick *guests*))  
> *unlucky*  
curly  
> (define *new-guests* ((residue *unlucky*) *guests*))  
> *new-guests*  
("set" moe larry)
```

Dismantling a Set

```
Welcome to Dr. Scheme.  
> (define *unlucky* (pick *guests*))  
> *unlucky*  
larry  
> (define *new-guests* ((residue *unlucky*) *guests*))  
> *new-guests*  
("set" moe curly)
```

PICK Returns a Random Element

- The elements of a set are not ordered.
- There is no first, second, ..., or last element.
- How can we PICK an element of a set?
- It's natural to make the choice random.
- Two successive applications of PICK to the same set may not yield the same value:

```
(pick *guests*)            →       curly  
(pick *guests*)            →       larry
```

Access Functions for Sets

```
(define set-tag "set")  
  
(define the-empty-set (cons set-tag '()))  
  
(define empty-set?  
  (lambda (s) (equal? s the-empty-set)))  
  
(define set?  
  (lambda (arg) (and (pair? arg) (equal? (car arg) set-tag))))
```

The ADJOIN Procedure

(OBJ × SET) SET

```
(define adjoin  
  (lambda (e s)  
    (cons set-tag (cons e (cdr s)))))
```

The PICK Procedure

SET OBJ

```
(define pick
  (lambda (s)
    (let ((lst (cdr s)))
      (if (null? lst)
          (error "pick: The set is empty.")
          (list-ref lst (random (length lst)))))))
```

The RESIDUE Procedure

OBJ (SET SET)

```
(define residue
  (lambda (e)
    (lambda (s)
      (let ((lst (remove-all e (cdr s))))
        (if (null? lst)
            the-empty-set
            (cons set-tag lst))))))
```

The REMOVE-ALL Procedure

(OBJ × LIST) LIST

```
(define remove-all
  (lambda (item lst)
    (cond ((null? lst) '())
          ((equal? (car lst) item) (remove-all item (cdr lst)))
          (else (cons (car lst) (remove-all item (cdr lst))))))
```

The CARDINAL Procedure

SET INTEGER

```
(define cardinal
  (lambda (s)
    (if (empty-set? s)
        0
        (let ((e (pick s)))
          (+ 1 (cardinal ((residue e) s))))))
```

Making a Set All at Once

Welcome to Dr. Scheme.

```
> (define *guests* (make-set 'curly 'larry 'moe))
```

```
> *guests*
```

```
("set" moe larry curly)
```

The MAKE-SET Procedure

(OBJ × ... × OBJ) SET

```
(define make-set
  (lambda (args)
    (letrec ((list-make-set
              (lambda (arg-lst)
                (if (null? arg-lst)
                    the-empty-set
                    (adjoin (car arg-lst)
                            (list-make-set (cdr arg-lst)))))))
      (list-make-set args))))
```

Alternative Representation of SETS

- Our representation allows an element to appear multiple times on the list that represents the set.
- The RESIDUE procedure removes all occurrences of the element from the list.
- An alternative representation would allow an element to appear only once on the list that represents the set.
- The ADJOIN and MAKE-SET procedures would check for and avoid repetitions.

The SET-MAP Procedure

$((\text{OBJ OBJ}) \times \text{SET}) \text{ SET}$

```
(define set-map
  (lambda (fn s)
    (if (empty-set? s)
        the-empty-set
        (let ((e (pick s)))
          (adjoin (fn e) (set-map fn ((residue e) s)))))))
```

Comparing MAP and SET-MAP

```
(define map
  (lambda (fn lst)
    (if (null? lst)
        '()
        (cons (fn (car lst)) (map fn (cdr lst))))))

(define set-map
  (lambda (fn s)
    (if (empty-set? s)
        the-empty-set
        (let ((e (pick s)))
          (adjoin (fn e) (set-map fn ((residue e) s)))))))
```

Why won't this work?

```
(define set-map
  (lambda (fn s)
    (if (empty-set? s)
        the-empty-set
        (adjoin (fn (pick s))
                (set-map fn ((residue (pick s)) s))))))
```

The two occurrences of (pick s) might return different elements of s.

SET-COMPREHENSION

$\text{SET} \times (\text{OBJ BOOLEAN}) \text{ SET}$

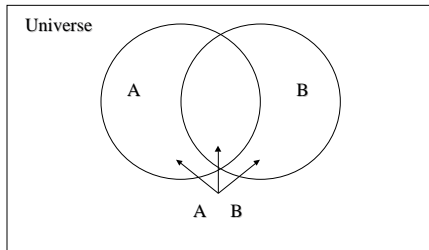
```
(define set-comprehension
  (lambda (set p?)
    (if (empty-set? set)
        the-empty-set
        (let* ((e (pick set)) (rest ((residue e) set)))
          (if (p? e)
              (adjoin e (set-comprehension rest p?))
              (set-comprehension rest p?))))))
```

Comparing COMPREHENSION and SET-COMPREHENSION

```
(define comprehension
  (lambda (lst p?)
    (cond ((null? lst) lst)
          ((p? (car lst))
           (cons (car lst) (comprehension (cdr lst) p?)))
          (else (comprehension (cdr lst) p?))))

(define set-comprehension
  (lambda (set p?)
    (if (empty-set? set)
        the-empty-set
        (let* ((e (pick set)) (rest ((residue e) set)))
          (if (p? e)
              (adjoin e (set-comprehension rest p?))
              (set-comprehension rest p?))))))
```

Union of Two Sets



The UNION Procedure

(SET×SET) SET

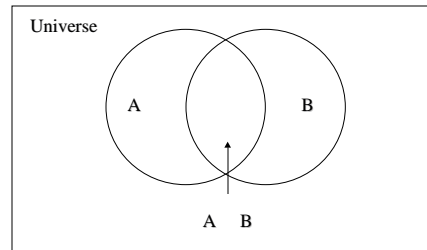
```
(define union
  (lambda (s1 s2)
    (letrec ((helper (lambda (s1)
                       (if (empty-set? s1)
                           s2
                           (let ((e (pick s1)))
                             (if (not ((contains s2) e))
                                 (adjoin e (helper ((residue e) s1)))
                                 (helper ((residue e) s1)))))))
            (helper s1))))))
```

Why is this a bad idea?

```
(define union
  (lambda (s1 s2)
    (apply make-set (append (cdr s1) (cdr s2)))))
```

This definition of union won't work properly if we change the representation of sets to require that each element appear only once on the list representing the set.

Intersection of Two Sets



The INTERSECTION Procedure

(SET×SET) SET

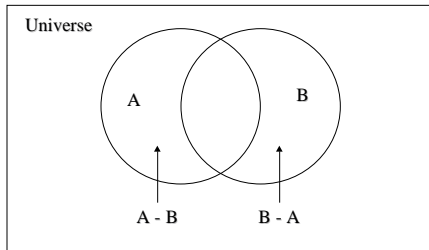
```
(define intersection
  (lambda (s1 s2)
    (letrec ((helper (lambda (s1)
                       (if (empty-set? s1)
                           the-empty-set
                           (let ((e (pick s1)))
                             (if ((contains s2) e)
                                 (adjoin e (helper ((residue e) s1)))
                                 (helper ((residue e) s1)))))))
            (helper s1))))))
```

What is wrong with this?

```
(define intersection
  (lambda (s1 s2)
    (if (empty-set? s1)
        the-empty-set
        (let* ((e (pick s1))
              (temp (intersection ((residue e) s1) s2)))
          (if ((contains s2) e)
              (adjoin e temp)
              temp))))))
```

The second argument of the intersection procedure is the same on each recursive call. In using `letrec` to define a local helper, we need only pass one argument to the recursive helper procedure.

Difference of Two Sets



The DIFFERENCE Procedure

```
(SET×SET) SET
(define difference
  (lambda (s1 s2)
    (letrec ((helper (lambda (s1)
                      (if (empty-set? s1)
                          the-empty-set
                          (let ((e (pick s1)))
                            (if (not ((contains s2) e))
                                (adjoin e (helper ((residue e) s1)))
                                (helper ((residue e) s1)))))))
      (helper s1))))
```

Testing for Equality of Sets

```
> (define *guests1* (make-set 'curly 'larry 'moe))
*guests1*
> *guests1*
("set" moe larry curly)
> (define *guests2* (make-set 'moe 'larry 'curly))
*guests2*
> *guests2*
("set" curly larry moe)
> ((set-equal *guests1*) *guests2*)
#t
```

Testing for Equality of Sets

```
> (define *guests1* (make-set 'curly 'larry 'moe))
*guests1*
> *guests1*
("set" moe larry curly)
> (define *guests2* (make-set 'curly 'larry 'moe 'larry))
*guests2*
> *guests2*
("set" larry moe larry curly)
> ((set-equal *guests1*) *guests2*)
#f
```

Comparing SET-EQUAL and EQUAL?

- The SET-EQUAL procedure tests whether two sets have the same elements, ignoring the order of elements and repetitions of elements.
- The EQUAL? procedure tests whether two lists have the same members, in the same order, with the same repetitions.
- Since sets are implemented as lists, we can use EQUAL? on sets, but we may not get the right answer!

The SET-EQUAL Procedure

```
OBJ (OBJ BOOLEAN)
(define set-equal
  (lambda (o1)
    (lambda (o2)
      (or (and ((neither set?) o1 o2)
                (equal? o1 o2))
          (and ((both set?) o1 o2)
                ((subset o1) o2)
                ((subset o2) o1))))))
```

Two objects are SET-EQUAL to each other if either one of the following things is true: (1) Neither object is a set, and the two objects are EQUAL? (2) Both objects are sets and each set is a subset of the other.

The BOTH and NEITHER Procedures

(OBJ BOOLEAN) ((OBJ × OBJ) BOOLEAN)

```
(define both
  (lambda (p)
    (lambda (arg1 arg2)
      (and (p arg1) (p arg2)))))

(define neither
  (lambda (p)
    (lambda (arg1 arg2)
      (not (or (p arg1) (p arg2))))))
```

SUBSET and SUPERSET

SET (SET BOOLEAN)

```
(define superset
  (lambda (s1)
    (lambda (s2)
      ((for-all (contains s1) s2))))
```

Tests if s1 is a superset of s2.

```
(define subset
  (lambda (s1)
    (lambda (s2)
      ((superset s2) s1))))
```

Tests if s1 is a subset of s2.

Set s1 is a superset of set s2 if s1 contains all elements of s2. Set s1 is a subset of set s2 if s2 is a superset of s1.

ELEMENT

OBJ (SET BOOLEAN)

```
(define element (compose there-exists set-equal))
```

...or alternatively ...

```
(define element (lambda (x) (there-exists (set-equal x))))
```

An object is an element of a set if there exists something in the set that is SET-EQUAL to it.

CONTAINS

SET (OBJ BOOLEAN)

```
(define contains (lambda (s)
  (lambda (obj) ((element obj) s))))
```

A set s contains an object obj if obj is an element of s.

Sets of Sets of Sets ... Etc.

- How to determine if S is a member of set T, when S is itself a set?
- Try to find an element E of T such that S and E are equal sets.
- Notice that ELEMENT indirectly calls SET-EQUAL.
- Notice that SET-EQUAL indirectly calls ELEMENT.
- “Mutual Recursion”.

The NONE Procedure

(OBJ BOOLEAN) (SET BOOLEAN)

```
(define none
  (lambda (obj-pred?)
    (lambda (s)
      (letrec ((set-pred? (lambda (s)
        (or (empty-set? s)
            (let ((e (pick s)))
              (and (not (obj-pred? e))
                  (set-pred? (residue e) s)))))))
        set-pred?)))
```

NONE takes a predicate OBJ-PRED? on objects and returns a predicate SET-PRED? on sets. (SET-PRED? S) is true if and only if (OBJ-PRED? E) is false for each member E of S.

FOR-ALL and THERE-EXISTS

(OBJ BOOLEAN) (SET BOOLEAN)

```
(define for-all  
  (lambda (p) (none (compose not p))))
```

FOR-ALL takes a predicate on objects and returns a predicate on sets. ((FOR-ALL P) S) is true if and only if (P E) is true for all elements E of S.

```
(define there-exists  
  (lambda (p) (lambda (s) (not ((none p) s)))))
```

THERE-EXISTS takes a predicate on objects and returns a predicate on sets. ((THERE-EXISTS P) S) is true if and only if (P E) is true for at least one element E of S.

POWER-SET

SET SET-OF-SETS

```
(define power-set  
  (lambda (s)  
    (if (empty-set? s)  
        (make-set the-empty-set)  
        (let ((e (pick s)))  
          (let ((new (power-set ((residue e) s))))  
            (union new  
                  (set-map (lambda (s) (adjoin e s))  
                          new)))))))
```

POWER-SET takes a set as input and returns a set as output. (POWER-SET S) is the set containing all the subsets of S.