Quick and dirty set implementation

Higher-order functions lead to compact code:

```
-> (val emptyset '( ))
-> (define member?      (x s)
     (exists? ((curry equal?) x) s))
-> (define add-element (x s)
     (if (member? x s) s (cons x s)))
-> (define union      (s1 s2) (foldl add-element s1 s2))
-> (define set-from-list (l) (foldl add-element '( ) l))
-> (union '(1 2 3 4) '(2 4 6 8))
(8 6 1 2 3 4)
```

A few higher-order functions go a long way

Example: Quicksort in 10 lines

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Generalized equality for alists

"Built-in" equal won't do for association lists

A-lists equal if each has same associations as the other:

$$al_1 = al_2 \text{ iff } al_1 \subseteq al_2 \wedge al_2 \subseteq al_1$$

```
(define sub-alist? (al1 al2)
  (not (exists?
          (lambda (pair)
             (not (equal? (cadr pair)
                          (find (car pair) al2))))
          al1)))
```

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Equality from subset

```
-> (define =alist? (al1 al2)
       (if (sub-alist? al1 al2) (sub-alist? al2 al1) #f))
-> (=alist? '( ) '( ))
#t
-> (=alist? '((E coli)(I Magnin)(U Thant))
            '((E coli)(I Ching)(U Thant)))
#f
-> (=alist? '((U Thant)(I Ching)(E coli))
            '((E coli)(I Ching)(U Thant)))
#t
```

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Sets of alists

Where to put the equality function?

1. **Extra argument — awkward**

```
(define member? (x s eqfun)
   (exists? ((curry eqfun) x) s))
(define add-element (x s eqfun)
   (if (member? x s eqfun) s (cons x s)))
```

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Sets of alists, continued

2. Make set pair `(eqfun . elems)` of equality function, elements:

```
(define mk-set (eqfun elements) (cons eqfun elements))
(define eqfun-of    (set) (car set))
(define elements-of (set) (cdr set))
(val emptyset (lambda (eqfun) (mk-set eqfun '( ))))
(define member? (x s)
   (exists? ((curry (eqfun-of s)) x) (elements-of s)))
(define add-element (x s)
   (if (member? x s) s
       (mk-set (eqfun-of s) (cons x (elements-of s)))))
```

Works, but costs an extra cons cell per instance

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Sets of alists, continued

3. Curry — equality function as part of operations:

```
(val mk-set-ops (lambda (eqfun)
   (list2
      (lambda (x s) (exists? ((curry eqfun) x) s))
      (lambda (x s)
          (if (exists? ((curry eqfun) x) s) s
                           (cons x s))))))
(val al-nullset '( ))
(val list-of-al-ops (mk-set-ops =alist?))
(val al-member?      (car list-of-al-ops))
(val al-add-element (cadr list-of-al-ops))
```

Must create new ops for every new equality test

best w/few types, static checking

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Continuations — what to do next

Direct style: functions finish by returning a value

Continuation-passing style (CPS): functions finish by "throwing" value to continuation

- Not like a call, because it never returns
- "Goto with arguments"

Can simulate with ordinary tail call

Direct:	`return answer;`
True CPS:	`throw k answer;`
Simulated CPS:	`return k(answer);`

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Application of continuations

`find` can't distinguish "unbound" from "bound to nil"

```
(define find (x alist)
  (if (null? alist) '( )
    (if (equal? x (caar alist))
      (cadar alist)
      (find x (cdr alist)))))
```

Could use different kinds of return values, e.g.,

- if found, `(cons #t (caar alist))`
- if not found, `(cons #f 'irrelevant)`

But this is clunky — code gets cluttered, easy to forget test

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Success and failure continuations

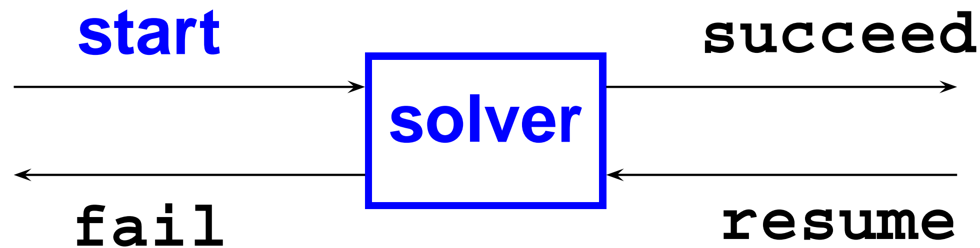
```
(define find-c (key alist success-cont failure-cont)
  (letrec
     ((search (lambda (alist)
                  (if (null? alist)
                      (failure-cont)
                      (if (equal? key (caar alist))
                          (success-cont (cadar alist))
                          (search (cdr alist)))))))
    (search alist)))
```

Example: table with default

```
(define find-default (key table default)
       (find-c key table (lambda (x) x)
                         (lambda ( ) default)))
```

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Continuations for search problems



start	Begin with partial solution
`fail`	Partial solution won't work
`succeed`	Pass improved solution to next step
`resume`	If improved solution won't work, backtrack and try something else

A composable unit! (We will use it again later this semester to implement μProlog)

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Moral: functions are cheap

Use lots of them

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Semantics and implementation of μScheme

Key changes from Impcore:

- New language constructs: let, lambda, function application
- New values, including functions (closures)
- Single environment
- Environments get copied
- Environment maps names to mutable locations (not values)

©Copyright 2006 Norman Ramsey. All Rights Reserved.

μScheme vs Impcore

New abstract syntax:

- LET (keyword, names, expressions, body)
- LAMBDAX (formals, body)
- APPLY (exp, actuals)

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Evaluation rules

Judgment $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$

σ **is the store**

$$\frac{x \in \operatorname{dom} \rho \qquad \rho(x) \in \operatorname{dom} \sigma}{\langle \mathsf{VAR}(x), \rho, \sigma \rangle \Downarrow \langle \sigma(\rho(x)), \sigma \rangle} \quad (\text{VAR})$$

$$\frac{x \in \operatorname{dom} \rho \qquad \rho(x) = \ell \qquad \langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \mathsf{SET}(x, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma'\{\ell \mapsto v\} \rangle} \quad (\text{ASSIGN})$$

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Implementation of closures

Key issue: values of free variables

Static scoping:

- at the location of `lambda`, "look outward" for ρ
- keep that ρ until we need it

$$\frac{}{\langle \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle \langle\!\langle \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho \rangle\!\rangle, \sigma \rangle} \quad (\text{MkClosure})$$

So, create closure in `eval` by

```
case LAMBDAX:
    return mkClosure(e->u.lambdax, env);
```

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Applying closures

Saved environment for free variables

Arguments for bound variables (≡ formal parameters)

$$\frac{\begin{array}{c}\ell_1, \ldots, \ell_n \notin \mathrm{dom}\,\sigma \\ \langle e, \rho, \sigma \rangle \Downarrow \langle \langle\!\langle \mathsf{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e_c), \rho_c \rangle\!\rangle, \sigma_0 \rangle \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e_c, \rho_c\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}, \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle\end{array}}{\langle \mathsf{APPLY}(e, e_1, \ldots, e_n), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad (\textsf{ApplyClosure})$$

```
nl = f.u.closure.lambda.formals;
return eval(f.u.closure.lambda.body,
            bindalloclist(nl, vl, f.u.closure.env));
```

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Locations in closures

The key is shared mutable state

```
-> (val resettable-counter-from
     (lambda (n)
        (list2
          (lambda ( ) (set n (+ n 1)))
          (lambda ( ) (set n 0)))))
-> (val twenty  (resettable-counter-from 20))
-> ((car twenty))
21
-> ((car twenty))
22
-> ((cadr twenty))
0
-> ((car twenty))
1
```

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Real closures

As in our book, real closures are stored on the heap.

BUT:

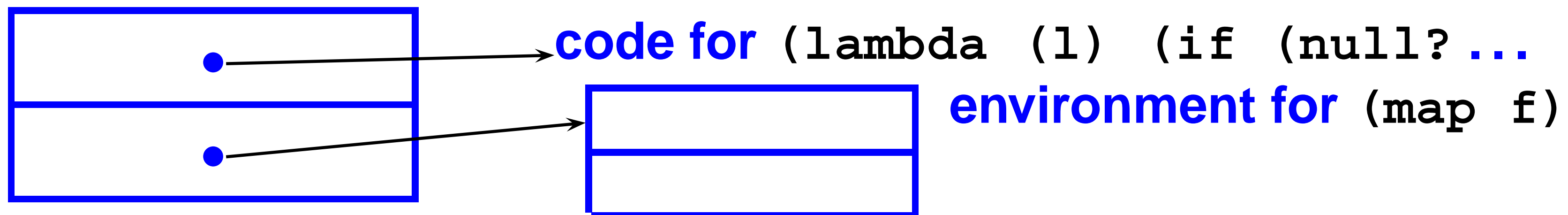
- contain only free vars needed, not entire ρ
- name lookup can be done at compile time

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Closures in pictures

```
(val map (lambda (f)
  (lambda (l)
    (if (null? l) '( )
      (cons (f (car l)) ((map f) (cdr l)))))))
```

closure for `(map f)`



So top-level names not stored in closure

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Closure optimizations

Major issue in making functional programs efficient

Heavy static analysis for:

- keeping closures on the stack
  - (when used as downward funargs)
- sharing closures
  - (e.g., mutually recursive functions)
- eliminating closures
  - (e.g., when functions never escape)

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Scoping Alternatives for Language Designers

© Copyright 2006 Norman Ramsey. All Rights Reserved.

Scoping

Problem also called *name resolution*:

given an occurrence of variable `x`, which declaration of `x` is meant?

or

what is denoted by this occurrence of `x`?

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Scoping — denotations

In basic Impcore's interpreter, denotations include

- locations, or the values stored in them
- functions

In Scheme, we have only locations/values

A function is a closure is a value stored in a location

Scoping rules come in two categories

static — can answer for each `x` at compile time

dynamic — can't determine until run time

(might change during program execution!)

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Static Scoping

What we all know and love

Works by examining source text

Scope of a declaration **of `x`**

region of source text in which occurrences of `x` resolve to that declaration

Typical scoping strategy:

- **divide source regions into "blocks"**
- **scope runs from declaration to end of block (definition before use)**
- **used in Algol, Pascal, C, Java, Scheme, ML, ...**

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Alternative static scoping

Requiring definition before use makes mutually recursive types and functions ugly

Idea: make scope *entire* unit containing declaration including before declaration

- **Makes recursive types, functions trivial**
- **Found in Modula-3, Haskell**

©Copyright 2006 Norman Ramsey. All Rights Reserved.

More Static Scoping

Sometimes blocks can be nested:

- **inner declaration can “hide” outer declaration**
- **outer declaration now suffers from “hole in the scope”**

Scope rules easy to express as computations on environments

- **like our interpreters, but can also be done at compile time**

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Scoping — block structure

(Misleading) term block structure refers to nested functions

- without block structure, no funarg problems (since all nonlocals must be global and live forever)
- therefore, functions are perfectly good values
- C, C++, Icon fit this category

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Static Scoping with Multiple Name Spaces

One name, one region, many meanings

Example, C:

- variable names
- typedef names
- struct and union tags

Each "name space" is simply an environment

More C:

- members of `struct` define their own name space
- typical compilers keep a tiny environment with the declaration of the structure type

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Multiple Name Spaces, continued

Recall Impcore: functions in one name space, variables in another

```
-> (define f (x) (+ x 1))
-> (set f 3)
-> (f f)
4
```

Multiple name spaces have good and bad points:

- permits more natural use of names
- too many name spaces can confuse the user (author's opinion: C is pushing the limit)

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Dynamic Scoping

Has origins in an implementation bug in early LISP

- resolve free variables using "currently active" functions

  (walk up the call stack looking for parameter names)
- makes `lambda` essentially useless
- but environments are never copied (no closures)

Other languages have other dynamic scope rules

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Object-oriented Dynamic Scoping

- Can't tell what "method" will be invoked at compile time
- But, can sometimes get some static checking anyway
- Most O-O languages have static (early) scoping for variables and dynamic (late) scoping for methods

©Copyright 2006 Norman Ramsey. All Rights Reserved.

Dynamic Scoping — a bug or a feature?

Definitely makes programs harder to understand:

- changing the names of formal parameters can change the meaning of a function!

Precludes much static analysis

- can't find mistakes until run time — if then!

Lessons of history?

- LISP rule is almost certainly wrong
- Object-oriented rule will save the world

©Copyright 2006 Norman Ramsey. All Rights Reserved.