CS603 Programming Language Organization

Lecture 16

Spring 2004

Department of Computer Science

Overview

- Questions
- Finishing μ-Scheme
- Reading for next time

HOF for Polymorphism: List of Functions

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

Pair Up:

- Draw a diagram (of environment, cons cells and closures) for `(val list-of-al-ops (mk-set-ops =alist?))`

HOF for Polymorphism: List of Functions

```
-> (val al-member? (car list-of-al-ops))
-> (val al-add-element (cadr list-of-al-ops))
```

So uses will look like:

```
->(val emptyset '())
->(val s (al-add-element '((U Thant)((I Ching)))
                         emptyset)
```

Pair Up: What is the value of s?

```
(((U Thant) (I Ching)))
->(val s (al-add-element '((E coli)(I Ching)) s))
```

Pair Up: What is the value of s?

```
(((E coli) (I Ching)) ((U Thang) (I Ching)))
```

A polymorphic, higher-order sort

```
->(define mk-insertion-sort (lt)
      (letrec (
          (insert (lambda (x l)
              (if (null? l) (list1 x)
                  (if (lt x (car l))
                      (cons x l)
                      (cons (car l) (insert x (cdr l)))))))
          (sort (lambda (l)
              (if (null? l)
                  '()
                  (insert (car l) (sort (cdr l)))))))
      sort))
```

A polymorphic, higher-order sort (cont.)

```
->(val sort< (mk-insertion-sort <))
->(val sort> (mk-insertion-sort >))
->(sort< '(6 9 1 7 4 3 8 5 2 10))
(1 2 3 4 5 6 7 8 9 10)
->(sort> '(6 9 1 7 4 3 8 5 2 10))
(10 9 8 7 6 5 4 3 2 1)
->(define pair< (p1 p2)
      (or (< (car p1) (car p2)
          (and (= (car p1) (car p2))
               (< (cadr p1) (cadr p2)))))
->((mk-insertion-sort pair<) '((4 5) (2 9) (3 3) (8 1) (2 7)))
((2 7) (2 9) (3 3) (4 5) (8 1))
```

μ-Scheme Concrete Syntax

toplevel ::= *exp*
| `(use` *file-name*`)`
| `(val` *variable-name exp*`)`
| `(define` *function-name* `(`*formals*`)` *exp*`)`

exp ::= *literal*
| *variable-name*
| `(if` *exp exp exp*`)`
| (while exp exp)
| `(set` *variable-name exp*`)`
| `(begin` {*exp*}`)`
| (*exp* {*exp*})
| (*let-keyword* ({(*variable-name exp*)}) *exp*)
| `(lambda` (*formals*) *exp*`)`
| *primitive*

let-keyword ::= `let` | `let*` | `letrec`

μ-Scheme Concrete Syntax (cont.)

formals ::= {*variable-name*}
literal ::= *integer* | `#t` | `#f` | `'` *S-exp* | `(quote` *S-exp* `)`
S-exp ::= *literal* | *symbol-name* | `(`{*S-exp*}`)`
primitive ::= `+` | `-` | `*` | `/` | `=` | `<` | `>` | `print` | `error`
| `car` | `cdr` | `cons`
| `number?` | `symbol?` | `pair?` | `null?` | `boolean?`
| `procedure?`
integer ::= sequence of digits, possibly prefixed with a minus sign
**-name* ::= sequence of characters not an *integer* and not containing `(`, `)`, `;`, or whitespace