Imperative Programming in Scheme

CS496

Imperative Programming in Scheme

1. Variable assignment and sharing
   - Assignment
   - Local state
   - Sharing
   - Sequencing
2. Data structure mutation
3. I/O

Imperative Features in Scheme

Scheme (as seen so far) is purely functional

- every expression is evaluated solely for its value

This lack of side-effects has an important consequence

- purely functional languages are said to enjoy referential transparency
  - This means that the order in which subexpressions are evaluated, in some large expression, is irrelevant

As a result, one

- can use standard algebraic equations (eg. $a + b = b + a$) to reason about programs
- can easily parallelize

Imperative Features in Scheme

However sometimes imperative features are needed

- variable assignment and destructive update of data structures (specially for efficiency reasons)
- I/O: communication with some external device

Therefore, Scheme is enriched with expressions that are evaluated solely for their effects

- variable assignment and destructive update of data structures:

  `set!`, `set-car!`, `set-cdr!`, ...

- I/O: communication with some external device:

  `display`, `read`, `write`, ...

Mutable Variable Assignment

To introduce variable assignment, we must think of variables as being bound to locations.

```
(set! var exp)
```

- `(set! var exp)` assigns the value of `exp` to `var`
- "!" signifies "use with caution"
- We'll see some examples

Examples

```
> (define x 1)
> (set! x 2)
> x
2
> (set! z 2)
ERROR:
 set!: assignment disallowed;
 cannot set variable before its definition
  variable: z
```

Examples

- What does this expression evaluate to?

```
> (let ((y 3))
    (let ((dummy (set! y 4)))
        y))
```

Examples

- What does this expression evaluate to?

```
>   (let ((y 3))
      (let ((w (+ y y))
            (dummy (set! y 4)))
       w))
```

- It helps illustrate that Scheme uses static scoping
- If it used dynamic scoping what would the result have been?

Value denoted by an Assignment

```
(set! x 3)
```

- Is evaluated to cause an effect (not to return a value)
- However, since all expressions in Scheme denote values, what value does an assignment denote?
  - A special value #<void>
- Not typically visible

```
> (define x 2)
> (set! x 3)
```

- Can be made visible as follows:

```
> (define x 2
> (write (set! x 3))
#<void>
> x
3
```

Assessment

- Consider an expression such as

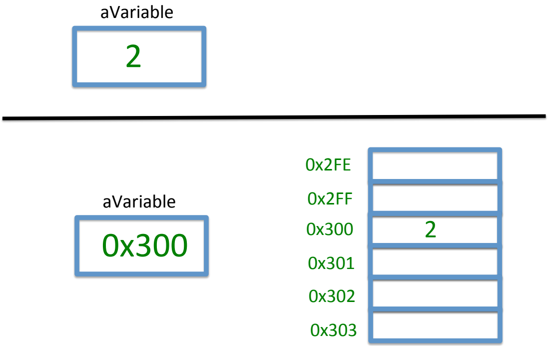
```
(set! x (+ x 1))
```

- There are two references to `x` (distinguished with two colors, blue and red)
- These references do not denote the same thing
  - x refers to the address or location of `x`
  - x refers to the contents of `x`

Assessment

`(set! x (+ x 1))`

- Before:
  - Variables were bound to values
- With assignment:
  - Variables are bound to locations (aka references)
  - There is a level of indirection



L/R-values

```
(set! x (+ x 1))
```

- L-value: name used for x (refers to its location Eg. 0x300)
- R-value: name used for x (refers to its value Eg. 2)

Note that:

- Locations are denotable (they may be bound to identifiers)
- but not expressable (they cannot be the result of an expression).

Sequencing

- Now that expressions can be evaluated to cause an effect (rather than return a value) the order in which they are evaluated is important

```
> (define x 2)
> x
2
> (set! x 3)
```

```
> (define x 2)
> (set! x 3)
> x
3
```

- So we need a way of putting expressions in a sequence so that they are evaluated in the order in which they appear

Sequencing

- Can be achieved using the `begin e1 e2 ... en` construct

```
> (define x 2)
> (begin x (set! x 3))
```

```
> (define x 2)
> (begin (set! x 3) x)
3
```

- This construct evaluates each argument in turn and returns the value of the last one

Local State

```
> (define counter
    (let ((local-state 0))
      (lambda ()
        (let ((dummy (set! local-state (+ local-state 1))))
          local-state))))
> (counter)
1
> (counter)
2
> (eq? (counter) (counter))
#f
```

- We know that, in algebra, $x + x = 2 * x$ for x any number
- Can we therefore replace `(+ (counter) (counter))` with `(* 2 (counter))`?

Imperative Features in Scheme

Mutable Pairs and Lists

Simple Input/Output

Implementing a Mutable Cell using Mutable Vectors

Mutable Pairs and Lists

- There is a mutable version of cons cells in Racket to build mutable lists
- The function `mcons` (instead of `cons`) builds a mutable cell.
- The functions `set-mcar!` and `set-mcdr!` change the car and cdr of the cell.
- There are additional functions like `mlist` and `mlength` that correspond to functions like `list` and `length` for ordinary lists.

Mutable Pairs and Lists

```
> (define c (mcons 1 2))
> (define d (mcons 0 c))
> (define e (mcons 0 c))
> c
(mcons 1 2)
> d
(mcons 0 (mcons 1 2))
> e
(mcons 0 (mcons 1 2))
> (equal? d e)
#t
> (eq? d e)
#f
```

Mutable Pairs and Lists

```
> (define c (mcons 1 2))
> (define d (mcons 0 c))
> (define e (mcons 0 c))
> (set-mcdr! c 5)
> d
(mcons 0 (mcons 1 5))
> e
(mcons 0 (mcons 1 5))
```

Mutable Pairs and Lists

- A circular list

```
> (define circ (mcons 'a (mcons 'b '())))
> (set-mcdr! (mcdr circ) circ)
> (mcar circ)
'a
> (mcar (mcdr (mcdr circ)))
'a
```

Reverse!

```
>  (define reverse!
    (letrec ((loop
               (lambda (last ls)
                 (let ((next (mcdr ls)))
                   (set-mcdr! ls last)
                   (if (null? next)
                       ls
                       (loop ls next))))))
      (lambda (ls)
        (if (null? ls)
            ls
            (loop '() ls)))))
> (define ls (mcons 1 (mcons 2 (mcons 3 '()))))
> (reverse! ls)
(3 2 1)
> ls
(1)
>
```

Parameter Passing

What is the value of “????”?

```
> (define x 2)
> (define (modify y)
    (set! y 5))
> (modify x)
> x
????
```

Parameter Passing

What is the value of "????"?

```
> (define x 2)
> (define (modify y)
    (set! y 5))
> (modify x)
> x
????
```

- The value is 2 (Racket passes numbers by copying them)
- Mutable lists are, however, are passed by reference

Parameter Passing

```
> (define x (mcons 1 2))
> (define (modify y)
      (set-mcdr! y 5))
> (modify x)
> x
(mcons 1 5)
```

Simple I/O

```
 > (write "pistol pete")
pistol pete

> (write (+ 2 2))
4

> (newline)
```

Other I/0

- `(write <exp>)`: prints literal representation
- `(read-char)`: reads a character
- `(read)`: reads a datum

Printing a Value

Racket provides three ways to print an instance of a built-in value:

- `print`, which prints a value in the same way that is it printed for a REPL result; and
- `write`, which prints a value in such a way that read on the output produces the value back; and
- `display`, which tends to reduce a value to just its character or byte contentat least for those datatypes that are primarily about characters or bytes, otherwise it falls back to the same output as write.

`https://docs.racket-lang.org/guide/read-write.html`

Sequencing

- As mentioned, in the presence of side effects, a specified order of evaluation is critical
- We introduced explicit sequencing through the following expression:

```
(begin <exp1> <exp2> ... <expn>)
```

- Since I/O operations also cause effects, we can use this construct to produce them in a specific order

Example

```
> (begin (display "Hi")
         (newline)
         (display "there!")
         (newline))
Hi there!
> (begin (display "Hello") (+ 1 2) 55)
Hello
55
>
```

Example (Read-eval-print loop)

Built-in interpreter: `(eval <exp>)`

```
> '(+ 1 2)
(+ 1 2)
> (eval '(+ 1 2))
3
> (eval (list '+ 1 2))
3
> (eval (list (list 'lambda (list 'x) 'x) (list '+ 1 2)))
3
```

Example (Read-eval-print loop)

```
> (define read-eval-print
    (lambda ()
      (display "--> ")
      (write (eval (read)))
      (newline)
      (read-eval-print)))
> (read-eval-print)
--> (+ 1 2)
3
--> (car (cons "foo" 'foo))
"foo"
--> (cdr 3)
ERROR: cdr: Wrong type in arg1 3
>
```

Imperative Features in Scheme

Mutable Pairs and Lists

Simple Input/Output

Implementing a Mutable Cell using Mutable Vectors

Vector-set!

`vector-set!`:

```
> (define v (vector 1 2 3))
> (vector-set! v 1 4)
>v
#(1 4 3)
```

Cell ADT

```scheme
(define cell-tag "cell")
(define make-cell
  (lambda (x)
    (vector cell-tag x)))
(define cell?
  (lambda (x)
    (and (vector? x)
         (= (vector-length x) 2)
         (eq? (vector-ref x 0) cell-tag))))
(define cell-ref
  (lambda (x)
    (if (cell? x)
        (vector-ref x 1)
        (error "Invalid argument:" x))))
(define cell-set!
  (lambda (x value)
    (if (cell? x)
        (vector-set! x 1 value)
        (error "Illegal argument:" x))))
```

Cell ADT (cont)

```
(define cell-swap!
  (lambda (cell-1 cell-2)
    (let ((temp (cell-ref cell-1)))
      (cell-set! cell-1 (cell-ref cell-2))
      (cell-set! cell-2 temp))))
Example:
> (define c (make-cell 3))
> (define c1 (make-cell 100))
> (cell-set! c 8)
> (cell-ref c)
> (cell-swap! c c1)
> (cell-ref c)
>
```