CSEP 505: Programming Languages

Lecture 10
March 12, 2015

```
#define SQR(x)    x * x
```

```
#define SQR(x)    x * x


SQR(1 + 2)
```

```
#define SQR(x)    x * x
```

```
SQR(1 + 2)  →    1 + 2  *  1 + 2
```

```
#define SQR(x)   (x)*(x)
```

```
SQR(1 + 2)  →  (1 + 2) * (1 + 2)
```

```
#define SQR(x)   (x)*(x)
```

```
100 / SQR(5)   →
```

```
#define SQR(x)   (x)*(x)
```

```
100 / SQR(5)   →   100 /   (5) * (5)
```

```
#define SQR(x) ((x)*(x))
```

```
100 / SQR(5)  →  100 / ((5) * (5))
```

```
(define (member elt lst)
  (if  (not (empty? lst))
       (if (eq? elt (first lst))
           true
           (member elt (rest lst)))
       false))
```

```
(define (member elt lst)
  (and (not (empty? lst))
       (or (eq? elt (first lst))
           true
           (member elt (rest lst)))
       false))
```

```
(define (or e1 e2)
  (if e1 true e2))
```

```
(defmacro or (e1 e2)
  `(if ,e1 true ,e2))
```

```
(defmacro or (e1 e2)
  `(if ,e1 ,e1  ,e2))
```

```
(defmacro or (e1 e2)
  `(let ([tmp ,e1])
     (if tmp tmp ,e2)))
```

```
(defmacro or (e1 e2)
  `(let ([tmp ,e1])
     (if tmp tmp ,e2)))

(let ([tmp (f y)])
  (or (> tmp x) (zero? tmp)))
```

```
(defmacro or (e1 e2)
  `(let ([tmp ,e1])
     (if tmp tmp ,e2)))

(let ([tmp (f y)])
  (let ([tmp (> tmp x)])
    (if tmp tmp (zero? tmp))))
```

```
(defmacro or (e1 e2)
  (let ([tmp (gensym)])
    `(let ([,tmp ,e1])
       (if ,tmp ,tmp ,e2))))
```

```
(define-syntax or
  (syntax-rules ()
    [(_ e1 e2)
     (let ([tmp e1])
       (if tmp tmp e2))]))
```

```
(define-syntax or
  (syntax-rules ()
    [(_) false]
    [(_ e) e]
    [(_ e es …)
     (let ([tmp e])
       (if tmp tmp (or es …)))]))
```

```
(define-syntax delay
  (syntax-rules ()
    [(_ e)
     (let ([v undefined])
       (lambda ()
         (when (undefined? v)
           (set! v e))
         v))]))
```

```
(define (helper fn)
  (let ([v undefined])
    (lambda ()
      (when (undefined? v)
        (set! v (fn)))
      v)))
```

```
(define (helper fn)
  (let ([v undefined])
    (lambda ()
      (when (undefined? v)
        (set! v (fn)))
      v)))
(define-syntax delay
  (syntax-rules ()
    [(_ e) (helper (λ () e))]))
```

```
(require delay)

(define foo
  (delay (expensive arg …)))
```

```
(require delay)

(define foo
  (helper
    (λ () (expensive arg …))))
```

```
(require delay)

(define foo
  (helper
    (λ () (expensive arg …))))


(helper 3)
```

```
(require delay)

(define foo
  (helper
    (λ () (expensive arg …))))


(helper 3)
```

```
(require delay)

(define foo
  (let ([helper (λ (y) (* y y))])
    (delay (expensive (helper …)))))


(helper 3)
```

```
(require delay)

(define foo
  (let ([helper (λ (y) (* y y))])
    (helper
      (λ () (expensive (helper …))))))

(helper 3)
```

```
(define (helper fn) … )

(define-syntax delay
  (syntax-rules ()
    [(_ e) (helper (λ () e))]))

(provide delay)
```

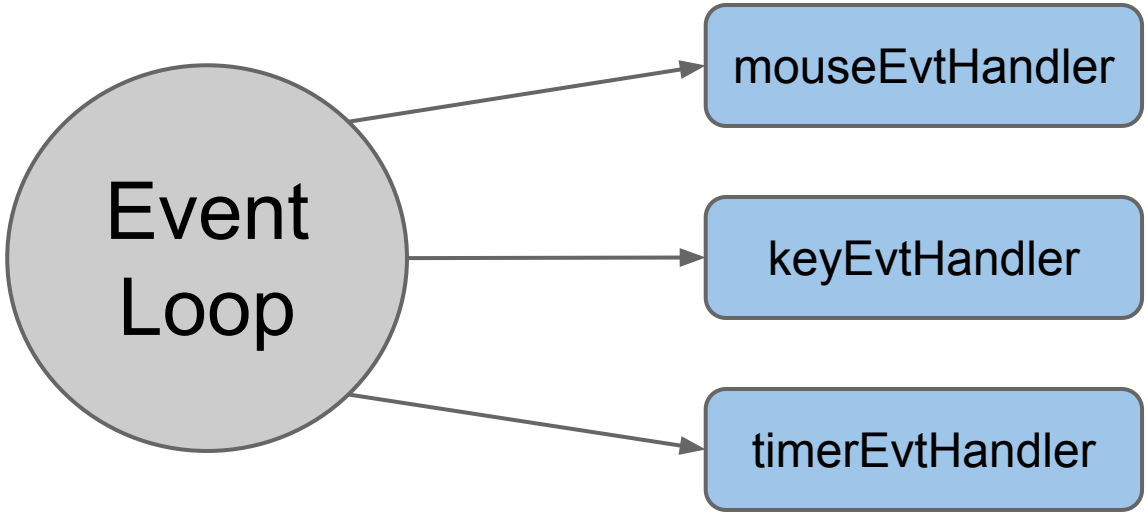
What Can We Do With This?

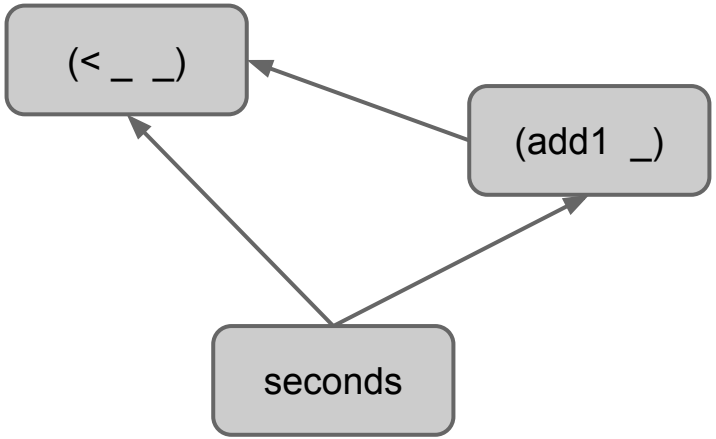
- Define modular language extensions
- Implement programming tools (type checkers, debuggers)
- Define complete “little languages” or language variations

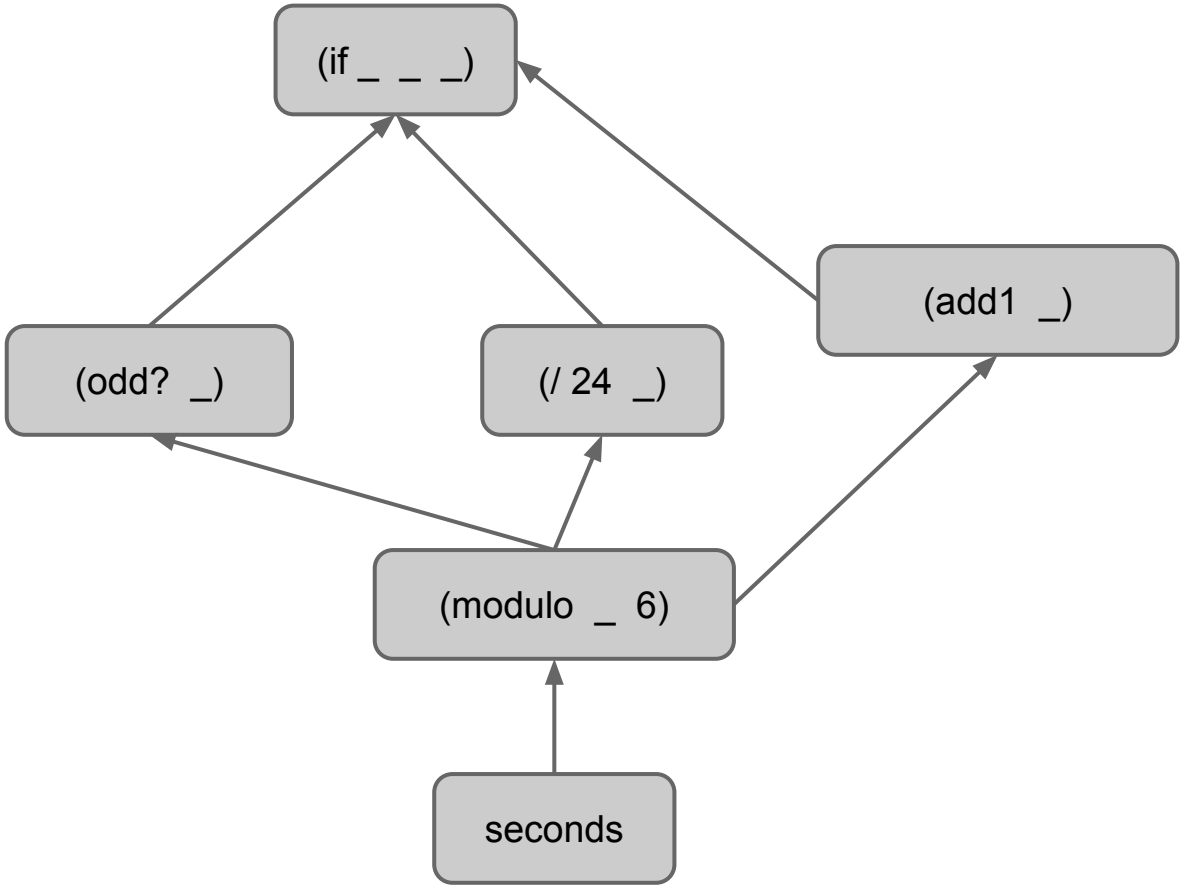
```
f(Request req) {
  val = readFromStore(req.key());
  newState = update(val, req.op());
  result = writeToStore(newState, … );
  return result.isOk();
}
```

```
f(Request req) {
  readFromStore(req.key(), fn(val) {
    newState = update(val, req.op());
    writeToStore(newState, fn (res) {
      response.setStatus(res.isOk());
    }
  });
}
```

```
onMouseMove = function(pos, btnState) {
  …
};

onMouseDown = function() { … };

onKeyEvent = function(key) { … };

onTimer = function() { … };
```


Event
Loop
mouseEvtHandler
keyEvtHandler
timerEvtHandler


(< _ _)
(add1 _)
seconds


(if _ _ _)
(add1 _)
(odd? _)
(/ 24 _)
(modulo _ 6)
seconds

Closing Thoughts

- Structured data + functions can represent anything
- Choice of representation determines what's easy
- Names (and scopes) need careful thought
- Static types are (almost always) your friend
- Hope we've challenged how you think about programming and programming languages