

Scheme

Textbook, Sections 13.1 – 13.3, 13.7

1

Functional Programming

- Based on mathematical functions
- Take argument, return value
- Only function call, no assignment
- Functions are *first-class* values
- E.g., functions can return functions
- Requires *garbage collection*

2

Lambda Calculus

- Mathematical functional language
- Language constructs
 - Variables: x
 - Primitive functions: $*$
 - Function abstraction: $\lambda(x) x*x$ or $\lambda x.x*x$
 - Function call: $f(x)$ or $f x$
- Example
 $(\lambda(x) x*x*x)(2)$ yields 8

3

Scheme

- Dialect of LISP (from MIT, ca. 1975)
- Lists as built-in data type
- Same syntax for lists and programs
- Functions are (first-class) data
- Dynamic, strong typing
- Interaction (*read-eval-print*) loop

4

Scheme Syntax

- **S-expressions**

s-expression ::= literals
 | *symbols*
 | (*s-expression . . .*)

- Special forms for specific language constructs

5

Atomic Data Types

- **Numbers**

3.14, 42, 4101

- **Booleans**

#t, #f

- **Characters, Strings**

#\a, #\newline, "Hello World!"

- **Symbols**

f, x, +, *, foo, bar, null?, set!

6

Variable Definitions

- Variable definitions

```
(define x 3)
(define y 5)
```

- This is not an assignment!

- A new variable is defined
- Like `int x = 3;` in C

7

Functions

- Function definition

```
(define (double x) (+ x x))
```

- Function application (call)

```
(* 3 4)
(+ 2 x y)
(double 3)
(* (+ 3 4) (double 3))
```

8

Anonymous Functions

- Anonymous function definitions

```
(lambda (x) (+ x x))
```

- Variable definitions

```
(define double
  (lambda (x) (+ x x)))
(define * +)
```

- Function application

```
(double 3)
((lambda (x) (+ x x)) 3)
```

9

Lists

- **Quotes**

`(quote x)`, `'x`

- **Empty list**

`'()`, often assigned to the variable `nil`

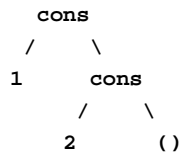
- **Nonempty list**

`'(1 2 3)`, `'(a b 3)`, `'(1 (2 3) 4)`

10

List Representation

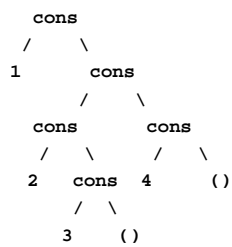
The list `'(1 2)` is stored as



11

List Representation

The list `'(1 (2 3) 4)` is stored as



12

List Functions

- **Predefined functions**

`car, cdr, cons`

- **Examples**

```
(car '(1 2)) returns 1
(cdr '(1 2)) returns (2)
(cons 1 '(2)) returns (1 2)
(cons 1 2) returns (1 . 2)
```

13

S-Expressions

List Syntax	S-Expression Syntax
<code>(1 2 3)</code>	<code>(1 . (2 . (3 . ())))</code>
<code>((1 2))</code>	<code>((1 . (2 . ())) . ())</code>
Not a list	<code>(1 . 2)</code>

14

More Examples

Expression	Value
<code>(- 24 (* 4 3))</code>	<code>12</code>
<code>(car (cdr '(1 2 3)))</code>	<code>2</code>
<code>(cadr '(1 2 3))</code>	<code>2</code>
<code>(cons 1 '())</code>	<code>(1)</code>
<code>(cons 1 2)</code>	<code>(1 . 2)</code>
<code>(cons '() '())</code>	<code>(())</code>
<code>(car '())</code>	<code>()</code>

15

More Examples

Expression	Value
<code>(define x 3)</code>	
<code>(+ x 2)</code>	5
<code>'x</code>	<code>x</code>
<code>(double x)</code>	6
<code>'(double x)</code>	<code>(double x)</code>
<code>(define - double)</code>	
<code>(* (- x) 7)</code>	42

16

More List Functions

Expression	Value
<code>(null? '())</code>	<code>#t</code>
<code>(null? '(a b))</code>	<code>#f</code>
<code>(list 1)</code>	<code>(1)</code>
<code>(list 'a 'b 'c)</code>	<code>(a b c)</code>
<code>(append '(1) '(2))</code>	<code>(1 2)</code>
<code>(map double '(1 2 3))</code>	<code>(2 4 6)</code>

17

Higher-Order Functions

```
(define (twice f x) (f (f x)))  
or  
(define twice  
  (lambda (f x) (f (f x))))  
  
(twice double 2) ; returns 8  
(twice (lambda (x) (* x x)) 2)  
                ; returns 16
```

18

Higher-Order Functions (cont'd)

```
;; Curried version of twice
(define (twice f)
  (lambda (x) (f (f x))))

(define quadruple (twice double))
(quadruple 2)           ; returns 8
((twice double) 2)     ; returns 8
((twice quadruple) 2)  ; returns 32
((twice (twice double)) 2) ; returns 32
```

19

Curried Function in C Syntax

```
(int (*)(int))
twice (int (*)(int)) {
  int foo (int x) {
    return f(f(x));
  }

  return foo;
}

(twice(double))(2);
```

20

Conditionals

• If-then-else

```
(if (= x y) 1 2)
; in C: (x == y) ? 1 : 2
```

• Cond

```
(cond ((= x y) 1)
      (< x y) 2)
      (else 3))
```

• Anything non-#f is considered true

```
(if 1 2 3) ; returns 2
```

21

Equality

Expression	Value
<code>(eq? x x)</code>	<code>#t</code>
<code>(eq? 'x 'x)</code>	<code>#t</code>
<code>(eq? '() '())</code>	<code>#t</code>
<code>(eq? 2 2)</code>	<code>undefined</code>
<code>(eq? (list 1) '(1))</code>	<code>#f</code>
<code>(eqv? 2 2)</code>	<code>#t</code>
<code>(equal? (list 1) '(1))</code>	<code>#t</code>

22

Recursion

```
(define (fac n)
  (if (= n 0)
      1
      (* n (fac (- n 1)))))

(fac 5) ; returns 120
(fac 1000) ; that's 2568 digits
```

23

Map Function

```
(define (mymap f l)
  (if (null? l) '()
      (cons (f (car l))
            (mymap f (cdr l)))))

(mymap double '(1 2 3 4 5))
; returns (2 4 6 8 10)
(mymap (lambda (x) (+ x 1)) '(1 2))
; returns (2 3)
```

24

Let

- Defines local name binding

```
(let ((x 3)
      (y 5))
  (* x y))
```

- Syntactic sugar for function call

```
((lambda (x y) (* x y)) 3 5)
```

25

Quicksort

Problem: sort list of numbers

Algorithm:

- If list is empty, we are done
- Choose pivot n (e.g., first element)
- Partition list into lists A and B with elements $< n$ in A and elements $> n$ in B
- Recursively sort A and B
- Append sorted lists and n

26

Quicksort (cont'd)

```
(define (quicksort ls)
  (if (null? ls) '()
      (let ((p (car ls)))
        (append
         (quicksort (gt p (cdr ls)))
         (list p)
         (quicksort (le p (cdr ls)))))))
```

27

Partitioning

```
(define (gt p ls)
  (filter (lambda (x) (> p x)) ls))

(define (le p ls)
  (filter (lambda (x) (<= p x)) ls))
```

28

Filtering a List

```
(define (filter pred ls)
  (cond ((null? ls) '())
        ((pred (car ls))
         (cons (car ls)
               (filter pred (cdr ls))))
        (else (filter pred (cdr ls)))))
```

29

Assignment (but don't use it)

Expression	Value of x
(define x 3)	3
(set! x 5)	5
(define x (list 1 2))	(1 2)
(set-car! x 3)	(3 2)
(set-cdr! x 4)	(3 . 4)

30
