

---

## Functional Programming— Illustrated in Scheme

---

### References:

- Dybvig,
- Sebesta 5th ed., chapter 15,
- or 4th ed., chapter 14.

Original Lisp slides © Diane Horton 2000.  
Adapted for Scheme © Eric Joanis 2000, 2002.  
Additional slides use material taken from Scheme  
notes © Gary Baumgartner 2001.

1

---

## Jumping right in

---

### A Scheme procedure

```
(define increment  
  (lambda (n)  
    (+ n 1)  
  )  
)
```

### A call to the procedure

```
(increment 21)
```

2

---

## The Spirit of Lisp-like Languages

---

We shall first define a class of **symbolic expressions** in terms of ordered pairs and lists. Then we shall define five elementary **functions and predicates**, and build from them by **composition, conditional expressions and recursive definitions** an extensive class of functions of which we shall give a number of examples. We shall then show how these **functions can themselves be expressed as symbolic expressions**, and we shall give a **universal function *apply*** that allows us to compute from the expressions for a given function its value for given arguments. Finally, we shall define some **functions with functions as arguments** and give some useful examples.

McCarthy, J, [1960]. Recursive functions of symbolic expressions and their computation by machine, Part I. *Comm. ACM* 3:4; quoted in Sethi.

3

---

## Pure Functional Programming

---

- A program consists of function definitions and calls. There is no other structure.
- No loops. Just recursion.
- No side effects – a function may not change its parameters, or do input or output.
- No variables. Just parameters.
- Implicit memory mgmt: no new or free.
- One data structure: the list. Can represent anything, including code.
- Functions can be:
  - passed as an argument;
  - returned from a function;
  - represented by a data structure;
  - and such a data structure can then be evaluated.

4

## The MIT Scheme Interface

```
werewolf 1% scheme
Scheme Microcode Version ...
```

```
1 ]=> (+ 8 3 5 16 9)
;Value: 41

1 ]=> (define increment (lambda (n) (+ n 1)))
;Value: increment

1 ]=> (increment 21)
;Value: 22

1 ]=> (load "incr")
;Loading "incr.scm" -- done
;Value: increment-list

1 ]=> (increment-list (1 32 7))
;The object 1 is not applicable.
;To continue, call RESTART with an option number:
;(RESTART 2) => Specify a procedure to use in its place.
;(RESTART 1) => Return to read-eval-print level 1.

2 error> (restart 1)
;Abort!

1 ]=> (increment-list '(1 32 7))
;Value 1: (2 33 8)
```

5

---

## Procedure Application

---

The main form of a Scheme expression is the procedure application. (Terminology: in Scheme, the official name for what you would think of as a function is *procedure*.)

*(procedure arg1 arg2 ... argn)*

### Evaluation

- Each argument is evaluated.
- The procedure is applied to the results.

Exception: **syntactic forms**.

Syntactic forms violate the rule—they are built in to the language to handle cases the rule above can't handle. Examples: **define**, **if**, **cond**, **lambda**—more on this later.

7

```
1 ]=> (trace increment-list)
;Unspecified return value

1 ]=> (increment-list '(1 32 7))

[Entering #[compound-procedure 2 increment-list]
  Args: (1 32 7)]
[Entering #[compound-procedure 2 increment-list]
  Args: (32 7)]
[Entering #[compound-procedure 2 increment-list]
  Args: (7)]
[Entering #[compound-procedure 2 increment-list]
  Args: ()]
[()]
  <== #[compound-procedure 2 increment-list]
  Args: ()]
[(8)
  <== #[compound-procedure 2 increment-list]
  Args: (7)]
[(33 8)
  <== #[compound-procedure 2 increment-list]
  Args: (32 7)]
[(2 33 8)
  <== #[compound-procedure 2 increment-list]
  Args: (1 32 7)]
;Value 3: (2 33 8)

1 ]=> (exit)

Kill Scheme (y or n)? Yes
Happy Happy Joy Joy.
werewolf 2%
```

6

### Examples

- $(- 1) \Rightarrow -1$
- $(* 5 7) \Rightarrow 35$
- $(+ 1 2 (* 2 3)) \Rightarrow 9$
- $(+ (- 6 3) (/ 10 2) 2 (* 2 3)) \Rightarrow 16$
- $(\cos 0) \Rightarrow -1$

Exercice: run Scheme and try the arithmetic operators with 0, 1, 2 and 3 arguments, and figure out how the results make sense.

8

---

## Variables

---

To bind a name to a value:

```
(define var value)
```

```
(define a 2)
```

```
=> a
```

```
a
```

```
=> 2
```

```
(+ a 2)
```

```
=> 4
```

```
(define b 3)
```

```
=> b
```

```
(define c (+ a (* 4 b)))
```

```
=> c ; LISP: Lots of Silly Parentheses
```

```
c
```

```
=> 14
```

Could **define** be a procedure?

9

## Naming a procedure

Like any other value, a procedure can be bound to a name using **define**.

### Example

```
(define increment
```

```
  (lambda (n) (+ n 1))
```

```
)
```

```
=> increment
```

```
(increment c)
```

```
=> 15
```

### Example

```
(c 15)
```

```
=> The object 14 is not applicable.
```

What does that mean?

11

---

## Procedures

---

The **lambda** syntactic form is used to create procedures:

```
(lambda (a1 a2 ... an) body)
```

### Example

```
(lambda (n) (+ n 1))
```

Could **lambda** be a procedure?

### Applying the procedure

```
((lambda (n) (+ n 1)) 21)
```

In procedure application, not only the arguments are evaluated first, but the procedure itself as well!

10

---

## Conditionals

---

Booleans: **#t**, **#f**.

(**#f** is the same as **()** in some implementations, including MIT Scheme.)

```
(if test consequent [alternative])
```

### Example

```
(if (negative? c) (- c) c)
```

```
=> 14
```

*alternative* is optional. Why would you leave it out? What value would be returned if *test* was false?

Could **if** be a procedure?

12

## Example

```
(define abs
  (lambda (x)
    ((if (positive? x) + -) x)
  )
)
=> abs
(abs -1)
=> 1
```

How does that work?

13

---

## More Conditionals

---

**cond** is like a switch or case statement

```
(cond (test1 exp11 exp12 ...)
      (test2 exp21 exp22 ...)
      ...
      (else expe1 expe2 ...))
)
```

Lazy **or** evaluation

```
(or exp1 exp2 exp3 ...)
```

Lazy **and** evaluation

```
(and exp1 exp2 exp3 ...)
```

Exercise: run Scheme and try these three syntactic forms with 0 arguments. Does the result make sense?

Could any of these be procedures?

14

---

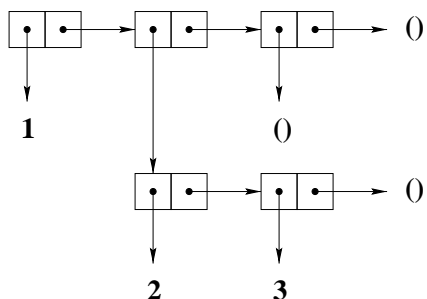
## Lists

---

A simple but powerful general-purpose datatype. (How many datatypes have we seen so far?)

```
(1 #t 1)
()
(1 (2 3) ())
```

Building block: the cons cell.



Note: Sebasta uses NIL. That is LISP notation! In Scheme, we use ().

15

## Creating lists

- Quote: `'(1 (2 3) ()) => (1 (2 3) ())`  
or `(quote (1 (2 3) ())) => (1 (2 3) ())`
- Apply *list*: `(list 1 '(2 3) ()) => (1 (2 3) ())`
- Build it, piece by piece:  
`(cons 1 (cons (cons 2 (cons 3 ())) (cons () ())))`

## More list manipulation

```
(define lst '(1 (2 3) ()))
```

- First element: `(car lst) => 1`
- Rest of the list: `(cdr lst) => ((2 3) ())`
- *car* of *cdr* of ...: `(cadadr lst) => 3`
- Appending lists:  
`(append lst '(4 5)) => ((1 (2 3) ()) 4 5)`

What preconditions do all these procedures have?

16

---

## Useful predicates

---

### Testing for equality

- (*eq?* *a b*): Returns *#t* iff *a* and *b* are the same Scheme object. (Don't use *eq?* with numbers!)
- (*=* *a b*): Returns *#t* iff *a* and *b* are numerically equal. Pre: *a* and *b* must evaluate to numbers.
- (*eqv?* *a b*): Similar to *eq?*, but works for numbers and characters. More expensive than *eq?*, however.
- (*equal?* *a b*): Returns *#t* iff *a* and *b* have the same structure and contents. Thus, *equal?* recursively tests for equality. The most expensive equality predicate.

Reference: Dybvig §6.1.

17

### More pre-defined predicates

- (*null?* *a*): Returns *#t* iff *a* is the empty list (or *#f*, depending on the implementation).
- (*pair?* *a*): Returns *#t* iff *a* is a pair, i.e., a cons cell.
- (*number?* *a*): Returns *#t* iff *a* is a number.

Lots more in Dybvig §6.

### Code as Data—Eval

Scheme code is simply data that is treated as code. If you build an expression, using any data processing technique, and you want to evaluate it as code, use *eval*:

```
(define a (+ 4 6))  
a => 10  
(define b '(+ 4 6))  
b => (+ 4 6)  
(eval b ()) => 10
```

More on this later...

18

---

## Some Interesting Examples

---

Write procedures to:

1. Return the sum of the first *n* integers.
2. Increment every integer in a list.
3. Reverse the elements of a list.
4. Build and return a list of the first *n* odd numbers.
5. Return the last element of a list
6. Take a list of lists, and return a new list containing that last element of each.
7. Count the number of atoms, at all levels, in a list.
8. Return a list of all the atoms, at all levels, in a list.

19