Functional Programming—
Illustrated in Scheme

References:
- Dybvig,
- Sebesta 5th ed., chapter 15,

Scheme slides © Suzanne Stevenson, Diana Inkpen 2001.
Modified and updated Sheila McIlraith 2004.
Additional slides use material taken from Scheme

Jumping right in

A Scheme procedure

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

or

(define (increment n)
  (+ n 1))

A call to the procedure

(increment 21)

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.


Pure Functional Languages

Fundamental concept: application of (mathematical) functions to values

1. Referential transparency: The value of a function application is independent of the context in which it occurs
   • value of f(a,b,c) depends only on the values of f, a, b and c
   • It does not depend on the global state of computation
     ⇒ all vars in function must be parameters
Pure Functional Languages (cont.)

2. The concept of assignment is **not** part of functional programming
   - no explicit assignment statements
   - variables bound to values only through the association of actual parameters to formal parameters in function calls
   - function calls have no side effects
   - thus no need to consider global state

3. Control flow is governed by function calls and conditional expressions
   ⇒ no iteration
   ⇒ recursion is widely used

---

A Functional Program

A program includes:

1. A set of function definitions
2. An expression to be evaluated

E.g. in Scheme:

```scheme
1 ]=> (define (abs-val x)
    (if (>= x 0)
      x
      (- x)))

;Value: abs-val

1 ]=> (abs-val (- 3 5))

;Value: 2
```

---

Pure Functional Languages (cont.)

4. All storage management is implicit
   - needs garbage collection

5. Functions are *First Class Values*
   - Can be returned as the value of an expression
   - Can be passed as an argument
   - Can be put in a data structure as a value
   - Unnamed functions exist as values

---

LISP

- Functional language developed by John McCarthy in the mid 50’s
- Semantics based on Lambda Calculus
- All functions operate on lists or atomic symbols: (called “S-expressions”)
- Only five basic functions: list functions cons, car, cdr, equal, atom and one conditional construct: cond
- Useful for list-processing applications
- Programs and data have the same syntactic form: S-expressions
- Used in Artificial Intelligence
Commonalities between LISP and SCHEME

- Expressions are written in prefix, parenthesized form
  - (function arg1 arg2 ...argn)
  - (+ 4 5)
  - (+ (* 3 4 5) (- 5 3))
- In order to evaluate an expression:
  1. evaluate function to a function value
  2. evaluate each argi in order to obtain its value
  3. apply the function value to these values

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)
1] => (trace increment-list)
; Unspecified return value
1] => (increment-list '(1 32 7))

)(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.

Examples

• (- 1) => -1
• (* 5 7) => 35
• (+ 1 2 (* 2 3)) => 9
• (+ (- 6 3) (/ 10 2) 2 (* 2 3)) => 16
• (cos 0) => -1

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

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
c => 14

Could define be a procedure?
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!

Naming a procedure

Like any other value, a procedure can be bound to a name using define. Two syntaxes:

1. (define <fcn-name> <fcn-value>)

   E.g.,

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

   (define square
     (lambda (x) (* x x)))

2. (define (<fcn-name> <fcn-params>)
   <expression>)

   E.g.,

   (define (increment n)
     (+ n 1))

   (define (square x)
     (* x x))

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?
More Conditionals

; cond is like a switch or case statement
(cond (test1 exp11 exp12 ...)
  (test2 exp21 exp22 ...) ... 
  (else exp1 exp2 ...) )

; 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?

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?

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

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.

More predefined 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...

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.