Functional Programming—Illustrated in Scheme

References:
- Dybvik, (available online and in the library)
- Sethi, chapter 10,
- Mitchell Chapter 3, 4.2.
- Sebesta 6th ed., chapter 15,

Modified and updated © S. McIlraith 2004.
Modified and updated © Iqbal Mohamed 2005.
Additional slides use material taken from © G. Baumgartner 2001.

Jumping right in

A function that searches a list

1 ]⇒ (define find (lambda (x y)
  (cond ((equal? y '()) '())
         ((equal? x (car y)) x)
         (else (find x (cdr y)))))

;Value: find
1 ]⇒ (find 'apple '(pear peach apple fig banana))
;Value: apple
1 ]⇒ (find 'apple '(pizza fries pepsi))
;Value: ()

Jumping right in

A Scheme procedure

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

;Value: increment
1 ]⇒ (* 1 2 3 4 5 6 7 8 9 10)
;Value: 3628800
1 ]⇒ (+ (* 4 0) (+ 2 2 2 2))
;Value: 20
1 ]⇒ (exit)

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 (i.e., given the same parameters, it always returns the same results). Or alternatively, a language is referentially transparent if we may replace one expression with another of equal value anywhere in a program without changing the meaning of the program. This is achieved by not having side effects in programs, e.g.,
- 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

The main advantage of referential transparency is that it makes it much easier to reason about programs, and to apply program transformations.

See Mitchell, page 78 for further discussion.

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 (abc-val x)
    (if (> x 0)
        x
        (- x)))
;Value: abc-val
1 ]=> (abc-val -3 5))
;Value: 2
```

Jumping Back In

Contents of file incr.scm

```scheme
(define increment-list
  (lambda (x)
    (cond ((null? x) '())
          ((number? x) (+ x 1))
          (else (cons (increment-list (car x))
                    (increment-list (cdr x)))))
```
Scheme: A Functional Programming Language

1958: Lisp
1975: Schémé (revised over the years)
1980: Common Lisp ("CL")
1980s: Lisp Machines (e.g., Symbolics, TI Explorer, etc.)

Lisp, Scheme and C-like languages contrasted on following pages.

Some features of Scheme:
• denotational semantics based on the λ-calculus.
  i.e., the meaning of programming constructs in the language is defined in terms of mathematical functions.
• lexical scoping
  i.e., all free variables in a λ-expression are assigned values at the time that the λ-defs are defined (i.e., evaluated and returned).
• arbitrary control structures w/ continuations.
• functions as first-class values
• automatic garbage collection.

LISP
• Functional language developed by John McCarthy in 1958.
• Semantics based on λ-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
• Uses dynamic scoping
• Useful for list-processing applications
• Programs and data have the same syntactic form: S-expressions
• Used in Artificial Intelligence

Expressions

Common structure for both procedures and data. In Scheme, functions are called procedures.

When an expression is evaluated it creates a value or list
of values that can be embedded into other expressions. Therefore programs can be written to manipulate other programs.

<expression> ::= <variable> 
  | <integer>
  | <procedure call>
  | <lambda expression>
  | <conditional>
  | <assignment>
  | <derived expression>

SCHEME

• Developed in 1975 by G. Sussman and G. Steele
• A version of LISP
• Consistent syntax, small language
• Closer to initial semantics of LISP
• Provides basic list processing tools
• Allows functions to be first class objects
• Provides support for lazy evaluation
• Lexical scoping of variables

COMMON LISP (CL)

• Implementations of LISP did not completely adhere to semantics
• Semantics redefined to match implementations
• COMMON LISP has become the standard
• Committee-designed language (1980s) to unify LISP variants
• Many defined functions
• Simple syntax, large language
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.

Examples

• (+ 1) -> 1
• (+ 5 7) -> 35
• (+ 1 2 (* 2 3)) -> 9
• (+ (* -6 3) (/ 10 2) 2 (* 2 3)) -> 16
• (cons 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 ; LISP: Lots of Silly Parentheses
c -> 14

Could define be a procedure?

Built-In Procedures

• eq?: identity on atoms
• null?: is list empty?
• car: selects first element of list
• cdr: selects rest of list
• cons element list): constructs lists by adding element to front of list
• quote or ’: produces constants

Built-In Procedures

• ’() is the empty list
• (car ’(a b c)) =
• (car ’((a) b (c d))) =
• (cdr ’(a b c)) =
• (cdr ’((a) b (c d))) =

• car and cdr can break up any list:
  → (car (cdr (cdr ’((a) b (c d))))) =
  → (caddr ’((a) b (c d)))

• cons can construct any list:
  → (cons ’a ’()) =
  → (cons ’d ’(a)) =
  → (cons ’(a b) ’(c d)) =
  → (cons ’(a b c) ’((a) b)) =
Lists

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

\[
\begin{align*}
1 & \rightarrow 1 \\
2 & \rightarrow 1 \\
3 & \rightarrow 1
\end{align*}
\]

Building block: the cons cell.

Note: Sometimes you'll see NIL. This is Lisp notation! In Scheme, we use \(\) .

More about lists

Proper lists:

\[
\begin{align*}
\emptyset, (a \ (b \ c \ d \ e)) \\
(\text{cons} \ 'a' \ '(b)) & \rightarrow (a \ b)
\end{align*}
\]

Dotted pairs (improper lists):

\[
\begin{align*}
(\text{cons} \ 'a' \ 'b') & \rightarrow (a \ b) \\
(\text{car} \ '(a \ . \ b)) & \rightarrow a \\
(\text{cdr} \ '(a \ . \ b)) & \rightarrow b \\
(\text{cons} \ a \ '(b \ . \ c)) & \rightarrow (a \ b \ c) \\
(a \ b \ c) & \rightarrow (a \ . \ (b \ . \ (c \ . \ ()())
\end{align*}
\]

Things you should know about cons, pairs and lists

The pair or cons cell is the most fundamental of Scheme's structured object types.

A list is a sequence of pairs; each pair's cdr is the next pair in the sequence.

The cdr of the last pair in a proper list is the empty list. Otherwise the sequence of pairs forms an improper list, i.e., an empty list is a proper list, and any pair whose cdr is a proper list is a proper list.

An improper list is printed in dotted-pair notation with a period (dot) preceding the final element of the list. A pair whose cdr is not a list is often called a dotted pair.

cons vs. list: The procedure cons actually builds pairs, and there is no reason that the cdr of a pair must be a list, as illustrated on the previous page.

The procedure list is similar to cons, except that it takes an arbitrary number of arguments and always builds a proper list.

E.g., \(\text{list} \ 'a' \ 'b' \ 'c') \rightarrow (a \ b \ c)

Other (Predicate) Procedures

Predicate procedures return \#t or \#f (i.e., true).

- \(+ \ - \ * /\) numeric operators, e.g.,
  \(\star 5 \ 3) = 15, \ (/ 5 \ 3) = 1.666666\)

- \(= \ < \ > \ <= \ >=\) number comparison ops

- Run-time type checking procedures:
  - All return Boolean values: \#t and \#f
    - (number? \ 5) is \#t
    - (zero? \ 0) is \#f
    - (symbol? \ 'aam) is \#t
    - (list? \ '(a b)) is \#t
    - (null? \ '()) is \#t

Other Predicate Procedures

- (number? \ 'aam) evaluates to \#f
- (null? \ '(a)) evaluates to \#f
- (zero? \ '(- 3 3)) evaluates to \#t
- (zero? \ '(- 3 3)) \Rightarrow \text{type error}
- (list? \ '(* 3 4)) evaluates to \#f
- (list? \ '(* 3 4)) evaluates to \#t