CHAPTER 10 : FUNCTIONAL LANGUAGES

CMPSC 460 – Principles of Programming Languages

Functional Programming Languages

- Pure Functional languages:
  - Pure (original) Lisp
  - Miranda
  - Haskell
  - Single Assignment C
- Typed Functional languages:
  - ML
  - OCaml
  - Haskell
  - F#

Functional Programming

- Can Programming be liberated from the von Neumann Style?

  John Backus

  - Purely functional languages are better than imperative language
    - More readable
    - More reliable
    - More likely to be correct

Functional Programming

Lisp is the medium of choice for people who enjoy free style and flexibility.

Gerald J. Sussman

A Lisp Programmer knows the value of everything, but the cost of nothing.

Alan Perlis

Scheme

- Expressions:
  - Atomic
  - Compound

Data Types

- Atom
  - Number
  - Boolean
  - String
  - Character
  - Symbol
- List

Data Types

- numbers:
  - `+, -, *, /`
  - `zero?`
  - `number?`
  - `=, <, >=, <=`
- symbols:
  - `quote`
  - `symbol?`
  - `eq?`

Scheme - Primitive Functions

- **Examples:**
  - `(+ 3 4)`
  - `(quote (+ 3 4))`
  - `(+ 3 4 5 7)`
  - `(/ 40 4 5)`
  - `(/ 3)`
  - `(- 1.0 0.9)`
  - `(- 1000.0 999.9)`
  - `((+ 1 2))`

Scheme – Prefix notation

- how do we write :

1. $8 + 9 \times 4$
2. $(9000 + 900 + 90 + 9) - (5000 + 500 + 50 + 5)$

Scheme – Predicate Functions

- Boolean values
  - #t
  - #f

- boolean?, even?, odd?, zero?, negative?,
- or, and, not

Scheme – Predicate Functions

Examples:

- `(not (even? 6))`
- `(negative? -8)`
- `(< 9 87 100)`
- `(= (/ 1 2) (/ 8 16))`
- `(boolean? 'y)`

Scheme - Lists

- *List* is a recursive structure:
  - Empty list
  - A pair

- *List form*: parenthesized collections of sublists and/or atoms

  e.g., `‘(A B (C D) E)`

Scheme - Lists

- `LIST` takes any number of parameters; returns a list with the parameters as elements
- `CONS` takes two parameters
  1. either an atom or a list
  2. a list

  returns a new list that includes the first parameter as its first element and the second parameter as the remainder of its result

Scheme - Lists

- Example:
  1. `(cons 'A '(B C))`
  2. `(list 'A '(B C) 'd)`
  3. `(cons '(a b (c)) '())`
  4. `(cons 'A 'B)`
  5. `(list 'A 'B)`
  6. `(list 'A '(B))`

Scheme - Lists

- `APPEND` takes two or more lists as parameters

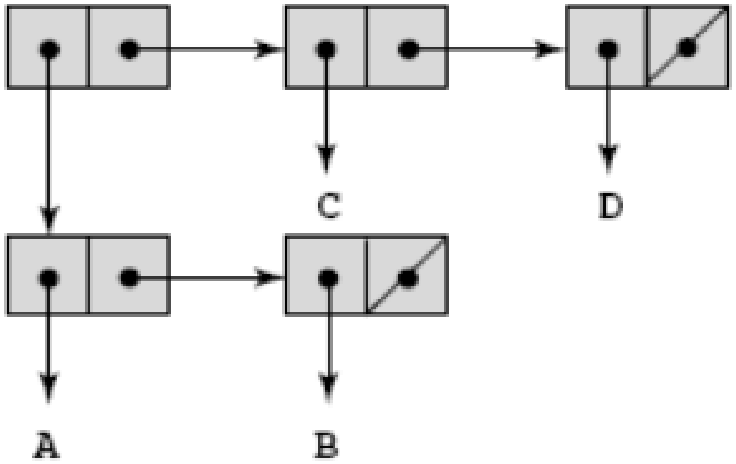
  returns a new list that contains all of the elements of the parameter lists in the specified order

Scheme - Lists

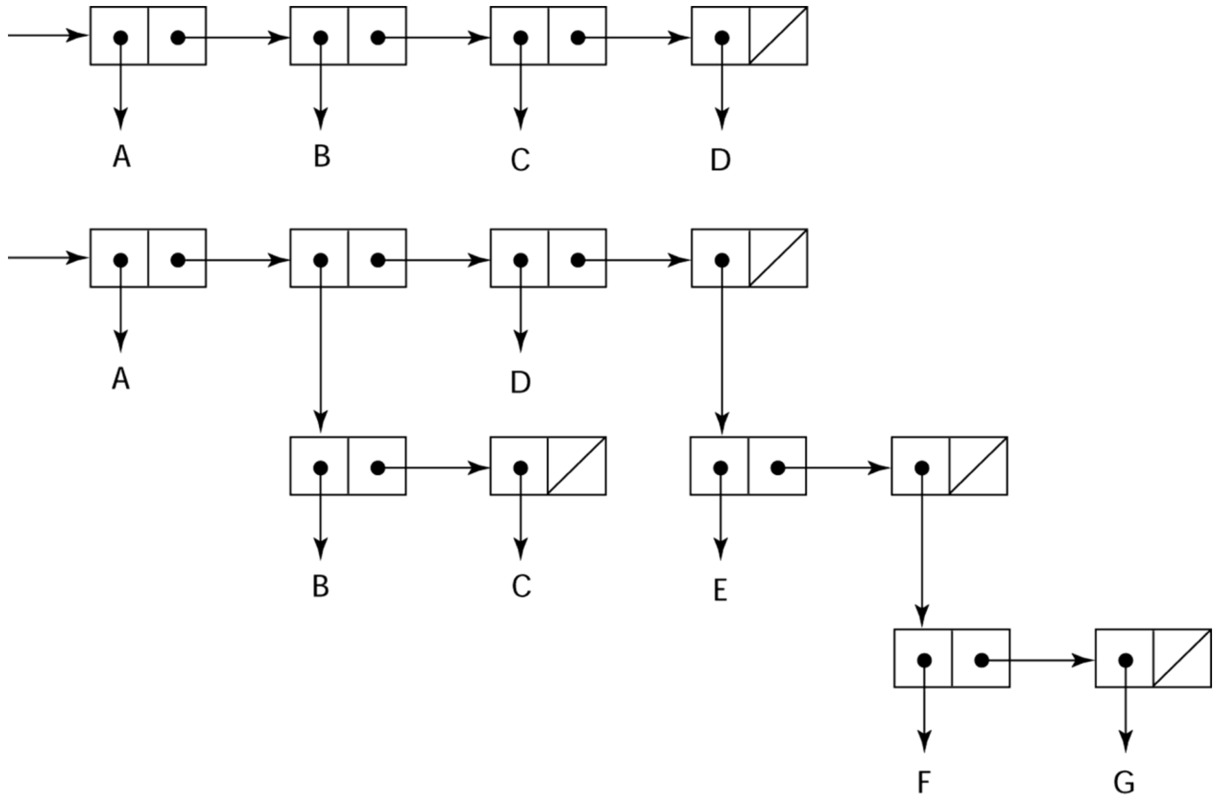
- Example:
  1. `(append '(A) '(B C))`
  2. `(append '(A B) '(B C) '(D))`
  3. `(append '(a b (c)) '())`
  4. `(append '((a b) (c d))`
     `'((e f) (g h)))`
  5. `(append '(A) 'B)`
  6. `(append 'A '(B))`

Scheme - Lists

```
(CONS '(A B) '(C D))
```



Scheme - Lists



Scheme - Lists

- `car` takes a *pair*; returns the first element of that list

  Examples:

  1. `(car '(A B C))`
  2. `(car (car '((A B) C D)))`
  3. `(car 'A)`
  4. `(car '(A))`
  5. `(car (car '(A)))`
  6. `(car '())`

Scheme - Lists

- `cdr` takes a pair parameter; returns the list after removing its first element

  Examples:

  1. `(cdr '(A B C))`
  2. `(cdr '((A B) C D))`
  3. `(cdr 'A)`
  4. `(cdr '(A))`
  5. `(cdr '())`

Predicate functions for Lists

- `list?`
- `null?`

Examples:

1. `(list? 'A)`
2. `(list? '(A N))`
3. `(list? '())`
4. `(list? (+ 9 8))`
5. `(null? 'A)`
6. `(null? '())`

pair? and equal?

1. `(pair? 1)`
2. `(pair? (+ 9 8))`
3. `(pair? (list 1 2))`
4. `(pair? '(1 2))`
5. `(pair? '())`

1. `(eq? '(A N) '(A N))`
2. `(equal? '(A N) '(A N))`

Scheme – Define

- To bind a identifier to an expression

Examples:

```
(define pi 3.141593)
(define two_pi (* 2 pi))
```

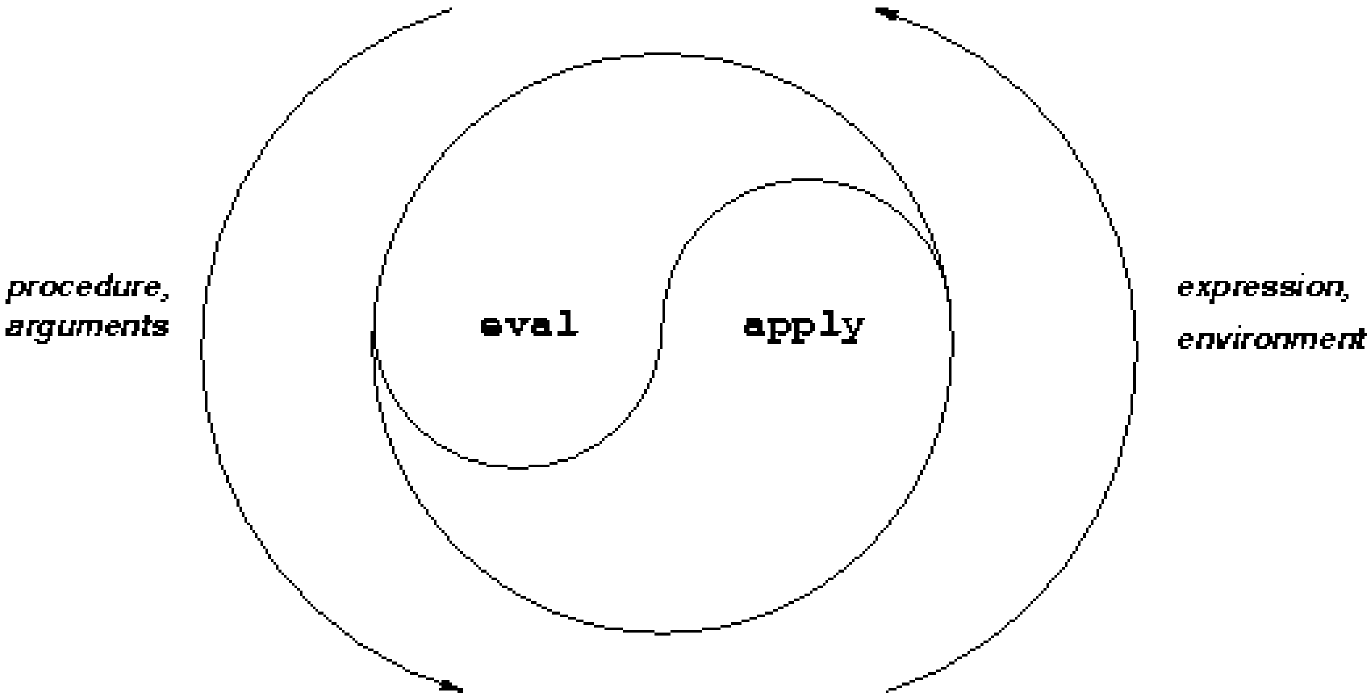
Example

```
(define lst (cons - (cons + '())))

((car lst) 2 4)

((cadr lst) 2 4)
```

Scheme - Simplicity

- Homogeneity:
  - Program and data have the same representation
  - Parentheses are NOT just grouping, as they are in Algol-family languages
  - `quote` - takes one parameter; returns the parameter without evaluation

Meta-circular Evaluator



H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs*, Fig. 4.1, p. 364

Functions

- The function constructor builds **anonymous functions**
- A function is a list containing three things

```
(lambda  (<parameters>) <body>)
```

Scheme – Define

- To bind names to lambda expressions
  - `(define square (lambda (x) (* x x)))`

Example use: `(square 5)`

Functions - Examples

1.
```
(define sum1
    (lambda (x y)
      (+ x y)))
```

2.
```
(define sum2
    (lambda() (lambda (x y)
                (+ x y))))
```

Scheme – Control Flow

- Multiple Selection - the special form, `COND`

  General form:

  (`cond`

  (*predicate_1 expr* { *expr* })

  (*predicate_1 expr* { *expr* })

  . . .

  (*predicate_1 expr* { *expr* })

  (`else` *expr* { *expr* }))

Scheme – Control Flow

Example:

```
(define(compare x y)
(cond
((> x y) "x is greater than y")
  ((< x y) "y is greater than x")
  (else "x and y are equal")
))
```

Scheme – Control Flow

```
(cond
   ((< 2 1) 2)
   ((< 1 2) 1))
```

```
(cond
   ((< 2 1)2)
   ((< 3 2) 3))
```

Scheme – Control Flow

- Selection- the special form, `IF`

```
(if predicate then_exp else_exp)
```

Example:

```
(if (= count 0)
    0 (/ sum count))
```

Scheme – Control Flow

Example:

```
(( (if (null? '(a)) cdr   car)
       (cons cdr (cons car '())))) '(9 8 7))
```

Local Binding

```
(let
    ((var1 exp1)... (varn expn))
  body)
```

Local Binding

```
(let ((x 2) (y 3))
  (* x y))
```

Local Binding

```
(let ((x 2) (y 3))
  (let ((x 7)
        (z (+ x y)))
    (* z x)))
```

```
(let ((a 2) (b (* a a)))
  (* a b))
```

Example 1 – Recursive Functions

Write a function that returns the length of a list *l*

Example 2 – Recursive Functions

Write a function that returns the sum of a list of numbers *l*

Example 3 – Recursive Functions

- Write a function that takes a list of numbers as input. The function should then return the list where each element is double its original value.

Example 3 – Recursive Functions

```
(define(double lst)
(cond
((null? lst) '())
(else (cons (* (car lst) 2) (double
   (cdr lst))))
))
```

Example 4 – Recursive Functions

- Write a function called *member* that takes two args: a symbol *s* and a list of symbols *ls*. The function returns the index of the first element (0-based), if such an element exists. Otherwise, the function returns #f.
  - `(member 'x '(a b c))       => #f`
  - `(member 'a '(a b c))       => 0`
  - `(member 'b '(w x a b c)) => 3`

Scheme vs Lisp

Lisp	Scheme	Lisp	Scheme
defun	define	rplacaset	car!
defvar	define	rplacdset	cdr!
car, cdr	car, cdr	mapcar	map
cons	cons	t	#t
null	null?	nil	#f
atom	atom?	nil	nil
eq, equal	eq?, equal?	nil	'()
Setq	set!	progn	begin
cond . . . t	cond . . . else		

Functions That Build Code

- DRRacket → *Pretty Big language*

```
(define adder
  (lambda (lst)
    (cond
      ((null? lst) 0)
      (else (eval (cons '+ lst))))
    ))
```

Variable Arity Procedures

```
(define fun
  (lambda x x))
```

Apply-to-All Functions

```
(map (lambda (num)
            (* num num num)) '(2 3 4 5))
```

```
(map * '(2 3 4 5) '(1 2 3 4))
```

Variable Arity Procedures

```
(define plus (lambda x
    (cond
        ((null? x) "Undefined")
        ((andmap number? x) (apply + x))
        (else #f))))
```

First Class Functions

When a function can be

1. passed to another function
2. returned from a function
3. stored in a data structure

First Class Functions

```
(define rem  (lambda (test? a l)
    (cond
      ((null? l) '())
      ((test? (car l) a) (cdr l))
      (else
        (cons (car l) (rem test? a (cdr l))))))))
```

Functional Programming in Perspective

- Advantages of Scheme
  - Simple semantics
  - Simple syntax
  - Lack of side effects makes programs easier to understand
    - Programs can automatically be made concurrent
  - Programs are short
    - 10 to 25% as large

Functional Programming in Perspective

- Disadvantages
  - Was not designed to benefit from the von Neumann architecture
  - Requires a different mode of thinking by the programmer
  - Execution time:
    - lots of copying of data through parameters
    - heavy use of pointers
    - frequent recursive calls
    - requires garbage collection

References

- Michael L. Scott, Programming Language Pragmatics, Morgan Kaufmann, 3rd edition, 2009.
- Robert W. Sebesta, Concepts of Programming Languages, Addison Wesley, 10th edition, 2012.
- John C. Mitchell, Concepts in Programming Languages, Cambridge University Press, 2002.