Principles of Programming Languages
Lecture 9B

Wael Aboulsaadat

wael@cs.toronto.edu

http://portal.utoronto.ca/

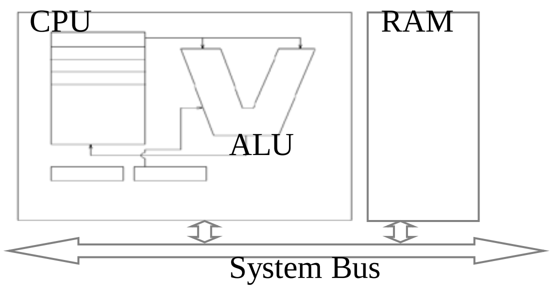
Acknowledgment: parts of these slides are based on material by Diane Horton & Eric Joanis @ UoT

References: Scheme by Dybvig; PL Concepts and Constructs by Sethi; Concepts of PL by Sebesta; ML for the Working Prog. By Paulson; Prog. in Prolog by Clocksin and Mellish; PL Pragmatics by Scott

Functional Programming Languages (FPL)

- **"Can programming be librated from the Von Neumann style?"**
  John Backus

- **Problems with Imperative programming languages:**
  - Von Neumann bottle neck (i.e. fetching words across bus)
    - Assignment
  - Side-effects
  - State-based transformation



- **FPL alternative:**
  - Goal? mimic mathematical functions to the greatest extent possible
  - How? Use calculus for the computation
  - The program is a mathematical function

- **FPL-based solutions:**
  - Hardware: Symbolics Machine, TI Explorer…
  - Software: Lisp (Scheme && Racket) , ML, Haskell, Miranda …

FPL: Mathematical v.s. Imperative

- **Recall: how do imperative functions work?**
  - Specify a sequence of operations on values in memory to produce a value
  - Evaluation is controlled by sequencing and iteration

- **Why are mathematical functions different?**
  - The value is defined and not produced
  - Evaluation order is controlled by recursion and conditional expressions

- **Example:**
  - Write a procedure to implement the following function *f*(x) = x * x / 3
    - *Imperative:*

```
procedure float foo( var int x )
        int product;
        float quotient ;
        product  := x * x;
        quotient := product/ 3;
        return quotient;
```

    - *Functional:* ??

FPL: Desiderata

1. A program consists of:

- Function definitions
- Function calls
- There is no other structure.

2. Control flow:

- Recursion and function application is the only way to achieve repetition

3. No assignment

- Values are bound to values only through parameter association

FPL: Desiderata

4. No side effects

- A function may not change its parameters
- A function cannot do input or output

5. No variable declaration

- No explicit typing

6. Implicit memory management:

- no new or free (malloc/realloc/delete)
- Program unaware of underlying memory structure

7. Referential transparency:

- Execution of a function always produce the same result when given the same parameters
- Implication: all variables in a function body must be local to that function; why?

FPL: Desiderata

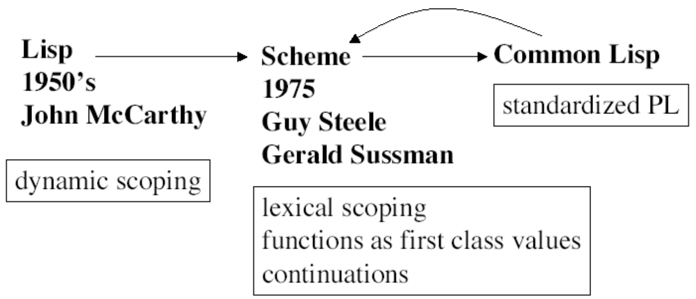
8. Functions can be:

- Passed as an argument
- Returned from a function
- Represented by a data structure and that data structure can then be evaluated

- **Is this possible?**

Scheme: Introduction

- **History:**



- **Scheme has a denotational semantics based on the lambda calculus:** *that is the meaning of all syntactic programming constructs in the language are defined in terms of mathematical functions*

- **A Scheme program consists of function definitions and calls. There is no other structure.**

- **A variable assumes the type of the value that is bound to them at run-time. So, the type of a variable changes dynamically during execution**

- **Automatic garbage collection.**

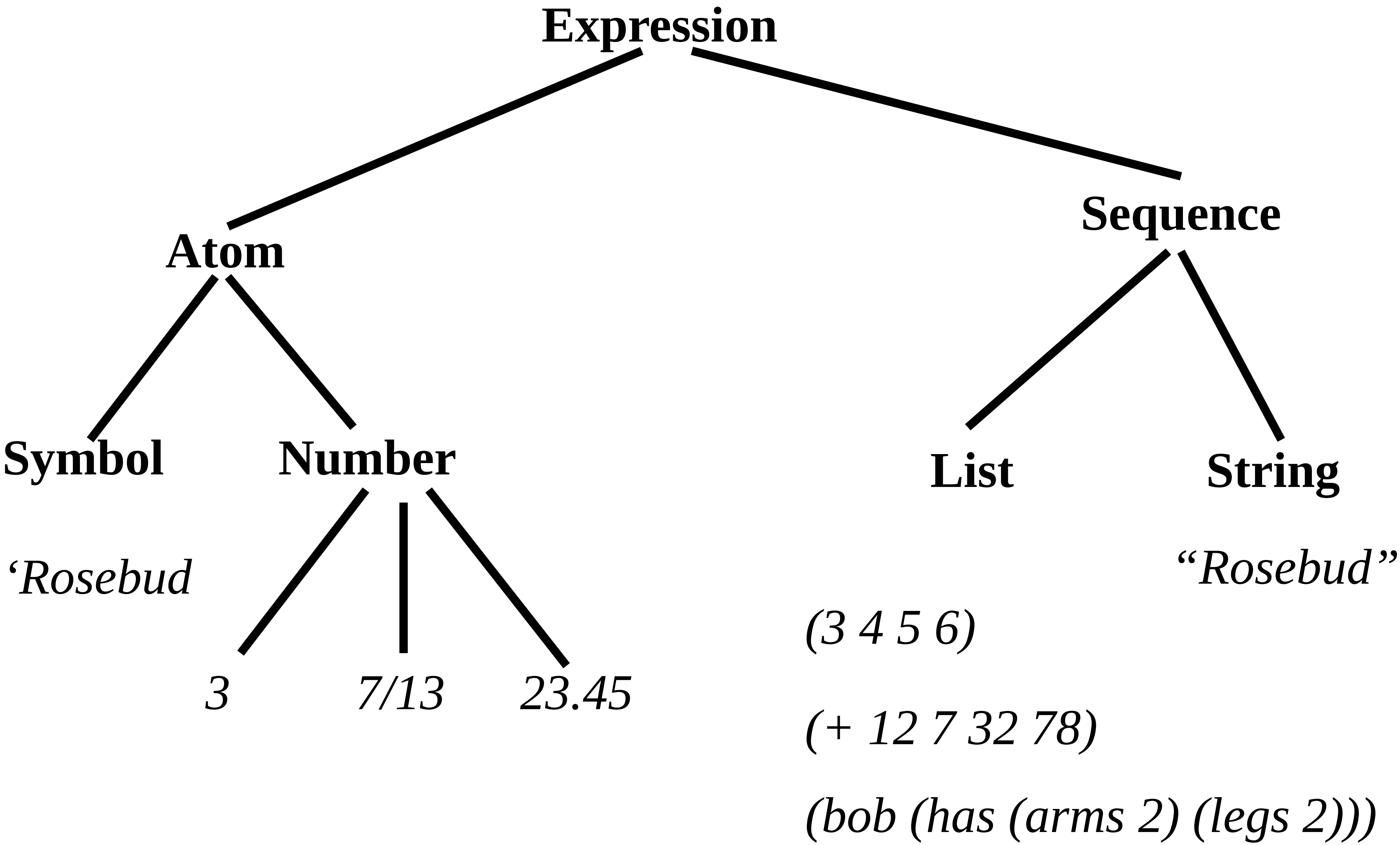
Scheme: Expressions

- **An expression in Scheme has the form (E_1 E_2 E_3…. E_n)**
  - E_1 evaluates to an operator
  - E_2 through E_n are evaluated as operands

- **Examples:**
  - (+ a b c) ; (a + b + c)
  - (+ 1 (* 2 3) 4 5) ; (1 + (2 * 3) + 4 + 5)
  - (+ (- 6 3) (/ 10 2) 2 (* 2 3)) ; 16
  - (<= (- 5 3) (+ 2 (* 3 3)) 14)
  - (not (= (sqrt (+ (expt 3 x) 1) y))
  - (max (+ 2 3) (abs –4) (remainder 12 5))

- **Postfix v.s. Infix:**
  - Scheme expressions use prefix notation while imperative languages use infix notation, *which is better*?

Scheme: Basic Data Types



Scheme: Evaluating Expressions

- **Using** *eval FORM*
  - Evaluate ⇔ compute/fetch value of an expression
  - Form ⇔ an expression to be evaluated
  - Rules:
    - A number evaluates to itself

      76 ➔ 76
    - A variable evaluates to its value

      (define x 54) ➔ x = 54
    - A quoted symbol evaluates to the symbol itself:

      ‘z ➔ z
    - A string evaluates to itself

      “trondheim” ➔ “trondheim”
    - A single quoted list evaluates to a simple list of symbols

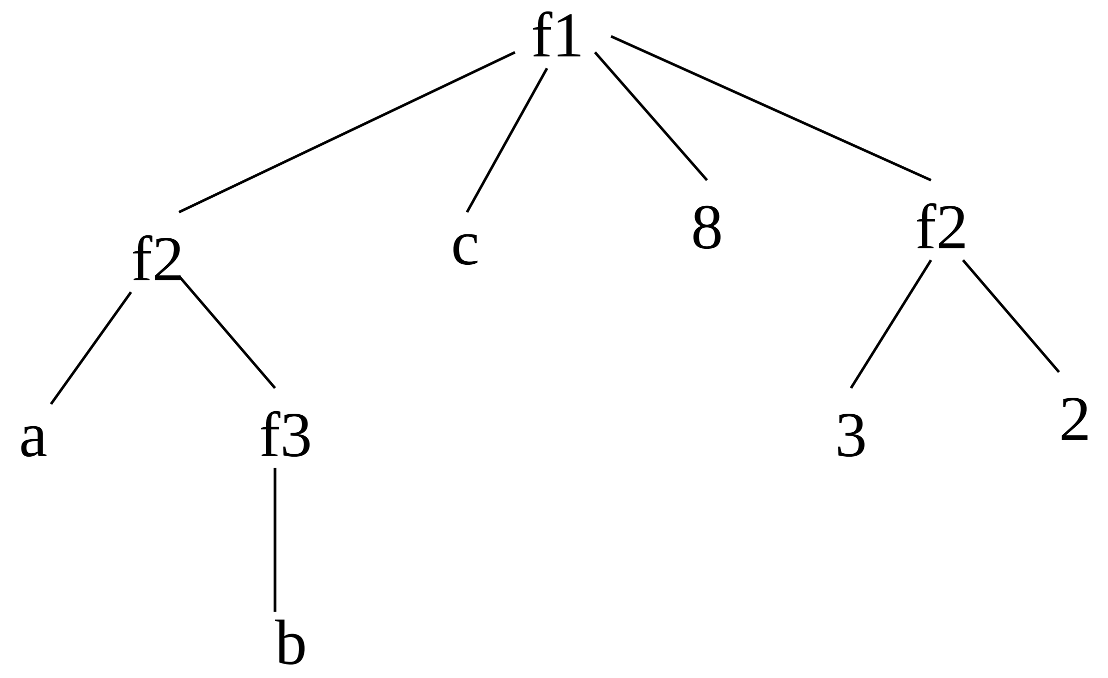
      ‘(+ 2 3) ➔ (+ 2 3)
    - An unquoted list evaluates to a function call

      (+ 2 3) ➔ 5

      (a b c) ➔ ERROR: attempt to call an undeclared function ‘a

Scheme: Evaluating Order

- **Scheme follows a depth-first applicative evaluation order**
- **Example: (f1 (f2 a (f3 b)) c 8 (f2 3 2))**

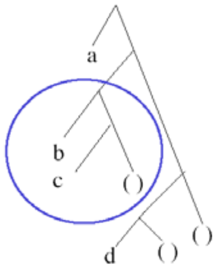


Scheme: Lists

- **A list is denoted by a collection of items enclosed in parentheses**

- **The empty list is denoted ()**
  - The list (2 4 6 8 10) is the same as (2 . (4 . (6 . (8 . (10 . ( ))))))
  - Improper list: a list that does not end with an empty list.

- **Example:**
  - **(a (b c) (d))**



Scheme: Lists

- **Note: Lists should be quoted when fed to the interpreter, otherwise the interpreter will try to apply the first item in the list to the other items**
  - **E.g.**

```
]=> (2 4 6 8)
error: procedure application: expected procedure, given: 2;
arguments were: 4 6 8

]=> ' (2 4 6 8)
(2 4 6 8)

]=> (quote (2 4 6 8))
(2 4 6 8)
```

Scheme: Lists cont'd

- **Constructing Lists:**
  - ***(cons* arg1 arg2)**
    - The <u>second</u> argument to *cons* must be a list
    - E.g.

      (cons 'peanut '(butter and jelly)) ; (peanut butter and jelly)

      (cons '(banana and) '(peanut butter and jelly)) ; ((banana and) peanut butter and jelly)

  - ***(append* arg1 arg2)**
    - Returns the list formed by joining the elements of a and b together.
    - Precondition: arg1 and arg2 must be lists

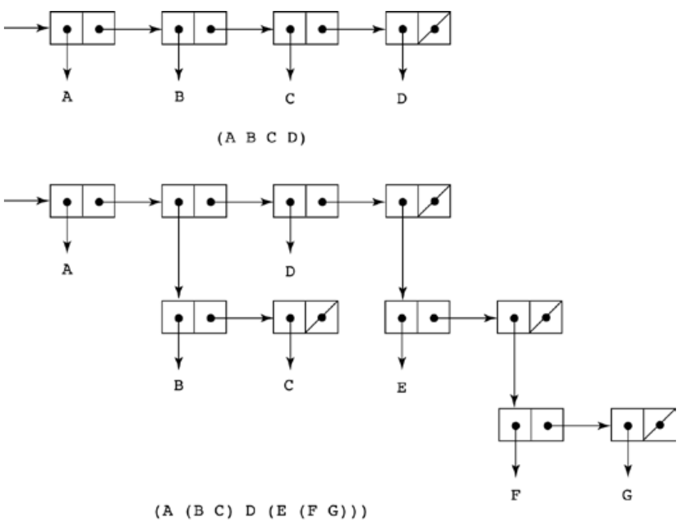
  - ***(list* arg1 arg2 … $\mathbf{arg_n}$)**
    - E.g.

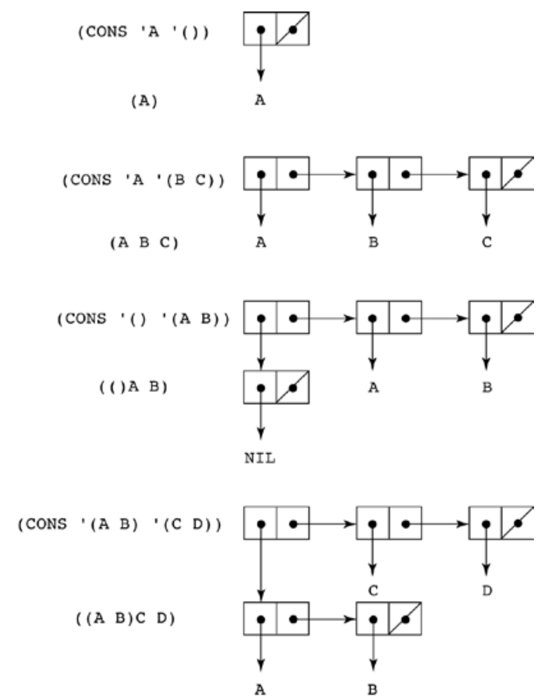
      (list 2 4 6 8 10)

Scheme: Lists cont'd

- **Internal implementation**
  - Linked list storage management used
  - Head: first member of the list.

    Tail: everything else other than the head



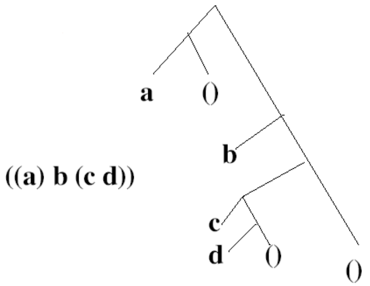


Scheme: Lists cont'd

- **Useful Operations:**
  - (car list) ; return head of the list , *pronounced car*
  - (cdr list) ; return tail of the list , *pronounced coulder*
  - (cadr list) ; eqv to (car (cdr list)) , *pronounced cahder*
  - (cdar list) ; eqv to (cdr (car list)) , *pronounced couldaher*
  - (caar list) ; eqv to (car (car list)) , *pronounced cahar*
  - (cddr list) ; eqv to (cdr (cdr list)) , *pronounced coulduhder*
  - (cadar list) ; eqv to (car (cdr (car list))) , *pronounced cahdauher*
  - (caadr list) ; eqv to (car (cadr list)) , *pronounced cahader*
  - (cdddr list) ; eqv to (cdr (cddr list)) , *pronounced couldduhduhder*
  - (cadadr list) ; eqv to ….
  - (reverse list) ; reverse the order of the elements in list
  - (member element list);

- **Example:**

(car '(a b c)) is a
(car '((a) b (c d))) is (a)
(cdr '(a b c)) is (b c)
(cdr '((a) b (c d))) is (b (c d))



Scheme: Lists cont'd

- **Examples:**

```
]=> (define lista '(1 2 5 67 3 2 5 88))
]=>(define fruits '(apple pear orange banana))
]=>(define colors '(red blue green yellow orange))
]=>(define prices '((banana 0.98) (orange 0.33) (lemon 0.20)))
```

```
]=>(car lista)
1

]=>(car colors)
red

]=>(cdr lista)
(2 5 67 3 2 5 88)

]=>(cdr colors)
(blue green yellow orange)

]=>(cadr colors)        ; (car  (cdr list))
blue

]=>(cadr fruits)
pear
```

```
]=>(caddr fruits) ; (car  (cdr  (cdr list)))
orange

]=>(cdddr fruits)    ; (cdr  (cdr  (cdr list)))
(banana)

]=>(car prices)
(banana 0.98)

]=>(caar prices)      ; (car  (car list))
banana

]=>(cadar prices) ;   (car  (cdr (car list)))
0.98
```

Scheme: Expressions/ Short Circuit Eval

- **(and ….)**
  - E.g.
    ```
    (and (try-first-thing)
         (try-second-thing)
         (try-third-thing)
    )
    ```
  - If the three calls all return true values, and returns the value of the last one.
  - If any of them returns #f, however, none of the rest are evaluated, and #f is returned as the value of the overall expression.
- **(or ….)**
  - E.g.
    ```
    (or (try-first-thing)
        (try-second-thing)
        (try-third-thing)
    )
    ```
  - Likewise, it stops when it gets a true value