Functional Pgming

Program execution = expression evaluation

Referential transparency

- value of expression independent of context (so static scope for globals)

Control flow only through function application (and recursion)

Functions are first class values

- can be passed as arguments
- can be saved in data structures
- can be returned as value
- unnamed functions as returned values
- ?run-time construction of functions?

(Implicit storage management)

Lisp (symbolic computation) --> Scheme (typed, cleaner)

Scheme

An *expression* is a

- a constant: #t,#f, 13, 4.55, "abe", 'red, #\a
- a *"variable identifier"* (bound *once* to a value)
- (**expr0 expr1 expr2 ...)** which will be interpreted as a function call

e.g., **(+ 2 3)** **--> 5**
(min 2 3) --> 2

note: prefix notation, no commas but spaces

No procedural variables in 'pure functional' language. i.e., ***x = x + 1;*** *does not exist, if the two x's are the same*

Binding names to values:

- some names are pre-bound (built-in constants, functions like **+, number?,...**)
- at top level, use **define** function

```
(define pi 3.14)
(define (square y)
       (* y y) )
```

- argument/parameter passing
  (square 3) causes **y <-->3** inside square
- can create "local variables" using **let**

```
(let  ((v 2)
        (w 3)
       )
     (* v (+ w 1)))
```

warning: **(let ((w (+ w 1)) ...** second w is from outside the let

Expressions and their values

Read-Eval-Print loop:
1. Read in an S-expression;
2. Evaluating:

constant yields itself
*identifier yields value it was bound to "most recently" (scoping!) -- **binding context** is the set of such visible identifiers and values*
expressions of the form **(e1 e2 e3 ...)**

- evaluate e1, expecting to get a function (code)
- evaluate e2,... to get arguments
- apply the function to the arguments

Control structures are function-like; they build expressions:

- conditional **if**

```
(if  arg1   arg2  arg3)
```

evaluates to the value of arg2 if arg1 evaluates to #t, and to the value of arg3 if arg1 evaluates to #f
e.g., **(if (> y 0) (+ 0 y) (- 0 y))**
will compute absolute value of y.

- multi-branch conditional **cond**

```
(cond
  ( (w < 0)  ('negative) )
  ( (w > 0)  ('positive) )
  (else      'zero)
)
```

3. Print result of evaluation

What's a function: definition & use

SYNTAX:

1a. at top level **(define (area h b) (/ (* b h) 2))**
use (area 3 4)

2• nameless function
What does a function need
- parameters **lambda (h b)**
- a body to evaluate **(/ (* h b) 2))**

put them together:
(lambda (h b) (/ (* h b) 2))
use
((lambda (h b) (/ (h b) 2)) 3 4)*

1b• **(define area (lambda (h b) (/(* h b) 2))**

What it means for function to be first class:
instead of **(if (> y 0) (+ 0 y) (- 0 y))**
note that the above returned expressions are of the form **(f 0 y)** where **f** is either **+** or **-** ;
So, consider:

```
(  (if (> y 0) + -)
    0
    y)
```

Yes, **(if (> y 0) + -)** has value **+** or **-** !!!

More DEF'S:

```
(define (square y)
   (if
    (= 1 y)
    1
    (+
      (square (- y 1))
      y
      y
      -1
     )))
```



```
evaluate (square 3)
  square --> [code]
  3  --> 3
  apply square code  to 3
    enter binding context with   y --> 3
   |evaluate (if (= y 1) 1 (+ (square (- y 1))...
   |  if --> form
   |  evaluate (= y 1) ... --> #f
   |  evaluate (+ (square (- y 1)) y y -1 )
   |   |+  --> code for addition
   |   |evaluate (square (- y 1))
   |   | square --> [code]
   |   | evaluate  (- y 1) --> ... --> 2
   |   | apply square code to 2
   |   |     enter binding context with y --> 2
   |   |    evaluate  (if (= y 1) 1 (+ (square ...)
   |   |    |   ...
   |   |    | evaluate (+ (square (- y 1) y y -1)
   |   |    |   + -> code for addition
   |   |    |   evaluate  (square (- y 1))
   |   |    |      square --> code
   |   |    |      (- y 1) --> 1
   |   |    |      apply square code to 1
   |   |    |          enter binding context y --> 1
   |   |    |            (if ...) --> 1
   |   |    |          exit  binding context with return 1
   |   |    |    y --> 2
   |   |    |    y --> 2
   |   |    |    -1 --> -1
   |   |    |   apply addition  --> 4
   |   |    |value of (if ...) -->  4
   |   |    exit binding context with return 4
   |   |y  --> 3
   |   |y  --> 3
   |   |-1  -->  -1
   |     apply addition  --> 9
   |
```

Lists

Recursive definition:

- empty list is a list **`()`**
- if **`v`** is some value, and **`L`** is a list, then **`(v L)`** is a list.

e.g. **`(a b c d)`** vs. **`(a (b c) (d))`**

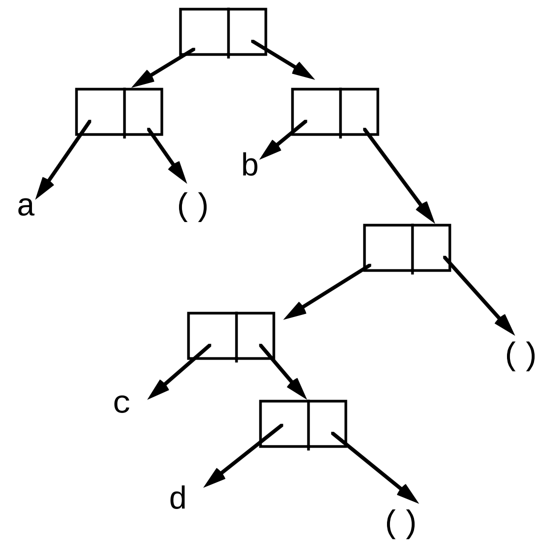
List constants: look like function calls!! Need to "block" evaluation. Use quote function

`'(a b c)` same as **`(quote (a b c))`**

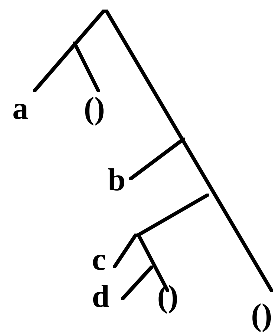
Functions on lists

- check if list is empty **`null?`**
  `'()` vs. **`'(())`** vs. **`'(()())`**
- to get *first* value of non-empty:
  (<u>car</u> <list>**)**
  `(car '(a b)) --> a`
- to get *rest*
  (<u>cdr</u> <list>**)**
  `(cdr '(a b)) --> '(b)`
- *create* a new list
  (<u>cons</u> <value> <list>**)**
  `(cons 1 '(f 3)) --> ( 1 f 3)`
  (<u>list</u> <value1> <value2> ...**)**
  `(list 1 2 b) --> (1 2 b)`
  (<u>append</u> <list1> <list2>**)**
  `(append '(a b) '(3 (b) c))`
  `--> (a b 3 (b) c)`

`( (a) b (c d) )` as cons cells



Examples



Can compose these operators in a short-hand manner. Can reach any arbitrary list element by composition of car's and cdr's.

(car (cdr (cdr '((a) b (c d))))) **=** *can also be written* **(caddr '((a) b (c d)))**

(car (cdr '(b (c d))) =>

(car '((c d)) => (c d)

(cons '(a b c) '((a) b (c d))) =>

((a b c) (a) b (c d))

(cons 'd '(e)) => (d e)

(cons '(a b) '(c d)) => ((a b) c d)

(car '()) --> run-time error; arg must be a pair.

Suppose f=2,g=3. **'(f g)** vs **('f 'g)** vs **'('f 'g)** vs **(list f g)** vs **(cons f g)** vs **'(list f g)**

•• Types/predicates in Scheme, plus constants,ops:

	true	***false***
boolean?	**#t #f**	**2 'a**
and, or, not		
number?	**43 1.45**	**'b**
+ * = >		
symbol?	**'bob**	**2**
eq?		
procedure?	**+**	**2**
apply, eval		
list?	**'((2 3) 5)**	**1 'b**
pair?	**'(3)**	**() 2**
null?	**()**	**'(a) 'b 3**
car,cdr,cons		
list		
eq?		
equal?		

special 'forms' (do not evaluate all arguments)

if

cond

•• ***Eval and quote**: when Scheme sees*

- *a non-list, it tries to look up its value*

  [2->2, #t->#t,

  identifier -> value bound to variable named it]

  e.g. **`+ --`**> #procedure

- *a list (b c ...), it sees a function call to function b on arguments c ...*

To stop this, use **quote `'` : (quote n)** or **'n**

`'2  '#t  ->` result is themselves 2, #t

`'n` --> symbol **`n`**

To reverse the process: **eval**

`(eval '(+ 1 2)) -->  3`

`(eval 'c) -->` lookup value bound to variable c