Functional Programming

- Pure functional PLs
- S-expressions
  - cons, car, cdr
- Defining functions
- read-eval-print loop of Lisp interpreter
- Examples of recursive functions
  - Shallow, deep
- Equality testing

Pure Functional Languages

- Referential transparency
  - value of an expression is independent of context where the function application occurs
  - means that all variables in a function body must be local to that function; why?
- There is no concept of assignment
  - variables are bound to values only through parameter associations
  - no side effects

Pure Functional Languages

- Control flow accomplished through function application (and recursion)
  - a program is a set of function definitions and their application to arguments
- Implicit storage management
  - copy semantics, needs garbage collection
- Functions are 1st class values!
  - can be returned as value of an expression or function application
  - can be passed as an argument
  - can be put into a data structure and saved

Pure Functional Languages

  - Unnamed functions exist as values
- Lisp designed for symbolic computing
  - simple syntax
  - data and programs have same syntactic form
    - S-expression
  - function application written in prefix form

    (e1 e2 e3 … ek) means
    - Evaluate e1 to a function value
    - Evaluate each of e2,…,ek to values
    - Apply the function to these values

    (+ 1 3) evaluates to 4

History

Lisp → **Scheme** → **Common Lisp**

Lisp
1950's
John McCarthy

dynamic scoping

Scheme
1975
Guy Steele
Gerald Sussman

lexical scoping
functions as first class values
continuations

Common Lisp

standardized PL

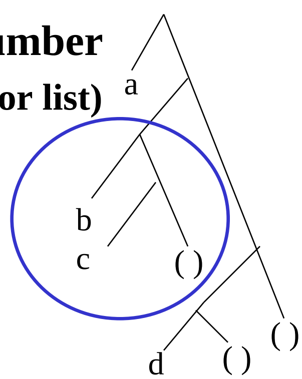
S-expressions

S-expr ::= Name | Number | ({ S-expr })

Name is a symbolic constant, some string of chars which starts off with anything that can't start a Number

Number is an integer or real number

- **E.g., (a (b c) (d)) is an S-expr (or list)**
- **car selects the first element**
  - **car of this S-expr is a**
- **cdr selects the rest of the list**
  - **cdr of this S-expr is ((b c) (d))**



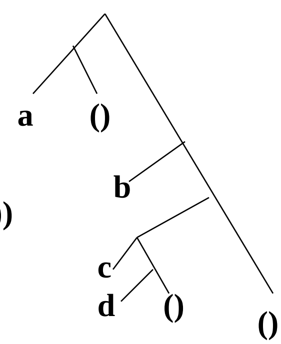
List Operators

- *Car* and *cdr*
  - Given a list, they decompose it into first element, rest of list portions
- *Cons*
  - Given an element and a list, cons builds a new list with the element as its car and the list as its cdr
- () means the empty list in Scheme

Examples

(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))

((a) b (c d))



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).

Examples

(cons ‘(a b c) ‘((a) b (c d))) is ((a b c) (a) b (c d))

(cons ‘d ‘(e)) is (d e)

(cons ‘(a b) ‘(c d)) is ((a b) c d)

Useful predicates in Scheme. Note the quote prevents evaluation of the argument as an S-expr.

(**symbol?** ‘sam) returns #t (symbol? 1) returns #f

(**number?** ‘sam) returns #f (number? 1) returns #t

(**list?** ‘(a b)) returns #t (list? ‘a) returns #f

(**null?** ‘()) returns #t (null? ‘(a b)) returns #f

(**zero?** 0) returns #t (zero? 1) returns #f

Can compose these.

(zero? (- 3 3)) returns #t *note that since this language is fully parenthesized, there are no precedence problems in the expressions!*

Scheme

Fcn-def ::= (define (Fcn-name {Param}) S-expr)

Fcn-name should be a new name for a fcn.

Param should be variable(s) that appear in the S-expr which is the function body.

Fcn-def ::= (define Fcn-name Fcn-value)

Fcn-value::= (lambda ({Param}) S-expr)

where Param variables are expected to appear in the S-expr; called a *lambda expression.*

Scheme Examples

(define (zerocheck? x) (if (= x 0) #t #f))

If-expr ::= (if S-expr0 S-expr1 S-expr2)

where S-expr0 must evaluate to a boolean value; if that value is true, then the If-expr returns the value of S-expr1, else the value of S-expr2.

(zerocheck? 1) returns #f, (zerocheck? (* 1 0)) returns #t

(define (atom? object) (not (pair? object)))

where *pair? returns #t if argument is non-trivial S-expr* (something you can take the cdr of), *else returns #f*

not is a logical operator

Scheme Examples

(define square (lambda (n) (* n n)))

- This associates the Fcn-name square with the function value (lambda (n) (* n n)))
- *Lambda calculus* is a formal system for defining recursive functions and their properties
  - Set of functions definable using lambda calculus (Church 1941) is same as set of functions computable as Turning Machines (Turing 1930's)

Trace of Evaluation

(define (atom? object) (not (pair? object)))

(atom? ‘(a))

-obtain function value corresponding to *atom?*

-evaluate ‘(a) obtaining (a)

-evaluate (not (pair? object))

- **-obtain function value corresponding to *not***
- **-evaluate (pair? object)**
  - **-obtain function value corresponding to *pair?***
  - **-evaluate object obtaining (a)**
  - **-return value *#t***
- **-return *#f***

-return *#f*

Read-eval-print loop

- **How does a Scheme interpreter work?**
  - ***Read* input from user**
    - **A function definition or abstraction**
    - **A function evaluation**
  - ***Evaluate* input**
    - **Store function definition**
    - **(e1 e2 e3 … ek)**
      - **Evaluate e1 to obtain a function**
      - **Evaluate e2, … , ek to values**
      - **Execute function body using values from previous step as formal parameter values**
      - **Return value of function**
  - ***Print* return value**

Conditional Execution

(if e1 e2 e3)

(cond (e1 h1) (e2 h2)…(en-1 hn-1) (else hn))

- Cond is like a nested if-then-elseif construct

```
(define (zerocheck? x)
      (cond ((= x 0) #t) (else #f)))
```

OR

```
(define (zchk? x)
  (cond ((number? x) (zero? x))
          (else #f)) )
```


Recursive Functions

(define (len x) (cond ((null? x) 0) (else (+ 1 (len (cdr x))))))

(len ‘(1 2)) should yield 2.

len is a shallow recursive function

```
Trace: (len ‘(1 2)) --top level call
          x = (1 2)
                (len ‘(2)) --recursive call 1
                x = (2)
                      (len ‘( ) ) -- recursive call 2
                      x = ( )
                      returns 0 --return for call 2
                returns (+ 1 0) =1 --return for call 1
          returns (+ 1 1) = 2 --return for top level call
```

(len ‘((a) b (c d))) returns 3

List Append

(define (app x y)
(cond ((null? x) y)
((null? y) x)
(else (cons (car x) (app (cdr x) y)))))

(app ‘( ) ‘( )) yields ( )

(app ‘( ) ‘(1 4 5)) yields (1 4 5)

(app ‘(5 9) ‘(a (4) 6)) yields (5 9 a (4) 6)

another shallow recursive function

- **Can we write a function that counts the number of atoms in a list? (this will have to be a *deep function*)**

Atomcount Function

atomcount is a deep recursive function

(define (atomcount x)
(cond ((null? x) 0)
((atom? x) 1)
(else (+ (atomcount (car x)) (atomcount (cdr x))))))

~~**(atomcount ‘(1)) yields 1**~~

(atomcount ‘(1 (2 (3)) (5)))yields 4

Trace: (atomcount ‘(1 (2 (3)))
1> (+ (atomcount 1) (atomcount ‘((2 (3)))))
2> (+ (atomcount ‘(2 (3))) (atomcount ‘( )))
3> (+ (atomcount 2) (atomcount ‘((3))) etc.
4> (+ (atomcount ‘(3)) (atomcount ‘( )))
5> (+ (atomcount 3) (atomcount ‘( )))
1
0

Equality Testing

eq?

- **predicate that can check atoms for equal values**
- **doesn't work on lists**

eql?

- **comparison function for lists**

```
(define (eql?  x  y)
   (or  (and (atom?  x) (atom? y) (eq?  x  y))
          (and  (not (atom? x)) (not (atom? y))
                  (eql?  (car x)  (car y))
                  (eql?  (cdr x)  (cdr y)) ))
```


Examples

(eql? '(a) '(a)) yields #t
(eql? 'a 'b) yields #f
(eql? 'b 'b) yields #t
(eql? '((a)) '(a)) yields #f

(eq? 'a 'a) yields #t
(eq? '(a) '(a)) yields #f

How does eq? work?

(define (f x   y) (list x y))

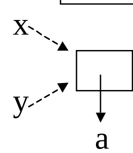
so (f 'a 'a) yields (a a).

How does Scheme implement this?

It binds both x and y to the same atom a.

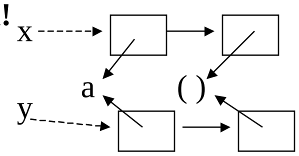
eq? checks that x and y both point to the same place

the atom: a



Say we called (f '(a) '(a)). then x and y don't point to the same list at all!

the arguments: (a)



Functional Programming, CS314 Fall 01© BGRyder

21