

Functional Programming

- **Pure functional PLs**
- **S-expressions**
- **Defining functions**
- *read-eval-print* loop
- **Examples of recursive functions**
 - shallow vs. deep
- **Equality testing**

1

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
- **There is no concept of assignment**
 - variables are bound to values only through parameter associations
 - no side effects

2

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 first class values!**
 - can be returned as value of an expression or function application
 - can be passed as an argument
 - can be stored as part of a data structure

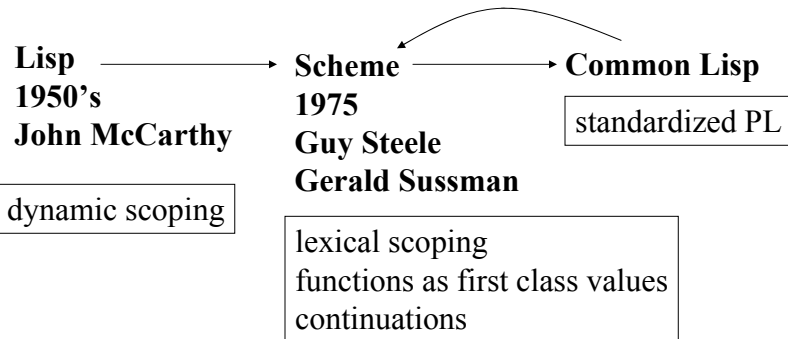
3

Pure Functional Languages

- **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
 - example: (+ 1 3) evaluates to 4

4

History



5

S-expressions

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

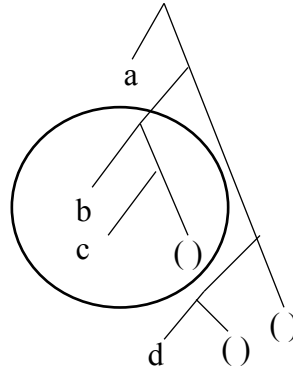
- **Name is a symbolic constant, some string of chars which starts off with anything that can't start with a Number**
- **Number is an integer or real number**
- **Parenthesis () are used to form lists of s-expressions**
 - **lists are heterogenous (contain elements of different types)**

6

Lists as Data

$\langle \text{datum} \rangle ::= \langle \text{number} \rangle \mid \langle \text{symbol} \rangle \mid \langle \text{list} \rangle \mid \dots$
 $\langle \text{list} \rangle ::= '(\langle \text{datum} \rangle \{ \langle \text{datum} \rangle \} . \langle \text{datum} \rangle)'$ |
 $'(\{ \langle \text{datum} \rangle \})'$ |

- (a (b c) (d)) is a list
- () is the empty list
- (b c) is a proper list
- (a . b) is an improper list,
also called a “dotted pair”
- (a b . c) is an improper list



7

Function Application

To evaluate a function application, e.g., (+ 3 5):

- evaluate the first element
+ \rightarrow [machine code for addition]
- evaluate the remaining elements
3 \rightarrow 3
5 \rightarrow 5
- apply the value of the first to the values of the rest
(+ 3 5) \rightarrow 8

This is like the mathematical concept of a function:

- plus(3, 5) = 8

8

Function Abstraction

How to evaluate `((lambda (x) (+ x x)) 4)` ?

`(lambda (x) (+ x x))` → [function ...]

`4` → `4`

- apply [function ...] to `4` by creating a new binding context in which `x` has the value `4`, and then evaluating

`(+ x x)` in this context:

→ ([machine code for addition] `4 4`)

→ `8`

11

Function Definitions

If we want this new function to be available in a global context, we can give it a name at the “top level”:

`(define double (lambda (x) (+ x x)))`

We can then use this name in a function application:

`(double 4)` → `8`

We can abbreviate the form of a function definition:

`(define (double x) (+ x x))`

Note: different books use different styles

12

Operations on Lists

Basic Functions:

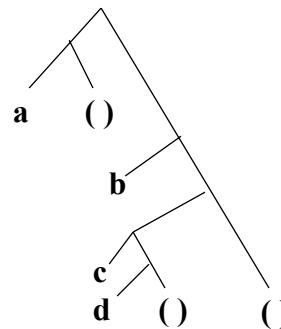
- `(car list)` returns the first element of `list`
- `(cdr list)` returns the rest of `list`
- `(cons element list)` constructs a new list by adding `element` to the front of `list`
- `(null? list)` returns `#t` if `list` is empty, otherwise it returns `#f`
- `(quote (a b c))` or `'(a b c)` is a special form that “quotes” its arguments without evaluation.

13

Operations on Lists

Examples:

- `(car '(a b c))` → `a`
- `(cdr '(a b c))` → `(b c)`
- `(car '((a) b (c d)))` → `(a)`
- `(cdr '((a) b (c d)))` → `(b (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)`



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

14

Logical operators

- **#f** false
- **#t** true (really, all other values except #f)
- **(or <expr1> ... <exprN>)**
returns the leftmost expression in the list that is **not** false, or #f if they are all false
- **(and <expr1> ... <exprN>)**
returns <exprN> if none of the expressions are #f, or #f if any are false
- **(not <expr>)**
negates <expr>, returning only #t or #f

15

Predicates

(symbol? 'x)	returns #t	(symbol? 1)	returns #f
(number? 1)	returns #t	(number? 'x)	returns #f
(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

Of course, these are functions and can be composed:

(zero? (- 3 3)) returns #t

Since this language is fully parenthesized, there are no precedence problems in the expressions!

16

Read-Eval-Print Loop

- ***Read* input from user**
 - A function (or symbol) definition
 - A function application
- ***Evaluate* input**
 - Store function (or symbol) definition
 - Evaluate function application
- ***Print* return value**
 - The name of the function (or symbol) being defined
 - The value of the function application