Functions, Conditionals & Predicates

York University CSE 3401

Vida Movahedi

Overview

- Functions as lambda terms
- Defining functions
- Variables (bound vs. free, local vs. global)
- Scoping (static vs. dynamic)
- Predicates
- Conditionals

[ref.: Wilensky Chap 3-4]

Functions as lambda terms

- In lambda calculus, function $f(x)=x^2$ is written as

$$(\lambda x.(* \ x \ x))$$

- In LISP it is written as:

(lambda (x) (* x x))

- In lambda calculus, an application of above function to 5 is written as

$$((\lambda x.(* \ x \ x)) \, 5)$$

- In LISP, it is written as

((lambda (x) (* x x)) 5)

Functions as lambda terms

- In the LISP interpreter environment, we type in a lambda term for evaluation (β-reduction).
- If we write a list for evaluation, the first element is always assumed to be a function (We input a β-redex).

```
> ((lambda (x) (* x  x))   5)
25
> ((lambda (x y) (cons x y))  1 '(2 3 4))
(1 2 3 4)
```

A β-redex: A lambda abstraction applied to a term

- It is not convenient to write the lambda definition of functions every time we need them, therefore we can give them names using **defun**.

Defining Functions

- Defining functions using defun

```
> (defun half(x) (/ x 2))
HALF
> (half 3)
3/2
> (defun average (x y) (/ (+ x y) 2.0))
AVERAGE
> (average 3 2)
2.5
```

- Formal parameters
  - x and y above are formal parameters of function definition of average
  - Formal parameters must be symbols
  - The value of formal parameters is not confused with their value outside the definition

More generally ...

- Function definition:
  (defun fname (p1 p2 ...pn)
  (... body 1 ...)
  (... body 2 ...) ...
  (... body m ...))

- Calling above function
  (fname a1 a2 ...an)

- Evaluation:
  (1) Definition of fname is retrieved
  (2) Supplied arguments *a1...an* are evaluated to get actual arguments *b1...bn*
  (3) Formal arguments *p1...pn* are assigned actual arguments *b1...bn* respectively
  (4) Each code in *body 1* to *body m* is evaluated in the order
  (5) The value returned by *body m* is returned.

Bound vs. free variables

```
> (setq x 5)
5
> (defun funone (x) (* x 2))
FUNONE
> (funone 6)
12
> x
5

> (defun funtwo (y) (* x y))
FUNTWO
> (funtwo 6)
30
> y
Error! Variable Y has no value
```

x is the formal parameter of funone and is a bound variable

Value of x at top-level does not change.

X is NOT the formal parameter of funtwo and therefore is free here. Its value is the same as top-level.

Bound vs. free (cont.)

```
> (setq sum 0)
0
> (defun getsum(x)
    (setq y 10)
    (setq sum (+ x y)))

> (getsum 5)
15
> sum
15
> y
10
```

x is bound,
y and sum are free

Changes to free variables are **NOT** local to a function!

setq creates a variable at top-level!

Bound vs. free (cont.)

- Each time a function is called, a new variable is created for each formal parameter.
- Changing value of formal parameters does not effect value of symbols with the same name at interpreter level (top-level).
- Formal parameters are **bound** variables.
- A symbol (variable) used in definition of a function that is not a formal parameter of the function is a **free** variable.
- Changing value of free variables changes their value at top-level.

Local vs. global variables

- Bound variables are **local** variables, can only be accessed inside function calls.
- A variable at the top-level (interpreter level) is a **global** variable.
- It can be accessed at the top-level, or inside function calls.
- Free variables refer to the same global variables at top-level.
- If **setq** is used with free variables, it can create global variables, or change value of existing ones! Do not use inside function definitions unless you intentionally want global effect. Not a good idea!

Let and Let*

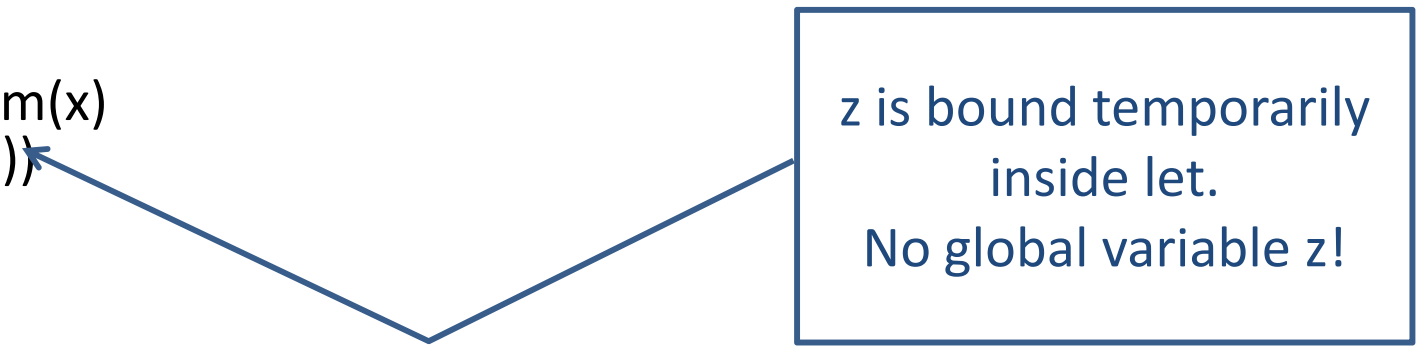
- Use **let** for temporary binding

  (let ((*var1 value1*)
  (*var2 value2*) ...
  (*varm valuem*))
  (body1)
  ***(body2)* ...**
  ***(bodyn)*)**

  – The value returned by body n is returned
  – **Let** assigns all values in parallel, use **let*** if you need sequential assignment

- Example

```
> (defun getsum(x)
      (let  ((z 10))
         (+ x z)))
> (getsum 10)
20
> z
Error! Variable z has no value
```



Static vs. Dynamic Scoping

- Scoping
  - referring to the method of choosing which variable is referenced by a symbol inside a function

  1. Static (or lexical) scoping: A given symbol refers to a variable in its local environment, and depends on the function definition in which it is accessed.
  2. Dynamic scoping: A given symbol refers to the most recently created variable with that name (at any level).

  - Common LISP uses lexical scoping for formal parameters of a function.

Static vs. Dynamic (cont.)

- Example:

```
>(defun f (x y)
     (+ x (g y)))
>(defun g (y)
     (* 10 x))
```

<u>By static scoping:</u>

```
> (f 2 3)
Error! X has no value in G!
```

Static scoping is used in Common LISP.

<u>By dynamic scoping:</u>

```
> (f 2 3)
22
```

Saving function in files

- Save your function definitions in files
- Load into LISP using **load**
- Examples:

```
(load 'test.lsp)
(load 'mylisp/test.lsp)
(load "c:\\lispcode\\first.L")
```

LOGIC AS FUNCTIONS

True and False

- The constant **nil** indicates false
- The constant **t** indicates true
- Any non-nil value is actually true as well
- Some built-in predicates, such as **not**, **and**, **or**, ...

Some built-in predicates

(**atom** arg)	returns T if arg is an atom (NIL otherwise)
(**listp** arg)	returns T if arg is a list
(**null** arg)	returns T if arg is nil
	(note: **not** and **null** have the same behaviour)
(**symbolp** arg)	returns T if arg is a symbol
(**numberp** arg)	returns T if arg is a number
(**integerp** arg)	returns T if arg is an integer
(**equal** arg1 arg2)	returns T if the two arguments look alike
(**typep** arg type)	returns T if arg is of type 'type'
	type can be 'number, 'atom, 'symbol, 'list, 'null, ...

Examples

```
> (setq x 5)
5
> (atom x)
T
> (atom 'x)
T
> (atom 5)
T
> (atom '(1 2 3))
NIL
> (listp '(1 2 3))
T
> (listp x)
NIL
```

```
> (numberp x)
T
> (numberp 'x)
NIL
> (symbolp x)
NIL
> (symbolp 'x)
T
> (typep 'x 'number)
NIL
> (typep x 'number)
T
```

```
> (typep 'x 'symbol)
T
> (atom nil)
T
> (listp nil)
T
> (numberp nil)
NIL
> (symbolp nil)
T
> (null nil)
T
```

Built-in predicates for arithmetic

(**zerop** arg) returns T if arg is zero

(**oddp** arg) returns T if arg is an odd number

(**evenp** arg) returns T if arg is an even number

(**>** arg1 arg2) returns T if arg1 is greater than arg2

(<, >=, <= are also defined)

(= arg1 arg2) returns T if arguments are equal numbers

Conditionals: cond

- **cond** is similar to "if ... then "
  - Example: If x is a list, return its head

```
(cond
      ( (listp x)  (car x)) )
```

  - Example: If x is a list, return its head, otherwise return x itself.

```
(cond
      ( (lisp x) (car x) )
      (   T          x     ) )
```

T acts like "otherwise"

Two **cond** clauses in this example.
If the test in one **cond** clause is true, the rest of clause is evaluated and all other cond clauses are ignored.

Conditionals: cond

- **General form:** **(cond**
  (exp11 exp12 exp 13 ...)
  (exp21 exp12 exp 13 ...)
  :
  :
  (expn1 exp12 exp 13 ...))
  - Each of the above list of expressions (1 to n) is called a **cond clause**.
  - Each cond clause can have one to as many expressions. The first expression in each cond clause is the **test**.
  - The cond clauses will be examined in the order. If the test of a cond clause evaluates to **false**, the rest of expressions in that cond clause will be ignored and the next cond clause will be examined.
  - If the test evaluates to **true**, the rest of the expressions in the cond clause will be evaluated. The value returned by the last expression is returned as the value of cond. All remaining cond clauses will be ignored.
  - If none of the tests evaluates to true, cond will evaluate to **nil**.

Example

```
(defun mymin (x y z)
      (cond
            ( (not (and (numberp x)
                        (numberp y)
                        (numberp z)))    nil)
            ( (and (<= x y) (<= x z))    x)
            ( (and (<= y x) (<= y z))    y)
            ( T z)))
```

- It accepts 3 arguments. If all arguments are numbers, it returns the minimum. Otherwise it will return nil.

Example

```
(defun nilzero (x)
   (cond
              ((null x))
              ((and (numberp x)(zerop x))) ))
```

- If the given argument is **nil** or **zero,** it will return T, otherwise nil.
- Both cond clauses have only one expression (only the test). If the test is true, its value true (**T**) will be returned.
- If none of them evaluates to true, **nil** will be returned.
- In the second cond clause, we cannot use zerop without numberp. Why?

Other conditionals

- If

  These two expressions perform the same function,
  (Add e to list lst if it is not already a member of lst)
  (member is a pre-defined function in Common LISP)

```
(if (member e lst) lst
    (cons e lst))
```

=

```
(cond ((member e lst) lst)
      (t (cons e lst)))
```

- If evaluates its first argument, if true, evaluates and returns the second argument. Otherwise evaluates and returns the third argument (if present).
- Others such as **when**, **unless**, **case**