1

Scoping of Variables

What is the value of variable x?

Variables in the Outer level

In the outer level the initial value of variable (atom) `x` is *undefined*

`setq` will assign a value to an atom:

```
-->  x
   undefined
-->  (setq x '(a b c))
   (a b c)
-->  x
   (a b c)
```

Variables that have values at the top level are sometimes called global variables.

©1993 John Mylopoulos

2

Variables as Formal Parameters

Consider the following example:

```
-->  y
   undefined
-->  (defun change-y (y)  (setq y 3)))
   change-y
-->(change-y 4)
   3
--> y
   undefined
```

Before the body of a function is evaluated with respect to its arguments,

the formal parameter `y` is bound to the value of the corresponding argument.

The value of variable `y` before the binding is saved, and then `y` is set to the value of the corresponding argument.

After the value of the function has been determined, and before the value is returned,

the saved value of each formal parameter is restored.

©1993 John Mylopoulos

3

Bound Variables

Variables serving as formal parameters of a function are bound variables within the function definition.

For example the following definition takes two arguments and returns a 2-element list consisting of the first of the first argument and the rest of the second argument:

```
(lambda  (y z)
   (  (lambda (x  z)(list  x  z))
        (first  y )
        (rest  z)    ))
```

In the outer lambda expression `y` is bound to the value of the first argument, while `z` is bound to the value of the second argument

In the inner lambda expression `x` is bound to the `first` of the value of first expression, `z` is bound to the `rest` of the value of second expression

Note that even though `z` is already bound in the outer lambda expression, it can be bound again within the inner lambda expression. The inner `z` shadows the outer `z`.

©1993 John Mylopoulos

4

Free Variables

Variables used in the body of a function that are not bound are called free variables. For example consider the following function `foo` when called with arguments `f1` and `f2`:

```
(defun  foo  (x y) (list x y z))
```

`x, y` will be bound to the values of `f1` and `f2`
`z` will be free

What is the value of `z` when `foo` is called? Case 1

```
((lambda (x z) (foo 'a  (cdr x)))  '(5)  t)
```

When `(foo 'a (cdr x))` is evaluated,
`x` gets bound to `(5)`
`z` gets bound to `t`

Evaluation of `(list x y z)` inside the body of `foo` leads to
`x` getting bound to `a`
`y` getting bound to `()`
`z` getting bound to `t`

value returned: `(a () t)`

©1993 John Mylopoulos

5

What is the value of `z` when `foo` is called? Case 2

```
(defun  foo  (x y) (list x y z))
```

`x, y` will be **bound** to the values of `f1` and `f2`
`z` will be **free**

```
((lambda (x) (foo 'a  (cdr x)))  '(5) )
```

Now, evaluation of `(lambda ...)` leads to
- `x` getting bound to `(5)`
- `z` remains *undefined*

Inside `(foo 'a (cdr x))` evaluation of `foo` leads to
- `x` getting bound to `a`
- `y` getting bound to `()`
- `z` remains *undefined*

Evaluation of `list` is *undefined* !!!

Common Lisp: `foo` is only defined when the variable `z` is a global variable;
Franz Lisp: `foo` is defined only if `z` is a global variable or `foo` is used (directly or indirectly) as a helping function to a function `f1`, that binds `z`.

©1993 John Mylopoulos

6

Static versus Dynamic Scoping

In dynamic scoping the value of a variable depends on the run-time program state!

For example in languages such as Franz Lisp, Snobol

In static scoping the value of a variable is determined by the program definition within which the variable appears

For example, in languages such as Scheme, C, Pascal, Turing,...

Static scoping example:

	Static	Dynamic
foo: declare x		
bar:		
put x	**(prints 5)**	**(prints 7)**
glob: declare x; x <- 7		
bar()		
put x	**(prints 7)**	**(prints 7)**
x <- 5; glob()		

©1993 John Mylopoulos

7

Contexts for Dynamic Scoping

To keep track of the values associated with a given variable in different contexts, each variable is associated with a LIFO (last-in-first-out) stack of values.

The current value of a variable is at the top of the stack. (the variable is *undefined* if the stack is empty)

• Initializing/updating a variable's stack: Initially, `var`'s stack is empty.

```
(setq var  form)
```
--> reset top of `var`'s stack to the value of `form`.

• When entering function `fn` with parameters `a1`, ..., `an` bound to values `v1`, ..., `vn`
--> push the value `vi` onto `ai` 's stack for each i

• When exiting function `fn`
--> pop stack for each of the function parameters `ai`

©1993 John Mylopoulos

8

Maintaining Contexts - An Example

• To evaluate:

```
((lambda (x y) ((lambda (z x)(list z x y))
        'a (cdr x))) '(a b c)   '(d e f))
```

• Initially, the context is:
`x/[]  y/[]  z/[]`

• Upon entrance of the outer lambda the context becomes:
`x/[(a b c)]  y/[(d e f )]  z/[]`

• Upon entrance of inner lambda:
`x/[(b c)(a b c)]  y/[(d e f )]  z/[a]`

• Exit inner lambda with value `(a (b c)(d e f))` and restore context to:
`x/[(a b c)]  y/[(d e f)]  z/[]`

• Exit outer lambda with value `(a (b c) (d e f))` and restore context to:
`x/[]  y/[]  z/[]`

--> `(a (b c) (d e f))`

©1993 John Mylopoulos

9

Another Example (Franz Lisp):

```
(defun foo (x y) (append x (bar y)))

(defun bar (p)
   (cond ((null p) x)
         (t (foo y (cdr p)))))
```

Evaluate: `(foo '(a) '(b c))`

1. Enter `foo {x<--(a), y<--(b c)}`
   2. Enter `bar {p<--(b c) }`
      3. Enter `foo {x<--(b c), y<--(c)}`
         4. Enter `bar {p<--(c)}`
            5. Enter `foo {x<--(c), y<--()}`
               6. Enter `bar {p<-()}`
               6. Return `bar` with `(c)`
            5. Return `foo` with `(c c)`
         4. Return `bar` with `(c c)`
      3. Return `foo` with `(b c c c)`
   2. Return `bar` with `(b c c c)`
1. Return `foo` with `(a b c c c)`

This won't work for Common Lisp!
Don't use free variables!

©1993 John Mylopoulos

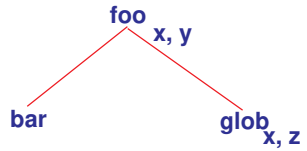
10

Contexts for Static Scoping

To keep track of the different scopes of variables, we now need to maintain a tree structure which is based on the static structure of a function definition.

```
(defun foo (x y) (...
        (defun bar () (...
            (put x) ...
            ...
        (defun glob (x z) ...
            (setf  x  7)
            (bar)
            (glob...)  (put x)       ...)

    (setf x  5)
    (glob)      ...)
```



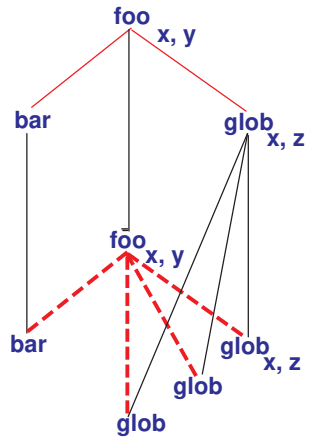
©1993 John Mylopoulos

11

Contexts for Static Scoping

• When a function is called, the corresponding node of the context tree is instantiated with an execution context.

• The value of a variable is determined by searching up the execution context tree until a value is found.



©1993 John Mylopoulos

12

Local Variables

Local variables in a function can be introduced through the macro `let`. The first argument of `let` is a list of variable-value pairs declaring local variables and their initial values, followed by a sequence of forms to be evaluated.

The general form of `let` is:

```
(let ((var1 val1) (var2 val2)
                   ...(varn valn))
                  exp1 exp2 ... expm )
```

The `let` macro actually turns this form into a lambda expression function call:

```
((lambda (var1 var2 ... varn)
            exp1 exp2 ... expm)
              val1 val2 ... valn )
```

For example the function `distance` finds the distance between two points `P1` and `P2`, where each is a two-element list giving the `X` and `Y` coordinates:

```
(defun  distance (P1 P2)
 (let ((xdiff (- (first P1)(first P2)))
      (ydiff (- (second P1)(second P2))))
      (sqrt (+ (* xdiff xdiff)
               (* ydiff ydiff)))   )  )
```


©1993 John Mylopoulos