Scope of Variables

- Consider function

  `(LAMBDA (y z) ⟨form⟩)`

  Within ⟨`form`⟩:

  “y” refers to 1^{st} arg

  “z” refers to 2^{nd} arg

- Consider *nested* functions

```
(LAMBDA (y z)
   (   (LAMBDA (x v) ⟨form⟩)
       (CDR y) 'A )))
```

  Variables usable within ⟨`form`⟩:

  “x” is bound to `CDR` of 1^{st} arg

  “v” is bound to value `A`

  “y” refers to 1^{st} arg

  “z” refers to 2^{nd} arg

Variables (con't)

- The function

```
(LAMBDA (y z)
   (  (LAMBDA (x y) (LIST x y z))
      'A (CDR y) ))
```

  takes 2 args, and

  evaluates to 3 element list:

  (A
    CDR of 1^{st} arg
    2^{nd} arg
)

- Notation:

  In inner λ-expr

  Variable "x" and "y" are BOUND

  Variable "z" is FREE

  y's value is "shadowed" by (CDR y)

Dealing with Free Variables

- Def'n:
  Formal variables of a function are
  bound within the function definition.
  All other variables are **free**.

- Consider function

  ```
  (DEFUN foo (z x) (LIST z x y))
  ```

  When (foo f_1 f_2) is called,
  z, x will be BOUND
  [to f_1 and f_2]
  y will be FREE

- Will (foo $\langle$f1$\rangle$ $\langle$f2$\rangle$) be defined?

Evaluating `foo`

- Case 1

  `( (LAMBDA (x y) (foo 'A (CDR x))) '(5) t)`

  Evaluates "`(foo 'A (CDR x))`" with
  `x` bound to `(5)`
  `y` bound to `t`

  Evaluates "`(LIST z x y)`" with
  `z` bound to `A`
  `x` bound to `()`
  `y` bound to `t`

  which returns: `(A () t)`

- Case 2

  `( (LAMBDA (x) (foo 'A (CDR x))) '(5) )`

  Evaluates `(foo 'A (CDR x))` with

  `x` bound to `(5)`

  Evaluates `(LIST z x y)` with
  `z` bound to `A`
  `x` bound to `()`
  `y` **undefined**

  which is *undefined*!!

Scope: Dynamic vs Static

- Dynamic Scoping:

  Value of variable depends on
  RUN-time situation!

  EG: *Lisp*

- Static Scoping

  Value of variable determined by
  COMPILE-time declaration.

  EG: *Pascal*, *Turing*, . . .

- Examples . . .

Example of Static Scoping

```
Foo
   x ← 5
      Bar
            ··· x ···
            put x;              {value is "5"}

      Glob
            x ← 7
            ··· Bar() ···        {prints 5}

            put x;       {prints 7}

   ... Glob( ) ...
```

Example of Dynamic Scoping

```
→ (SETQ x 20)
20
→ (SETQ y 10)
10
→ (DEFUN plusy (x) (+ x y))
plusy                   ;;; y's value found at run-time
→ (plusy 5)
15
→ (+ x y)
30
→ (SETQ y 20)
20
→ (plusy 5)
25
```

Contexts

- Identify each variable with a
  (LIFO) STACK of values.

- Variable's current "value" is top of stack

- Initializing/Updating Variable's Stack
  - Initally, each variable's stack is [`undefined`]
  - If `(SETQ a v)`,
    reset top of `a`'s stack to (value of) `v`.
  - When enterring function **`fn`**
    with args $\mathtt{a}_1, \ldots, \mathtt{a}_n$
    bound to values $\mathtt{v}_1, \ldots, \mathtt{v}_n$

    `PUSH` the value of $\mathtt{v}_i$ onto $\mathtt{a}_i$'s stack
    for each i
  - When exiting function,
    `POP` stack of each of function's variables
    [Here, each of $\mathtt{a}_i$.]

Maintaining Contexts

Initially, context is $\left\{\begin{array}{l}\texttt{x/[]}\\ \texttt{y/[]}\\ \texttt{z/[]}\end{array}\right\}$

To evaluate:

```
( (LAMBDA (x y) ((LAMBDA (z x) (LIST z x y))
                  'A    (CDR x)))
 '(A B C)  '(D E F) )
```

Enter outer λ:

(Context becomes $\left\{\begin{array}{l}\texttt{x/[(A B C)]}\\ \texttt{y/[(D E F)]}\\ \texttt{z/[ ]}\end{array}\right\}$)

Enter inner λ:

(Context becomes $\left\{\begin{array}{l}\texttt{x/[(B C) (A B C)]}\\ \texttt{y/[(D E F)]}\\ \texttt{z/[A]}\end{array}\right\}$)

Exit inner λ with

`(A (B C) (D E F))`

Exit outer λ with

`(A (B C) (D E F))`

Examples of Tracing

```
(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 $\{ \texttt{X} \leftarrow \texttt{(A)}, \texttt{Y} \leftarrow \texttt{(B C)} \}$
  2. enter BAR $\{ \texttt{P} \leftarrow \texttt{(B C)} \}$
    3. enter FOO $\{ \texttt{X} \leftarrow \texttt{(B C)}, \texttt{Y} \leftarrow \texttt{(C)} \}$
      4. enter BAR $\{ \texttt{P} \leftarrow \texttt{(C)} \}$
        5. enter FOO $\{ \texttt{X} \leftarrow \texttt{(C)}, \texttt{Y} \leftarrow \texttt{()} \}$
          6. enter BAR $\{ \texttt{P} \leftarrow \texttt{()} \}$
          6. return (C)
        5. return (C C)
      4. return (C C)
    3. return (B C C C)
  2. return (B C C C)
1. return (A B C C C)

Functional Arguments – Revisited

- Can take a function as argument

  treat it as an s-expr

  "apply" it

- References:

  Ex: p60

  Main application: Mapping Functions

  p61,63

- Dynamic vs Static Scoping

  `QUOTE` vs `FUNCTION`

Mapping Functions

- Apply function to each element of list.

- Eg 1: Add 1 to each element:

```
(DEFUN plus1 (l)
   (COND   ((NULL l) nil )
           ( t        (CONS ( 1+  (CAR l))
                            (plus1 (CDR l))) )))
```

Function `1+` is successor

→ <u>`(1+ 5)`</u>
6

→ <u>`(1+ 19)`</u>
20

→ <u>`(plus1 '(3 5 0))`</u>
`(4 6 1)`

- Eg 2: Take `CAR` of each element:

```
(DEFUN CarAll (l)
   (COND   ((NULL l) nil )
           ( t        (CONS ( CAR  (CAR l))
                            (CarAll (CDR l))) )))
```

→ <u>`(CarAll '((A B) (C D E) (t) (5 A)))`</u>

`(A C t 5)`

Mapping Function – MAPCAR

`Mapcar` of list `l` and function `fn`
applies `fn` to each element in `l`.

List	Function	result
(3 5 0)	1+	(4 6 1)
((4) (t Q))	CAR	(4 t)
((4) (t Q))	CDR	(() (Q))
((4) (t))	LISTP	(t t)
(A B (C D))	ATOM	(t t nil)
()	ATOM	()
(A B C)	(LAMBDA (x) (CONS x '(t)))	
		((A t) (B t) (C t))

```
(DEFUN MAPCAR (l fn)
   (COND  ((NULL l) nil )
          ( t        (CONS ( fn  (CAR l))
                                 (MAPCAR (CDR l) fn))
 )))
```

Mapping Function – AnyOf

`AnyOf` of list `x` and function `fn` determines if any element of `x` satisfies predicate `fn`.

Function	List	result
ATOM	(A B (C D))	t
ATOM	((4) (t Q))	nil
LISTP	((4) (t Q))	t
CAR	((4) (t))	t
CDR	((4) (t))	nil
ATOM	()	nil
(LAMBDA (y) (EQ y 'A))	(B A C)	t

```
(DEFUN AnyOf (fn x)
   (COND  ((NULL x)        nil )
          (( fn  (CAR x)) t )
          ( t              (AnyOf fn (CDR x)))  ))
```

Mapping Functions

- Apply function to each [element | sublist] of list, returning list of values.

  `MapCar` applies function to each element of list, returning list of values.

  `MapList` — like `MAPCAR`, but uses successive SUBLISTS (not elements)

  `MapCan`, `MapCon` ... destructive ...

- Apply function to each [element | sublist] of list, returning `nil`.
  (used for side effect – eg printing values)

  `MapC` — like `MapCar`, but returns `nil`.

  `MapL` — like `MapList`, but returns `nil`.

- "Boolean" Functions

  `ANYOF` determines if *any* element satisfies predicate.

  `ALLOF` determines if *all* elements satisfy predicate.

Function Argument Problem

- Using functions with free variables
  can cause problems.

  [Partly due to Dynamic Scoping.]

- Consider:

```
(DEFUN memq (at l)
    (AnyOf '(LAMBDA (i) (EQ i at))
              l ))
```

  which return `t` if
  `at` is an element of `l`.

- . . . not always . . .

- Notice: `at` is
  Free Variable within inner λ-expr.

Tracing MEMQ

$\rightarrow$ (memq 'a '(b a c))

Enter memq $\{$ at $\leftarrow$ a, l $\leftarrow$ (b a c) $\}$

Enter AnyOf $\left\{ \begin{array}{l} \texttt{fn} \leftarrow \texttt{(LAMBDA (i) (EQ i at))} \\ \texttt{x} \leftarrow \texttt{(b a c)} \end{array} \right\}$

Enter λ (fn) $\{$ i $\leftarrow$ b $\}$

EVAL (EQ i at) $\left\{ \begin{array}{l} \texttt{i} \leftarrow \texttt{b} \\ \texttt{at} \leftarrow \texttt{a} \end{array} \right\}$ $\leadsto$ nil

Exit λ (fn) nil

Enter λ (fn) $\{$ i $\leftarrow$ a $\}$

EVAL (EQ i at) $\left\{ \begin{array}{l} \texttt{i} \leftarrow \texttt{a} \\ \texttt{at} \leftarrow \texttt{a} \end{array} \right\}$ $\leadsto$ t

Exit λ (fn): t

Exit AnyOf: t

Exit memq: t

t

Same MEMQ?

(DEFUN memq ($\boxed{\text{x}}$ l)
    (AnyOf '(LAMBDA (i) (EQ i $\boxed{\text{x}}$))
        l))

$\rightarrow$ (memq 'a '(b a c))

Enter memq { x $\leftarrow$ a, l $\leftarrow$ (b a c) }

Enter AnyOf { fn $\leftarrow$ (LAMBDA (i) (EQ i x)),
$\boxed{\text{x} \leftarrow \text{(b a c)}}$ }

Enter λ { i $\leftarrow$ b }

EVAL (EQ i $\boxed{\text{x}}$) $\left\{ \begin{array}{l} \text{i} \leftarrow \text{b} \\ \text{x} \leftarrow \text{(b a c)} \end{array} \right\}$

$\leadsto$ ERROR, as x is (b a c)

"FunArg" Problem

- As `AnyOf` uses variable, `x`,

  ```
  (LAMBDA (at L) (AnyOf L '(LAMBDA (i) (EQ i at)) )
  ```

  and

  ```
  (LAMBDA ( x  L) (AnyOf L '(LAMBDA (i) (EQ i  x )) )
  ```

  are different functions!

- Here, want variable `x` evaluated
  STATICALLY (based on program definition)
  rather than
  DYNAMICALLY (based on run-time environ.)

- *Lisp* usually evaluates free variables
  DYNAMICALLY.

  To get STATIC evaluation
  use new special form: `FUNCTION`

`QUOTE` is for Dynamic Scoping

- Dynamic Scoping

```
(DEFUN memq1 (x l)
   (AnyOf  ( QUOTE  (LAMBDA (i) (EQ i x)))
           l ))
```

- Free variables in

  (`QUOTE` (`LAMBDA` (a_1 $\cdots$ a_n) $\langle$`form`$\rangle$))

  evaluated DYNAMICALLY

  Hence: value of `x` in `memq1`'s

  `(QUOTE (LAMBDA (i) (EQ i x)))`

  is value of `AnyOf`'s 2^{nd} arg.

  $\rightsquigarrow$ FunArg problem!

`FUNCTION` is for Static Scoping

- Static Scoping

```
(DEFUN memq2 (x l)
   (AnyOf (FUNCTION (LAMBDA (i) (EQ i x)))
          l ))
```

- Free variables in

  (`FUNCTION` (LAMBDA (a_1 $\cdots$ a_n) $\langle$form$\rangle$))

  evaluated STATICALLY

  Hence: value of `x` in `memq2`'s

  `(FUNCTION (LAMBDA (i) (EQ i` $\boxed{\texttt{x}}$`)))`

  is value of `memq2`'s 1^{st} arg.

  $\rightsquigarrow$ "correct" answer!

`FUNCTION` Special Form

- `FUNCTION` behaves exactly like `QUOTE` except wrt evaluation of free variables:

- `FUNCTION` $\approx$ STATIC EVALUATION

  [based on (compile-time) function definition]

- `QUOTE` $\approx$ DYNAMIC EVALUATION

  [based on current (run-time) context]

- *Lisp*'s Compiler can compile

  ```
  (function (LAMBDA (...)  ...))
  ```