F.G.H.
6/6/85

LispCourse #34: Variable Binding and the Interlisp Stack

Variable Reference Inside Functions: Bound and Free Variables

Consider the following function definition from the solution to Homework #30:

```
(DEFINEQ
   (LC.ParseNameString
        (LAMBDA (String)
              (SETQ String (CONCAT String))
              (SETQ Comma (STRPOS "," String))
              (SETQ Space (STRPOS " " String Comma))
              (create LC.Name
                      Last _
                           (MKATOM
                             (SUBSTRING String 1 (SUB1 Comma)))
                      First _
                           (MKATOM
                             (SUBSTRING String (ADD1 Comma ) (SUB1 Space)))
                      Middle _
                           (MKATOM
                             (SUBSTRING String (ADD1 Space))))
```

In this function, there are three variables: *String*, *Comma*, and *Space*.

String differs from *Comma* and *Space* in that before the function is entered during a function call evaluation, the value of *String* is temporarily set (i.e., ***bound***) to the value of the argument in the function call.

Moreover, the old value of *String* is reset to its previous value when the function is exited, despite the new value assigned to *String* by the SETQ in the function.

In contrast, *Comma* and *Space* have unknown values when the function is entered and the SETQ operations on *Comma* and *Space* permanently change the value of these variables.

String is known as a ***bound variable*** within the function LC.ParseNameString.

A bound variable is one whose value is set when the function is entered and reset to the previous value when the function is exited.

In particular, a bound variable has the following property: You can change the name of the variable, and the operation of the function will not change.

Substituting *NameString* for *String* throughout the definition of LC.ParseNameString would not change how the function worked.

Comma and *Space* are known as ***free variables*** within the function LC.ParseNameString.

A variable is a free variable in a function definition if it is not bound within that function definition.

The value of a free variable when the function is entered cannot be specified *a priori*. The free variable may have a value or it may not.

Moreover, when the function is exited, the free variable is not reset to its previous value.

In particular, a free variable has the following property: If you change the name of the variable, and the operation of the function may change, depending on the context of the evaluation.

For example, if the variables *ListSize* and *StringSize* were used by the Lisp Exec to hold important information, then changing *Comma* to *ListSize* and *Space* to *StringSize* in the definition of LC.ParseNameString might have serious side-effects on the Lisp Exec, since evaluating a call to LC.ParseNameString would change the value of these variables.

The problem with free variables in a function definition is that there is some ambiguity as to what is being refered to by the free variable.   Interpreting the free variable reference depends on some context outside of the function itself.

Consider the following example:

```
1_ (DEFINEQ
        (LC.FindComma
                    (LAMBDA (String SearchLetter)
                            (SETQ SearchLetter ",")
                            (LC.FindLetter String)))
            (LC.FindLetter
                    (LAMBDA (String)
                            (STRPOS SearchLetter String))))
```

(LC.FindComma LC.FindLetter)

```
2_ (SETQ SearchLetter "+")
```

"+"

```
3_ (LC.FindLetter "AB,CD+EF")
```

6

```
4_(LC.FindComma "AB,CD+EF")
```

????

Problem: What is the value of this function call?

There are two possible answers, depending on how the free variable *SearchLetter* is resolved in the call to *LC.FindLetter*.

1. If *SearchLetter* in LC.FindLetter refers to the global value in the Exec environment, then the result would be 6 since LC.FindLetter would be searching for a "+" as determined by event 2.

2. If *SearchLetter* in LC.FindLetter refers to the most recent binding of *SearchLetter* anywhere, then the result would be 3 since LC.FindLetter would be searching for a "," as determined by the binding and SETQ statements in LC.FindComma.

In fact, in Interlisp the value returned by (LC.FindComma "AB,CD+EF") in event 4 would be 3, because a free variable reference in a function always references the most recent binding of that variable.

The following several sections, explain in detail how Interlisp handles variable references, both free and bound, within function definitions.

The main point of these sections is that the model of variables we have been using until now is far too simple.

> Until now, we have assumed that the value of a variable in a function definition is the value of the atom with the same name as the varaible.
>
> This is true "at the top level", i.e. for variable references typed directly to the Lisp Exec.
>
> However, for variable references within a function, Interlisp uses a much more complex scheme to set and determine the value of an variable.

Review: Evaluating S-expressions using EVAL and APPLY

Recall that all of the significant work in Lisp is done by the Lisp evaluator during the EVAL part of the read-EVAL-print loop.

Recall also that the Lisp evaluator is built around two functions: ***EVAL*** and ***APPLY***.

Hence, understanding Lisp function evaluation requires understanding the two functions EVAL and APPLY.

EVAL

The following defines a function that could be used by the Lisp evaluator to evaluate arbitrary S-expressions (i.e., EVAL):

```
(DEFINEQ
  (EVAL
        (LAMBDA (SExpr)
          (COND
                ((NLISTP SExpr) (LookUpValue SExpr))
                (T
                  (APPLY (CAR SExpr)
                        (FOR Arg IN (CDR SExpr)
                              COLLECT (EVAL Arg)))
```

Note: The function ***LookUpValue*** is some mysterious function that can look up the value of atoms, arrays, etc. in the Lisp environment.

In natural language, EVAL does the following:

If *SExpr* is not a list, then ***look up the value*** of *SExpr* and return it.

If *SExpr* is a list, then ***APPLY*** the function named by the first element (i.e., CAR) of *SExpr* to the list obtained by collecting the evaluation each item in the rest (i.e., CDR) of *SExpr*.

Aside from *LookUpValue* (which will remain mysterious for a while), the central function used in defining EVAL is the function ***APPLY***.

APPLY

APPLY could be defined as follows:

```
(DEFINEQ
  (APPLY
        (LAMBDA (Function Arguments)
                (FOR Parameter IN (CADR (GetFunctionDefn Function))
                        AS Argument IN Arguments
                        DO (Bind Parameter Argument))
                (FOR SExpr IN (CDDR (GetFunctionDefn Function))
                        AS Ctr FROM 1 TO
                                (DIFFERENCE
                                        (LENGTH (GetFunctionDefn
                                        Function)) 3)
                        DO (EVAL SExpr))
                (SETQ Result
                    (EVAL (CAR (LAST (GetFunctionDefn Function)))))
                (FOR Parameter IN (CADR (GetFunctionDefn Function))
                        DO (Unbind Parameter))
                Result)))
```

Note: The functions ***GetFunctionDefn***, ***Bind***, and ***Unbind*** are some mysterious functions that can look up the defintion of a function, bind a parameter to a value, and unbind a parameter, respectively.

In natural language, APPLY does the following:

For each parameter in the parameters list in the function definition of *Function*, ***bind*** that parameter to the corresponding argument in the *Arguments* list.

Then, for each S-expression in the body of the function definition except for the last, EVAL the S-expression.

Then, EVAL the last S-expression in the body of the function definition and hold on to the resulting value.

Then, for each parameter in the parameters list in the function definition of *Function*, ***unbind*** that parameter.

Finally, return the result already derived by EVALing the last S-expression.

EXAMPLE (from LispCourse #3, page 6)

```
(DEFINEQ
    (MOVEFILE
        (LAMBDA (FromFile ToFile)
            (COPYFILE FromFile ToFile)
            (DELFILE FromFile)
            (QUOTE AllDone))))
```

Evaluating ***(MOVEFILE ’OLD.LISP ’NEW.LISP)*** proceeds as follows:

EVAL:

1. ’OLD.LISP evaluates to OLD.LISP
2. ’NEW.LISP evaluates to NEW.LISP
3. APPLY the function MOVEFILE to (OLD.LISP NEW.LISP)

APPLY:

1. FromFile is bound to OLD.LISP
2. ToFile is bound to NEW.LISP
3. (COPYFILE FromFile ToFile) is evaluated using current bindings of FromFile and ToFile (i.e., FromFile will evaluate to OLD.LISP and ToFile will evaluate to NEW.LISP).
4. (DELFILE FromFile) is evaluated similarly.
5. (QUOTE AllDone) is evaluated to AllDone.
6. FromFile and ToFile are reset to their previous values (if any).
7. APPLY returns AllDone.

Missing Pieces

The foregoing explanation references several functions that have not yet been explained.

In particular, ***LookUpValue***, ***Bind***, ***Unbind***, and ***GetFunctionDefn*** are critical functions used by EVAL and/or APPLY in the process of evaluating S-expressions.

The following sections explain the operation of these functions in the Interlisp-D evaluator.

The Lisp Stack, Variable Binding, Variable Lookup, and Related Topics

In the APPLY phase of evaluating a Lisp function call, the first step is to ***bind*** the parameters of the function to the corresponding arguments in the function call.

In LispCourse #3 (page 6), binding a parameter to an argument was described as "temporarily SETQing the parameter to the argument value". The implication was that binding simply temporarily resets the value assigned to the atom of the same name as the parameter.

In actuality, binding is much more complex and relies on a special data structure called the ***stack***.

The Lisp Stack

The stack is a data structure that contains a variable number of ***stack frames*** arranged in an ordered one-dimensional vector.

Each stack frame contains a variable number of ***binding records***.

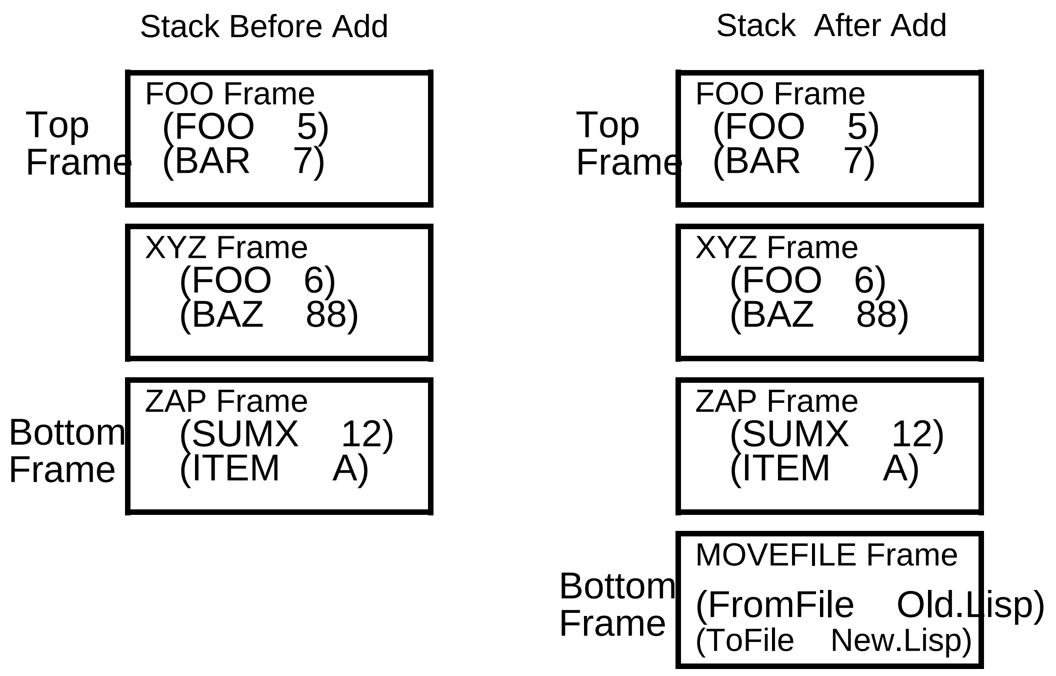
Each binding record is a pair consisting of a *variable name* (i.e., a litatom) and a *value*.

The stack has a ***top*** and a ***bottom***.

Stack frames can only be added to the **bottom** of the stack.

Stack frames can only be removed from the **bottom** of the stack.

Adding a stack frame to the bottom of the stack



The stack implements a *first-in/last-out* access scheme.

If we add Frame X to the bottom of the stack and subsequently add N more frames to the bottom of the stack, then we have to remove the last N frames before we can remove Frame X.

The analogy is to a stack of boxes. To remove the Nth box down in the stack, you have to first remove the N-1 boxes sitting on top of that box.

(Contrast the stack with the queue data structure in Homework #32, which implemented a *first-in/first-out* access scheme.)

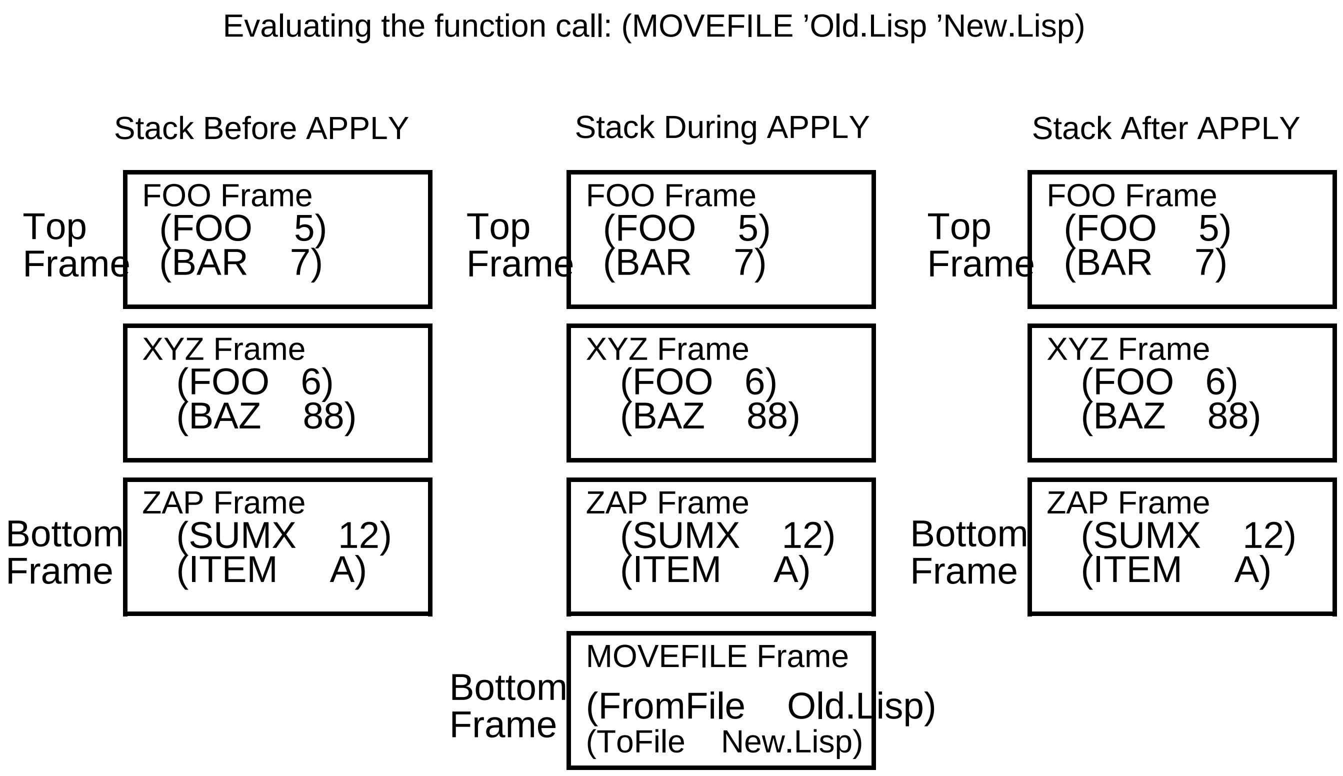
Variable Binding

Every time the Lisp evaluator evaluates a new function call, APPLY creates a new stack frame and adds it to the bottom of the Interlisp stack.

When APPLY **binds** the parameters of the function to the arguments of the function call, it simply adds a binding record to this stack frame for each parameter.

When APPLY **unbinds** the parameters at the end, it simply removes this stack frame from the stack.

Example:



Looking up the Value of a Variable

After binding all of the parameters, APPLY calls EVAL on each S-expression in the function body. These S-expression often refer to variables.

Example: Inside the body of MOVEFILE is the S-expression ***(COPYFILE FromFile ToFile)***, which refers to 2 variables *FromFile* and *ToFile*.

EVALualting these S-expressions involves EVALuating these variables.

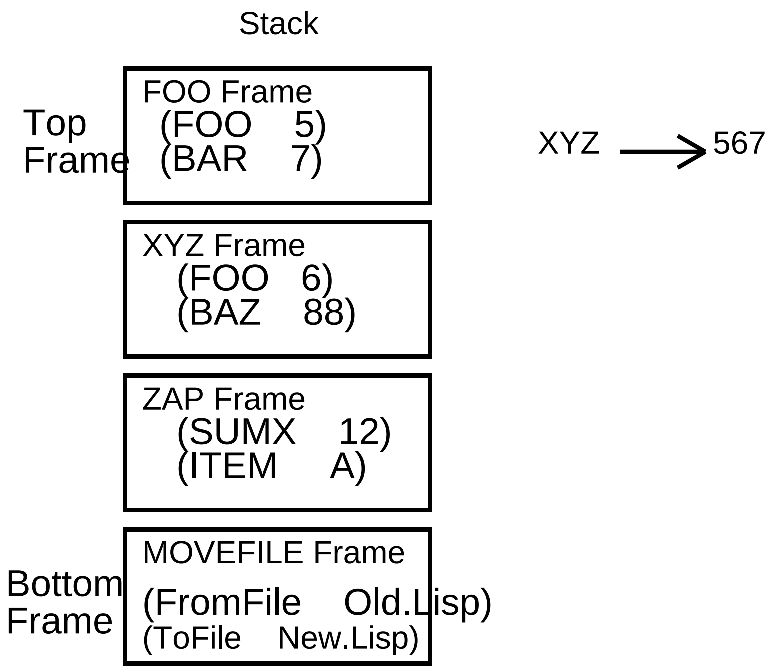
Looking at EVAL, evaluating a variable (i.e., a NLISTP) involves a function, ***LookUpValue***, that looks up the value of the variable.

The ***LookUpValue*** function works as follows:

> Starting at the bottom of the stack, search each stack frame for a binding record with a variable name EQ to the variable being looked up.
>
> Return the value associated with the FIRST such binding record found on the stack.
>
> If there is no binding record on the stack, then get the value attached to the atom with the same name as the variable being looked up.
>
> If the atom has no value, then break with an "unbound atom" error.

Examples:

Given the following stack and atom values:



(LookUpValue 'FromFile) would return *Old.Lisp*

(LookUpValue 'FOO) would return *6*

(LookUpValue 'BAR) would return *7*

(LookUpValue 'XYZ) would return *567*

(LookUpValue 'PRQ) would return *u.b.a. error*

Setting the Value of a Variable

Within a function body being processed by APPLY, the SET functions work in a manner analogous to variable lookup.

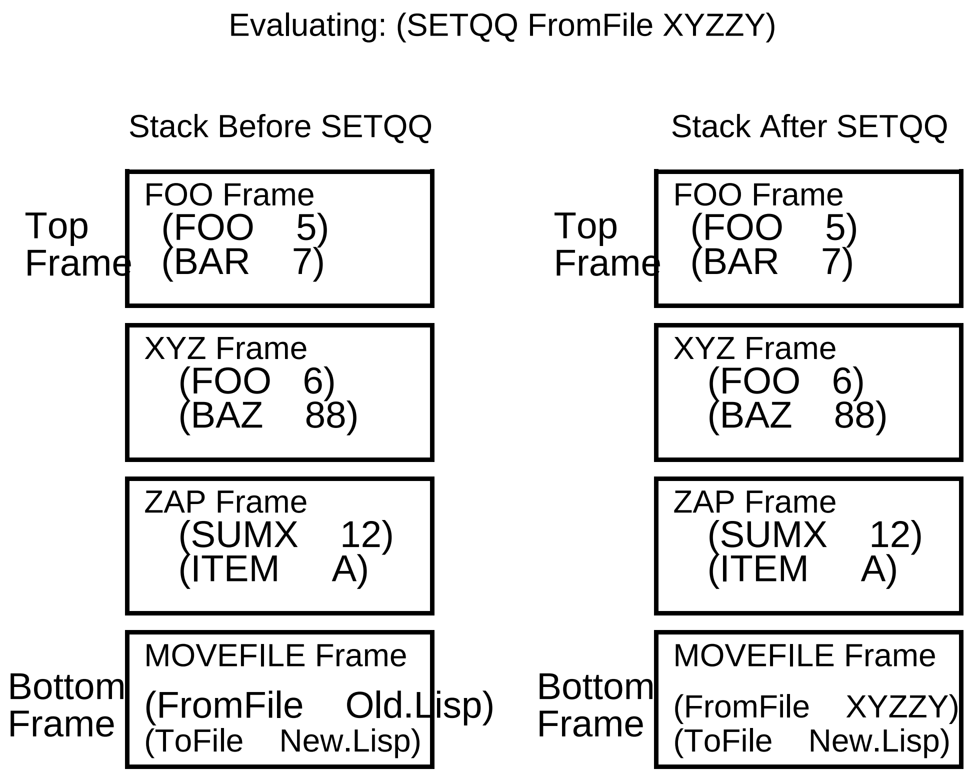
In particular, **(SETQQ *Variable Value*)** will search up the stack starting at the bottom for a binding record for the variable *Variable*. If it finds one, it will change the value portion of this binding record to be *Value.*

If no binding record is found on the stack, then SETQQ will set the value of the atom *Variable*.

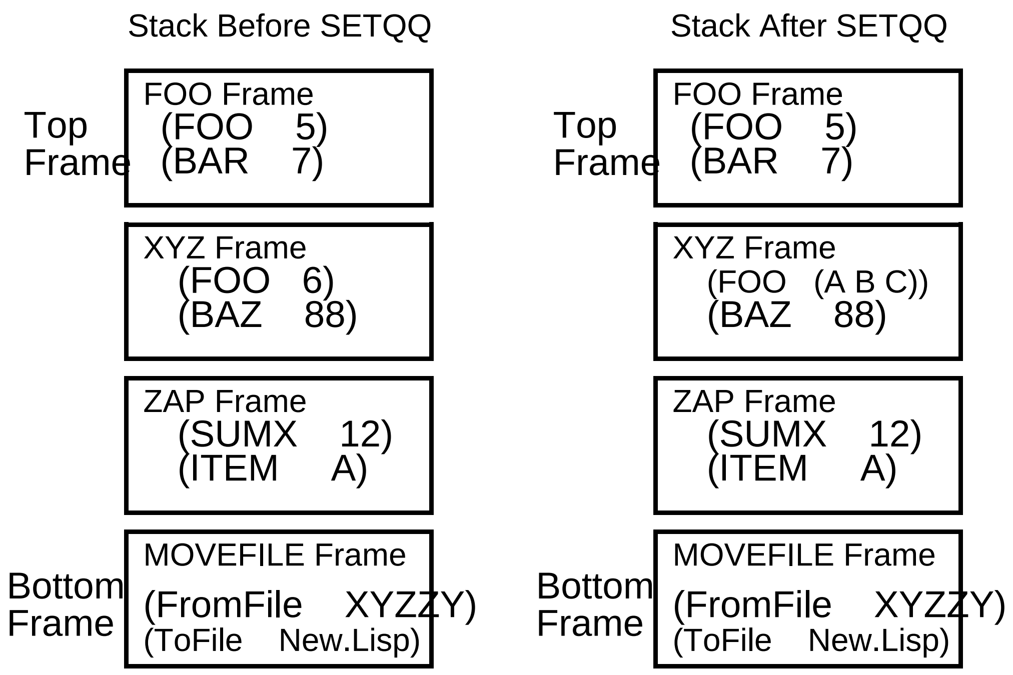
SET and SETQ work analogously.

Examples of variable setting:

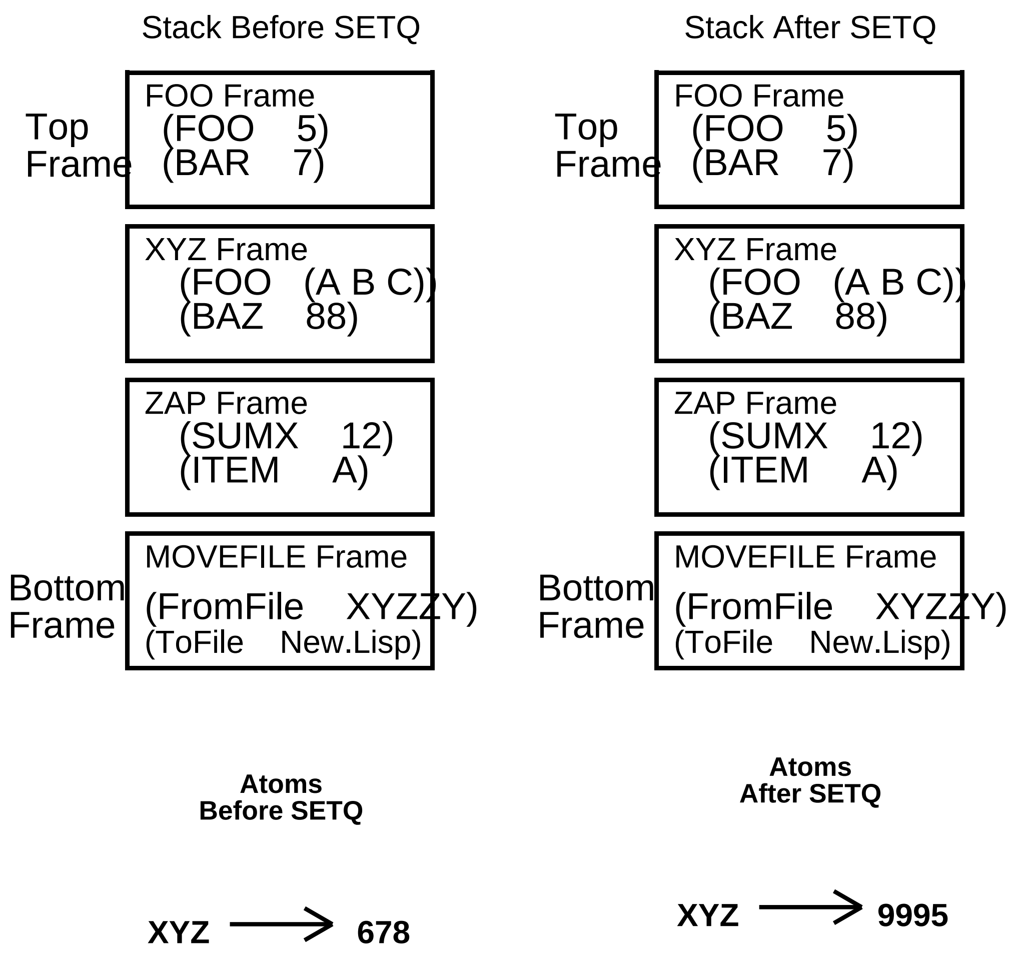
All of the following assume that the SET statement is part of the MOVEFILE function definition.



Evaluating: (SETQQ FOO (A B C))



Evaluating: (SETQ XYZ 9995)



Function Definition Lookup

There is no such thing as binding a function name in Interlisp.

All function names in Interlisp refer to the function definition attached to the atom of the same name.

The function ***GetFunctionDefn*** in the definition of APPLY simply accesses the function definition for the named atom.

In particular, it does not search the stack for new bindings of the function name.

Note: This is different in other dialects of Lisp. Some Lisps allow binding of function names as well as variables names on the stack.

Variable Reference in Practice: LET and PROG are used to bind variables

Avoid Free Variables!

In practice the rule is: *Bound variables are good, free variables are bad.*

The effect of evaluating a function with bound variables is always predictable.

In contrast, evaluating a function that uses free variables can lead to differing results, depending on the context of the evaluation.

Examples:

Interference between variable names

```
1_ (DEFINEQ
        (Circumference (LAMBDA (Radius)
                (TIMES 2 PI Radius))))
(Circumference)
2_ (SETQ PI 3.1416)
3.1416
3_ (DEFINEQ
        (MakeProgrammersInterface (LAMBDA NIL
                (SETQ PI (CREATE ProgInt ....)))))
(MakeProgrammersInterface)
4_ (MakeProgrammersInterface)
{ProgInt}#32,33412
5_ (Circumference 5)
NON-NUMBERIC ARG
{ProgInt}#32,33412
```

Inadvertantly altering system parameters

```
6_ (DEFINEQ
       (FontNameToFontNumber (LAMBDA (FontName)
           (SETQ DEFAULTFONT 1)
           (FOR Name in ’(Helvetica TimesRoman
           Gacha Modern)
               AS Number FROM 1
               WHEN (EQ FontName Name)
               DO (SETQ DEFAULTFONT
               Number))
           DEFAULTFONT)
(FontNameToFontNumber )
7_ (FontNameToFontNumber ’Gacha)
3
8_ (TEDIT "XXXXXX")
ILLEGAL ARG
{FONTCLASS}#71,10500
       (because DEFAULTFONT has been reset to an
       illegal value)
```

The Funarg Problem

```
9_ (DEFINEQ
       (Sum (LAMBDA (List Transform)
           (FOR N IN List SUM (APPLY* Transform
           N)))))
(Sum)
10_(DEFINEQ
       (SumSquares#1 (LAMBDA (List)
           (Sum List (FUNCTION SQUARE))))
       (SumSquares#2 (LAMBDA (List)
           (SETQ N 2)
           (Sum List (FUNCTION NthPower))))
```

```
        (SumCubes (LAMBDA (List)
            (SETQ N 3)
            (Sum List (FUNCTION NthPower))))
        (NthPower (LAMBDA (X)
            (EXPT X N))))
    (SumSquares#1 SumSquares#2 SumCubes NthPower)

    11_ (SumSquares#1 (LIST 1 2 3 4 5))

    55

    12_ (SumSquares#2 (LIST 1 2 3 4 5))

    3413

    13_ (SumCubes (LIST 1 2 3 4 5))

    3413
```

The examples illustrate the problems involved in using free variables in defining functions.   Because of these problems, it is best to avoid using free variables.

The LET Special Form

Consider the following function that computes the average of all the numbers in a list of numbers and litatoms:

```
(DEFINEQ
    (AverageOfNumbers (List)
        (SETQ Sum 0.0)
        (SETQ N 0)
        (FOR Item IN List
            WHEN (NUMBERP Item)
            DO
                (SETQ Sum (PLUS Sum Item))
                (SETQ N (ADD1 N)))
        (COND
            ((NOT (ZEROP N))
                (QUOTIENT Sum N))
(T NIL)))))
```

It would not be possible to write this function without the **Sum** and **N** variables, since they are used to store intermediate results as the function iterates through the list.

> There is no reason, however, for **Sum** and **N** to be free variables. Their function should be limited to the scope of the AverageOfNumbers function body.
>
> On the other hand, **Sum** and **N** are not a parameters either and therefore should not be made into a bound variables by placing them in the parameter list of the function.

The ***LET*** special form provides the means to bind variables without making them part of the parameter list.

LET has the format:

> **(LET *BindingList S-Expression1 S-Expression2* ...)**
>
> > ***BindingList*** is a list of variable-value pairs, i.e., **(*VariableName InitialValue*).** A variable can also be expressed by just its ***VariableName***, which is equivalent to the list **(*VariableName* NIL).**
> >
> > The ***S-Expressioni*** are arbitrary Lisp S-expressions to be evaluated.
>
> LET works as follows:
>
> > The variables specified in the binding list are bound on the stack and set to the specified initial values**.**
> >
> > The S-expressions are evaluated in order.
> >
> > The LET form returns the value of the last S-expression, unbinding the variables in the binding list before it exits.

Variables in the binding list are bound "in parallel" and thus the order of mention in the binding list is unimportant.

If the binding list is *((X 55) (Y X))*, the value of *X* within the LET will be 55 and the value of *Y* will be whatever was the value of *X* before the LET (and **not** 55 as might be expected).

The same effect could have been achieved by the binding list *((Y X) (X 55))*.

The proper defintion for AverageOfList would thus be:

```
(DEFINEQ
    (AverageOfList (List)
        (LET ((Sum 0.0)(N 0))
            (FOR Item IN List
                WHEN (NUMBERP Item)
                DO
                    (SETQ Sum (PLUS Sum
                    Item))
                    (SETQ N (ADD1 N)))
            (COND
                ((NOT (ZEROP N))
                    (QUOTIENT Sum N))
                (T NIL)))))
```

In this definition, List, **Sum**, and **N** are all bound variables.

When the function is enetered, List is bound since it is in the parameter list.

When the LET is entered, **Sum** and **N** are bound on the stack and set to their initial values of zero.

When the LET is exited, **Sum** and **N** are unbound.

When the function is exited, List is unbound.

As a second example, consider a function (slightly modified) from the solution to Homework#32:

```
(DEFINEQ
        (LC.PrintQueue
           (LAMBDA (Q)
                 (COND
                        ((NOT (LC.QueueEmptyP Q))
                           (SETQ **NextPtr** (fetch (LC.Queue Head)
                        of Q))
                           (PRINT (CAR **NextPtr**))
                           (until (EQ
                                        (SETQ **NextPtr** (CDR
                                      **NextPtr**))
                                        (fetch (LC.Queue Tail) of
                                      Q)
                              DO (PRIN1 (CAR **NextPtr**)))
                          (TERPRI))))
```

Note the unnecessary use of **NextPtr** as free variable.

The proper defintion for LC.PrintQueue is:

```
(DEFINEQ
        (LC.PrintQueue
           (LAMBDA (Q)
                 (LET (**NextPtr**)
                    (COND
                        ((NOT (LC.QueueEmptyP Q))
                           (SETQ **NextPtr** (fetch (LC.Queue Head)
                        of Q))
                           (PRINT (CAR **NextPtr**))
                           (until (EQ
                                        (SETQ **NextPtr** (CDR
                                      **NextPtr**))
```

```
                                    (fetch (LC.Queue Tail) of
                                   Q)
                              DO (PRIN1 (CAR NextPtr)))
                           (TERPRI)))))
```

In this definition, **NextPtr** is bound within the context of the LET statement (and hence within the entire function body).

The PROG Special Form

The ***PROG*** special form is very much like the LET special form, but it allows a little more flexibility in how and when the special form is exited.

PROG has the format:

(PROG *BindingList S-Expression1 S-Expression2* ...)

BindingList is as in the LET special form.

The ***S-Expressioni*** are arbitrary Lisp S-expressions to be evaluated. One or more of these S-expressions may contain a sub-expression of the form **(RETURN *S-expression*)**

PROG works as follows:

The variables specified in the binding list are bound on the stack and set to the specified initial values**.**

The S-expressions are evaluated in order until an expression (or sub-expression) of the form **(RETURN *S-expression*)** is evaluated.

Evaluating this RETURN expression evaluates the embedded *S-expression*, then causes the PROG special form to be exited, returning the value of this evaluation. On exit, the variables in the binding list are unbound.

If no RETURN statement is encountered while evaluating the S-expressions, PROG unbinds the variables in the binding list and returns NIL.

As in LET, variables in the PROG binding list are bound "in parallel" and thus the order of mention in the binding list is unimportant.

PROG should be used where one might want to exit at one of several places in a function, depending on certain conditions.

For example, the following is a function that transfers a list of numbers to an array and then places the sum of the numbers in the last cell of the array. The program immediately exits with NIL if the Array is not on larger than the length of the list.

```
(DEFINEQ
    (TransferListToArray (LAMBDA (List Array)
        (PROG
            ((LstLen (LENGTH List))
              (ArrSize (ARRAYSIZE Array))
              (Sum 0.0))
            (COND
                ((OR
                    (ZEROP LstLen))
                    (NEQ LstLen (SUB1
                    ArrSize)))
                 (RETURN NIL))
            (FOR Item IN List
                AS Index FROM 1
                DO
                    (ELT Array Index
                    Item)
                    (SETQ Sum (PLUS
                    Sum Item)))
            (ELT Array ArrSize Sum)
            (RETURN Sum))
```

FOR and WHILE are implemented using PROG

The FOR and WHILE clisp forms are implemented using the PROG special form.

An important side-effect of this is that the (RETURN S-expr) form can be used to exit from FOR or WHILE loops before their normal termination.

For example, the following will sum the item in a list until a negative number is reached.

```
(DEFINEQ
    (SumTillMinus (LAMBDA (List)
        (PROG ((Sum 0.0))
            (FOR Item IN List
            DO  (COND
                    ((MINUSP
                    Item)(RETURN
                    NIL)))
                (SETQ Sum (PLUS Sum
                Item)))
            (RETURN Sum)))))
```

In this example, the RETURN inside the FOR loop will exit only out of the PROG implicit in the FOR. It will not exit out of the PROG that contains the FOR loop.

This is true for all RETURN statements: each RETURN exits out only the lowest level enclosing PROG and not out of any PROGs that in turn enclose the lowest level PROG.

Note also that the previous example, would probably best be written using a LET as follows:

```
(DEFINEQ
    (SumTilLMinus (LAMBDA (List)
                (LET ((Sum 0.0))
                    (FOR Item IN List
                    DO  (COND
                                ((MINUSP
                                Item)(RETURN
                                NIL)))
                            (SETQ Sum (PLUS Sum
                            Item)))
                    Sum))))
```

As another example, the following function takes a list of numbers and returns a list of as many of the initial numbers as necessary to sum to just of 100. If any of the items in the initial list is non-numeric the function exits and returns the bad item.

```
(DEFINEQ
    (FirstHundred (LAMBDA (List)
        (LET ((Sum 0.0))
            (WHILE (LESSP Sum 100)
                FOR Item IN List
                COLLECT
                    (COND
                        ((NOT (NUMBERP
                        Item))
                          (RETURN Item)))
                    (SETQ Sum (PLUS Sum
                    Item))
                    Item)))))
```

Note: The RETURN statement inside of a COLLECT loop will cause the COLLECT loop to return with the value of the S-expr in the RETURN clause instead of the list being COLLECTed.

```
3_ (FirstHundred (LIST 1 2 3 44 55 66 77 88))
```

(1 2 3 44 55)

```
4_ (FirstHundred (LIST 1 2 'A 44 55 66 77))
```

A

A

Global Variables: Free variables used as system or package parameters

Free variables are sometimes necessary.

In particular, system or package parameters (e.g., DEFAULTPRINTINGHOST, CHAT.FONT, DEFAULTFONT, LAFITEDEFAULTHPOST&DIR, etc.) are set outside of any function (e.g., in an Init file or in the Lisp Exec) but must be used (and sometinmes set) within various system or user functions.

These system/package parameters must be globally accessible, i.e., avaialable to all functions in all contexts.

Therefore, they cannot be bound on the stack inside of some particular function.

When setting or retrieving the value of one of these global parameters, you want to go directly to the value attached to the atom, skipping the search up the stack for other bindings.

For example, when a function definition includes the form *(SEND.FILE.TO.PRINTER '{DSK}FOO (CAR DEFAULTPRINTINGHOST)),* the *DEFAULTPRINTINGHOST* variable should reference the global value of this variable. If one of the calling functions of this function has stupidly rebound *DEFAULTPRINTINGHOST*, the rebinding should probably be ignored.

The functions ***GETTOPVAL*** and ***SETTOPVAL*** can be used to directly access the global value of a variable (i.e., the value attached to the atom) skipping the search for rebindings on the stack.

GETTOPVAL takes a single LITATOM as an argument and returns the value attached to that LITATOM.

SETTOPVAL takes a LITATOM and a value and sets the value of the LITATOM to be value, returning the value.

Example:

```
1_ (SETQ LineLength 107)
```

```
107
2_ (DEFINEQ
        (Tester
                (LAMBDA NIL
                        (PROG ((LineLength 223))
                                (PRINT
                                        (CONCAT "Initial values: " LineLength "   "
                                        (GETTOPVAL 'LineLength)))
                                (SETQ LineLength 55)
                                (SETTOPVAL 'LineLength 88)
                                (PRINT
                                        (CONCAT "LineLength: " LineLength "
                                        TopVal of LineLength: " (GETTOPVAL
                                        'LineLength)))))))
(Tester)
3_ (Tester)
Initial values: 223   107
LineLength: 55     TopVal of LineLength: 88
NIL
```

References

In the IRM:

PROG is on Page 4.3. LET is not in the IRM, but is essentially the same as PROG, except for the RETURN and GO features.

GETTOPVAL and SETTOPVAL are on page 2.5

Free variables and binding are covered in both Winston & Horn and Touretzky. But be careful. CommonLisp uses a different sort of binding scheme than does Interlisp, so not everything said in these books applies to Interlisp.

Look at Chapter 5 of Touretzky and page 53 of Winston and Horn.

Also LET is explained starting on page 57 of W&H and on page 250 of T.

Exercise

Overall Task: Write a simple Lisp evaluator.

For each of the functions you write, put a print statement at the ned of the function that prints on the screen some information about what the function just did. This way, you can watch you evaluator in action when it all gets put together.

1. Write a function to do variable binding on a stack.

   The stack should just be a list: use CONS and CDR to add and remove items.

   The binding function should take two equal length lists, one of variables and one of values.

   It should put a marker on the stack (e.g., the litatom MARKER) and then put a binding pair on the stack for each variable and its corresponding value in the lists.

2. Write a function that unbinds variables. It should basically remove items from the stack up to and including the next marker.

3. Write a function that looks up the value of a variable on the binding stack. If its not on the stack, get its top level value using GETTOPVAL.

4. Write a function that sets the value of a variable. If the variable is on the binding stack, then just reset the value in that binding. Otherwise, set its top level value using SETTOPVAL.

5. Rewrite the EVAL and APPLY procedures from the course notes.

   A. Rename the functions so as not to mess up the original EVAL and APPLY in Interlisp.

   B. Use your bind, unbind and lookup functions.

   C. In your apply, if the function name is SET, SETQ, or SETQQ, then use your set value of variable function to carry out the appropriate setting action instead of looking the definition of SET, SETQ, or SETQQ.

   D. In your apply use GETD to get a function definition. If the value of GETD is a list, then proceed as in the course notes. If the value of GETD is not a list, then assume it is a primitive function and just invoke the standard APPLY function instead of the rest of your apply function.

6. Write a procedure (called CountAtoms) that recursively counts all the atoms in a list. (See page 12 of LispCourse #5).

7. Use your eval to evaluate *(CountAtoms ’(A (B C D) (E (F (G (H) I) J K) L)))* and watch the evaluator in action as it prints out its action summaries.