6 Environments and Bindings

What's in This Set of Notes ?

6.1 Definitions

- A *binding* is an association of a name with a value.
- A *bound variable* is a variable for which a binding exists.
- A *unbound* variable has no binding.
- The duration of time that a variable is bound is called its *extent* or lifetime

  Comments:
  Bindings for local variables, for example, are created and destroyed by procedure entry or exit

- A *name* can have at most one binding at a given time
- An *environment* consists of a set of bindings (name:value) pairs
- In Scheme we say environment is a sequence of *frames*
- Each *frame* is table of bindings
- Each frame has a *pointer* to its enclosing environment unless the frame is considered global

  Comments:
  The environment is used as a lookup table. In most programs the environment actually consists of a set of nested environments. Given the name of something in a program (usually a variable of procedure) the associated value can be found by searching the set of bindings for the current program according to the scoping rules for the language.

- The root or outermost environment is called *initial environment*
- There are two common forms of maintaining bindings: *dynamic* and *lexical* scoping
- *Dynamic Scoping* is the method used to define the scope and extent of variables in Lisp and APL. The environments are searched in the reverse order of invocation. It is as if the return "address" is used to find the environments of the callers.
- *Lexical Scoping* (sometimes called static) is the method used to

define the scope and extent of variables in Scheme, Common Lisp, Pascal and Algol. Lexical scoping requires keeping track of the lexical order as well as the dynamic execution order at runtime. This means that a Scheme procedure call will of necessity be more expensive than a similar call in C or Fortran
- *Deep Binding:* Most common implementation. Each environment consists of a collection of variables names and values. The environments are linked according to the scoping rules. When a name is used in a procedure the environments are searched from the inner to the outer nesting level.
- *Shallow Binding:* Associate bindings for each variable and then search the bindings associated with the variable to find the binding for the current environment. The major advantage is that it reduces the search time for variables that are bound in the global environment.

Issues:

- Parameter Passing
  - What can be passed?
  - Are values shared or copied?
- Scoping
  - How are free variables searched for in outer environments?
- Representation of Environments
  - Which operations are more efficient?

6.2 Contour Model

- A graphical model of runtime environments for block structured programming languages.
- Describes the runtime execution of a program as a sequence of *snapshots* which illustrate the nested environments and hence the scoping and binding mechanisms used by a programming language
- References: Data Structures in Programming Languages - ACM Sigplan Notices, 1973
  1. The Contour Model of Block Structured Processes, J.B. Johnston, pp 55-82.

Model

- The model consists of a set of *contours* each corresponding to a given environment both lexical and dynamic.
- Each *snapshot* It is a record of execution which depicts the state of the machine as a nested collection of contours.
- An *access environment* is a relationship between processor(s) and the set of environments A typical access environment is the traditional dynamic/static link.
- Labels are used for flow control.
- Each contour has a *declaration array* containing the variables

which are allocated in that environment.
- A process is modeled as a pair of pointers *(ep, ip)* where the ep(*environment pointer*) points to a contour and the ip(*instruction pointer*) points to the next instruction to be executed.
- It is often useful to show where the process is to return control to once it is finished. For this we use an rp(return pointer)

6.3 Visualization with Contour Model

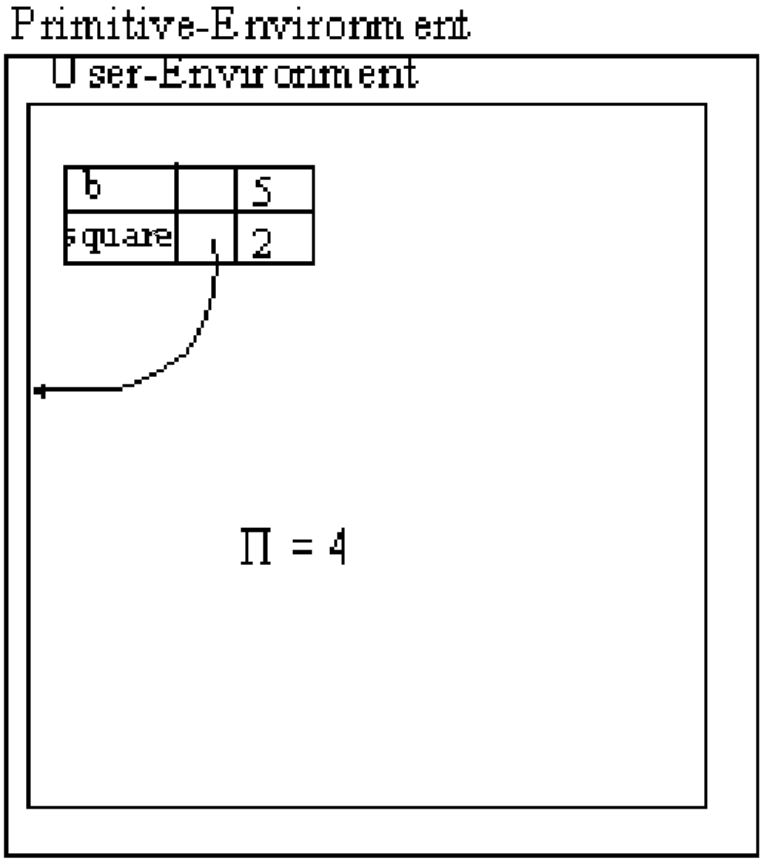
- So what happens when we execute the following:

```
line 1:   (define b 5)
line 2:   (define (square num)
line 3:      (* num num))
line 4:   (define result (square 5))
```

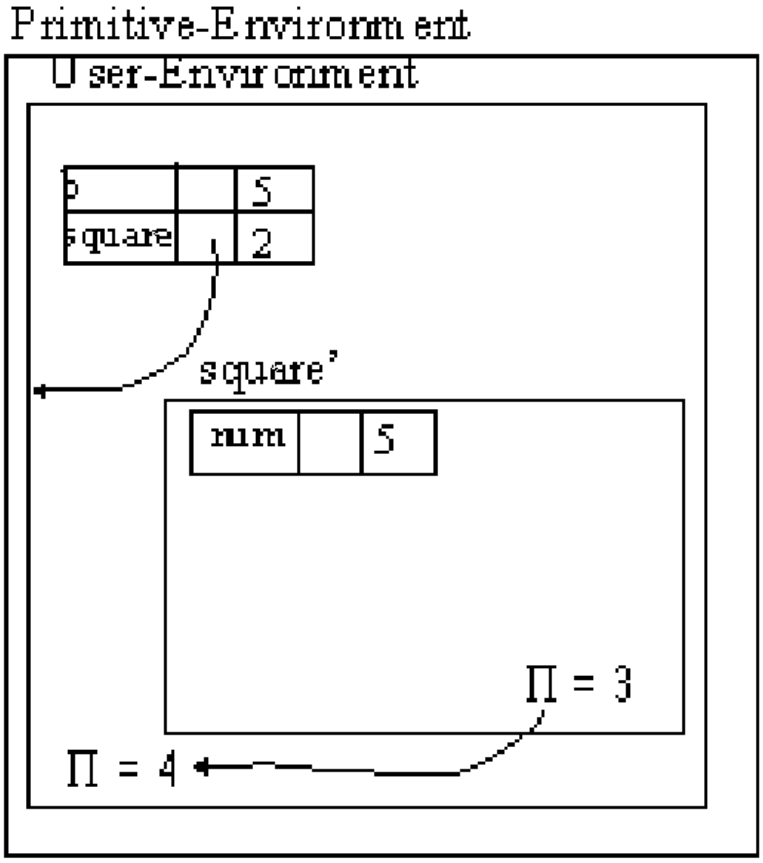
- Recall the steps for evaluation:
  1. Evaluate each of the subexpressions (within a given environment)
  2. Create a new frame in which the formal parameter names of the function are bound with the results of evaluating the operands
  3. Add this frame to the a new environment created within the ep of the procedure
  4. Evaluate the body of the function with respect to the new frame and return the result

---

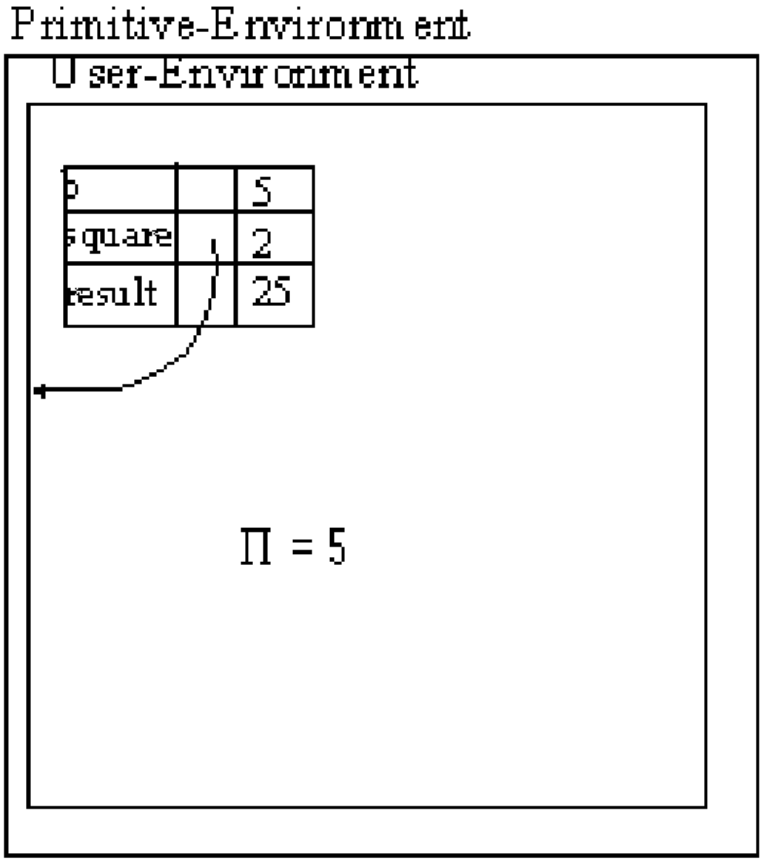
- Notes:
  1. Primitive Environment is created by Scheme
  2. User-Environment is one generated by read-eval-print loop
  3. Bind variables *b* and *square* to associated values in frame in user-environment



4. Create new environment for square and add frame for local argument
5. Execute body of square

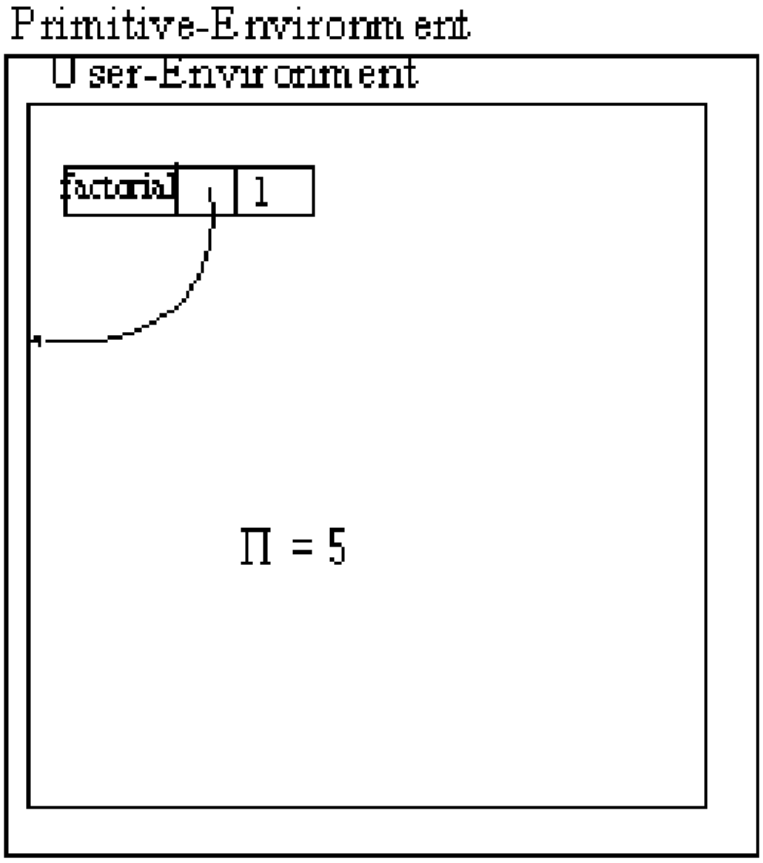


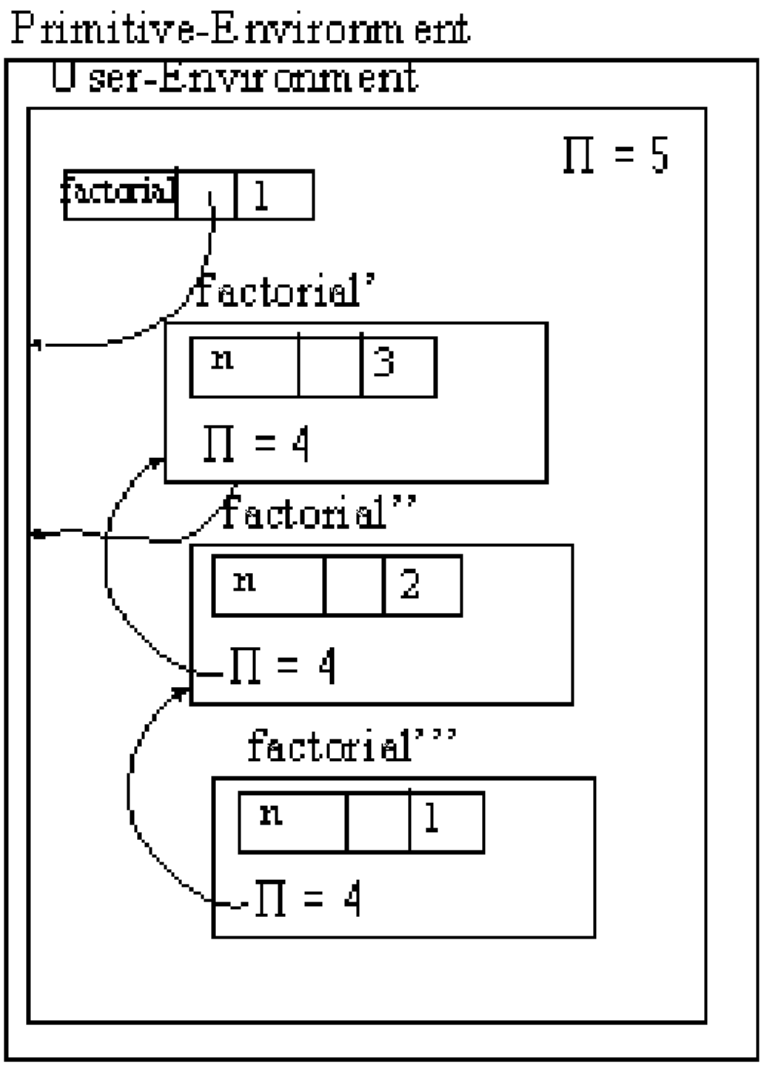
6. Bind values to result variable in user-environment

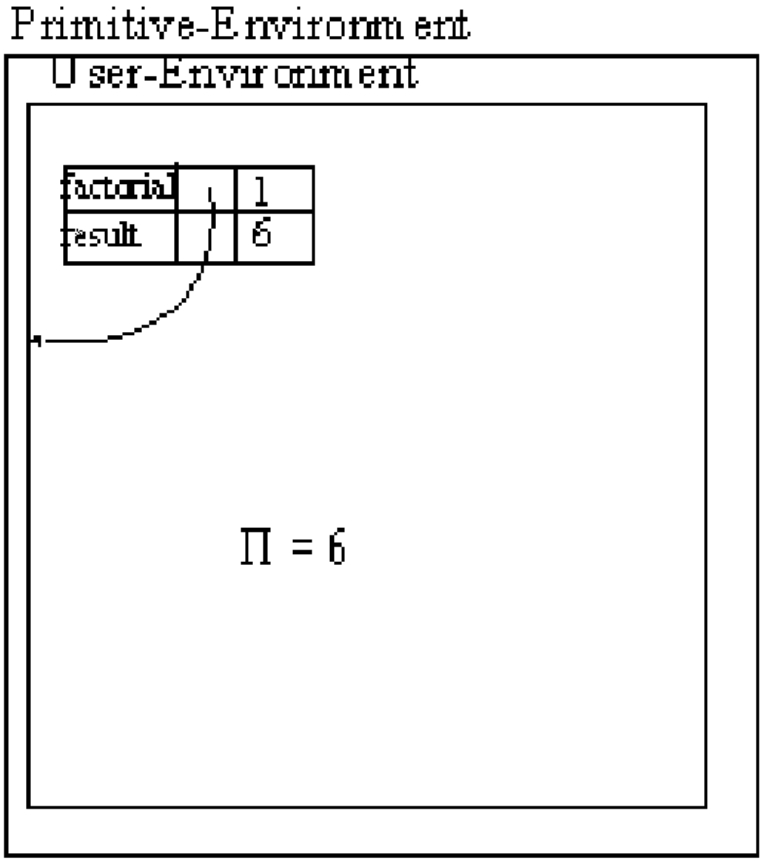


Another Example with Recursion

```
line 1:   (define (factorial n)
line 2:         (if (< n 2)
line 3:               1
line 4:               (* n (factorial (- n 1)))))
line 5:   (define result (factorial 3))
```


Primitive-Environment
User-Environment
factorial
1
Π = 5


Primitive-Environment
User-Environment
Π = 5
factorial
1
factorial'
n
3
Π = 4
factorial''
n
2
Π = 4
factorial'''
n
1
Π = 4



6.4 Scoping

Scoping of Variables

- The value associated with a variable name is found by searching nested environments
- The scoping rules of a language state how environments are nested
- Two main formats: lexical and dynamic

Lexical Scoping

- Binding is searched for in current procedure environment.
- If not found, search is continued in the procedure in which current procedure is defined, and so on.
- We must keep track of the lexical order in which procedures are defined, as well as the execution order at runtime.
- Same approach in Pascal, Scheme, Algol

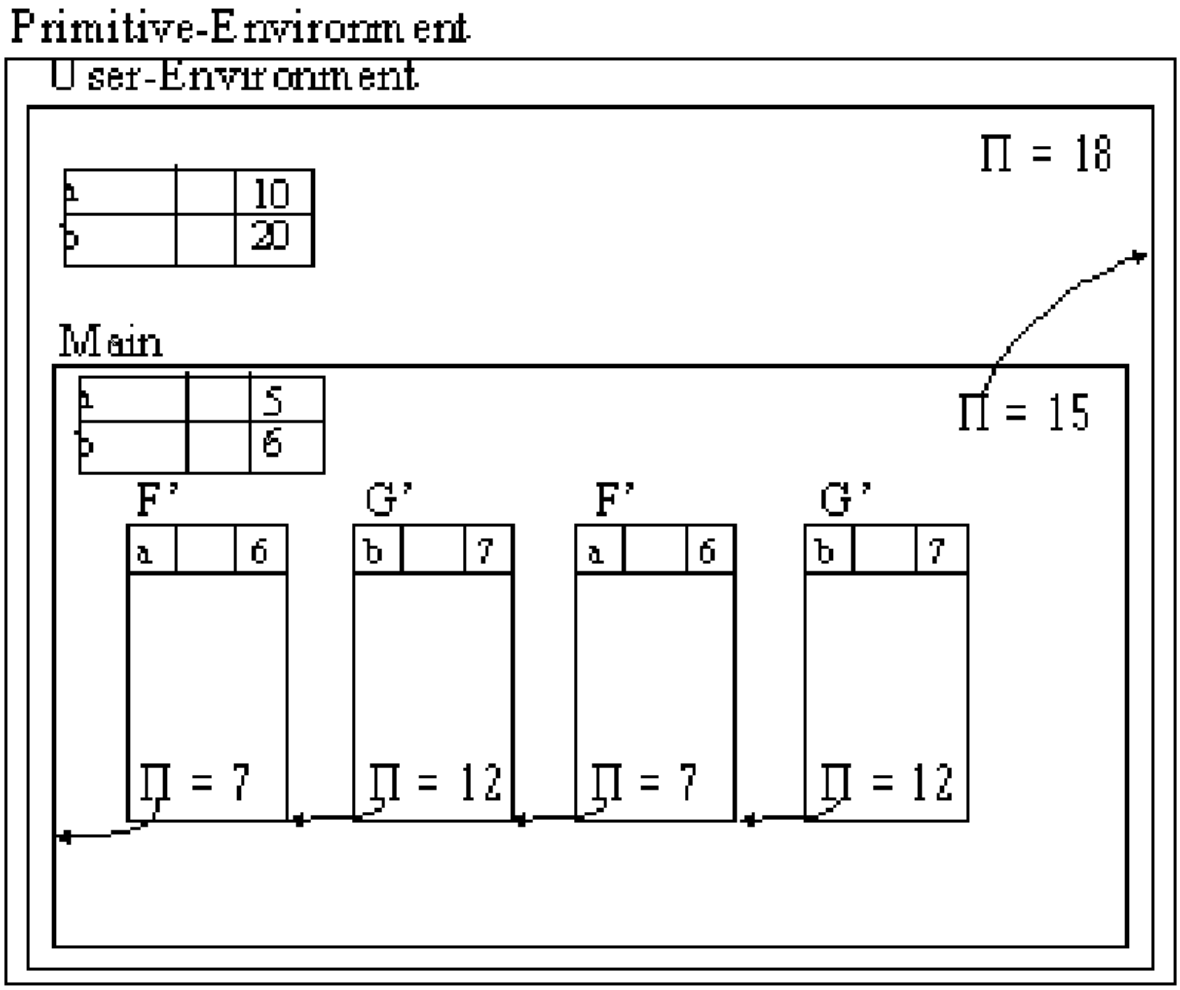
Dynamic Scoping

- Environments are searched in reverse order of function invocation.
- If variable not found in current environment, then search in environment of function which called the current function.

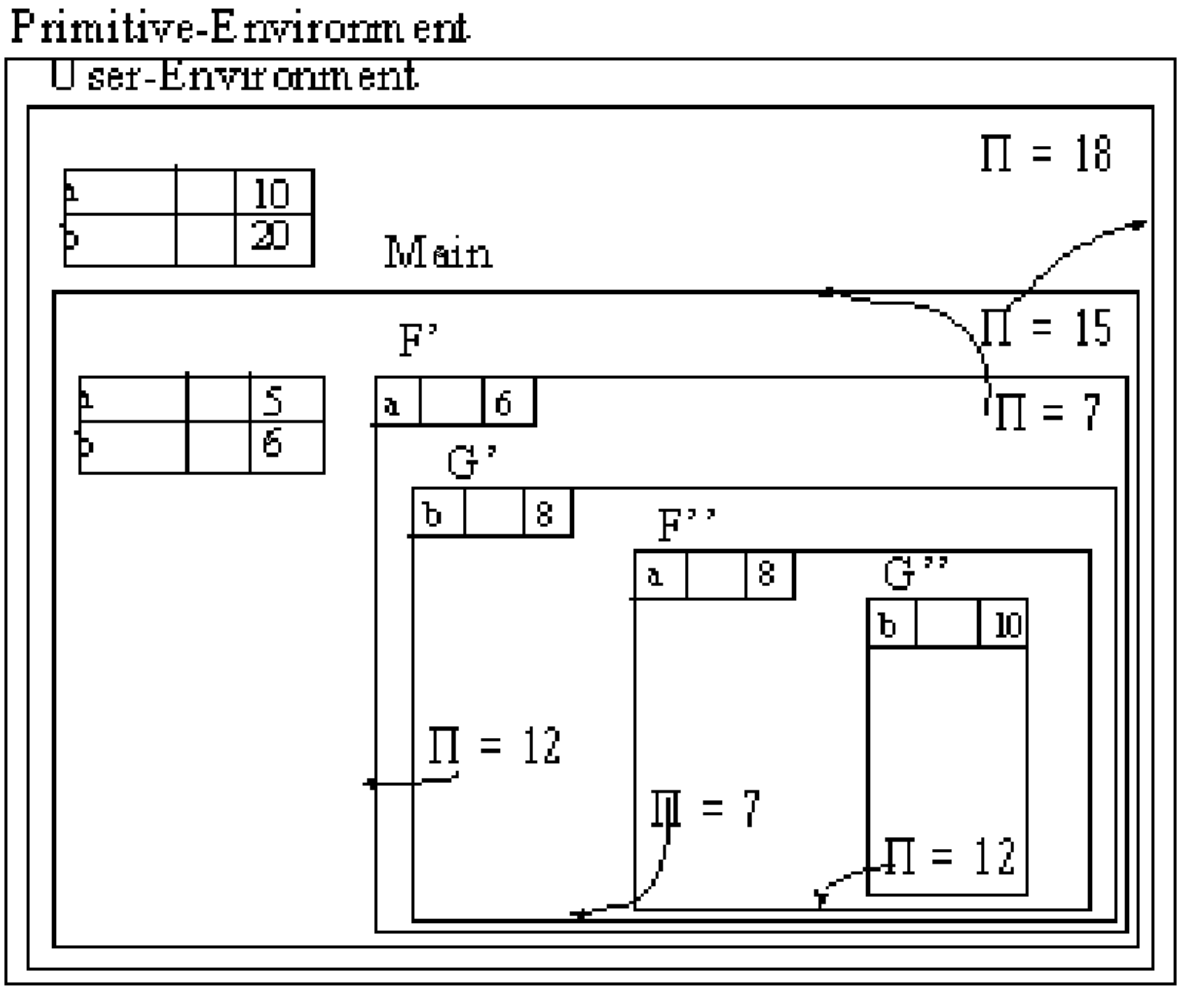
Example

```
Line1:      (define (main)
Line 2:          (let (( a 0) (b 0))

Line3:          (define (f)
Line4:            (let ((a 0))
Line5:               (set! a b)
Line6:                (writeln "in f: A and B are - " a b)
Line7:               (g)))

Line8:          (define (g)
Line9:            (let ((b 0))
Line10:               (set! b (+ a 2))
Line11:               (writeln "in g: A and B are - " a
b)
Line12:                (f)))

Line13:         (set! a 5)
Line14:         (set! b 6)
Line15:         (f)))

Line16:    (define a 10)
Line17:    (define b 20)
Line 18:   (main)
```

Execution Contours Using Lexical Scoping



Execution Contours Using Dynamic Scoping



5.5 Notes

- Since let is just syntactic sugar for lambda it has the same contour representation.
- A Lambda expression and its definition environment are saved into a data structure called a closure.
- A closure formalizes the notion of freezing the value of free variables.