**Part 1.4 Closure and Lexical Scope**

1. **Introduction to Closure and Lexical Scope**
   1. **Closures**

A closure is a combination of a function together with references to its surrounding state (the lexical environment). Closures give a function access to all the variables of its lexical environment, even when the function is executed outside that environment. In other words, a closure "closes over" the variables from the environment in which it was defined.

Formally, a closure can be represented as a pair:

* The function code itself (with its parameters and body)
* The environment that was in effect when the function was defined

This combination allows the function to remember and access its lexical environment even when invoked outside that environment.

* 1. **Lexical Scope**

Lexical scope (also known as static scope) is a variable resolution strategy where the scope of a variable is determined by its position in the source code. In a language with lexical scoping:

1. A variable's scope begins at the point where it is declared
2. The scope extends to all nested blocks within its declaring block
3. Variable references are resolved by searching the current block and progressively outward through enclosing blocks

This contrasts with dynamic scope, where variable references are resolved based on the calling context rather than the defining context.

1. **The Stack-Like Structure of Environments**

**2.1 Environment as a Stack**

In our TLS interpreter, environments are implemented as tables that behave like a stack:

The environment begins as an empty list. When new frames are created using the new-entry function, they are added to the front of the environment using *extend-table*, which effectively pushes a new frame onto the environment stack. Each frame is a list of (variable . value) pairs, constructed using the *build-pair* function. This structure supports nested scopes and lexical scoping during evaluation.

Each time the interpreter enters a new scope (e.g., when evaluating the body of a lambda expression), a new frame is pushed onto the environment stack. When that scope is exited, the frame is no longer used, effectively mimicking a stack pop as evaluation returns to the enclosing environment.

The environment stack has these critical properties:

1. **LIFO (Last-In-First-Out) behavior**: The most recently defined variables are found first.
2. **Shadowing**: Inner scope variables with the same name hide (shadow) outer scope variables.
3. **Persistence**: Outer environments remain unchanged when inner environments are created.

This stack-like implementation is what enables lexical scoping, as it preserves the hierarchical relationship between nested scopes.

**2.2 Variable Lookup in the Stack**

Variable lookup in our interpreter navigates the stack-like environment structure using the *lookup-in-table* function.

This function operates as follows:

1. It first checks the top frame, which represents the most recent (innermost) scope.
2. If the variable is not found in that frame, it recursively searches deeper in the environment stack, moving through older (outer) scopes.
3. If the entire stack is exhausted without finding the variable, it calls table-f, typically resulting in an "unbound variable" error.

This recursive stack traversal mirrors the behavior of lexical scoping, in which variable resolution starts from the innermost scope and moves outward to enclosing environments.

**3. Implementation of Closures Using the Stack**

**3.1 Capturing the Stack**

When a lambda expression is evaluated, the interpreter does not immediately execute its body. Instead, it calls the *\*lambda* function, which is responsible for handling lambda expressions. This function creates a closure, which is a compound object that packages together two things:

* The lambda expression itself (its parameter list and body), and
* The environment at the time of definition, i.e., the current environment stack.

This captured environment represents the lexical context in which the lambda was defined. By saving this stack snapshot, the closure can later evaluate its body with the correct variable bindings, even if it’s applied in a different scope.

Capturing the environment in this way is essential for lexical scoping, variables in the closure body are resolved based on where the function was defined, not where it was called.

**3.2 Extending the Stack During Application**

When a closure is applied, the interpreter uses the *tlsi-apply-closure* function to evaluate the function body. This involves extending the closure’s captured environment stack with new bindings for the parameters.

This operation works in three steps:

1. It creates a new frame that binds the closure’s parameters to the provided arguments using new-entry.
2. It pushes this frame onto the front of the closure's captured environment using extend-table, creating a new extended environment stack.
3. It evaluates the closure’s body using this extended stack, not the caller's environment.

The important thing here is that the closure carries its own lexical environment, and that is what's extended and used during application, not the environment from the call site. This is the heart of lexical scoping: functions evaluate in the context where they were defined, not where they were called.

**Examples**

**Example 1: Nested Scope**

(value '((lambda (x)

((lambda (y)

x) ; x refers to outer scope

10))

5)) ; Returns 5

**Example 2: Closure Capture**

(value '((lambda (x)

((lambda (f)

(f))

(lambda () x))) ; Closure captures x=5

5)) ; Returns 5

**Example 3: Shadowing**

(value '((lambda (x)

((lambda (x)

x) ; Inner x shadows outer x, Returns 10

10))

5))

**Example 4:**

(value '((lambda (x)

((lambda (f)

((lambda (x)

(f)) ; Closure uses captured x=5, not local x=10

10))

(lambda () x)))

5)) ; Returns 5