Team: LingJie Pan, Xin Gao, Joquanna Scott

1.1

Walkthrough of TLS Interpreter This document walks through the TLS (The Little Schemer) interpreter implementation. It breaks down each section and explains what each function does.

SECTION 1: HELPER FUNCTIONS

- **build:** Creates a two-element list. Used to create environment entries or closures.
- **first, second, third:** Simple accessors for the first, second, and third elements of a list.

SECTION 2: ENVIRONMENT (TABLE) OPERATIONS

- **new-entry:** Creates a new environment entry (names and values).
- **lookup-in-entry:** Searches a single entry for a name and returns the associated value.
- **extend-table:** Adds a new entry to the front of the environment.
- **lookup-in-table:** Searches through all environment entries (table) for a name.
- **initial-table:** Default error handler when a name is unbound (not found in any entry).

SECTION 3: TYPE PREDICATES AND EXPRESSION CLASSIFICATION

- **atom?:** Determines if a value is an atom (non-list, non-null).
- **primitive?:** Checks if a value is a tagged primitive (e.g., '(primitive add1)).
- **non-primitive?:** Checks if a value is a tagged user-defined function (closure).
- **expression-to-action:** Main classifier that dispatches based on expression type.
- **atom-to-action:** Handles classification of atomic expressions.
- **list-to-action:** Handles classification of list expressions (e.g., quote, lambda, cond, application).

SECTION 4: ACTION IMPLEMENTATIONS

- ***const:** Evaluates constants and primitive operations.
- ***quote:** Returns the quoted portion of a 'quote expression.
- **text-of:** Retrieves the text inside a quote.
- ***identifier:** Looks up a variable's value in the environment.
- ***lambda:** Constructs a closure from a lambda expression.
- **table-of, formals-of, body-of:** Accessors for closure components.
- **else?:** Checks whether a cond clause is an 'else clause.
- **question-of, answer-of, cond-lines-of:** Helpers for processing cond expressions.
- **evcon:** Evaluates cond clauses in order, returning the first matching result.
- ***cond:** Top-level cond evaluator that calls evcon.
- **evlis:** Evaluates a list of arguments.
- **function-of, arguments-of:** Accessors for the components of a function application.

● **apply-primitive:** Applies a primitive operation to arguments.
● **apply1:** Dispatches to either apply-primitive or apply-closure depending on function type.
● **apply-closure:** Applies a user-defined function (closure) using its saved environment.
● ***application:** Handles function applications by evaluating function and arguments.

SECTION 5: TOP-LEVEL EVALUATION FUNCTIONS

● **meaning:** Core evaluator that dispatches based on expression classification.
● **value:** Top-level function that evaluates an expression starting from an empty environment.

SECTION 6: TEST CASES

These tests the interpreter using examples from quoting, list manipulation, arithmetic, Boolean checks, conditionals, lambda expressions, closures, and function applications. Each (value '(...)) call demonstrates how the interpreter behaves for a specific kind of Scheme expression.

1.2

Helper Function

- **(find? Target set):** Search through the set to check whether the target exists in the set or not.
- **(Repeated? Set)** Check whether the list contains duplicated elements. Used to detect if the parameters of lambda are duplicated.
- **(param-check-help name):** Return the corresponding number of arguments for the primitive
- Helpers for Free Variable Collection
  - **(element-lst? element lst):** Returns whether an element is contained within a list
  - **(union-lst lst1 lst2):** Returns the union set of lst1 and lst2
- **Primitives**: Defined a list of primitives

Syntax Checking Procedures

- **(lambda-checker expr):** Check whether the input expression is a valid lambda expression
- **(cond-checker expr):** Check whether the input expression is a valid cond expression
- **(param-check expr):** Check whether the input primitive expression has the correct number of arguments
- **(unbound-var? expr):** Returns whether the expression has any free variables.
- **(free-vars expr vars):** Returns the list of free variables from lambda expressions

Top Level Function

- **(Syntax-checker expr):** Top level function that recursively search through an expression,

and check whether the syntax is valid.

1.3

Helper Function

- Accessor Functions for *Let:
  - **(*let-body)** - Returns the body of a *let expression
  - **(*let-init)** - Returns the initial values provided in the bindings
  - **(*let-vars)** - Returns the variables/names provided in the bindings
  - **(*let?)** - Predicate for determining if a given expression is a *let expression
- **(merge lst1 lst2)** - Returns the elements of lst1 and lst2 as a merged interwoven list

Top Level Function

- **(*let expr table) -** Action function meant to evaluate the components of a let expression and maintain its current environment.
  - As of 5/8/25: Does not perform as intended (as syntactic sugar for lambda)
- **(new-binding entry) -** Changes the representation of a TLS entry to (name value) using TLS' provided accessors

1.4

Base Case:
Case 1: exp is a constant, (value exp) returns exp
Case 2: exp is an identifier, (value exp) returns the corresponding value of exp in the environment or table using lookup-in-table

Induction Step：
Assume that for all expression exp' smaller than the expression exp, evaluating exp' respects lexical scoping through closure.

For a lambda-expression: (lambda (parameters) (body)), (value lambda-expression) returns:

(non-primitive (env (parameters) (body)))

Where it creates closure that captures environment env at definition time, which ensures that any free variable in the exp is resolved using the environment at definition, thus lexical scope is preserved.

For evaluation of an exp, (meaning exp env) either returns a primitive tagged function or a non-primitive.

- If it returns primitive tagged function, then it is not closure, so it does not reference any environment, hence lexical scope is preserved.
- If it returns a non-primitive, then by the induction hypothesis, each argument of the expression respects lexical scoping. When we use apply-closure, it extends the closure's

saved lexical environment with the new frame binding each identifier in the parameter to its corresponding evaluated argument. The function body is then evaluated in this extended environment. Thus, lexical scoping is preserved.

Therefore, by structural induction, all expression respect lexical scope.

1.5

Base Case:

Case 1: If the expression is a constant (such as a number or boolean), the evaluation function simply returns the constant value: (value exp) returns exp when exp is a constant.

Case 2: If the expression is an identifier (a variable), the evaluation function looks up its corresponding value in the environment: (value exp) returns the value of exp found using lookup-in-table. This ensures that symbols correctly reference values from the environment and follow lexical scoping.

Inductive Step:

- Assumption: For any expression exp' smaller than exp, the evaluation respects lexical scoping via closure.
- For a Lambda Expression:
    - (lambda (parameters) (body)):
        - (value lambda-expression) returns a non-primitive closure that captures the environment (env) at the time of definition, including the parameters and body of the function.
        - This ensures that any free variables in the body of the lambda are resolved using the environment at the time the lambda was created, preserving lexical scoping.
- Evaluation of Expressions:
    - When evaluating an expression (meaning exp env), the evaluation function either returns a primitive function or a non-primitive (closure).
        - If a primitive function is returned, it does not reference any environment because it is self-contained, thus lexical scoping is preserved.
        - If a non-primitive is returned (a closure), by the inductive hypothesis, each argument of the expression respects lexical scoping. When applying the closure:
            - The closure's environment is extended by creating a new frame that binds each identifier in the parameter list to its corresponding evaluated argument.
            - The function body is then evaluated in this extended environment.

This guarantees that the closure correctly references variables in the environment where it was defined, preserving lexical scoping.

By structural induction, it is proven that all expressions in the TLS interpreter respect lexical scoping. The evaluation of expressions works as follows:

- Constants return their own values.
- Identifiers are looked up in the correct environment.
- Lambda expressions create closures that capture and preserve the environment where they were defined.
- Function applications extend the environment with arguments and ensure that the body is evaluated in the correct extended environment, thereby maintaining lexical scoping.

Therefore, the TLS interpreter correctly handles lexical scoping through closures, and its evaluation process preserves this property for all types of expressions.