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

The interpreter's environment subsystem manages the ((name)(value)) pair entries that makeup the environment with the following properties:

- Environment structure: nested list
- Entry structure: "An entry is a pair of lists whose first list is a set. Also, the lists must be of equal length."
    - Implies that every valid variable in TLS should be bound
- Lexical scoping: In the environment level, maintained by extend-table. Cons'ing the new entries to the front of the stack is analogous to going deeper into scope frames as new closures are created.

The primary operations of the subsystem are new-entry, lookup-in-entry, lookup-in-table, and extend-table. The action functions use the primary operations to define types in the interpreter.

Claim: The environment subsystem of TLS is structured in a way that allows it as an abstract data type. Therefore, a change in representation should satisfy the previously stated specifications and work with the rest of the interpreter. This can be shown by comparing the results of the action functions with the new and old representation.

Top Level Function

All of the functions in this section are the primary operations changed to fit with the new representation. Therefore, they have the same name and specs as the original.

1.4

Closure: the function to capture environment at definition time, so that functions can have access to variables from its outer function, even after the outer function executed.

Lexical scope: the scope of the variables is determined by its position in the code's structure. For a nested function, the lexical environment for the outer-most function is the global scope, and the lexical environment of the inner functions is its parent functions' scope, where it can access variables from its parent function's scope(including the global scope). The lexical scope is determined at the time of definition not execution, this allows access to variables in outer function even after execution.

```
;Ex:
(define (plus-1 x)
   (lambda () (+ x 1)))

(plus-1 1) ;when you call plus-1 directly, it'll return a procedure, which is the inner lambda
```

and it creates a closure where it captures the lexical environment, where it contains the binding of variable x with 1.

((plus-1 1)); when you try to call the inner lambda, it will return 2 since we passed the argument 1 and the inner lambda should increment the input by 1. This is a result of closure. When plus-1 is called, it captures the lexical environment with x=1, then because of lexical scope, inner lambda would have access to the variable x because the varibale x is in its parent function's scope.

((plus-1 2));->3
((plus-1 100));->101
(newline)

;To prove that our TLS implements closure and lexical scope correctly we can use structural induction.

;Base Case:

;For atom expression(number, boolean, primitives, identifiers), TLS handles closure and lexical scoping correctly.

;Correct because number and boolean are self-evaluating constants that do not depend on environment. For a primitive, if they are referenced directly, they will be tagged as primitive and treated as a constant, which do not require environment as well. When applied, the primitive itself is not a closure and does not capture any environment. If it needs to look up its arguments, it will just refer to its' current environment. If it's in a closure, then it will refer to the saved environment captured by closure, which ensures that the variables are resolved in their lexical definition.
;For identifiers, it also does not create a closure that captures the environment, it just refers to its environment passed with the function meaning. In a closure form, its environment will be the the environment captured by closure, which ensures the identifier will be resolved using its lexical definition.

;Hence, atomic expression will be trivial to the proof of closure and lexical scoping as they do not involve creating closure.

;Assume for all subexpression 'exp of expression exp, the TLS scheme correctly implements closure that captures the environment and evaluates the variables according to the lexical environment.

;Induction step:

;In the TLS scheme, when defining a lambda expression, the function *lambda would return the closure: (non-primitive (env (parameters) (body))). In which it tagged the user defined function as non-primitive, so that when evaluating the function, the interpreters know it's a closure and can apply closure. The environment env is the lexical environment captured by

closure, which will be the parent scope of the function, where it would capture future bindings of variables so that its inner function can access variables from outer function. By induction hypothesis, since all variables are resolved according to the lexical environment, when the function body is later evaluated it will look up its arguments in the saved environment using look-up-table, ensuring lexical scope, where it would have the access to variables in the outer function as they are stored in the environment.

;Ex:
(value '(lambda (x) (cons x 1)));->(non-primitive (() (x) (+ x 1))). In this case, the first part of exp is the non-primitive tag, and the second part is the environment env with the formal body of the lambda expression. The empty list is the table or environment

;During function application, if the function is a primitive function, then according to our base case, they do not involve the creation of closure, hence the interpreter would handle their closure and lexical scope correctly. If the function is non-primitive, then the function will be processed by *lambda to create a closure that captures environment, which is empty at top-level. When evaluating, the function apply-closure would evaluate the function body in the environment obtain by closure, and extend the environment with a new entry of the variable with its evaluated arguments, which means now the inner function have access to variables in the outer function as they are stored in the environment, whereas the outer function could never access the variables in the inner function, as by the time the outer function are evaluated with their respected arguments, variables of the inner function are not yet recorded in the table, so they can't find it in the table even if they try to. Hence, lexical scope is preserved.

;Therefore, the TLS scheme correctly implements closure and lexical scope.

Ex:
(value '(((lambda (x) (lambda (y) (cons x y))) 2) 3))
(value '((((lambda (x) (lambda (y) (lambda (z) (cons x (cons y z))))) 2) 3) 4))
(value '(((lambda (x) (lambda (y) (car (cons x y)))) 2) 3))

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.

1.6

The TLS (The Little Schemer) interpreter is implemented as a pure Scheme program running on top of the R5RS-compliant DrRacket environment. This design means that TLS depends heavily on DrRacket's underlying Scheme system, especially in the execution of primitive operations and the mechanics of function calling.

Dependence on R5RS of DrRacket

TLS defines an interpreter that processes Scheme expressions by representing and evaluating them explicitly as data structures (lists, atoms, etc.). However, TLS does not implement all aspects of Scheme from scratch; instead, it leverages DrRacket's R5RS Scheme runtime for:

- **Primitive operations:** Basic arithmetic, list operations, boolean logic, and other built-in primitives are directly performed by DrRacket's native implementation.
- **Low-level evaluation and environment management:** TLS uses custom environment structures but relies on DrRacket to manage actual memory, variable bindings, and function calls at the system level.
- **Function call mechanics for primitives:** When TLS evaluates a primitive function application, it invokes DrRacket's built-in function directly. TLS itself only dispatches to these primitives without implementing their internal mechanics.

Mechanics of Function Calling: TLS vs. DrRacket

In TLS:

- Function application expressions are parsed and classified.
- TLS evaluates the function expression to determine if it is a primitive or a user-defined closure.
- If it is a user-defined closure, TLS constructs a new environment frame extending the saved closure environment with the bindings of parameters to argument values.
- TLS then recursively evaluates the function body expression in this extended environment.

In DrRacket

- TLS is itself a Scheme program running inside DrRacket, so every TLS function call — including recursive calls in the interpreter — is executed by DrRacket's function calling mechanism.
- When TLS applies a primitive function (e.g., +, car, cons), the call is delegated directly to DrRacket's built-in primitive function implementations.
- When TLS applies a user-defined closure, TLS's environment and evaluation rules govern the evaluation, but the recursive invocation of TLS evaluation functions is managed by DrRacket's call stack and execution engine.

Thus, TLS performs the semantic interpretation and environment handling at the Scheme language level, while relying on DrRacket's runtime system for the actual execution, stack

management, and primitive operation execution.