Project
SNU 4190.210, Programming Principles Fall 2024
Chung-Kil Hur
Due Date: 2024.12.23 (Mon) 23:59

This file provides the semantics of the toy language for which you will implement an interpreter. Refer to `src/lib.rs` for a basic overview and important notes about the project.

Problem 1 Implement the `eval` function to serve as an interpreter for the programming language E described below in Rust.

$$\texttt{eval} : E \to V$$

FD	::=	$(x) \Rightarrow E$	(anonymous) function definition
B	::=	$(f = FD)$	function binding
E	::=	n	integer
	|	$\texttt{(}E\ \texttt{+}\ E\texttt{)}$	addition
	|	$\texttt{(}E\ \texttt{-}\ E\texttt{)}$	subtraction
	|	$\texttt{(}E\ \texttt{*}\ E\texttt{)}$	multiplication
	|	$\texttt{(}E\ \texttt{/}\ E\texttt{)}$	division
	|	$\texttt{(if0}\ E\ E\ E\texttt{)}$	conditional
	|	x	name
	|	$\texttt{(}E\ \texttt{(}E\texttt{))}$	function call
	|	$\texttt{(let}\ x := E\ \texttt{in}\ E\texttt{)}$	name binding of `val`
	|	$\texttt{(letfun}\ B^*\ \texttt{in}\ E\texttt{)}$	name binding of `def`
	|	$\texttt{(letfun (}\star = FD\texttt{) in }\star\texttt{)}$	anonymous function

- For ill-typed inputs, use the `panic!` macro to raise errors (e.g., `panic!("error message")`). Note that the error message itself will not be graded.
- X^* indicates that X can appear zero or more times.
- Addition, subtraction, multiplication, and division are performed using operations for Rust's `i64`. There is no need to explicitly handle division by zero, as Rust's `i64` division already raises an error in such cases.
- The `if0` E_1 E_2 E_3 expression first evaluates E_1 to a value v. If v equals 0, the result of E_2 is returned; otherwise, the result of E_3 is returned.

- Identifiers (names) `x` must be alphanumeric and cannot start with a digit.
- `let/letfun` clauses create a new scope, similar to a block in Scala. Name bindings `def` and `val` work in a similar manner as in Scala.
  - `(let` x `:=` E_1`)` assigns the name x to the value obtained by evaluating E_1, then evaluates E_2.
  - `(letfun` B^* `in` E`)` defines functions in B^*, supporting mutual recursion within the same scope, and then evaluates E.
  - `(f = (x)` $\Rightarrow$ E) assigns the name `f` to the expression `E` with argument x. For example, `letfun (f x := (x - 1) in f(42)`.
  - Duplicate names cannot be defined within the esame scope.
- `Environment`(Env) is a collection of `Environment Entry`(EnvEntry). An `Environment Entry` is created whenever a new scope is introduced.
- For Problem 1, memoization is not required.
- Examples can be found in the test cases at the bottom of `src/eval.rs` and `src/eval_memo.rs`.
- `(letfun` $(\star = FD)^*$ `in` $\star$`)` is a syntactic sugar for anonymous functions using `letfun`.

Problem 2 (Extra Credit Problem) Enhance the language by incorporating memoization.

$$\texttt{eval\_memo} : E \to V$$

B	::=	$(f = FD)$	function binding without memoization
	|	`memo:`$(f = FD)$	function binding with memoization
E	::=	...	
	|	`(letfun memo:`$(\star = FD)$ `in` $\star$`)`	anonymous function with memoization

- For function applications in memoized functions, if the argument evaluates to a number, check if the computation result is already recorded in M. If it is, return the recorded value as the result. Otherwise, perform the computation and log the result in M.
- For function applications in non-memoized functions, follow the same process as described in Problem 1.
- `(letfun memo:`$(\star = FD)$ `in` $\star$`)` is a syntactic sugar for anonymous functions using `letfun` with memoization.