

Project 2: Scheme Interpreter

CSC 4101, Fall 2009

Due: 12 November 2009

For this project, you will implement a simple Scheme interpreter in C++ or Java. Your interpreter should be able to handle the same subset of the language as the parser for project 1, as well as some predefined functions.

In particular, your interpreter should implement

- symbols, 32 bit integers, booleans, strings, lists, and closures;
- the test `symbol?` for identifying an identifier;
- the test `number?` for identifying integers and the binary arithmetic operations `b+`, `b-`, `b*`, `b/`, `b=`, `b<`;
- the list operations `car`, `cdr`, `cons`, `set-car!`, `set-cdr!`, `null?`, `pair?`, `eq?`;
- the test `procedure?` for identifying a closure;
- the I/O functions `read`, `write`, `display`, `newline`;
- the functions `eval`, `apply`, and `interaction-environment`.

Build this interpreter on top of the code you wrote for project 1. Use the parse trees from project 1 as the internal data structure for lists. The arithmetic and list operations would then be simply implemented by the appropriate C++ (or Java) operations on parse trees.

The following built-in Scheme functions call parts of your interpreter:

- `read` calls the parser and returns a parse tree,
- `write` calls the pretty-printer,
- `eval` calls your C++ or Java `eval()` function,
- `apply` calls your C++ or Java `apply()` function,
- `interaction-environment` returns a pointer to your interpreter's global environment.

All other built-in Scheme functions, such as `and`, `or`, `not`, `>`, `>=`, `<=`, `list`, `cadr`, `caddr`, etc., `equal?`, `map`, `assq`, `reverse`, etc., can then be implemented as Scheme definitions using only the primitive operations implemented above. Since we have a general mechanism for defining functions with a variable number of arguments, the n-ary versions of the binary arithmetic operators can also be implemented in Scheme. We'll do that in project 3.

Environments

For keeping track of the values of variables during evaluation, we need a data structure for storing these values. That's the environment. Scheme is a language with nested scopes. Given the function definition:

```
(define (z x) (+ x y))
(define y 42)
(define x 17)
```

When evaluating the body of the function call `(z y)`, we need to look for the values of `+`, `x`, and `y`. For each variable, we first look in the local function scope, then in the outer file scope, then in the scope containing the built-in function definitions. We will find `x` in the function scope, `y` in file scope, and `+` in built-in scope (`+` is nothing special, it's a regular variable which has a function as its value). The environment data structure needs to be designed to allow this search.

The environment consists of individual *frames*, tables that contain the values for a single scope. We have one scope for built-in functions, the top-level (file) scope, and one scope for each function *call* and for *each evaluation* of a let-expression. Every frame is linked to the frame representing the *syntactically enclosing* scope. When looking up the value of a variable, we traverse these links to frames for enclosing scopes until we find the variable or until we reach the outermost scope.

If a variable name refers to a function, its value is represented as a *closure*. Unlike function pointers in C, which only point to the code, a closure consists of two pointers: a pointer to the code (a lambda expression) and a pointer to the environment in which the lambda expression was created.

As data structure for environments, you could either use a list of association lists or a list of hash tables. The implementation of environments is described below in the form of association lists in Scheme. This is also the implementation used in the Scheme interpreter written in Scheme. I suggest, though, that you use classes to define the environment data structure.

An association list is a lists of pairs with the first element (the `car`) of each pair being the key and the second element (the `cadr`) being the value. The Scheme function `assq` can be used to look up an entry in an association list. The function `assq` takes a key as first argument, an association list as second argument, and looks up the key in the association list with the comparison operation `eq?`. E.g., the call

```
(assq 'y '((x 17)
          (y 42)
          (z (closure
              (lambda (x) (+ x y))
              nil))))
```

will result in the list `(y 42)`.

Each scope in the program will be represented by such an association list. A complete environment is a list of association lists, from the innermost scope to the outermost scope. E.g., evaluating the Scheme definitions

```
(define (z x) (+ x y))
(define y 42)
(define x 17)
```

will result in the environment

```
((x 17)
 (y 42)
 (z (closure (lambda (x) (+ x y)) ...)))
```

where ... is the environment in which z is defined, i.e., a pointer to the entire environment. Since this results in a circular data structure, we need to use the assignment function `set-car!` for building it.

The circular data structure can be constructed in Scheme using `set-car!` as follows:

```
; Define the empty environment: a list containing an empty scope
(define env (list '()))
; Add a binding for z to the innermost scope
(set-car! env (cons
               (list 'z (list 'closure
                              '(lambda (x) (+ x y))
                              env))
               (car env)))
; Add a binding for y to the innermost scope
(set-car! env (cons
               (list 'y 42)
               (car env)))
; Add a binding for x to the innermost scope
(set-car! env (cons
               (list 'x 17)
               (car env)))
```

These `set-car!` operations are performed when evaluating the `define` special forms above.

The value associated with a variable can be updated using `set-cdr!`. For example, the redefinition (`define x 15`) or the assignment (`set! x 15`) would have the effect

```
(set-cdr! (assq 'x (car env)) (list 15))
```

When calling a function, we create a new (empty) association list for representing the function scope and `cons` it in front of the environment in which the function was defined (which is taken out of the closure for the function). Then we add variable definitions for the parameters to the new association list.

Given the function call (`z y`), the body of the function `z` will be evaluated in the environment

```
((x 42)
 (x 17)
 (y 42)
 (z (closure (lambda (x) (+ x y)) ...)))
```

where the first association list, `((x 42))`, contains the binding for the parameter `x` of function `z` to the value passed as argument.

A frame (association list) is added to the environment when entering a function body or when entering a `let` body. For each `define` that is encountered inside a function body or inside a `let` body, one binding is added to the front of the first frame in the environment.

The Scheme interpreter written in Scheme is started with the empty `global-environment`. However, conceptually, the environment also contains everything that was defined in the underlying Scheme

system. This way we can rely on the existing implementations for the data structures and primitive functions.

When implementing the interpreter in C++ or Java, we first need to construct a frame for built-in functions, then add all the built-in function definitions, then we construct the frame for user definitions and start processing the input.

A possible implementation in C++ or Java would be to define a class for implementing frames. The sample skeleton implementation is patterned after the Scheme implementation. For a better implementation, you could use a hash table that associates a value with a symbol name.

Evaluation

For evaluating Scheme programs, you will implement the two mutually recursive functions `eval` and `apply`. The main evaluation function, `eval`, takes two arguments, a Scheme expression and an environment. E.g., the call

```
(eval '(z y) env)
```

where `env` is defined as above, will result in the value 84.

The function `eval` needs to perform the following tasks:

- extend the environment for variable or function definitions,
- extend the environment for `let` expressions,
- update the environment for `set!` assignments,
- look up variables in the environment,
- handle the special forms `quote`, `lambda`, `begin`, `if`, and `cond` (including the `else` keyword in `cond`), and
- recursively call `apply` for function calls.

Note that some of the special forms as well as some of the built-in functions do not return a value.

A user-defined function is applied to its arguments using the function `apply`, which takes two arguments: a function and a list of arguments to the function. The function argument must be a closure. A closure is constructed by `eval` for a function definition or a lambda expression. It is represented as a list consisting of the symbol `'closure`, a lambda expression, and the environment in which the lambda expression was defined. BTW, since environments are circular structures, printing an environment or an entire closure will loop endlessly.

Before calling the function `apply`, its arguments are evaluated. The first argument must evaluate to a closure. The function `apply` then needs to perform the following tasks:

- extract the environment out of the closure;
- add a new frame to the environment that binds the parameters to the corresponding argument values;
- recursively call `eval` for the function body and the new environment.

The functions `eval` and `apply` as described above, should work correctly for most Scheme programs, including programs containing functions as parameters or functions as return values. What does not work are programs containing *special forms* (like `define`, `lambda`, `if`, etc.) other than the ones listed above.

For distinguishing between built-in functions and functions that were defined in Scheme, we need two different representations for closures. The Scheme interpreter written in Scheme uses lists of the form `(closure fun env)` for representing user-defined functions. For built-in functions, it uses the closure representation of the underlying Scheme48 implementation. For your interpreter in C++ or Java, you need two parse tree node classes, `Closure` and `Builtin`, for representing these different types of closures.

Top-Level Loop

Your Scheme interpreter `eval` should be run with a top-level loop such as the following:

```
(define (Scheme)
  (display "Scheme 4101> ")
  (let ((input (read)))
    (if (not (eof-object? input))
        (begin
          (write (eval input global-environment))
          (newline)
          (Scheme))
        (newline))))
```

This top-level loop could either be implemented directly in C++ or Java or read from a file of Scheme definitions. For this project, it will be easiest to implement this loop directly in your main function.

Administrative Stuff

You can compare the output of your interpreter with the behavior of `scheme48`.

Extend the `Makefile` appropriately and submit the program using `'make submit'` or `p_copy 2`. Make sure you submit a `README` file.