CS 131: Programming Languages, Spring 2003

Assignment 5
Scheme Interpreter

Due: Wednesday, March 5

Goals for this Assignment

1. Reading a language specification
2. Becoming familiar with Lisp/Scheme-like concrete syntax
3. Dealing with an interpreter for a realistically-sized language

Office Hours

The purpose of office hours is to provide you with a chance to discuss class-related topics, to ask questions about material you find particularly interesting or particularly confusing, and to provide comments about the class.

You can drop by Olin 1253 any time to see if the professor is around; times specifically set aside are Tuesday, Thursday, and Friday from 2:30–4pm. Grutors are available in the Terminal Room on Tuesdays 7–9pm and Sundays 2–4pm. Questions can also be sent to `cs131help@cs.hmc.edu` .

Instructions

To start this assignment, copy all the files from the directory `/cs/cs131/src/5`. The only file you should need to look at or modify is `scheme.sml`.

You will also need to refer to the *Revised*^{5} *Report on the Algorithmic Language Scheme*, available in PDF form on the assignments web page.

Please submit both your `scheme.sml` and `library.scm` files, using `cs131submit`.

Functions must have the exact names and types given in the assignment (though you are encouraged to define other helper functions as needed), and must be well-commented in order to allow the graders to maintain their sanity.

Introduction

Scheme is a functional language designed as a cleaned-up version of Lisp, e.g., to make it more uniform and statically-scoped. The core syntax of Scheme is quite simple. The *sval* nonterminal ranges over "Scheme values". (There's a little more to the actual Scheme language, such as floating-point values, but not much more.)

$$
\begin{array}{rcl}
\textit{sval} & ::= & \textit{integer} \\
 & & \textit{symbol} \\
 & & \textit{boolean} \\
 & & () \\
 & & \texttt{(}\ \textit{sval}\ \texttt{.}\ \textit{sval}\ \texttt{)} \\
 & & \\
\textit{boolean} & ::= & \texttt{\#t} \mid \texttt{\#f}
\end{array}
$$

The syntax is then extended slightly so that the user can write lists directly rather than in terms of the empty list `()` and cons pairs `( . )`; Thus, we can write `(a b 1)` and this gets interpreted by the compiler as the nested conses `(a . (b . (1 . ())))`. (This is exactly parallel to SML having bracket notation for lists instead of requiring the user to write `nil` and `::` everywhere).

Scheme and the Lisp languages are weird, because programs represented in terms of scheme values and the data they operate on at run time are also scheme values. Evaluation of a Scheme program is simply a process of turning one scheme value into another scheme value.

Scheme code is written in prefix, as a nested collection of lists. For example, the following program (which is also a list whose first element is the symbol `define` and whose third component is a list starting with the symbol `lambda`) defines the symbol `remove-duplicates` to be equal to a particular function, which when given a list argument `values`, tries to remove all the duplicate entries in that list.

```
(define remove-duplicates
  (lambda (values)
    (if (null? values)
      '()
      (let* ((recursion (remove-duplicates (cdr values))))
        (if (member (car values) recursion)
          recursion
          (cons (car values) recursion))))))
```

You can skim the Scheme definition (on the Assignments web page) to learn more about the language. It is suggested you look at it enough right now to figure out what `let*`, `if`, `car`, `cdr`, `member`, and `cons` are supposed to be doing.

A big issue in Scheme is quoting: treating scheme values as data rather than expressions to be evaluated. The expression `(quote` e`)` evaluates to e without further looking at e. Thus, for example, `(quote a)` evaluates to the symbol `a` by itself, whereas evaluating the

expression `a` looks up the definition of a (or a run-time error if `a` is undefined). Scheme parsers (including the one you were given) will let you write `'a` instead of `(quote a)` and `'(1 2)` instead of `(quote (1 2))`, but the two syntaxes are exactly equivalent.

Then the expression `(if #t 'a 'b)` evaluates to `a` (since `#t` is the syntax for logical truth and `'a` evaluates to `a`), while `(if #f 'a 'b)` evaluates to `b`. The expression `'(if #t 'a 'b)` evaluates to the list `(if #t (quote a) (quote b))`. The expression `(if #t a b)` evaluates to the value of `a`, if any.

There are a number of special list forms that get evaluated specially (e.g., expressions starting with `if`, `quote`, `set!`, `define`, `lambda`, `let*`, ...) but all others are simply treated as function applications. Thus `(+ 3 2)` applies the sum function to `3` and `2`, while `(f x)` applies the value of `f` to the value of `x` (where both should have been previously defined.)

One thing to be aware of is the slightly weird scoping in scheme. In addition to the "usual" environment, there is also a top-level environment checked when all else fails. A `define` adds a definition to this top-level environment, which gets changed imperatively. Thus code like

```
(define (f x) (+ x y))
(define y 1)
(f 3)
```

will work. Even though `y` wasn't defined when `f` was (and hence won't be in the closure), by the time `(f 3)` is evaluated `y` is in the top-level environment, where it will be found during the execution of `f`.

1 Scheme Programming (20%)

In the file `library.scm` are several helper functions that are built-in to most scheme implementations, but not the one you're building here. Add the missing definitions for `length`, and `member` as indicated. Your code should correspond to the description of these functions in R5RS (page 27).

You can test your code by, for example, running `guile` on turing; you can type in scheme expressions to be evaluated (without a trailing semicolon!) or enter `(load "library.scm")` to have it read your definitions. You can then type in tests such as `(append '(1 2) '(3 4))`

2 Primitives (40%)

The file `scheme.sml` has a fairly large amount of skeleton code provided for you. PLEASE START BY READING THROUGH THE COMMENTS TO SEE WHAT'S THERE!

The scheme language includes a number of built-in primitive operations such as `+`, `null?`, `newline`, etc. These are implemented in the interpreter in the function

```
applyPrim : string * sval list -> sval
```

which takes the name of the primitive, and a list of evaluated arguments.

Extend the interpreter to handle the following primitives: `list` (R5RS p. 27), `eqv?` (p. 17), `equal?` (p. 18), `not` (p. 25), `null?` (p. 26), `list?` (p. 26), `number?` (p. 21), `=` (p. 21), `+` (p. 22), `*` (p. 22), `-` (p. 22), and `pair?` (p. 26).

Then add at least three more primitives of your choice. Be sure they are added both to `applyPrim` and to the list `allPrims` (so that they appear in the top-level environment). Make sure to mark your extensions in your comments.

3 Main Interpreter (50%)

Complete the implementation of the function

```
eval : (sval ref) env * sval -> sval
```

to handle evaluation of `if` expressions (p. 10), the side-effect of `set!` expressions (p. 10), evaluating `lambda` expressions (p. 9) into closure values, evaluating `let*` expressions (p 11), and normal applications of closures. These are all marked in the code with `raise Unimplemented`.

If you choose to write helper functions outside of `eval`, note that all the functions from `applyPrim` to the end have to be connected with `and` so that `applyPrim` can call `main`.

Testing your code

Once your code compiles, you can test it by applying `Scheme.main` to a string containing the name of the input file. Several input files `*.scm` have been provided; you will likely want to create your own as well. These can be loaded into `guile` as well if you want to see what they are supposed to do.