INTERPRETERS AND MACROS Solutions

CS 61A

July 30, 2021

1 Interpreters

Interpreters Overview

An **interpreter** is essentially a program that understands and processes other programs.

The interpreter design we will be covering in 61A is the **Read-Eval-Print Loop**, which consists of the following steps:

1. Read the text input and load it into Python as a `Pair`
2. In each Scheme list, evaluate the operator (figure out if it's a +, car, etc.)
3. Recursively evaluate the operands (i.e. parameters) of the operation
4. Apply the operator to the operands and return the result

One of the challenges of designing interpreters is to represent the input in a way that the interpreter's language can understand. For example, since our Scheme interpreter is written in Python, we need to convert Scheme tokens to a Python representation. To achieve this, we will use the `Pair` object, which is essentially a Linked List that takes in `nil` instead of `Link.empty`.

As an example, `(list 1 2 3)` in Scheme can be converted to `Pair('list', Pair(1, Pair(2, Pair(3, nil))))`. This conversion is done in the Read step of the Read-Eval-Print loop. Note that nothing is evaluated in the Read step yet- everything is treated as just another token.

The following questions refer to the Scheme interpreter. Assume we're using the implementation seen in lecture and in the Scheme project.

1. What's the purpose of the read stage in a Read-Eval-Print Loop? For our Scheme interpreter, what does it take in, and what does it return?

   The read stage returns a representation of the code that is easier to process later in the interpreter by putting it in a new data structure. In our interpreter, it takes in a string of code, and outputs a Pair representing an expression (which is really just the same as a Scheme list).

2. What are the two components of the read stage? What do they do?

   The read stage consists of

   1. The lexer, which breaks the input string and breaks it up into tokens (individual characters or symbols)
   2. The parser, which takes that string of tokens and puts it into the data structure that the read stage outputs (in our case, a Pair).

3. Write out the constructor for the Pair object the read stage creates with the input string `(define (foo x) (+ x 1))`

   Pair("define", Pair(Pair("foo", Pair("x", nil)), Pair(Pair("+", Pair("x", Pair(1, nil))), nil)))

4. For the previous example, imagine we saved that Pair object to the variable `p`. How could we check that the expression is a `define` special form? How would we access the name of the function and the body of the function?

   We could check to see that it's a define special form by checking if `p.first == "define"`.
   We could get its name by accessing `p.second.first.first` and get the body of the function with `p.second.second.first`.

5. Circle or write the number of calls to `scheme_eval` and `scheme_apply` for the code below.

```
(if 1 (+ 2 3) (/ 1 0))
```

`scheme_eval`	1	3	4	6
`scheme_apply`	1	2	3	4

6 `scheme_eval`, 1 `scheme_apply`.

```
(or #f (and (+ 1 2) 'apple) (- 5 2))
```

`scheme_eval`	6	8	9	10
`scheme_apply`	1	2	3	4

8 `scheme_eval`, 1 `scheme_apply`.

```
(define (square x) (* x x))

(+ (square 3) (- 3 2))
```

`scheme_eval`	2	5	14	24
`scheme_apply`	1	2	3	4

14 `scheme_eval`, 4 `scheme_apply`.

```
(define (add x y) (+ x y))

(add (- 5 3) (or 0 2))
```

13 `scheme_eval`, 3 `scheme_apply`.

2 Macros

Macros Overview Whereas normal Scheme evaluation entails evaluating the operator, then evaluating the operands, before finally applying the operator on operands, macros evaluation involves three steps:

1. Evaluate the operator
2. Evaluate the body of the macro procedure without evaluating the operands
3. Evaluate the expression produced by the body and return the result.

Because the body is evaluated without evaluating the operands at first, macros are powerful and allow us to do more than scheme procedures, like implementing new special forms, control the order of evaluation, and more.

Below is a simple example of a macro. Note that even though we pass in `(print 'hello)` as an operands, we don't evaluate the expression and print right away. Instead we first evaluate the body of the macro procedure, and afterwards we evaluate the expression produced by the macro.

```
(define-macro (twice expr)
    (list 'begin expr expr)
)

scm> (twice (print 'hello))
hello
hello
```

When `twice` is called, it will first generate a Scheme list that looks like `(list 'begin '(print 'hello) '(print 'hello))` (the input is automatically quoted rather than evaluated).

The interpreter will then automaticaly call `eval` on this list of literals to treat it as if you had just typed it into the interpreter directly.

Quoting, Quasiquoting, Unquoting All Scheme expressions are lists except for atomic expressions like numbers and symbols; so call expressions and special forms are lists too; Example: (+ 1 2)

The `(quote expression)` special form, also denoted by a ´, simply returns `expression` - it does not evaluate it. This means we can write a Scheme expression and have the expression remain as an expression; if an expression is a call expression or special form, this means the expression will remain a list.

The `(quasiquote expression)` special form, `` ` ``, has the same effect as `quote`, except that any expression within `expression` can be unquoted by preceding it with `,` or the `unquote` special form; any unquoted expression is evaluated, whereas everything else within `expression` is not, as normal. Quasiquote and unquote are often used in the body of macro procedures to selectively evaluate certain parts.

`(eval expression)` is a procedure that simply evaluates its argument. Note that since `eval` is a procedure, `expression` is evaluated first before applying `eval`.

To build off of the `twice` example introduced earlier, it is possible to replace all `list` or `cons` operations with an expression involving quotes, quasiquotes, and unquotes to produce an identical result:

```
(define-macro (twice expr)
    `(begin ,expr ,expr)
)

scm> (twice (print 'hello))
hello
hello
```

1. What will Scheme output?

```
scm> (define x 6)
x

scm> (define y 1)
y

scm> '(x y a)
(x y a)

scm> `(,x ,y a)
(6 1 a)

scm> `(,x y a)
(6 y a)

scm> `(,(if (- 1 2) '+ '-) 1 2)
(+ 1 2)

scm> (eval `(,(if (- 1 2) '+ '-) 1 2))
3

scm> (define (add-expr a1 a2)
               (list '+ a1 a2))
add-expr

scm> (add-expr 3 4)
(+ 3 4)
```

```
scm> (eval (add-expr 3 4))
```

```
7
```

```
scm> (define-macro (add-macro a1 a2)
            (list '+ a1 a2))
```

```
add-macro
```

```
scm> (add-macro 3 4)
```

```
7
```

2. Implement `if-macro`, which behaves similarly to the `if` special form in Scheme but has some additional properties. Here's how the `if-macro` is called:
   `if <cond1> <expr1> elif <cond2> <expr2> else <expr3>`
   If cond1 evaluates to a truth-y value, expr1 is evaluated and returned. Otherwise, if cond2 evaluates to a truth-y value, expr2 is evaluated and returned. If neither condition is true, expr3 is evaluted and returned.

```
;Doctests
scm> (if-macro (= 1 0) 1 elif (= 1 1) 2 else 3)
2
scm> (if-macro (= 1 1) 1 elif (= 2 2) 2 else 3)
1
scm> (if-macro (= 1 0) (/ 1 0) elif (= 2 0) (/ 1 0) else 3)
3

(define-macro (if-macro cond1 expr1 elif cond2 expr2 else
   expr3)








)
```

```
(define-macro (if-macro cond1 expr1 elif cond2 expr2 else
   expr3)
    (list 'cond (list cond1 expr1)
                (list cond2 expr2)
                (list 'else expr3)))
```

Alternate solution with nested ifs:

```
(define-macro (if-macro cond1 expr1 elif cond2 expr2 else
   expr3)
    (list 'if cond1 expr1 (list 'if cond2 expr2 expr3)))
```

Alternate solution with quasiquoting:

```
(define-macro (if-macro cond1 expr1 elif cond2 expr2 else
   expr3)
    `(cond (,cond1 ,expr1)
           (,cond2 ,expr2)
           (else ,expr3)))
```

3. Could we have implemented `if-macro` using a function instead of a macro? Why or why not?

   Without using macros, the inputs would be evaluated when we evaluated the function call. This is problematic for two reasons:
   First, we only want to evaluate the expressions under certain conditions. If cond1 was false, we would not want to evaluate expr1. This might lead to errors!
   Secondly, some of the inputs to the call would be names which have no binding in the global frame. Elif, for example, is not supposed to be interpreted as a name but rather as a symbol. This would cause our code to error if we ran it as is!
   However, it is also possible to recreate a similar behavior to macros with a function by delaying the final evaluation. This makes it considerably more complicated to produce the desired behavior, since all inputs would have to be quoted and `eval` would have to be manually called on the result:

```
(define (if-macro-without-macro cond1 expr1 elif cond2 expr2
   else expr3)
  (list 'cond (list cond1 expr1)
              (list cond2 expr2)
              (list 'else expr3))
)

(eval (if-macro-without-macro '(= 1 0) '(/ 1 0) 'elif '(= 2 0)
    '(/ 1 0) 'else '3))
```

4. Implement `apply-twice`, which is a macro that takes in a call expression with a single argument. It should return the result of applying the operator to the operand twice.

```
;Doctests
scm> (define add-one (lambda (x) (+ x 1)))
add-one
scm> (apply-twice (add-one 1))
3
scm> (apply-twice (print 'hi))
hi
undefined

(define-macro (apply-twice call-expr)

    `(let ((operator ______________________________)

              (operand ______________________________))

              (________________________________________________________)))
```

```
(define-macro (apply-twice call-expr)
    `(let ((operator ,(car call-expr))
              (operand ,(car (cdr call-expr))))
           (operator (operator operand))))
```