Interpreters and Macros

CS 61A Group Mentoring

July 30, 2018

1 Let in Scheme

1. **let** is a special form in Scheme which allows you to create local bindings. Consider the example

   ```
   (let ((x 1)) (+ x 1))
   ```

   Here, we assign `x` to `1`, and then evaluate the expression `(+ x 1)` using that binding, returning `2`. However, outside of this expression, `x` would not be bound to anything.
   Each `let` special form has a corresponding lambda equivalent. The equivalent lambda expression for the above example is

   ```
   ((lambda (x) (+ x 1)) 1)
   ```

   The following line of code does not work. Why? Write the lambda equivalent of the `let` expressions.

   ```
   (let ((foo 3)
         (bar (+ foo 2)))
       (+ foo bar))
   ```

   Solution: The above function will error because it is equivalent to:
   `((lambda (foo bar) (+ foo bar)) 3 (+ foo 2))`

   In other words, foo has not been defined in the global frame. When bar is being assigned to `(+ foo 2)`, it will error. The assignment of foo to 3 happens in the lambda?s frame when it's called, not the global frame (this is relevant to the Scheme project – when the interpreter sees `lambda`, it will call a function to start a new frame).

   If we had the line `(define foo 3)` before the call to let, then it would return 8, because within let, foo would be 3 and bar would be `(+ 3 2)`, since it would use the foo in the Global frame.

2 Interpreters

The following questions refer to the Scheme interpreter. Assume we're using the implementation seen in lecture and in the Scheme project, as well as the Calculator subset seen in discussion.

2. 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?

Solution: 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).

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

Solution: 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).

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

Solution: Pair(”define”, Pair(Pair(”foo”, Pair(”x”, nil)), Pair(Pair(”+”, Pair(”x”, Pair(1, nil))), nil)))

5. 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?

Solution: 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`.

6. Write the number of calls to `scheme_eval` and `scheme_apply` for the code below.

```
scm> (+ 1 2)
3
```

Solution: 4 `scheme_eval`, 1 `scheme_apply`.

```
scm> (if 1 (+ 2 3) (/ 1 0))
5
```

Solution: 6 `scheme_eval`, 1 `scheme_apply`.

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

Solution: 8 `scheme_eval`, 1 `scheme_apply`.

```
scm> (define (square x) (* x x))
square
scm> (+ (square 3) (- 3 2))
10
```

Solution: 14 `scheme_eval`, 4 `scheme_apply`.

```
scm> (define (add x y) (+ x y))
add
scm> (add (- 5 3) (or 0 2))
2
```

Solution: 13 `scheme_eval`, 3 `scheme_apply`.

3 Macros

1. What will Scheme output?

```
scm> (define x 6)
```

Solution: `x`

```
scm> (define y 1)
```

Solution: `y`

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

Solution: `(x y a)`

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

Solution: `(6 1 a)`

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

Solution: `(6 y a)`

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

Solution: `(+ 1 2)`

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

Solution: `3`

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

Solution: `add-expr`

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

Solution: `(+ 3 4)`

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

Solution: `7`

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

Solution: `add-macro`

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

Solution: `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)






)
```

Solution:
```
(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?

Solution: 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!
Of course, we could have written out a `cond` or nested `if` expression instead of defining an `if-macro`. But the syntax for `if-macro` is more familiar, which is why we might want to do something like this!

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  ___________________________))

              (_____________________________________________________)))
```

Solution:

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