CS61A Lecture 28

Monday, November 4, 2019

Announcements

- Homework 6 and Lab 9 deadlines extended to today.
- Midterm 2 regrade requests due today.
- Homework 7 is due today.
- Cats composition revisions due next Tuesday.
- Scheme project due November 20.
  - Submit by the 19th for an early submission bonus point.
  - Complete the project by next Tuesday.
- Optional Project Fair on Sunday, December 15.

Interpreting Scheme

Today, we will discuss the Scheme project and the mindset you should have to correctly solve the project.

The interpreter has an `eval` function that can evaluate expressions. `eval` is a recursive function that has:

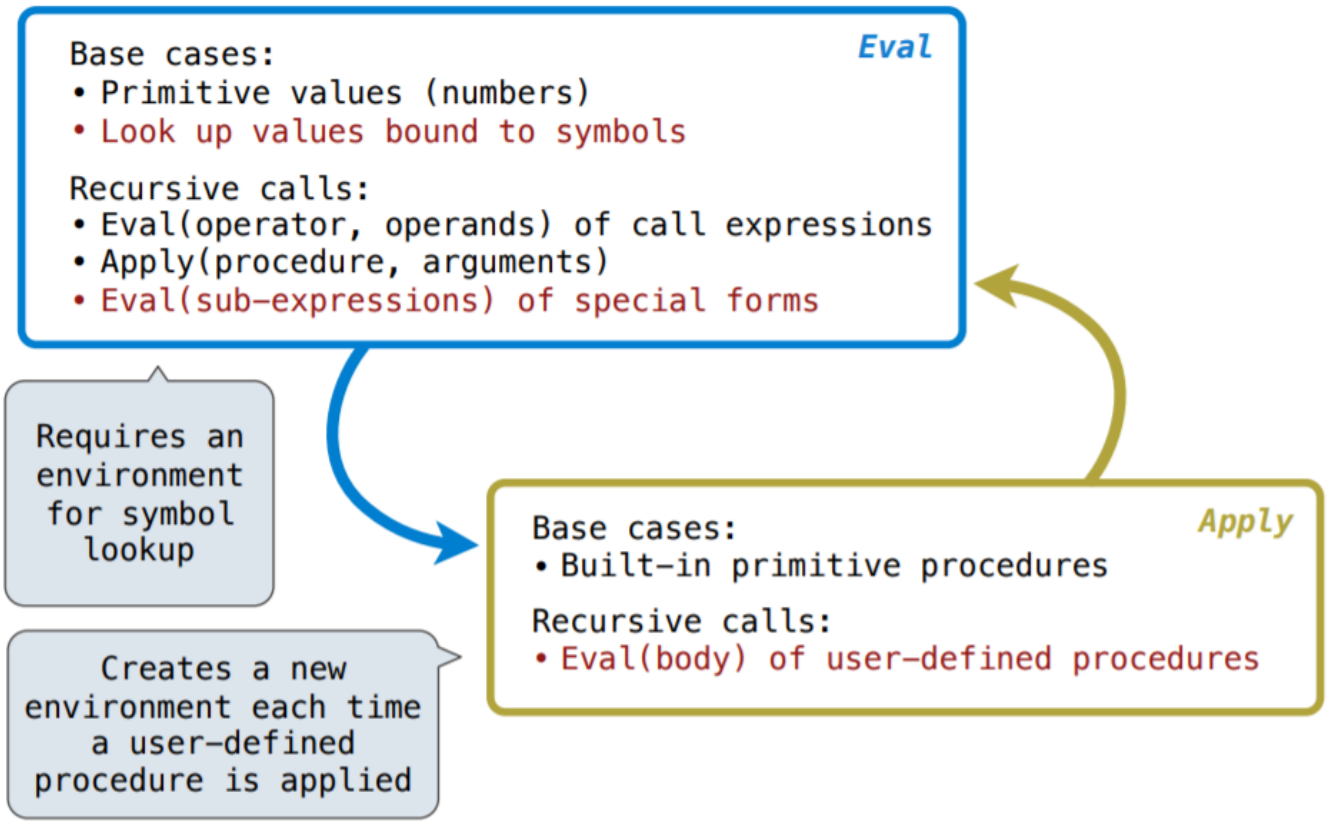
- base case: primitive values
- recursive calls:
  - `eval(operator,operands)` of call expressions
  - `apply(procedure,arguments)`

In the calculator language we implemented last Friday, we had both of these. But that's not enough for Scheme. We need to be able to look up symbols, special forms like `if` and `and` .

`apply` is another recursive function with:

- base cases: built-in primitive procediures
- recursive calls: `eval(body)` of user-defined procedures

As such, `eval` and `apply` are mutually recursive functions!



Special Forms

How can we evaluate special forms like `if` and `and`? There is the `scheme_eval` function that chooses behavior based on expression form:

- Symbols are looked up in the current environment.
- Self-evaluating expressions are returned as values (e.g. `1`, `#t`)
- All other legal expressions are represented as Scheme lists, called combinations (special forms are identified by the first element):
  - `(if ...)`
  - `(lambda (<formal parameters>) ...)`
  - `(define ...)`
  - `(<operator> ...)`

  Any combination that is not a known special form is a call expression.

So what does `scheme_eval` look like? Here's what it sort of looks like:

```
define scheme_eval(expr,parent,_=None):

    if <scheme_symbolp(expr):
        return  env.lookup(expr)
    elif self_evaluating(expr):
        return expr

    else:
        scheme_apply(expr,...)
```

Logical Forms

Logical forms may only evaluate some sub-expressions:

- `if` expression
- `and` and `or`
- `cond` expression

The way we'll do this is that we will define a Python function for each unique logical form that knows all the rules.

For example, the `do_if_function`:

- Evaluate the predicate.
- Choose a sub-expression: `<consequent>` or `<alternative>`.
- Evaluate the sub-expression to get the value of the whole expression.

Quotation

Your interpreter needs to handle quotation. The quote special form evaluates to the quoted expression, which is not evaluated. As such, the check for this happens in the parser function, and not the `eval` function.

```
> (quote (+ 1 2))
(+ 1 2)
```

The `<expression>` itself is the value of the whole quote expression.

Also `'<expression>` is shorthand for `(quote <expression>)`.

```
> (quote (1 2))
(1 2)
> '(1 2)
(1 2)
```

This checking happens in the `scheme_reader` file, which checks for the balancing of parentheses, and turns `'(list)` into `(quote (list))`.

Lambda Expressions

Lambda expressions evaluate to user-defined procedures:

```
(lambda (<formal parameters>) <body>)
```

And this is how the procedure is defined:

```
class LambdaProcedure:
    def __init__(self, formals, body, env):
    self.formals = formals
    self.body = body
    self.env = env
```

Frames and Environemnts

A frame represents an environment by having a parent frame.

Frames are Python instances with methods `lookup` and `define`.

In Project 4, frames do not hold return values.

```
>>> g = Frame(None)
>>> g
<Global Frame>
>>> f1 = Frame(g)
>>> f1
<{} -> <Global Frame>>
>>> g.define('y',3)
>>> g.define('z',5)
>>> g.lookup('y')
3
>>> f1.define('x',2)
>>> f1.define('z',4)
>>> f1
<{x: 2, z: 4} -> <Global Frame>>
>>> f1.lookup
2
>>> f1.lookup
4
```

Define Expressions

Define binds a symbol to a value in the first frame of the current environment.

```
(define <name> <expression>)
```

It works with these procedures:

1. Evaluate the `<expression>`
2. Bind `<name>` to its value in the current frame

Procedure definition is shorthand of define with a lambda expression.

```
(define (<name> <formal parameters>) <body>)
(define <name> (lambda (<formal parameters>) <body>))
```

Applying User-Defined Procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent in the `env` attribute of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame:

```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s)))))

(demo (list 1 2))
```