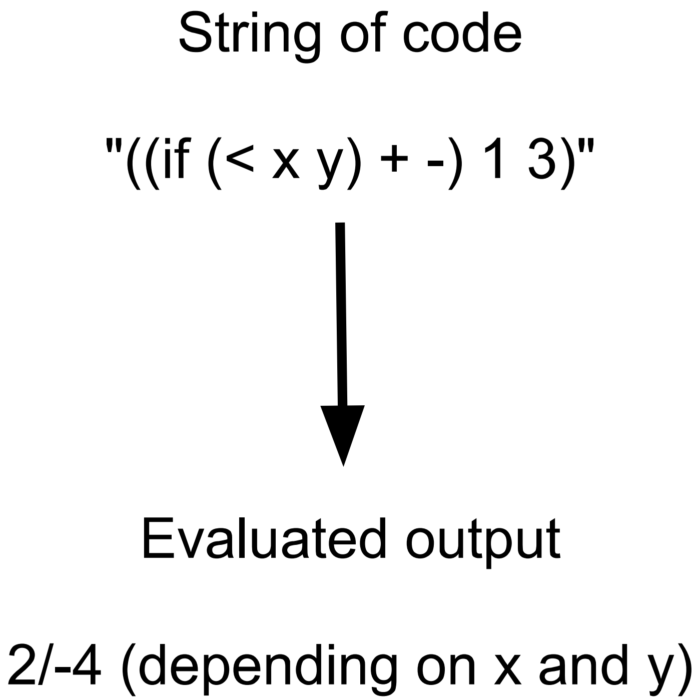
Interpreters

Initial thoughts

- I do not know what is on the final exam
- Historically, there are not many (if any) questions on interpreters

Purpose of an Interpreter



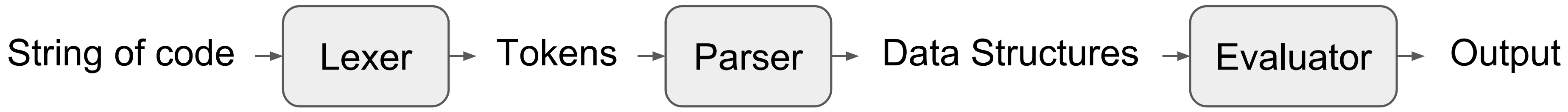
String of code

"((if (< x y) + -) 1 3)"

Evaluated output

2/-4 (depending on x and y)

Components of an Interpreter



String of code → Lexer → Tokens → Parser → Data Structures → Evaluator → Output

The Scheme Interpreter

Lexer - `scheme_tokens.py`

Parser - `scheme_reader.py`

Evaluator - `scheme.py`

What do we need to worry about?

~~Lexer~~

~~Parser~~

Evaluator

Steps of Evaluation

1. Evaluate the operator
2. Evaluate the operands
3. Apply the operator to the operands

General rule: Call apply once for every function call

Example Evaluation

`(+ 1 2 (* 3 4))`	1	0	Evaluate entire expression
`(+ 1 2 (* 3 4))`	2	0	Evaluate operator
`(+ 1 2 (* 3 4))`	5	0	Evaluate operands
`(+ 1 2 (* 3 4))`	6	0	Evaluate operator
`(+ 1 2 (* 3 4))`	8	0	Evaluate operands
`(+ 1 2 (* 3 4))`	8	1	Apply multiplication function
`(+ 1 2 12)`	8	2	Apply addition function

Final Question (Summer 2016)

How many calls to scheme_eval and scheme_apply in evaluating the following expression?

```
(- (* 6 11) 2 2 2)
```

Solution

```
(- (* 6 11) 2 2 2)
(- (* 6 11) 2 2 2)
(- (* 6 11) 2 2 2)
(- (* 6 11) 2 2 2)
(- (* 6 11) 2 2 2)
(- (* 6 11) 2 2 2)
    (- 66 2 2 2)
```

1	0
2	0
6	0
7	0
9	0
9	1
9	2

When does this system fail?

Special Forms!

- `(if (< 4 3) 1 2)`
  - We do not evaluate both the 1 and 2
  - Short circuiting is implemented at evaluation
- `(lambda (x) 4)`
  - We do not evaluate the body when defined
- Other forms: `and`, `or`, `cond`, `define`, `begin`

Final Question (Summer 2016)

How many calls to scheme_eval and scheme_apply in evaluating the following expressions?

```
(if (+ 0 (- 1 1)) (+ 4 5) 4)
```

```
((lambda (x) (* x x)) 57)
```

Solution

```
(if (+ 0 (- 1 1)) (+ 4 5) 4)
(if (+ 0 (- 1 1)) (+ 4 5) 4)
(if (+ 0 (- 1 1)) (+ 4 5) 4)
(if (+ 0 (- 1 1)) (+ 4 5) 4)
(if (+ 0 (- 1 1)) (+ 4 5) 4)
   (if (+ 0 0) (+ 4 5) 4)
      (if 0 (+ 4 5) 4)
      (if 0 (+ 4 5) 4)
      (if 0 (+ 4 5) 4)
```

1	0
2	0
5	0
8	0
8	1
8	2
9	2
12	2
12	3

Solution

```
 ((lambda (x) (* x x)) 57)
 ((lambda (x) (* x x)) 57)
 ((lambda (x) (* x x)) 57)
 ((lambda (x) (* x x)) 57)
 ((lambda (x) (* x x)) 57)
 ((lambda (x) (* x x)) 57)
 ((lambda (x) (* x x)) 57)
((lambda (x) (* 57 57)) 57)
```

1	0
2	0
3	0
3	1
4	1
5	1
7	1
7	2

More questions!

```
(begin (define (foo x) (+ x 3)) (foo (- 5 2)))
```

```
(define a 2)
(define (b) 3)
(let ((b a) (a (b))) ((lambda (x) (- x a b)) 5))
```

More solutions!

```
(begin (define (foo x) (+ x 3)) (foo (- 5 2)))
12 calls to eval, 3 calls to apply

(define a 2)
(define (b) 3)
(let ((b a) (a (b))) ((lambda (x) (- x a b)) 5))
15 calls to eval, 3 calls to apply
```