CS339: Abstractions and Paradigms for Programming

Metacircular Evaluator (Cont.)

Manas Thakur
CSE, IIT Bombay

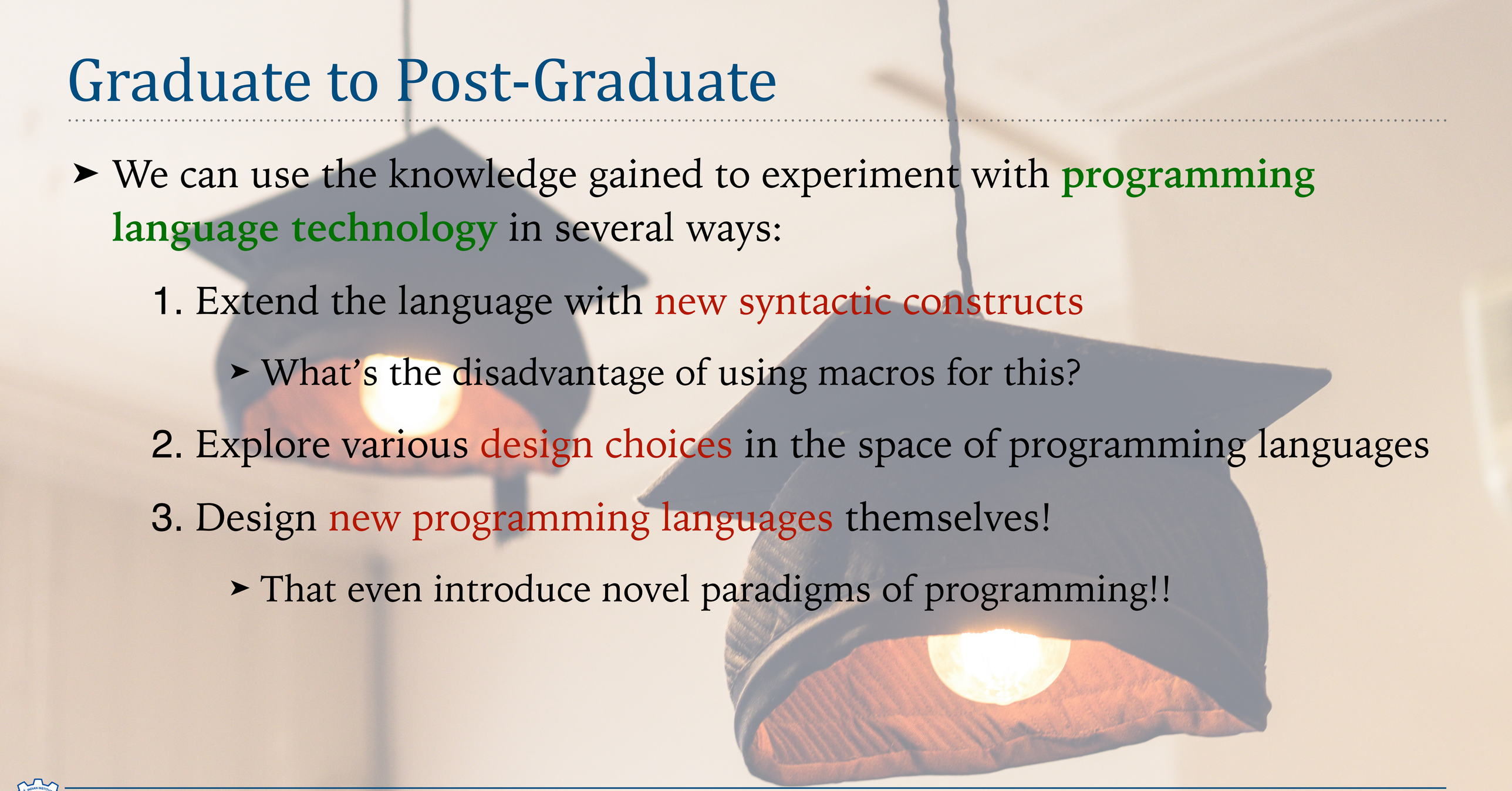
Autumn 2024

A Feat Accomplished

- What we have seen:
  - The kernel of an Eval/Apply Interpreter based on the Environment Model for a (good enough) subset of Scheme.
- Add the implementation of environment data structures:
  - We get a **working interpreter**!

Graduate to Post-Graduate

- We can use the knowledge gained to experiment with **programming language technology** in several ways:
  1. Extend the language with new syntactic constructs
     - What's the disadvantage of using macros for this?
  2. Explore various design choices in the space of programming languages
  3. Design new programming languages themselves!
     - That even introduce novel paradigms of programming!!



Exploring design choices: Eager vs Lazy Evaluation

Applicative Order and Normal Order

- Applicative order: All arguments evaluated before procedure application.
- Normal order: Evaluation of arguments delayed until the actual argument values are needed.
- What do we mean by *needed*?
  - Application of a primitive procedure
  - Printing the value on screen
  - As the predicate of a conditional
  - Value of a procedure operator (recall: we're first-class citizens!)

What about returning as a value?

Example

- Applicative order: Error
- Normal order: 1
- Scheme?
- Another useful example:

```
(define (try a b)
  (if (= a 0) 1 b))

> (try 0 (/ 1 0))
```

```
(define (unless condition usual-value exceptional-value)
  (if condition exceptional-value usual-value)

> (unless (= b 0)
          (/ a b)
          (begin (display “Exception”) 0))
```

More jargon: Strict vs Non-Strict Arguments

- If the body of a procedure is entered before evaluating an argument, then the procedure is ***non-strict*** in that argument, and vice-versa.
- Purely applicative-order language: All procedures are ***strict*** in each argument.
- Purely normal-order language: All compound procedures are non-strict in each argument, and primitive procedures may be strict or non-strict.
- **PCQ:** Example of a primitive procedure that can be non-strict?
  - Answer: cons!

More more jargon: Call by Xyz

- Call by value: Evaluate arguments before procedure application.
- Applicative order!
- Call by name: Delay evaluation until the argument is needed!
- Normal order!
- What is *lazy evaluation* then?

Flashback

```
(define (foo a)
  (define (square x) (* x x))
  (define (sum-squares x y)
    (+ (square x) (square y)))
  (sum-squares (+ a 1) (* a 2)))
```

```
(foo 5)
(sum-squares (+ 5 1) (* 5 2))
(+ (square 6) (square 10))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

Applicative-order evaluation

Normal-order evaluation

```
(foo 5)
(sum-squares (+ 5 1) (* 5 2))
(+ (square (+ 5 1)) (square (* 5 2)))
(+ (* (+ 5 1) (+ 5 1)) (* (* 5 2) (* 5 2)))
(+ (* 6 6) (* 10 10))
(+ 36 100)
136
```

Repeated computation!

Lazy Evaluation

- Lazy evaluation is the way to impart efficiency to normal-order evaluation, or in other words to non-strict parameters.
- Normal order + Memoization
- *Call by need* (confusing term, but acceptable)
- **Implementation:**
  - Instead of evaluation, arguments stored into a ***thunk***.
    - It's just an object (*aka* list in APP!).
  - Thunks are replaced with the evaluated value on first use.

Last jargon:
The opposite of lazy evaluation is *eager evaluation*.

Creating a thunk

- You already told it's a list!
- What's needed to evaluate (an argument) at a later point of time?
  - The (right) environment!
- So now we can create a thunk as simply as:

```
(define (create-thunk exp env)
  (list 'thunk exp env))
```

- With corresponding changes in `eval/apply` to decide when to create a thunk and when to evaluate the expression in a thunk.

Changes in *eval*

- We need an additional layer to actually evaluate the operator arguments:

```
(define eval
  (lambda (exp env)
    (cond ((number? exp) exp)
          ((application? exp)
           (apply (actual-eval (car exp) env) (cdr exp) env))
          (else (error "unknown expression type")))))

(define (actual-eval exp env)
  (eval-thunk (eval exp env)))

(define (eval-thunk obj)
  (cond ((thunk? obj)
         (actual-eval (cadr thunk) (caddr thunk)))
        (else obj)))
```

- Notice the purpose of recursion.

Changes in *apply*

- We need to evaluate the arguments of primitive procedures and *thunkize* (delay) the arguments of compound ones:

```
(define (apply
  (lambda (proc args env)
    (cond ((primitive-op? proc)
           (apply-prim-op proc (evallist-eager args env)))
          ((compound-op? proc)
            (eval (caddr proc)
                  (extend-environ (cadr proc)
                                  (evallist-lazy args env)
                                  (cadddr proc))))))))

(define evallist-eager
  (lambda (args env)
    (cond ((eq? args '()) '())
          (else
            (cons (actual-eval (car l) env)
                  (evallist-eager (cdr l) env))))))

(define evallist-lazy
  (lambda (args env)
    (cond ((eq? args '()) '())
          (else
            (cons (create-thunk (car l) env)
                  (evallist-lazy (cdr l) env))))))
```

Last one: Conditional

- We need to *actually* evaluate the predicate:

```
(define evalcond
  (lambda (clauses env)
    (cond ((eq? clauses '()) '())
          ((eq? (caar clauses) 'else)
             (eval (cadar clauses) env))
          ((false? (actual-eval (caar clauses) env))
             (evalcond (cdr clauses) env))
          (else
            (eval (cadar clauses) env)))))
```

But that was *call by name*, not *need*

- Recall **memoization**: Evaluate thunk on first use and replace it with the evaluated value.

```
(define (evaluated-thunk? obj) ((eq? (car obj) 'evaluated-thunk)))
(define (eval-thunk obj)
  (cond ((thunk? obj)
         (let ((result (actual-eval (cadr obj) (caddr obj))))
           (set-car! obj 'evaluated-thunk)
           (set-car! (cdr obj) result)
           (set-cdr! (cdr obj) '())
           result))
        ((evaluated-thunk? obj) (cadr obj))
        (else obj)))
```

- No change in `create-thunk`.
- Auxiliary procedures `set-car!` and `set-cdr!` mutate the `car` and the `cdr` of a list, respectively.

Laziness in action

Now Scheme can be magical!

Streams as lazy lists

- Now that we have laziness built in, do streams need to be something special any more?
  - The only change required: Make `cons` lazy.
  - An easy way: Switch to pairs being procedures!
- **Advantages:**
  - No explicit `delay/force` required.
    - Why were they bad?
  - No special `stream-*` procedures need to be defined.
    - Why is that good?
  - Even lazier:
    - Both `car` and `cdr` are lazy now!
  - **What would be the best?**

```
(define (cons x y)
  (lambda (m) (m x y)))
(define (car z)
  (z (lambda (p q) p)))
(define (cdr z)
  (z (lambda (p q) q)))
```

The End is Near

- Special classes on Haskell
  - Today 2 PM (F C Kohli Auditorium, KR Building)
  - Tomorrow 10:30 AM (usual, LH 101)
- Pending marks and queries
  - Throughout this week
- Last APP class of 2024
  - Thu, Nov 7, 11:30 AM, LH 101
- Final exam
  - Tue, Nov 12, 1:30 PM