CS 61A Fall 2023

Interpreters, More Scheme

Discussion 11: November 8, 2023

Note: For extended explanations of the concepts on this discussion, feel free to look at the **Appendix** section on the back of the worksheet.

Interpreters

Q1: From Pair to Expression

Recall that our Scheme interpreter is written in Python. The Python data structure that we use to represent Scheme lists is the `Pair` class, which is almost identical to the `Link` class you have already encountered.

Write out the Scheme expression with proper syntax that corresponds to the following `Pair` constructor calls.

```
>>> Pair('+', Pair(1, Pair(2, Pair(3, Pair(4, nil)))))
```

```
> (+ 1 2 3 4)
```

```
>>> Pair('+', Pair(1, Pair(Pair('*', Pair(2, Pair(3, nil))), nil)))
```

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

```
>>> Pair('and', Pair(Pair('<', Pair(1, Pair(0, nil))), Pair(Pair('/', Pair(1, Pair(0, nil
    ))), nil)))
```

```
> (and (< 1 0) (/ 1 0))
```

Box and pointers solutions Video walkthrough

Scheme Eval/Apply

Scheme evaluates an expression by obeying the following evaluation rules:

- Call `scheme_eval` on the input expression
  - If the input expression is self-evaluating (e.g. number, boolean), output that value.
  - If the input expression is a special form, follow the evaluation rules for that special form.
  - Otherwise, the input expression is a call expression, and you follow the rules of evaluation for call expressions.

The tricky part of this is the evaluation rules for compound expressions (call expressions and special forms). As an example, let's look at the evaluation rules for call expressions.

Here's how you learned how to evaluate a Scheme call expression:

- Evaluate the operator to get a procedure.
- Evaluate each operand to get arguments.
- Apply the procedure to the arguments.

And here's how the Scheme interpreter handles it in Python:

- Call `scheme_eval` on the operator to get a procedure.
- Call `scheme_eval` on each operand to get arguments.
- Call `scheme_apply` to apply the procedure to the arguments.

Notice the similarities here? Every time we need to evaluate some subexpression in Scheme, the underlying Python implementation calls `scheme_eval` recursively on that call expression!

Let's see if you can extend this to other special forms.

Q2: Connecting Scheme to Python

Recall the rules for how you handle a `define` special form in Scheme:

> To evaluate `(define <symbol> <expr>)`
>
> - Evaluate `<expr>` to get a value.
> - Bind that value to `<symbol>` in the current frame.

Now write out how the Scheme interpreter handles `(define <symbol> <expr>)` using `scheme_eval` and/or `scheme_apply`.

To evaluate `(define <symbol> <expr>)` * Call `scheme_eval` on `<expr>` to get a value. * Bind that value to `<symbol>` in the current frame.

Let's do `and` now. Write out the rules for how you handle an `and` special form in Scheme:

- Evaluate the first operand. If its value is false, immediately return that value.
- Do the same thing for each subsequent operand of the expression.
- If you get to the final operand, evaluate and return its value.

Now write out how the Scheme interpreter handles an `and` special form using `scheme_eval` and/or `scheme_apply`.

- Call `scheme_eval` on the first operand. If its value is false, immediately return that value.
- Do the same thing for each subsequent operand of the expression.
- If you get to the final operand, call `scheme_eval` on it and return its value.

Q3: Counting Eval and Apply

How many calls to `scheme_eval` and `scheme_apply` would it take to evaluate each of the following expressions?

Take the following Scheme expression as an example: `(* 2 (+ 3 4))`

To evaluate this entire expression, we…

- Call `scheme_eval` on `(* 2 (+ 3 4))`
  - Call `scheme_eval` on `*`
  - Call `scheme_eval` on `2`
  - Call `scheme_eval` on `(+ 3 4)`
    - Call `scheme_eval` on `+`
    - Call `scheme_eval` on `3`
    - Call `scheme_eval` on `4`
    - Call `scheme_apply` to apply `+` to `(3 4)` → `(+ 3 4)` evaluates to `7`
  - Call `scheme_apply` to apply `*` to `(2 7)` → `(* 2 (+ 3 4))` evaluates to `14`

In total, there are 7 calls to `scheme_eval` and 2 calls to `scheme_apply` to get our final result of `14`. A visualization of these specific calls are shown below, where each underline is a call to `scheme_eval` and each overline is a call to `scheme_apply`:

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

For this particular prompt please list out the inputs to `scheme_eval` and `scheme_apply`.

4 calls to eval: 1 for the entire expression, and then 1 each for the operator and each operand.

1 call to apply the addition operator.

Explicity listing out the inputs we have the following for `calc_eval`: , '+', 1, 2. `calc_apply` is given '+' for fn and (1 2) for args.

A note is that (+ 1 2) corresponds to the following Pair, Pair('+', Pair(1, Pair(2, nil))) and (1 2) corresponds to the Pair, Pair(1, Pair(2, nil)).

```
scm> (+ 2 4 6 8)
```

6 calls to eval: 1 for the entire expression, and then 1 each for the operator and each operand.

1 call to apply the addition operator.

```
scm> (+ 2 (* 4 (- 6 8)))
```

10 calls to eval: 1 for the whole expression, then 1 for each of the operators and operands. When we encounter another call expression, we have to evaluate the operators and operands inside as well.

3 calls to apply the function to the arguments for each call expression.

```
scm> (and 1 (+ 1 0) 0)
```

7 calls to eval: 1 for the whole expression, 1 for the first argument, 1 for `(+ 1 0)`, 1 for the `+` operator, 2 for the operands to plus, and 1 for the final 0. Notice that `and` is a special form so we do not run `calc_eval` on it.

1 calls to apply to evaluate the `+` expression.

```
scm> (and (+ 1 0) (< 1 0) (/ 1 0))
```

9 calls to eval: 1 for the whole expression, 1 for `(+ 1 0)`, 1 for the `+` operator, 2 for the operands to plus, and then 4 total for `(< 1 0)` (following the same breakdown as `(+ 1 0)`). Note that `and` is a special form so we do not run `calc_eval` on it, and also note that since `and` looks for the first false-y value (which in Scheme is just `#f`), it will return the `#f` without doing anything for the `(/1 0)`.

2 calls to apply, 1 for `+` and 1 for `<`.

Video Walkthrough

More Scheme

Q4: Reverse

Write the procedure `reverse`, which takes in a list `lst` and outputs a reversed list.

Hint: you may find the built-in append procedure useful.

```
(define (reverse lst)
    (if (null? lst)
        nil
        (append
            (reverse (cdr lst))
            (list (car lst))))
)
```

Q5: Longest Increasing Subsequence

Write the procedure `longest-increasing-subsequence`, which takes in a list `lst` and returns the longest subsequence in which all the terms are increasing. *Note: the elements do not have to appear consecutively in the original list.* For example, the longest increasing subsequence of `(1 2 3 4 9 3 4 1 10 5)` is `(1 2 3 4 9 10)`. Assume that the longest increasing subsequence is unique.

Hint: The built-in procedures length and filter might be helpful to solving this problem.

```
; helper function
; returns the values of lst that are bigger than x
; e.g., (larger-values 3 '(1 2 3 4 5 1 2 3 4 5)) --> (4 5 4 5)
(define (larger-values x lst)
    (filter (lambda (v) (> v x)) lst))

(define (longest-increasing-subsequence lst)
    (if (null? lst)
        nil
        (begin
            (define first (car lst))
            (define rest (cdr lst))
            (define large-values-rest
                (larger-values first rest))
            (define with-first
                (cons
                    (car lst)
                    (longest-increasing-subsequence large-values-rest)))
            (define without-first
                (longest-increasing-subsequence rest))
            (if (> (length with-first) (length without-first))
                with-first
                without-first))))

(expect (longest-increasing-subsequence '()) ())
(expect (longest-increasing-subsequence '(1)) (1))
(expect (longest-increasing-subsequence '(1 2 3)) (1 2 3))
(expect (longest-increasing-subsequence '(1 9 2 3)) (1 2 3))
(expect (longest-increasing-subsequence '(1 9 8 7 6 5 4 3 2 3)) (1 2 3))
(expect (longest-increasing-subsequence '(1 9 8 7 2 3 6 5 4 5)) (1 2 3 4 5))
(expect (longest-increasing-subsequence '(1 2 3 4 9 3 4 1 10 5)) (1 2 3 4 9 10))
```

Q6: Cons All

Implement `cons-all`, which takes in an element `first` and a list of lists `rests`, and adds `first` to the beginning of each list in `rests`:

```
scm> (cons-all 1 '((2 3) (2 4) (3 5)))
((1 2 3) (1 2 4) (1 3 5))
```

You may find it helpful to use the built-in map procedure.

```
(define (cons-all first rests)
  (map (lambda (rest) (cons first rest)) rests)
)
```

Q7: List Change

Implement the `list-change` procedure, which lists all of the ways to make change for a positive integer `total` amount of money, using a list of currency denominations, which is sorted in descending order. The resulting list of ways of making change should also be returned in descending order.

To make change for 10 with the denominations (25, 10, 5, 1), we get the possibilities:

```
10
5, 5
5, 1, 1, 1, 1, 1
1, 1, 1, 1, 1, 1, 1, 1, 1, 1
```

To make change for 5 with the denominations (4, 3, 2, 1), we get the possibilities:

```
4, 1
3, 2
3, 1, 1
2, 2, 1
2, 1, 1, 1
1, 1, 1, 1, 1
```

Therefore, your function should behave as follows for these two inputs

```
scm> (list-change 10 '(25 10 5 1))
((10) (5 5) (5 1 1 1 1 1) (1 1 1 1 1 1 1 1 1 1))
scm> (list-change 5 '(4 3 2 1))
((4 1) (3 2) (3 1 1) (2 2 1) (2 1 1 1) (1 1 1 1 1))
```

You may want to use `cons-all` in your solution.

You may also find the built-in append procedure useful.

```
;; List all ways to make change for TOTAL with DENOMS
(define (list-change total denoms)
  (cond (( = total 0) (list nil))
        ((or (< total 0) (null? denoms)) nil)
        (else (let ((denom (car denoms)) (rest-denoms (cdr denoms)))
                    (append (cons-all denom (list-change (- total denom) denoms))
                            (list-change total rest-denoms)))))
)
```

Appendix: Explanation of Material

Interpreters

An interpreter is a program that allows you to interact with the computer in a certain language. It understands the expressions that you type in through that language, and performs the corresponding actions in some way, usually using an underlying language.

In Project 4, you will use Python to implement an interpreter for Scheme. The Python interpreter that you've been using all semester is written (mostly) in the C programming language. The computer itself uses hardware to interpret machine code (a series of ones and zeros that represent basic operations like adding numbers, loading information from memory, etc).

When we talk about an interpreter, there are two languages at work:

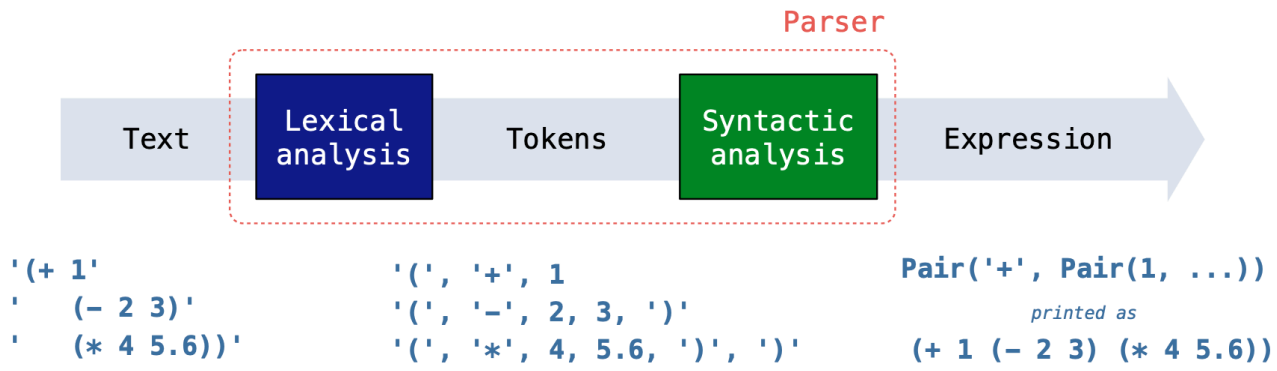
1. **The language being interpreted/implemented.** For Project 4, we are interpreting the Scheme language.
2. **The underlying implementation language.** For Project 4, we will implement an interpreter for Scheme using Python.

REPL

Many interpreters use a Read-Eval-Print Loop (REPL). This loop waits for user input, and then processes it in three steps:

- **Read:** The interpreter takes the user input (a string) and passes it through a parser. The parser processes the input in two steps:
  - The *lexical analysis* step turns the user input string into tokens that are like "words" of the implemented language. Tokens represent the **smallest** units of information.
  - The *syntactic analysis* step takes the tokens from the previous step and organizes them into a data structure that the underlying language can understand. For our Scheme interpreter, we create a `Pair` object (similar to a Linked List) from the tokens to represent the original call expression.
    - The first item in the `Pair` represents the operator of the call expression. The subsequent items are the operands of the call expression, or the arguments that the operator will be applied to. Note that operands themselves can also be nested call expressions.

Below is a summary of the read process for a Scheme expression input:



- **Eval:** Mutual recursion between `eval` and `apply` evaluate the expression to obtain a value.
  - `eval` takes an expression and evaluates it according to the rules of the language. Evaluating a call expression involves calling `apply` to apply an evaluated operator to its evaluated operands.

  - `apply` takes an evaluated operator, i.e., a function, and applies it to the call expression's arguments. Apply may call `eval` to do more work in the body of the function, so `eval` and `apply` are *mutually recursive*.
- **Print:** Display the result of evaluating the user input.

Here's how all the pieces fit together: