CS 61A
Spring 2023

Interpreters, SQL

Discussion 12: April 19, 2023

Interpreters

Recall the calculator interpreter you implemented in Lab 12. The following is an extension of that prompt.

Q1: Counting Eval and Apply

How many calls to `calc_eval` and `calc_apply` would it take to evaluate each of the following Calculator expressions?

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

For this particular prompt please list out the inputs to `calc_eval` and `calc_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

Q2: From Pair to Calculator

Write out the Calculator 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

SQL

SQL is an example of a declarative programming language. Statements do not describe computations directly, but instead describe the desired result of some computation. It is the role of the query interpreter of the database system to plan and perform a computational process to produce such a result.

> For this discussion, you can test out your code at sql.cs61a.org. The **records** table should already be loaded in. We can use a **SELECT** statement to create tables. The following statement creates a table with a single row, with columns named "first" and "last":

```
sqlite> SELECT "Ben" AS first, "Bitdiddle" AS last;
Ben|Bitdiddle
```

Given two tables with the same number of columns, we can combine their rows into a larger table with UNION:

```
sqlite> SELECT "Ben" AS first, "Bitdiddle" AS last UNION
...> SELECT "Louis", "Reasoner";
Ben|Bitdiddle
Louis|Reasoner
```

We can SELECT specific values from an existing table using a FROM clause. This query creates a table with two columns, with a row for each row in the records table:

```
sqlite> SELECT name, division FROM records;
Alyssa P Hacker|Computer
...
Robert Cratchet|Accounting
```

The special syntax SELECT * will select all columns from a table. It's an easy way to print the contents of a table.

```
sqlite> SELECT * FROM records;
Alyssa P Hacker|Computer|Programmer|40000|Ben Bitdiddle
...
Robert Cratchet|Accounting|Scrivener|18000|Eben Scrooge
```

We can choose which columns to show in the first part of the SELECT, we can filter out rows using a WHERE clause, and sort the resulting rows with an ORDER BY clause. In general the syntax is:

```
SELECT [columns] FROM [tables]
WHERE [condition] ORDER BY [criteria];
```

For instance, the following statement lists all information about employees with the "Programmer" title.

```
sqlite> SELECT * FROM records WHERE title = "Programmer";
Alyssa P Hacker|Computer|Programmer|40000|Ben Bitdiddle
Cy D Fect|Computer|Programmer|35000|Ben Bitdiddle
```

The following statement lists the names and salaries of each employee under the accounting division, sorted in descending order by their salaries.

```
sqlite> SELECT name, salary FROM records
...> WHERE division = "Accounting" ORDER BY salary desc;
Eben Scrooge|75000
Robert Cratchet|18000
```

Note that all valid SQL statements must be terminated by a semicolon (;). Additionally, you can split up your statement over many lines and add as much whitespace as you want, much like Scheme. But keep in mind that having consistent indentation and line breaking does make your code a lot more readable to others (and your future self)!

SQL Practice

The next part of the discussion will refer to the table **records**, which stores information about the employees at a small company.

records

name	division	title	salary	supervisor
Ben Bitdiddle	Computer	Wizard	60000	Oliver Warbucks
Alyssa P Hacker	Computer	Programmer	40000	Ben Bitdiddle
Cy D Fect	Computer	Programmer	35000	Ben Bitdiddle
Lem E Tweakit	Computer	Technician	25000	Ben Bitdiddle
Louis Reasoner	Computer	Programmer Trainee	30000	Alyssa P Hacker
Oliver Warbucks	Administration	Big Wheel	150000	Oliver Warbucks
Eben Scrooge	Accounting	Chief Accountant	75000	Oliver Warbucks
Lana Lambda	Administration	Executive Director	610000	Lana Lambda

Q3: Oliver Employees

Write a query that outputs the names of employees that Oliver Warbucks directly supervises.

```
SELECT name FROM records WHERE supervisor = "Oliver Warbucks";
```

Q4: Self Supervisor

Write a query that outputs all information about employees that supervise themselves.

```
SELECT * FROM records WHERE name = supervisor;
```

Q5: Rich Employees

Write a query that outputs the names of all employees with salary greater than 50,000 in alphabetical order.

```
SELECT name FROM records WHERE salary > 50000 ORDER BY name;
```

Q6: Raises

This last question refers to the following **salaries** table:

salaries

name	salary2022	salary2023
Ben Bitdiddle	60000	80000
Alyssa P Hacker	40000	80000
Cy D Fect	35000	74000
Lem E Tweakit	25000	28000
Louis Reasoner	30000	30000
Oliver Warbucks	150000	120000
Eben Scrooge	75000	76000
Robert Cratchet	18000	20000
Lana Lambda	610000	610000

Write a query that outputs the names of the top 3 employees with the largest salary raises from 2022 to 2023 along with their corresponding salary raises, ordered from largest to smallest raise.

```
SELECT name, salary2023 - salary2022 FROM salaries ORDER BY salary2023 - salary2022 DESC
    LIMIT 3;
```

Additional Exam Practice: Scheme Lists

Q7: 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` (documentation) and `filter` (documentation) 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))
```