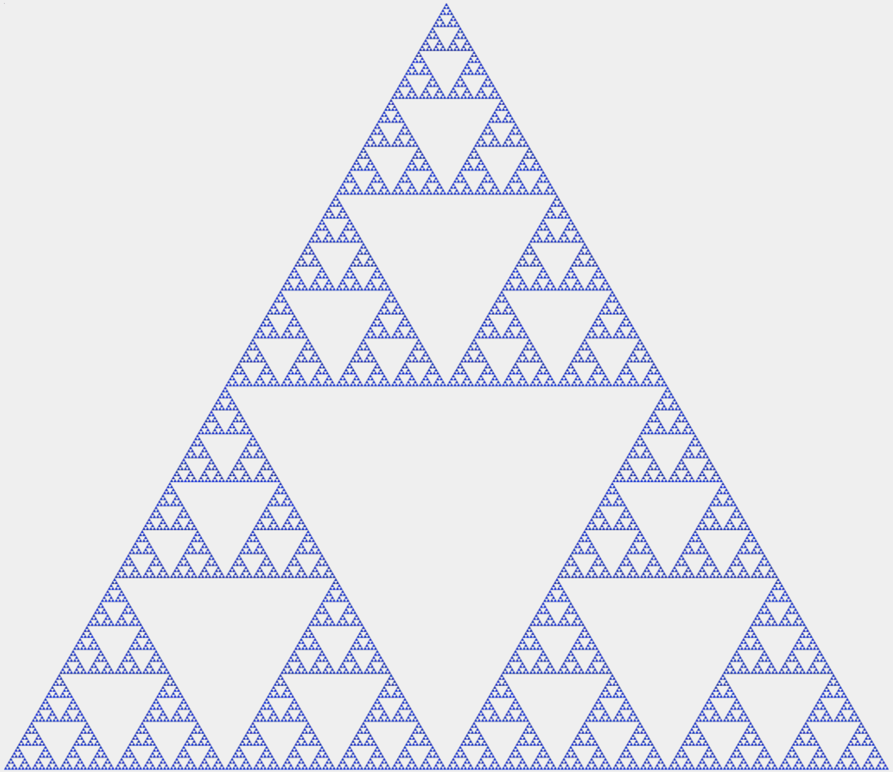
# Making Use of the Turing Completeness of SQL

# CTEs

```
WITH dan_brown_books AS (
  SELECT title FROM books WHERE author = ‘Dan Brown’
)
SELECT COUNT(*) FROM dan_brown_books;
```

# Sierpinski's Triangle



# Using math to generate Sierpinski's Triangle

```
7|*
6|**
5|* *
4|****
3|*   *
2|**  **
1|* * * *
0|********
  --------
  01234567
```

# Turning this into a SQL query.

1. Generate one SQL row for each cell in the grid.
2. Assign a string to each cell based on the bitwise AND of the row and column.
3. Form the grid by concatenating the string for every cell together.

# Generate the cells

```
> SELECT r, c FROM generate_series(0, 63) rows(r)
  CROSS JOIN generate_series(0, 63) cols(c);
```

```
 r  | c
----+----
  0 |  0
  0 |  1
  0 |  2
... |...
  0 | 63
  1 |  0
  1 |  1
  1 |  2
... |...
```

# Mark the points

```
> WITH points AS (
  SELECT r, c FROM generate_series(0, 63) rows(r)
  CROSS JOIN generate_series(0, 63) cols(c)
) SELECT r, c, CASE WHEN r & c != 0 THEN '  ' ELSE '**' END AS marker
  FROM points;

 r  | c  | marker
----+----+--------
  0 |  0 | **
  0 |  1 | **
... |... | ...
  1 |  1 |
  1 |  2 | **
  1 |  3 |
... |... | ...
```

# Combine each line of the output.

```
> WITH points AS (
    SELECT r, c FROM generate_series(0, 63) rows(r)
    CROSS JOIN generate_series(0, 63) cols(c)
  ), marked_points AS (
    SELECT r, c, CASE WHEN r & c != 0 THEN '  ' ELSE '**' END AS marker
    FROM points
    ORDER BY r DESC, c ASC
  ), rows AS (
    SELECT r, string_agg(marker, '') AS line
    FROM marked_points
    GROUP BY r
    ORDER BY r DESC
  ) SELECT string_agg(line, E'\n') FROM rows;
```

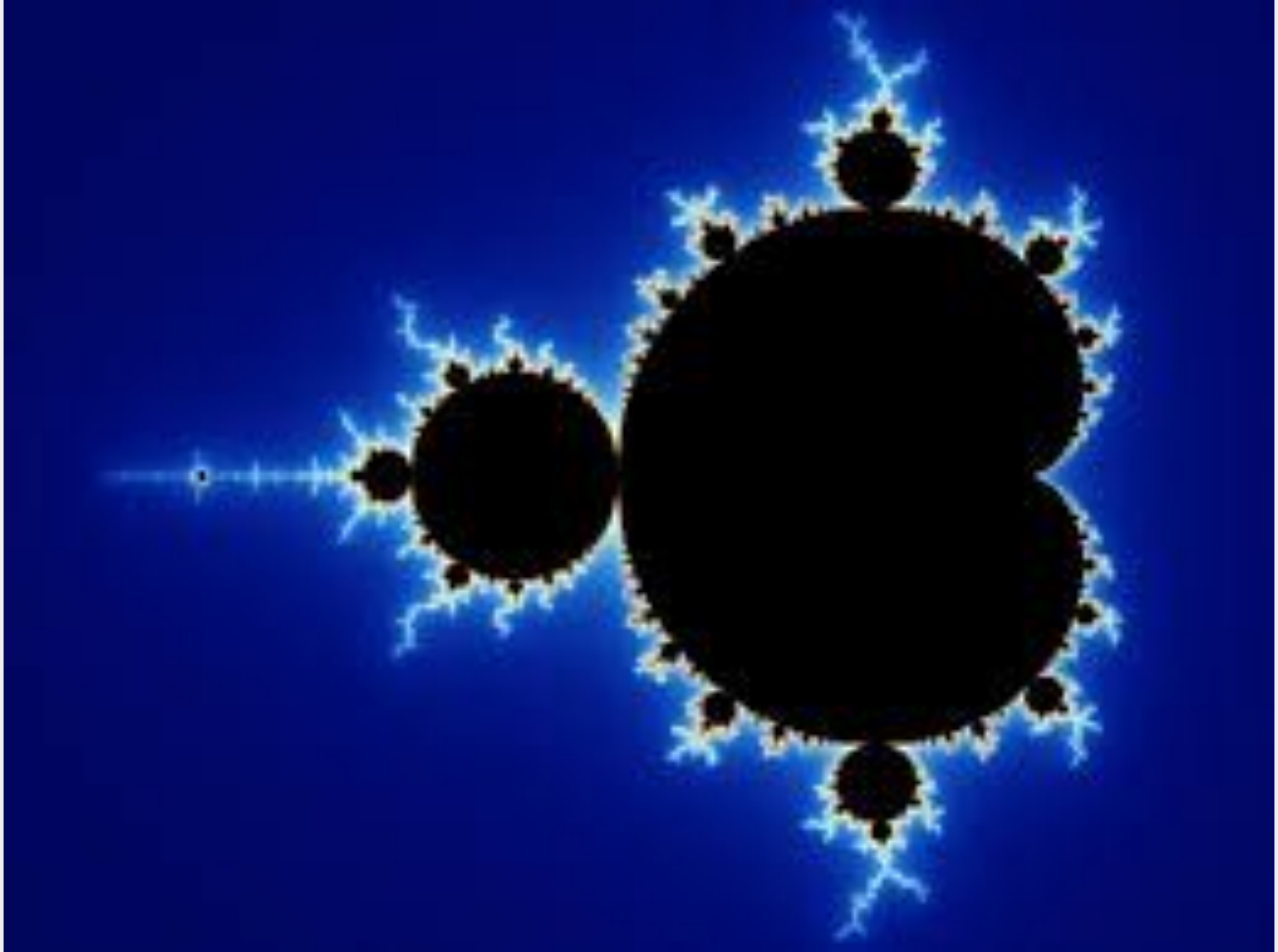
# The Output:

```
**
****
**  **
********
**      **
****    ****
**  **  **  **
****************
**              **
****            ****
**  **          **  **
********        ********
**      **      **      **
****    ****    ****    ****
**  **  **  **  **  **  **  **
********************************
**                              **
****                            ****
**  **                          **  **
********                        ********
**      **                      **      **
****    ****                    ****    ****
**  **  **  **                  **  **  **  **
****************                ****************
**              **              **              **
****            ****            ****            ****
**  **          **  **          **  **          **  **
********        ********        ********        ********
**      **      **      **      **      **      **      **
****    ****    ****    ****    ****    ****    ****    ****
**  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **
****************************************************************
**                                                              **
****                                                            ****
**  **                                                          **  **
********                                                        ********
**      **                                                      **      **
****    ****                                                    ****    ****
**  **  **  **                                                  **  **  **  **
****************                                                ****************
**              **                                              **              **
****            ****                                            ****            ****
**  **          **  **                                          **  **          **  **
********        ********                                        ********        ********
**      **      **      **                                      **      **      **      **
****    ****    ****    ****                                    ****    ****    ****    ****
**  **  **  **  **  **  **  **                                  **  **  **  **  **  **  **  **
********************************                                ********************************
**                              **                              **                              **
****                            ****                            ****                            ****
**  **                          **  **                          **  **                          **  **
********                        ********                        ********                        ********
**      **                      **      **                      **      **                      **      **
****    ****                    ****    ****                    ****    ****                    ****    ****
**  **  **  **                  **  **  **  **                  **  **  **  **                  **  **  **  **
****************                ****************                ****************                ****************
**              **              **              **              **              **              **              **
****            ****            ****            ****            ****            ****            ****            ****
**  **          **  **          **  **          **  **          **  **          **  **          **  **          **  **
********        ********        ********        ********        ********        ********        ********        ********
**      **      **      **      **      **      **      **      **      **      **      **      **      **      **      **
****    ****    ****    ****    ****    ****    ****    ****    ****    ****    ****    ****    ****    ****    ****    ****
**  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **  **
********************************************************************************************************************************
```

# If you look at the query we used...

```
WITH points AS (
    SELECT r, c FROM generate_series(0, 63) rows(r)
    CROSS JOIN generate_series(0, 63) cols(c)
  ), marked_points AS (
    SELECT r, c, CASE WHEN r & c != 0 THEN ' ' ELSE '*' END AS marker
    FROM points
    ORDER BY r DESC, c ASC
  ), rows AS (
    SELECT r, string_agg(marker, '') AS line
    FROM marked_points
    GROUP BY r
    ORDER BY r DESC
  ) SELECT string_agg(line, E'\n') FROM rows;
```

```
WITH points AS (
    SELECT r, c FROM generate_series(0, 63) rows(r)
    CROSS JOIN generate_series(0, 63) cols(c)
  ), marked_points AS (
    SELECT r, c, CASE WHEN r & c != 0 THEN '  ' ELSE '**' END AS marker
    FROM points
    ORDER BY r DESC, c ASC
  ), rows AS (
    SELECT r, string_agg(marker, '') AS line
    FROM marked_points
    GROUP BY r
    ORDER BY r DESC
  ) SELECT string_agg(line, E'\n') FROM rows;
```



# Definition of Mandelbrot Set

For each point using complex numbers, set

$Z_0 = 0$

$Z_{n+1} = Z_n^2 + x + i * y$

If $Z_n$ does not approach infinity, the point is in the Mandelbrot set.

# Iteration through recursive CTEs

**WITH RECURSIVE** <name> **AS** (

<initial query>

**UNION ALL**

<recursive query>

)

# A simple example: a counter

```
WITH RECURSIVE counter AS (
  SELECT 1 AS i
  UNION ALL
  SELECT i+1 FROM counter WHERE i < 5
) SELECT i FROM counter;
```

```
 i
---
 1
 2
 3
 4
 5
```

# Iteration for testing if a point is in the Mandelbrot Set

```
WITH RECURSIVE iterations AS (
  SELECT r, c, 0.0::float AS zr, 0.0::float AS zi, 0 AS iteration FROM points
  UNION ALL
  SELECT r, c, zr*zr - zi*zi + c AS zr, 2*zr*zi + r AS zi, iteration+1 AS iteration
  FROM iterations WHERE zr*zr + zi*zi < 4 AND iteration < 1000
)
```

# Changing the query from before

```
WITH RECURSIVE points AS (
  SELECT r, c FROM generate_series(-2, 2, 0.05) a(r)
  CROSS JOIN generate_series(-2, 1, 0.05) b(c)
  ORDER BY r DESC, c ASC
), iterations AS (
   SELECT r, c, 0.0::float AS zr, 0.0::float AS zc, 0 AS iteration FROM points
   UNION ALL
   SELECT r, c, zr*zr - zc*zc + c AS zr, 2*zr*zc + r AS zc, iteration+1 AS iteration
   FROM iterations WHERE zr*zr + zc*zc < 4 AND iteration < 1000
), final_iteration AS (
  SELECT * FROM iterations WHERE iteration = 1000
), marked_points AS (
   SELECT r, c, (CASE WHEN EXISTS (SELECT 1 FROM final_iteration i WHERE p.r = i.r AND p.c = i.c)
                  THEN '**'
                  ELSE '  '
                  END) AS marker
   FROM points p
   ORDER BY r DESC, c ASC
), lines AS (
   SELECT r, string_agg(marker, '') AS r_text
   FROM marked_points
   GROUP BY r
   ORDER BY r DESC
) SELECT string_agg(r_text, E'\n') FROM lines;
```

# The Result

```
                                                                              **


                                                                          **
                                                                      ********
                                                                      ********
                                                                      ********

                                                          **    ********************
                                                          ****************************  ****
                                                          **********************************
                                                        **********************************
                                                      **************************************
                                                    ******************************************
                                                    ******************************************
                                **    **          ********************************************
                                **************    ********************************************
                              ******************  ********************************************
                              ******************  ********************************************
                          **  ******************  ******************************************
******************************************************************************************
                          **  ******************  ******************************************
                              ******************  ********************************************
                              ******************  ********************************************
                                **************    ********************************************
                                **    **          ********************************************
                                                    ******************************************
                                                    ******************************************
                                                      **************************************
                                                        **********************************
                                                          **********************************
                                                          ****************************  ****
                                                          **    ********************

                                                                      ********
                                                                      ********
                                                                      ********
                                                                          **


                                                                              **
```

# Julia Set

# Julia Set Fractal

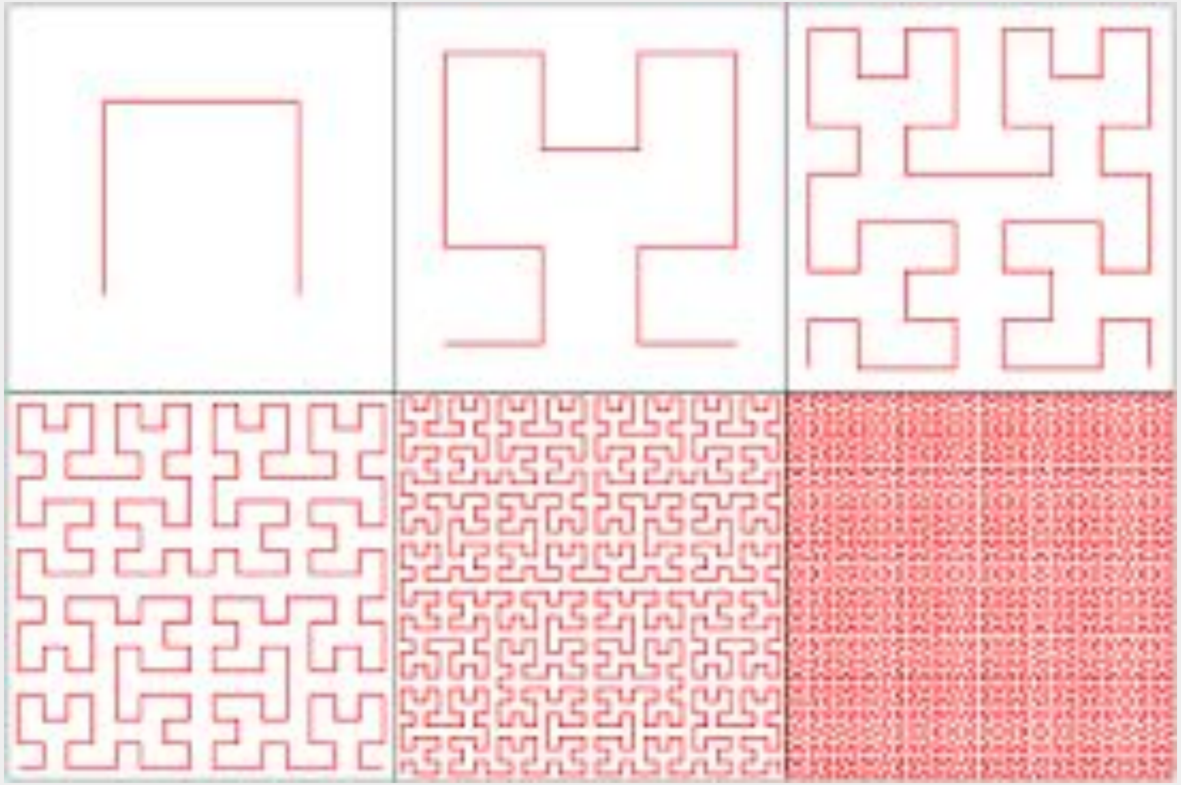
$Z_0 = x + i * y$

$Z_{n+1} = Z_n^2 + 1 - \varphi$ (the golden ratio)

```
WITH RECURSIVE points AS (
  SELECT r, c FROM generate_series(-2, 2, 0.05) a(r)
  CROSS JOIN generate_series(-2, 2, 0.05) b(c)
  ORDER BY r DESC, c ASC
), iterations AS (
   SELECT r, c, c::float AS zr, r::float AS zc, 0 AS iteration FROM points
   UNION ALL
   SELECT r, c, zr*zr - zc*zc + 1 - 1.61803398875 AS zr, 2*zr*zc AS zc, iteration+1 AS iteration
   FROM iterations WHERE zr*zr + zc*zc < 4 AND iteration < 1000
), final_iteration AS (
  SELECT * FROM iterations WHERE iteration = 1000
), marked_points AS (
   SELECT r, c, (CASE WHEN EXISTS (SELECT 1 FROM final_iteration i WHERE p.r = i.r AND p.c = i.c)
                 THEN '**'
                 ELSE '  '
                 END) AS marker
   FROM points p
   ORDER BY r DESC, c ASC
), lines AS (
   SELECT r, string_agg(marker, '') AS r_text
   FROM marked_points
   GROUP BY r
   ORDER BY r DESC
) SELECT string_agg(r_text, E'\n') FROM lines;
```

```
                                                        **
                                                        **
                                                    **********
                                                    **********
                                                    **********
                                          **  **********************  **
                                          ******************************
                                        **********************************
                                      **************************************
                    ****    **        **************************************        **    ****
                    **************    **************************************    **************
                  ******************************************************************************
                **********************************************************************************
              **************************************************************************************
              **************************************************************************************
      ******************************************************************************************************
      ******************************************************************************************************
  **************************************************************************************************************
******************************************************************************************************************
  **************************************************************************************************************
      ******************************************************************************************************
      ******************************************************************************************************
              **************************************************************************************
              **************************************************************************************
                **********************************************************************************
                  ******************************************************************************
                    **************    **************************************    **************
                    ****    **        **************************************        **    ****
                                      **************************************
                                        **********************************
                                          ******************************
                                          **  **********************  **
                                                    **********
                                                    **********
                                                    **********
                                                        **
                                                        **
```

# The Hilbert Curve



# How can we generate these fractals with SQL?

One way to describe self-similar fractals is with an “L-system”.

# Hilbert Curve L-system

```
Initial String: A

Replacement Rules

    A -> −BF+AFA+FB−

    B -> +AF−BFB−FA+
```

# Running the L-system

```
Iteration 0: A

Iteration 1: -BF+AFA+FB-

Iteration 2: -+AF-BFB-FA+F+-BF+AFA+FB-F-BF+AFA+FB-+F+AF-BFB-FA+-

...
```

# Crazy Idea

Let’s try to write a SQL query that processes L-systems!

# How can we do that?

1. Write a SQL query that runs the L-system and produces the L-system string.
2. Convert that string into the fractal.

# Writing a CTE to run the L-system

```
WITH RECURSIVE iterations AS (
  SELECT 'A' AS PATH, 0 AS iteration
  UNION ALL
  SELECT replace(replace(replace(PATH, 'A', '-CF+AFA+FC-'), 'B', '+AF-BFB-FA+'), 'C', 'B'), iteration+1 AS
iteration
  FROM iterations WHERE iteration < 3
) SELECT * FROM iterations;

                       path                               | iteration
----------------------------------------------------+-----------
 A                                                  |         0
 -BF+AFA+FB-                                        |         1
 -+AF-BFB-FA+F+-BF+AFA+FB-F-BF+AFA+FB-+F+AF-BFB-FA+- |         2
```

# Converting the string into the fractal

Three steps:

a. Convert the string into a set of line segments.
b. Convert the line segments into marked points in a grid.
c. Concatenate the characters for the points together.

# Recursive CTE that calculates the line segments

```
  SELECT 0 AS r1, 0 AS c1, 0 AS r2, 0 AS c2, 0 AS r3, 0 AS c3, 0 AS dr, 1 AS dc, (SELECT path FROM iterations
ORDER BY iteration DESC LIMIT 1) AS path_left
  UNION ALL
  SELECT r3 AS r1, c3 AS c1, r3 + dr * movement AS r2, c3 + dc * movement AS c2, r3 + 2 * dr * movement AS r3,
c3 + 2 * dc * movement AS c3, dr, dc, SUBSTRING(path_left FROM 2) AS path_left
  FROM (
    SELECT r1, c1, r2, c2, r3, c3,
      CASE WHEN SUBSTRING(path_left FOR 1) = '-' THEN -dc
           WHEN SUBSTRING(path_left FOR 1) = '+' THEN dc
           ELSE dr
      END AS dr,
      CASE WHEN SUBSTRING(path_left FOR 1) = '-' THEN dr
           WHEN SUBSTRING(path_left FOR 1) = '+' THEN -dr
           ELSE dc
           END AS dc,
           path_left,
      CASE WHEN SUBSTRING(path_left FOR 1) = 'F' THEN 1 ELSE 0 END AS movement
  FROM segments
  WHERE CHAR_LENGTH(path_left) > 0
  ) sub
```

# Mark the points

```
marked_points AS (
SELECT r, c, (CASE WHEN
    EXISTS (SELECT 1 FROM end_points e WHERE p.r = e.r AND p.c = e.c)
    THEN '*'

    WHEN EXISTS (SELECT 1 FROM segments s WHERE p.r = s.r2 AND p.c = s.c2 AND dc != 0)
    THEN '-'

    WHEN EXISTS (SELECT 1 FROM segments s WHERE p.r = s.r2 AND p.c = s.c2 AND dr != 0)
    THEN '|'

    ELSE ' '
    END
) AS marker
FROM points
```

# Combine the points together to produce the fractal

```
lines AS (
   SELECT r, string_agg(marker, '') AS row_text
   FROM marked_points
   GROUP BY r
   ORDER BY r DESC
) SELECT string_agg(row_text, E'\n') FROM lines;
```

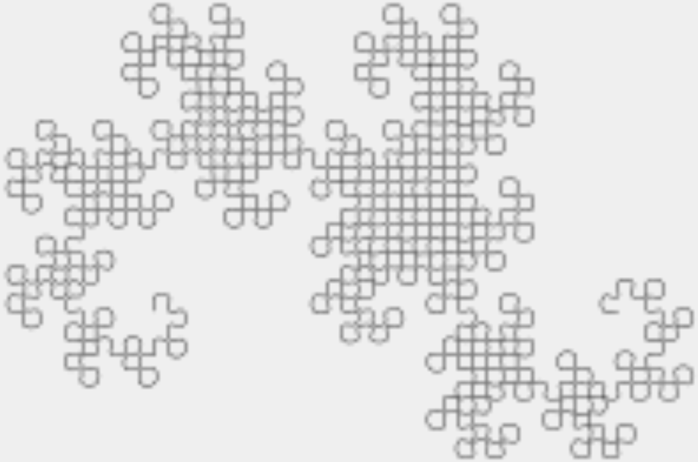
# The full query

```
WITH RECURSIVE iterations AS (
  SELECT 'A' AS PATH, 0 AS iteration
  UNION ALL
  SELECT replace(replace(replace(PATH, 'A', '-CF+AFA+FC-'), 'B', '+AF-BFB-FA+'), 'C', 'B'), iteration + 1 FROM iterations WHERE iteration < 2
), segments AS (
  SELECT 0 AS r1, 0 AS c1, 0 AS r2, 0 AS c2, 0 AS r3, 0 AS c3, 0 AS dr, 1 AS dc, replace(replace((SELECT path FROM iterations ORDER BY iteration DESC LIMIT 1), 'A', ''), 'B', '') AS path_left
  UNION ALL
  SELECT r3 AS r1, c3 AS c1, r3 + dr * movement AS r2, c3 + dc * movement AS c2, r3 + 2 * dr * movement AS r3, c3 + 2 * dc * movement AS c3, dr, dc, SUBSTRING(path_left FROM 2) AS path_left
  FROM (
    SELECT r1, c1, r2, c2, r3, c3,
      CASE WHEN SUBSTRING(path_left FOR 1) = '-' THEN -dc
           WHEN SUBSTRING(path_left FOR 1) = '+' THEN dc
           ELSE dr
      END AS dr,
      CASE WHEN SUBSTRING(path_left FOR 1) = '-' THEN dr
           WHEN SUBSTRING(path_left FOR 1) = '+' THEN -dr
           ELSE dc
           END AS dc,
           path_left,
      CASE WHEN SUBSTRING(path_left FOR 1) IN ('+', '-') THEN 0 ELSE 1 END AS movement
  FROM segments
  WHERE CHAR_LENGTH(path_left) > 0
  ) sub
),
end_points AS (SELECT r1 AS r, c1 AS c FROM segments UNION SELECT r3, c3 FROM segments),
points AS (
SELECT r, c FROM generate_series((SELECT MIN(r) FROM end_points), (SELECT MAX(r) FROM end_points)) a(r)
CROSS JOIN generate_series((SELECT MIN(c) FROM end_points), (SELECT MAX(c) FROM end_points)) b(c)
), marked_points AS (
SELECT r, c, (CASE WHEN
    EXISTS (SELECT 1 FROM end_points e WHERE p.r = e.r AND p.c = e.c)
    THEN '*'

    WHEN EXISTS (SELECT 1 FROM segments s WHERE p.r = s.r2 AND p.c = s.c2 AND dc != 0)
    THEN '-'

    WHEN EXISTS (SELECT 1 FROM segments s WHERE p.r = s.r2 AND p.c = s.c2 AND dr != 0)
    THEN '|'

    ELSE ' '
    END
    ) AS marker
FROM points p
), ROWS AS (
   SELECT r, string_agg(marker, '') AS row_text
   FROM marked_points
   GROUP BY r
   ORDER BY r DESC
) SELECT string_agg(row_text, E'\n') FROM ROWS;
```

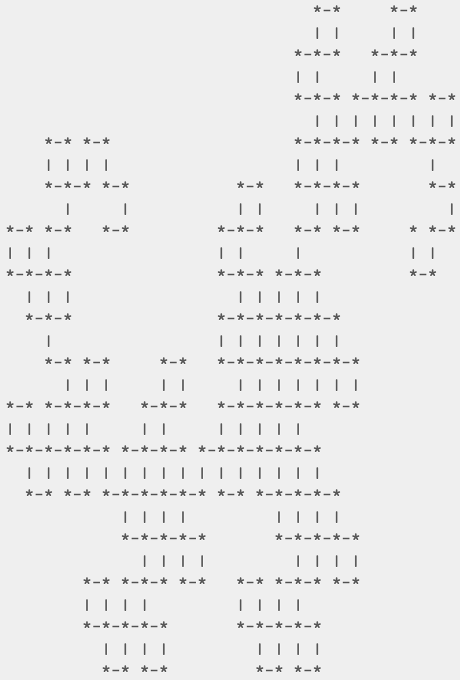
# Hilbert Curve

```
*-* *-* *-* *-*
| | | | | | | |
* *-* * * *-* *
|     | |     |
*-* *-* *-* *-*
  | |     | |
*-* *-*-*-* *-*
|             |
* *-*-* *-*-* *
| |   | |   | |
*-* *-* *-* *-*
    |     |
*-* *-* *-* *-*
| |   | |   | |
* *-*-* *-*-* *
```

# Dragon Curve



# Dragon Curve



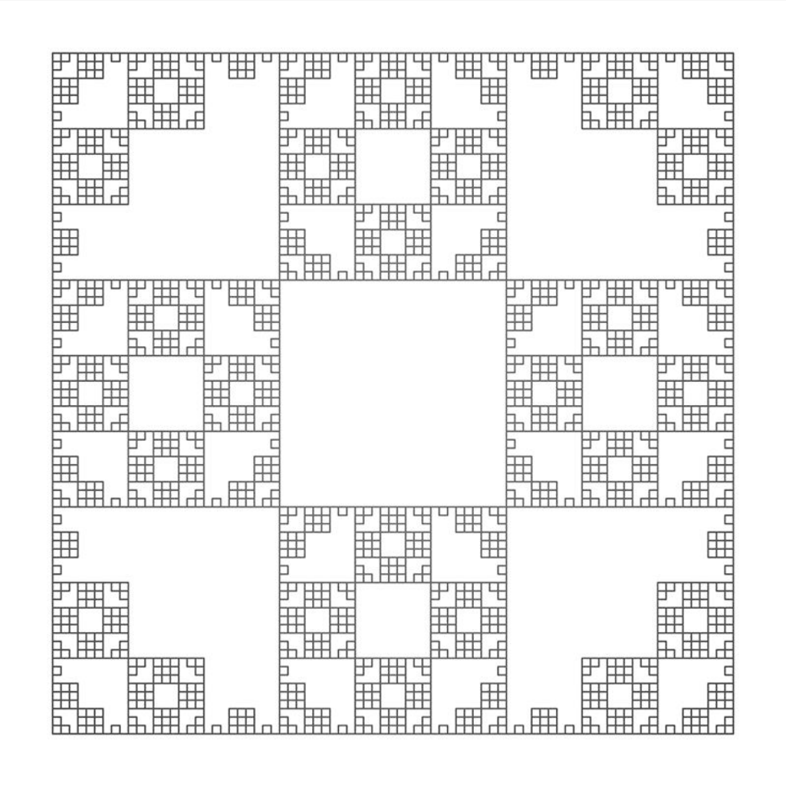
Initial String: FX

Replacement Rules

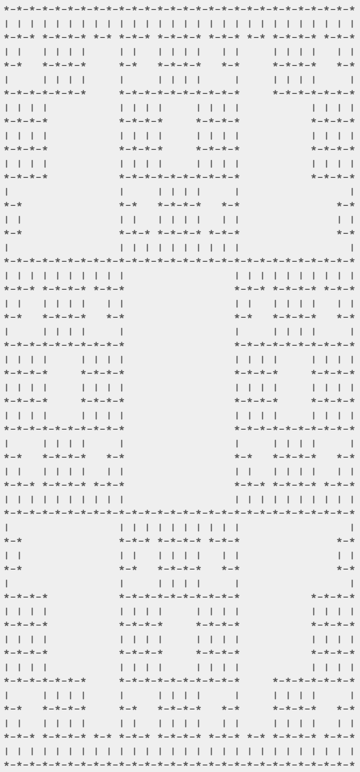
X -> X+YF+

Y -> -FX-Y

# Board



# Board



Initial String: F+F+F+F

Replacement Rules

F -> FF+F+F+F+FF

# Now, let’s try to build a programming language

# What Will the Language Look Like?

# Lisp is a great place to start!

```
(defun eval. (e a)
  (cond
    ((atom e) (assoc. e a))
    ((atom (car e))
     (cond
       ((eq (car e) 'quote) (cadr e))
       ((eq (car e) 'atom)  (atom   (eval. (cadr e) a)))
       ((eq (car e) 'eq)    (eq     (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) 'car)   (car    (eval. (cadr e) a)))
       ((eq (car e) 'cdr)   (cdr    (eval. (cadr e) a)))
       ((eq (car e) 'cons)  (cons   (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) 'cond)  (evcon. (cdr e) a))
       ('t (eval. (cons (assoc. (car e) a)
                        (cdr e))
                  a))))
    ((eq (caar e) 'label)
     (eval. (cons (caddar e) (cdr e))
            (cons (list (cadar e) (car e)) a)))
    ((eq (caar e) 'lambda)
     (eval. (caddar e)
            (append. (pair. (cadar e) (evlis. (cdr e) a))
                     a)))))

(defun evcon. (c a)
  (cond ((eval. (caar c) a)
         (eval. (cadar c) a))
        ('t (evcon. (cdr c) a))))

(defun evlis. (m a)
  (cond ((null. m) '())
        ('t (cons (eval.  (car m) a)
                  (evlis. (cdr m) a)))))
```

# Step #1: Write a Lisp Parser

# Nah... Let's Just Use JSON

# Postgres Has Really Good Support for JSON

```
jsonb_build_array(‘a’, ‘b’, ‘c’) => ["a", "b", "c"]

jsonb_build_object('a', 1, 'b', 2) => {"a": 1, "b": 2}

'["a", "b", "c"]'::jsonb -> 2 => ‘c’

'{"a": 1, "b": 2}'::jsonb -> 'a' => 1

jsonb_typeof('["a", "b", "c"]'::jsonb) => ‘array’
```

# Arrays

The Lisp program from before:

```
(+ 2 3)
```

Can be represented in JSON as:

```
[“+”, 2, 3]
```

# Implementing the Backend

# We Can Try Translating Another Interpreter...

```
(defun eval. (e a)
  (cond
    ((atom e) (assoc. e a))
    ((atom (car e))
     (cond
       ((eq (car e) 'quote) (cadr e))
       ((eq (car e) 'atom)  (atom   (eval. (cadr e) a)))
       ((eq (car e) 'eq)    (eq     (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) 'car)   (car    (eval. (cadr e) a)))
       ((eq (car e) 'cdr)   (cdr    (eval. (cadr e) a)))
       ((eq (car e) 'cons)  (cons   (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) 'cond)  (evcon. (cdr e) a))
       ('t (eval. (cons (assoc. (car e) a)
                        (cdr e))
                  a))))
    ((eq (caar e) 'label)
     (eval. (cons (caddar e) (cdr e))
            (cons (list (cadar e) (car e)) a)))
    ((eq (caar e) 'lambda)
     (eval. (caddar e)
            (append. (pair. (cadar e) (evlis. (cdr e) a))
                     a)))))

(defun evcon. (c a)
  (cond ((eval. (caar c) a)
         (eval. (cadar c) a))
        ('t (evcon. (cdr c) a))))

(defun evlis. (m a)
  (cond ((null. m) '())
        ('t (cons (eval.  (car m) a)
                  (evlis. (cdr m) a)))))
```

# Let’s Start With Something Simpler: A Calculator

We can support basic operations:

- We can support the operations “+”, “-”, “*”, “/”.
- You can nest these operations.

# Evaluating an Expression

- Numbers and operators (+, -, *, /) evaluate to themselves.
- To evaluate a compound expression (e.g. `[“+”, 2, 3])`, you recursively evaluate each argument. Then you pass the arguments to the given operator.

# An Example

To evaluate an expression like `[“+”, [“*”, 2, 3], [“+”, 4, 6]]`

- Recursively evaluate `“+”` `=>` `“+”`.
- Recursively evaluate `[“*”, 2, 3] => 6`.
- Recursively evaluate `[“+”, 4, 6] => 10`.
- Calculate the result of the `“+”` operator on the values `6` and `10`.
- This gives the result of the whole expression as `16`.

# Unfortunately SQL Doesn’t Have Recursion

# But We Can Fake It With a Stack!

# Implementing This in SQL

We can do this using primarily two SQL features:

- CASE
- Recursive CTEs

# Sketch of How the Interpreter Will Work

We maintain a “state” which is a JSON object that looks like:

```
{“stack”: [...], result: ...}
```

Every “stack frame” has a `type` and data specific to that type.

# Manipulating the State

Each iteration of the recursive CTE will look at the top stack frame and perform some computation associated with it.

# Basic State Manipulations

Create a new state:

```
jsonb_build_object('stack', ..., 'result', ...)
```

Push a new stack frame:

```
jsonb_build_array(<new frame>) || stack
```

Remove the top stack frame:

```
stack - 0
```

# Types of Stack Frames

- expr - a stack frame that will evaluate the given expression.
- eval_args - a stack frame that will keep track of our progress as we recursively evaluate each argument.
- eval_call - a stack frame that takes all the evaluated arguments and performs the actual function call.

# Interpreter Skeleton

```
WITH RECURSIVE loop AS (
    SELECT '{"stack": [{"type": "expr", "expr": ["+", ["*", 5, 6], 7]}]}'::jsonb AS STATE
  UNION ALL
    ...
) ...
```

# Alias Some Useful Values

```
WITH RECURSIVE loop AS (
    SELECT '{"stack": [{"type": "expr", "expr": ["+", ["*", 5, 6], 7]}]}'::jsonb AS state
  UNION ALL
    SELECT ...
    FROM (
      SELECT state -> 'stack' -> 0 ->> 'type' AS frame_type,
             state -> 'stack' -> 0 -> 'expr' AS expr,
             state -> 'stack' -> 0 ->> 'expr' AS expr_string,
             state -> 'stack' -> 0 -> 'expr' -> 0 AS op,
             state -> 'stack' -> 0 -> 'expr' ->> 0 AS op_string,
             state -> 'stack' -> 0 -> 'expr' -> 1 AS arg1,
             state -> 'stack' -> 0 -> 'expr' -> 2 AS arg2,
             state -> 'stack' -> 0 -> 'expr' -> 3 AS arg3,
             state -> 'stack' -> 0 -> 'expr' -> 4 AS arg4,
             state -> 'stack' -> 0 -> 'left' AS args_left,
             state -> 'stack' -> 0 -> 'done' AS args_done,
             state -> 'stack' -> 0 -> 'env' AS env,
             state -> 'result' AS result,
             state -> 'stack' AS stack
             FROM loop
    ) sub
) ...
```

# The CASE Statement Does All The Work

```
WITH RECURSIVE loop AS (
    SELECT '{"stack": [{"type": "expr", "expr": ["+", ["*", 5, 6], 7]}]}'::jsonb AS STATE
  UNION ALL
    SELECT CASE ... END
    FROM (
      SELECT state -> 'stack' -> 0 ->> 'type' AS frame_type,
             state -> 'stack' -> 0 -> 'expr' AS expr,
             state -> 'stack' -> 0 ->> 'expr' AS expr_string,
             state -> 'stack' -> 0 -> 'expr' -> 0 AS op,
             state -> 'stack' -> 0 -> 'expr' ->> 0 AS op_string,
             state -> 'stack' -> 0 -> 'expr' -> 1 AS arg1,
             state -> 'stack' -> 0 -> 'expr' -> 2 AS arg2,
             state -> 'stack' -> 0 -> 'expr' -> 3 AS arg3,
             state -> 'stack' -> 0 -> 'expr' -> 4 AS arg4,
             state -> 'stack' -> 0 -> 'left' AS args_left,
             state -> 'stack' -> 0 -> 'done' AS args_done,
             state -> 'stack' -> 0 -> 'env' AS env,
             state -> 'result' AS result,
             state -> 'stack' AS stack
             FROM loop

    ) sub
) ...
```

# First Case: Expression Stack Frames

```
WHEN frame_type = 'expr'
THEN CASE ...
     END
```

# Self-Evaluating Expressions

```
WHEN jsonb_typeof(expr) = 'number' OR jsonb_typeof(expr) = 'string'
THEN jsonb_build_object('stack', stack - 0,  'result', expr)
```

# Function Calls

To handle function calls, we first need to evaluate each argument. We do this by pushing an `eval_args` frame:

```
ELSE ...  jsonb_build_object('type', 'eval_args', 'left', expr, 'done', '[]'::jsonb) || (stack - 0)) ...
```

# Second Case: Argument Evaluation Stack Frames

```
WHEN frame_type = 'eval_args'
THEN CASE ...
     END
```

# Base Case: When there are no args left

```
WHEN result IS NULL AND jsonb_array_length(args_left) = 0
THEN ... jsonb_build_array( 'type', 'eval_call', 'expr', args_done) ...
```

# Recursive Case: Evaluate one argument

```
WHEN result IS NULL
THEN ...
     jsonb_build_object('type', 'expr', 'expr', args_left -> 0)
     jsonb_build_object('type', 'eval_args', 'left', args_left - 0, 'done', args_done)
     ....
```

# Last Case: We have the result of an evaluation

```
ELSE ... jsonb_build_object('type', 'eval_args', 'left', args_left, 'done', args_done || jsonb_build_array(result))) ...
```

# Last Stack Frame: Eval Call Frames

```
WHEN frame_type = 'eval_call'
THEN CASE WHEN op_string = '+'
          THEN jsonb_build_object('stack', stack - 0,'result', arg1::bigint + arg2::bigint)

          WHEN op_string = '*'
          THEN jsonb_build_object('stack', stack - 0,'result', arg1::bigint * arg2::bigint)

          WHEN op_string = '-'
          THEN jsonb_build_object('stack', stack - 0,'result', arg1::bigint - arg2::bigint)

          WHEN op_string = '/'
          THEN jsonb_build_object('stack', stack - 0,'result', arg1::bigint / arg2::bigint)
     END
END
```

# Terminating Condition of the Interpreter

We keep running the loop until we get to a state that has no stack frames.

```
SELECT state -> 'result' FROM loop WHERE jsonb_array_length(state -> 'stack') = 0 LIMIT 1;
```

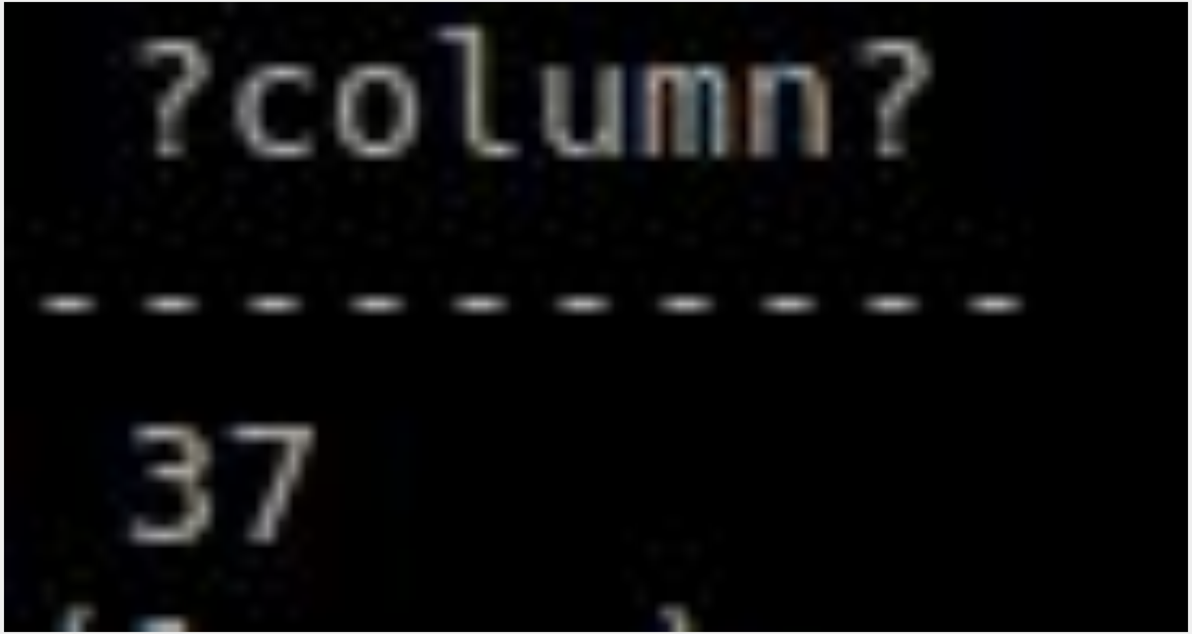
# Full Query

```
WITH RECURSIVE loop AS (
    SELECT '{"stack": [{"type": "expr", "expr": ["+", ["*", 5, 6], 7]}]}'::jsonb AS STATE
  UNION ALL
  SELECT
    CASE
      WHEN frame_type = 'expr'
      THEN CASE WHEN jsonb_typeof(expr) = 'number' OR jsonb_typeof(expr) = 'string'
                THEN jsonb_build_object('stack', stack - 0, 'result', expr)

                ELSE jsonb_build_object('stack', jsonb_build_array(jsonb_build_object('type', 'eval_args', 'left', expr, 'done', '[]'::jsonb))  || (stack - 0))
           END

      WHEN frame_type = 'eval_args'
      THEN CASE WHEN result IS NULL AND jsonb_array_length(args_left) = 0
                THEN jsonb_build_object('stack', jsonb_build_array(jsonb_build_object('type', 'eval_call', 'expr', args_done)) || (stack - 0))

                WHEN result IS NULL
                THEN jsonb_build_object('stack', jsonb_build_array(jsonb_build_object('type', 'expr', 'expr', args_left -> 0), jsonb_build_object('type', 'eval_args', 'left', args_left - 0, 'done', args_done)) || stack - 0)

                ELSE jsonb_build_object('stack', jsonb_build_array(jsonb_build_object('type', 'eval_args', 'left', args_left, 'done', args_done || jsonb_build_array(result))) || (stack - 0))
           END

      WHEN frame_type = 'eval_call'
      THEN CASE WHEN op_string = '+'
                THEN jsonb_build_object('stack', stack - 0, 'result', arg1::bigint + arg2::bigint)

                WHEN op_string = '*'
                THEN jsonb_build_object('stack', stack - 0, 'result', arg1::bigint * arg2::bigint)

                WHEN op_string = '-'
                THEN jsonb_build_object('stack', stack - 0, 'result', arg1::bigint - arg2::bigint)

                WHEN op_string = '/'
                THEN jsonb_build_object('stack', stack - 0, 'result', arg1::bigint / arg2::bigint)
           END
      END
    FROM (
      SELECT state -> 'stack' -> 0 ->> 'type' AS frame_type,
             state -> 'stack' -> 0 -> 'expr' AS expr,
             state -> 'stack' -> 0 -> 'expr' -> 0 AS op,
             state -> 'stack' -> 0 -> 'expr' ->> 0 AS op_string,
             state -> 'stack' -> 0 -> 'expr' -> 1 AS arg1,
             state -> 'stack' -> 0 -> 'expr' -> 2 AS arg2,
             state -> 'stack' -> 0 -> 'expr' -> 3 AS arg3,
             state -> 'stack' -> 0 -> 'left' AS args_left,
             state -> 'stack' -> 0 -> 'done' AS args_done,
             state -> 'result' AS result,
             state -> 'stack' AS stack
             FROM loop
  ) sub
) SELECT state -> 'result' FROM loop WHERE jsonb_array_length(state -> 'stack') = 0 LIMIT 1;
```

# Our Program

```
["+", ["*", 5, 6], 7]
```

# The result



# The Road to Turing Completeness

- Comparison Operators
- Lists
- If statements
- Lambda Functions

# Example: Fibonacci Numbers

```
[["lambda", ["f"],
    ["f", "f", 1, 0, 0]],
  ["lambda", ["self", "a", "b", "i"],
    ["if", ["=", "i", 10],
           ["empty"],
           ["cons", "b", ["self", "self", ["+", "a", "b"], "a", ["+", "i", 1]]]]]]
```

# Adding Comparison Operators

```
WHEN op_string = '-'
THEN jsonb_build_object('stack', stack - 0, 'result', arg1::bigint - arg2::bigint)

WHEN op_string = '/'
THEN jsonb_build_object('stack', stack - 0, 'result', arg1::bigint / arg2::bigint)

WHEN op_string = '>'
THEN jsonb_build_object('stack', stack - 0, 'result', arg1::bigint > arg2::bigint)

WHEN op_string = '<'
THEN jsonb_build_object('stack', stack - 0, 'result', arg1::bigint < arg2::bigint)

WHEN op_string = '='
THEN jsonb_build_object('stack', stack - 0, 'result', arg1 = arg2)

...
```

# Adding List Operations

```
WHEN op_string = '='
THEN jsonb_build_object('stack', stack - 0, 'result', arg1 = arg2)

WHEN op_string = 'head'
THEN jsonb_build_object('stack', stack - 0, 'result', arg1 -> 0)

WHEN op_string = 'tail'
THEN jsonb_build_object('stack', stack - 0, 'result', arg1 - 0)

WHEN op_string = 'cons'
THEN jsonb_build_object('stack', stack - 0, 'result', jsonb_build_array(arg1) || arg2)

WHEN op_string = 'empty'
THEN jsonb_build_object('stack', stack - 0, 'result', '[]'::jsonb)

...
```

# If Statements

We first add a new kind of expression:

```
CASE WHEN jsonb_typeof(expr) = 'number' OR jsonb_typeof(expr) = 'string'
     THEN jsonb_build_object('stack', stack - 0, 'result', expr)

     WHEN op_string = 'if'
     THEN ... jsonb_build_object('type', 'eval_if', 'expr', expr) ...

     ELSE ... jsonb_build_object('type', 'eval_args', 'left', expr, 'done', '[]'::jsonb)
...
END
```

# Implementation

```
CASE WHEN result IS NULL
     THEN jsonb_build_object('stack', jsonb_build_array(jsonb_build_object('type', 'expr', 'expr', arg1)) || stack)

     WHEN result IS NOT NULL AND result::boolean
     THEN jsonb_build_object('stack', jsonb_build_array(jsonb_build_object('type', 'expr', 'expr', arg2)) || (stack - 0))

     WHEN result IS NOT NULL AND NOT result::boolean
     THEN jsonb_build_object('stack', jsonb_build_array(jsonb_build_object('type', 'expr', 'expr', arg3)) || (stack - 0))
END
```

# One Last Feature: Lambda Functions

In order to support lambda functions, we need to be able to do the following:

- We need support for variables.
- Define a lambda function.
- Call a lambda function.

# Variables

```
WHEN jsonb_typeof(expr) = 'number' OR jsonb_typeof(expr) = 'string'
THEN jsonb_build_object('stack', stack - 0, 'result', expr)

WHEN jsonb_typeof(expr) = 'string'
THEN jsonb_build_object('stack', stack - 0, 'result', env -> expr_string)

WHEN op_string = 'if'
```

# Define a lambda function

```
[“lambda”, [<arguments>], <body>]
```

# Add a new `lambda` expression type.

```
WHEN op_string = 'if'
THEN ...

WHEN op_string = 'lambda'
THEN jsonb_build_object('stack', stack - 0, 'result', jsonb_build_object('args', arg1, 'body', arg2, 'env', env))
```

# Calling a Lambda Function

```
jsonb_build_object('type', 'expr',
                   'expr', (op -> 'body'),
                   'env', (op -> 'env') || jsonb_build_object(
                                             COALESCE(op -> 'args' ->> 0, 'null'), arg1,
                                             COALESCE(op -> 'args' ->> 1, 'null'), arg2,
                                             COALESCE(op -> 'args' ->> 2, 'null'), arg3,
                                             COALESCE(op -> 'args' ->> 3, 'null'), arg4)
```

# Example: First 10 Fibonacci Numbers

```
[["lambda", ["f"],
    ["f", "f", 1, 0, 0]],
  ["lambda", ["self", "a", "b", "i"],
    ["if", ["=", "i", 10],
           ["empty"],
           ["cons", "b", ["self", "self", ["+", "a", "b"], "a", ["+", "i", 1]]]]]]
```

# Hacking in Recursion

```
[["lambda", ["f"],
    ["f", "f", 1, 0, 0]],
  ["lambda", ["self", "a", "b", "i"],
    ["if", ["=", "i", 10],
           ["empty"],
           ["cons", "b", ["self", "self", ["+", "a", "b"], "a", ["+", "i", 1]]]]]]
```

# Actual Logic

```
[["lambda", ["f"],
    ["f", "f", 1, 0, 0]],
  ["lambda", ["self", "a", "b", "i"],
    ["if", ["=", "i", 10],
           ["empty"],
           ["cons", "b", ["self", "self", ["+", "a", "b"], "a", ["+", "i", 1]]]]]]
```

# Full Code

```
WITH RECURSIVE loop AS (
    SELECT '{"stack": [{"type": "expr", "env": {"+": "+", "-": "-", "*": "*", "/": "/", ">": ">", "<": "<", "=": "=", "head": "head", "tail": "tail", "cons": "cons", "empty": "empty"}, "expr": [["lambda", ["f"], ["f", "f", 1, 0, 0]], ["lambda", ["self", "a", "b", "i"], ["if", ["=", "i", 10], ["empty"], ["cons", "b", ["self", "self", ["+", "a", "b"], "a", ["+", "i", 1]]]]]]}]}'::jsonb AS STATE
  UNION ALL
  SELECT
    CASE
      WHEN frame_type = 'expr'
      THEN CASE WHEN jsonb_typeof(expr) = 'number'
                THEN jsonb_build_object('stack', stack - 0, 'result', expr)

                WHEN jsonb_typeof(expr) = 'string'
                THEN jsonb_build_object('stack', stack - 0, 'result', env -> expr_string)

                WHEN op_string = 'if'
                THEN jsonb_build_object('stack', jsonb_build_array(jsonb_build_object('type', 'eval_if', 'expr', expr, 'env', env))  || (stack - 0))

                WHEN op_string = 'lambda'
                THEN jsonb_build_object('stack', stack - 0, 'result', jsonb_build_object('args', arg1, 'body', arg2, 'env', env))

                ELSE jsonb_build_object('stack', jsonb_build_array(jsonb_build_object('type', 'eval_args', 'left', expr, 'done', '[]'::jsonb, 'env', env))  || (stack - 0))
            END

      WHEN frame_type = 'eval_args'
      THEN CASE WHEN result IS NULL AND jsonb_array_length(args_left) = 0
                THEN jsonb_build_object('stack', jsonb_build_array(jsonb_build_object('type', 'eval_call', 'expr', args_done, 'env', env)) || (stack - 0))

                WHEN result IS NULL
                THEN jsonb_build_object('stack', jsonb_build_array(jsonb_build_object('type', 'expr', 'expr', args_left -> 0, 'env', env), jsonb_build_object('type', 'eval_args', 'left', args_left - 0, 'done', args_done, 'env', env)) || stack - 0)

                ELSE jsonb_build_object('stack', jsonb_build_array(jsonb_build_object('type', 'eval_args', 'left', args_left, 'done', args_done || jsonb_build_array(result), 'env', env)) || (stack - 0))
            END

      WHEN frame_type = 'eval_call'
      THEN CASE WHEN op_string = '+'
                THEN jsonb_build_object('stack', stack - 0, 'result', arg1::text::bigint + arg2::text::bigint)

                WHEN op_string = '*'
                THEN jsonb_build_object('stack', stack - 0, 'result', arg1::text::bigint * arg2::text::bigint)

                WHEN op_string = '-'
                THEN jsonb_build_object('stack', stack - 0, 'result', arg1::text::bigint - arg2::text::bigint)

                WHEN op_string = '/'
                THEN jsonb_build_object('stack', stack - 0, 'result', arg1::text::bigint / arg2::text::bigint)

                WHEN op_string = '>'
                THEN jsonb_build_object('stack', stack - 0, 'result', arg1::text::bigint > arg2::text::bigint)

                WHEN op_string = '<'
                THEN jsonb_build_object('stack', stack - 0, 'result', arg1::text::bigint < arg2::text::bigint)

                WHEN op_string = '='
                THEN jsonb_build_object('stack', stack - 0, 'result', arg1 = arg2)

                WHEN op_string = 'head'
                THEN jsonb_build_object('stack', stack - 0, 'result', arg1 -> 0)

                WHEN op_string = 'tail'
                THEN jsonb_build_object('stack', stack - 0, 'result', arg1 - 0)

                WHEN op_string = 'cons'
                THEN jsonb_build_object('stack', stack - 0, 'result', jsonb_build_array(arg1) || arg2)

                WHEN op_string = 'empty'
                THEN jsonb_build_object('stack', stack - 0, 'result', '[]'::jsonb)

                ELSE jsonb_build_object('stack', jsonb_build_array(jsonb_build_object('type', 'expr',
                    'expr', (op -> 'body'),
                    'env', (op -> 'env') || jsonb_build_object(
                      COALESCE(op -> 'args' ->> 0, 'null'), arg1,
                      COALESCE(op -> 'args' ->> 1, 'null'), arg2,
                      COALESCE(op -> 'args' ->> 2, 'null'), arg3,
                      COALESCE(op -> 'args' ->> 3, 'null'), arg4)))
                  || (stack - 0))
            END

      WHEN frame_type = 'eval_if'
      THEN CASE WHEN result IS NULL
                THEN jsonb_build_object('stack', jsonb_build_array(jsonb_build_object('type', 'expr', 'expr', arg1, 'env', env)) || stack)

                WHEN result IS NOT NULL AND result::boolean
                THEN jsonb_build_object('stack', jsonb_build_array(jsonb_build_object('type', 'expr', 'expr', arg2, 'env', env)) || (stack - 0))

                WHEN result IS NOT NULL AND NOT result::boolean
                THEN jsonb_build_object('stack', jsonb_build_array(jsonb_build_object('type', 'expr', 'expr', arg3, 'env', env)) || (stack - 0))
            END
      END
    FROM (
      SELECT state -> 'stack' -> 0 ->> 'type' AS frame_type,
             state -> 'stack' -> 0 -> 'expr' AS expr,
             state -> 'stack' -> 0 ->> 'expr' AS expr_string,
             state -> 'stack' -> 0 -> 'expr' -> 0 AS op,
             state -> 'stack' -> 0 -> 'expr' ->> 0 AS op_string,
             state -> 'stack' -> 0 -> 'expr' -> 1 AS arg1,
             state -> 'stack' -> 0 -> 'expr' -> 2 AS arg2,
             state -> 'stack' -> 0 -> 'expr' -> 3 AS arg3,
             state -> 'stack' -> 0 -> 'expr' -> 4 AS arg4,
             state -> 'stack' -> 0 -> 'left' AS args_left,
             state -> 'stack' -> 0 -> 'done' AS args_done,
             state -> 'stack' -> 0 -> 'env' AS env,
             state -> 'result' AS result,
             state -> 'stack' AS stack
             FROM loop
  ) sub
) SELECT state -> 'result' FROM loop WHERE jsonb_array_length(state -> 'stack') = 0 LIMIT 1;
```

```
---------------------------------------
 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Email: michael@perfalytics.com

Twitter: @mmalisper