Befunge-93 in SQL

(Ab-)Using SQL's Turing Completeness

Tim Fischer
t.fischer@student.uni-tuebingen.de
Eberhard Karls Universität
Tübingen, Baden-Württemberg, Germany

ABSTRACT

Though SQL has been Turing complete since the SQL:1999 standard leveraging this feature to full effect still eludes many developers. One of the major roadblocks standing the way is the conceptual impedance mismatch between SQL's declarative semantics and the imperative semantics most developers are more familiar with. Where this becomes most evident is when trying to express complex control flow such as chained assignments, loops, and branches in a generic manner. In this seminar paper, we introduce techniques to represent such control flow constructs in plain SQL:1999 and demonstrate their use by implementing an interpreter for the esoteric programming language Befunge.

KEYWORDS

SQL, Befunge, Interpreters, Recursion

1 INTRODUCTION

To the layman, programming languages appear to be magical incantations that allow "computer wizards" to make computers obey their every whim. This mysticism around programming languages does not leave novice disciples of programming; the first fact they learn is that when developing these supposed magical incantations in any given programming language, they need to adhere to a strict set of rules. Comprising these rules are two major parts: *syntax* and *semantics*.

Though at first both *syntax* and *semantics* appear arbitrary at best, deeper study reveals the logic behind them. More often than not programming languages are built around a small set of core features that dictate the design of their semantics and syntax. For example, LISP [8] is built, both semantically and syntactically, around the concept of list-based symbolic processing.

Not all language design concepts are equally as expressive in all domains. Take Smalltalk [5], for example, it is a language in which all code is structured around the concept of objects that interact with one another. SQL [1] on the other hand is a query language the main goal of which is to enable developers to query relational data in a declarative manner.

When evaluating the expressiveness of programming language design concepts the property of Turing completeness elevates some languages above others. Though not a necessary property, for a language to be actually "useful", it nonetheless allows for gauging what kinds of programs can be expressed. In short, a Turing complete language can express any problem a Turing machine can solve and is thus capable of expressing most "sensible" computations.

All the programming languages referenced so far are Turing complete, this includes the obvious two, LISP and Smalltalk, and also, maybe surprisingly so, SQL. SQL has been Turing complete since the introduction of recursive common table expressions (`WITH RECURSIVE`) in the 1999 standard [17]. Which allow SQL developers to express any Turing complete problem in terms of fixpoint computation.

SQL's Turing completeness, in theory, allows developers to make use of textbook algorithms like the Dijkstra algorithm or the Edmonds-Karp algorithm; both of which are algorithms that can be used in many situations. In practice, however, developers opt to side-step writing these algorithms in SQL in favor of imperative languages. This often comes down to the impedance mismatch between the imperative pseudocode most textbooks depict these algorithms in and the declarative nature of SQL code.

In this paper, we will flex SQL's Turing complete muscles by demonstrating how to translate a Python-based Befunge interpreter into a single recursive SQL:1999 query in three major steps.

- In Section 2, we will look at what constitutes esoteric programming languages and what Befunge is. In doing so we will also argue why Befunge is an interesting vehicle for our cause.
- In Section 3, we will deconstruct an imperative Befunge interpreter into a general program shape and the individual types of control flow residing within.
- In Section 4, we will translate the Python-based Befunge interpreter into a recursive query; all the while only making use SQL:1999.

2 ESOTERIC PROGRAMMING LANGUAGES

Amongst all the groups of programming languages in which language designers like to claim their creations are Turing complete, one foregoes the most basic of constraints of making both the language's semantics and syntax sensible to read, write, let alone, reason in and about. Programming languages in this group are known as *esoteric programming languages*[3, 6].

One of the earliest examples of such a programming language is INTERCAL which was written by Don Wood and James Lyon in 1972 and revived by Eric S. Raymond in 1990 [15]. INTERCAL's sole design tenet is to satirize as many aspects of other programming languages as possible. For example, a proper INTERCAL program must contain an entirely undefined amount of the `PLEASE` keyword; too few and the compiler rejects the input program for *impoliteness*, too many and the compiler rejects for *excessive politeness*.

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

```
"!dlroW ,olleH">:#v_@
                ^ ,<
```

(a) Hello World

```
&>:1-:v v *_$.@
 ^    _$>\:^
```

(b) Factorial

```
211p&01p>121p          >21g1+21p 11g21g-v>11g21g%#v_v
                                        >|
                 v,,,,, ,,,,.g11"is prime"<
>       ^        >   v ^                           <
^_@#-g10p11:+1g11,*25<,,,,,,,,,,,,.g11"is not prime"<
```

(c) Prime Numbers

Figure 1: A collection of sample programs written in Befunge in increasing complexity.

Some esoteric programming languages take a more minimal approach, the most well-known examples of this are Wouter van Oortmerssen's FALSE [18] and Urban Müller's Brainfuck [9]. A simple *cat program*[1] in the latter can be written as `,+[-.,+]`. Both make use of a minimal amount of syntactic and semantic constructs to express arbitrarily complex computations. In short, Brainfuck simulates a Turing machine in a very direct manner, and FALSE operates what is known as a stack machine.

Brainfuck especially has become synonymous with esoteric programming languages in general. Aside from both its simplicity and absurdity, a core reason is that it is a Turing complete language [2] that is fairly direct to be understood as such. This feature has made Brainfuck the go-to target for reduction arguments concerning Turing completeness. Any language which can be used to implement a Brainfuck interpreter, with an "infinite" number of memory cells[2], is, in fact, Turing complete.

2.1 Befunge

Another esoteric programming language that follows in the footsteps of the likes of FALSE and Brainfuck is Befunge-93 [12] (from here on just Befunge). Befunge was invented by Chris Pressey in 1993 with the express intent to be difficult to compile. Befunge is a Turing complete, stack-based, self-modifying, two-dimensional language.

Befunge programs are arranged on a 2D character grid, dubbed the funge space, and each character represents a single command. At runtime, this funge space is traversed by the program counter, which can be in one of two execution modes: normal mode and string mode. During operation in normal mode, each command character the program counter "lands on" is executed, and any non-command character is treated as a comment. Some commands change the travel direction or step length of the program counter, some interact with the program stack to perform basic calculations and the like. In string mode, which is toggled by the `"` character, the ASCII value of all encountered characters is pushed onto the stack until the execution mode is toggled back.

In 1998, Pressey sought to extend both the feature set and the number of dimensions available in Befunge programs. In this process, he created then Funge-98 family of languages [13]. Aside from the newly introduced commands it also lifts the program size restriction of Befunge-93, which was capped at 80×25 characters. Without this size constraint, it is commonly accepted that

[1] A *cat program* is a program that simply copies its standard input to its standard output.

[2] In reality this is mostly a soft requirement for the most part, as many languages limit the program/stack/heap/etc. size artificially. The original Brainfuck reference implementation limited the number of memory cells to 30000.

```
1   def interpreter(source):
2       program = preprocess(source)
3       state   = make_inital_state(program)

5       while state.mode != "🏁":

7           match state.mode:
8               case "⚙": ...
9               case "🪡": ...

11          match state.direction:
12              case "➡": ...
13              case "⬇": ...
14              case "⬅": ...
15              case "⬆": ...

18      return state.result
```

Figure 2: Python-based skeleton implementation of a Befunge interpreter written in an iterative style and highlighted control flow regions.

Befunge-93 is Turing-complete, and with just a few Funge-98 specific commands it is possible to implement a Brainfuck interpreter [7].

In this paper, we are working towards showing the expressibility of Turing-complete control flow in SQL in a direct but fun manner. And as any Turing-complete language can be used to implement an interpreter for any other Turing-complete language, this makes a Befunge[3] interpreter a suitable sample program to flex SQL's muscles.

3 IMPERATIVE BEFUNGE INTERPRETER

There are many ways to styles of interpreters—tree-walk interpreters, bytecode interpreters, graph reduction interpreters, etc. Befunge lends itself to VM-like interpreters. Such interpreters are similar in style to bytecode interpreters, in that they emulate very simple semantics, the difference stems from VM-style interpreters not needing to compile given programs into bytecode beforehand. In short, our interpreter will retain the running program in exactly the form Befunge programs are supplied, in a 2D grid of ASCII values.

Figure 2 shows a rough sketch of a Befunge interpreter written in Python. Though not complete, the missing parts—i.e., the `...`-parts—are just more branching, some array-based stack manipulation, and simple integer arithmetic. The translation methods we

[3] The actual version of Befunge we are targeting is Befunge-93 without the program size constraint.

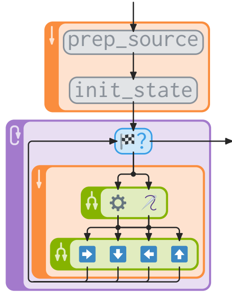

Figure 3: Control Flow Graph of the Befunge interpreter with the control flow regions highlighted respectively.

will discuss extend naturally over those parts as well. Both the complete translation and some Python scripts to make the SQL-based interpreter more useable can be found on Codeberg[4].

The interpreter ingests Befunge source code as a string which is first transformed into a 2D array for easier access. Afterward, we prepare the interpreter state which contains the funge space, the program counter, the stack and the execution mode. This program state is then iterated over until the execution mode signals termination. This iteration follows in two steps; in the first step, the interpreter ingests and processes the character the program counter resides on per the current processing mode. In the second step, the interpreter moves the program counter to the next character based on either the preexisting movement direction and step width or possibly updated variants of the two. On program termination, the interpreter then returns the generated output.

3.1 Concessions to Codd

In preparation for translating the interpreter to SQL, we need to make some concessions. Though Turing complete, there are some features that SQL does not and most likely will never support as they are entirely out of scope for a database query language. The biggest necessary concession is interactive I/O; Befunge's input and output commands usually interact directly with standard input and standard output, both of which SQL cannot interact with. To meet halfway our interpreter takes input up-front and returns all output at once upon completion.

3.2 Types of Control Flow

Out interpreter makes use of three major types of control flow: straight-line, branching, and looping. The boxes in Figure 2 box in the individual regions of different control flow and the control flow graph in Figure 3 illustrates how each of these control flow regions are linked to one another.

- Straight-line control flow pertains to a linear sequence of statements. This can be a series of assignments, bare expressions, etc.
- Branching control flow is, as the name implies, covers all branching, e.g., conditional branches like `if`, multiway branches like `switch`, etc. Though we will factor out fall-throughs in the latter in this paper to keep things simple. For example, Python's `match`, as used in Figure 2, can be used such that it fits this description.
- Looping control flow includes all looping constructs—be it while loops, for loops, for-each loops, you name it. Our interpreter only contains one loop and as such it only contains one region exhibiting non-linear control flow.

[4]The Git-Repository can be found at https://codeberg.org/timfi/befunge-sql.

```sql
WITH
  _1(program) AS (
    SELECT preprocess(source)
  ),
  _2(state) AS (
    SELECT make_initial_state(program)
    FROM   _1
  )
SELECT ... FROM _1, _2
```

(a) ① Consecutive CTEs

```sql
SELECT ...
FROM   LATERAL (
         SELECT preprocess(source)
       ) AS _1(program),
       LATERAL (
         SELECT make_initial_state(program)
       ) AS _2(state)
```

(b) ② Consecutive LATERAL-Joins

```sql
SELECT preprocess(source)          AS program,
       make_initial_state(program) AS state
```

(c) ③ Left-Looking Column-Alias-References

Figure 4: Linear control flow in multiple representations in SQL.

4 FLEXING SQL'S MUSCLES

No relational database management system (RDBMS) implements the entire SQL current standard, so we limit ourselves to the 1999 standard [17]. SQL:1999 introduced both recursive CTEs and lateral joins; both of which we will make use of to give general transformations for translating imperative control flow to SQL. Most mainstream RDBMSs today—i.e., PostgreSQL[11], SQL Server[16], MySQL[10], etc.—implement these two features. In the context of this work, we will work towards translations to PostgreSQL-style SQL.

4.1 Straight-Line Control Flow

There are multiple options to represent straight-line control flow in SQL, we will go through the pros, cons, and availability of three of these.

① The first option is to chain the individual statements in the linear sequence as individual common table expressions(CTEs)—as illustrated in Figure 4a. Most developers know of this technique as it is the go-to way to decompose complex queries into simpler constituent parts.

② The second option is to make use of `LATERAL`-joins—as illustrated in Figure 4b. These allow query authors to use row and

```sql
SELECT CASE direction
  WHEN '➡' THEN ...
  WHEN '⬇' THEN ...
  WHEN '⬅' THEN ...
  WHEN '⬆' THEN ...
END CASE
```

(a) ① CASE Expressions

```sql
SELECT ...
WHERE  direction = '⚙'
  UNION ALL
SELECT ...
WHERE  direction = '?'
```

(b) ② Mutually Distinct UNION ALL

Figure 5: Branching control flow in multiple representations in SQL.

column variables inside a table expression from other table expressions preceding it in the same FROM-clause.

③ The last option is to use left-looking column-alias-references; a non-standard feature not available in many RDBMS—as illustrated in Figure 4c. It allows for LATERAL-esque references to preceding expressions in the projection list of a SELECT-clause.

From a purely ergonomic perspective ③ left-looking column-alias-references win by a long shot. Sadly, few large RDBMSs support them as they are a non-standard feature. Additionally, in most implementations, there are limitations as to what expressions can be referenced. For example, in DuckDB [14] one can only reference aliases bound to what they call "simple expressions"—for example, this excludes subqueries.

Both ① CTEs and ② LATERAL-joins are equally supported by a very wide set of RDBMSs. There are multiple factors at play when choosing one over the other. For example, depending on how performant your RDBMS of choice implements CTEs there may be performance benefits or drawbacks over LATERAL-joins. In this paper, we will make use of the LATERAL-join variant due to how it interacts with the translations of the other two control flow types.

4.2 Branching Control Flow

SQL provides two options for representing branching behavior, one using SQL's expression language and one using SQL's relational design.

① SQL's expression language offers branching control flow in the form of CASE-expressions as in Figure 5a.

② Aside from the direct option, pure relational algebra can represent conditional evaluation as well. Inheriting from set theory, mutually distinct unions, as illustrated in Figure 5b, allow for expression of "optional execution".

Though the more direct solution of the two, ① CASE-expressions come with some difficulties depending on the targeted RDBMSs. The type-checking/-inference systems of some RDBMSs are weak and require type annotations. Some of these type systems do not support the ad-hoc generation of complex types in scalar positions, e.g., row types with named columns.

When trying to make things work out with CASE-expressions while skirting around the type system limitations, one option is to "push down" the CASE to each scalar value. But this leads to subpar performance in most RDBMSs. In contrast, RDBMSs are exceptional at performing relational workloads like computing ② mutually distinct unions. Some RDBMSs optimize such workloads in such a manner that the "lazy" behavior of branching control flow is retained.

```sql
WITH RECURSIVE
  loop(done, ...) AS (
    SELECT false, ...
      UNION ALL
    SELECT ...
    FROM   loop
    WHERE  NOT loop.done
  )
SELECT ...
FROM   loop
WHERE  loop.done
```

Figure 6: Looping control flow represented through a recursive CTE (i.e., WITH RECURSIVE).

4.3 Looping Control Flow

In contrast to the previous two types of control flow, looping control flow requires Turing completeness and there is only one SQL:1999 option for Turing completeness in SQL: WITH RECURSIVE. Recursive CTEs consist of two distinct parts, the base case and the recursive term. Starting from the base case the recursive term is iteratively applied to the previously produced rows. This iterative application is performed until no rows are produced, at which point the rows produced during each iteration are considered the result of the CTE.

Due to WITH RECURSIVEs bare-bone semantics, the only directly equivalent loops are do-while-loops. As such, translating all other loop constructs starts with translating them into do-while-notation. This usually entails adding extra control data like indexes or flags to the local program state.

There are some performance considerations to take into account when translating imperative loops to SQL. Depending on how your underlying RDBMS keeps track of the rows the individual iterations produce, a lot of data may need to be copied around in memory. A slight remedy for these performance pains is to keep loop-invariant and transient data out of the working table.

4.4 Putting It All Together

The translated versions of the three control flow types compose naturally to encapsulate the complex control flow of entire programs. This composition is accomplished by linking the translations of sequential regions together via LATERAL-joins and placing embedded regions into FROM-clause of their enclosing region. Figure 7 demonstrates how this composition of individually translated control flow regions looks like for our skeleton Befunge interpreter introduced in Figure 2.

Translations of complex imperative programs using these simple composition rules can be very verbose. Though this verbosity does not necessarily present problems for developers, the produced queries can be compacted quite a bit. One such simplification is to merge straight-line control flow preceding a section of looping control flow into the base case of the recursive CTE. A more complicated "simplification" is to preprocess the program such that all looping control flow is merged into one massive loop, i.e., trampolined style [4].

```sql
SELECT result
FROM   LATERAL (
         SELECT preprocess(source)
       ) AS _1(program),
       LATERAL (
         SELECT make_initial_state(program)
       ) AS _2(state),
       LATERAL (
         WITH RECURSIVE
           loop(done, ...) AS (
             SELECT false, state.*
               UNION ALL
             SELECT next.mode = '🏁', next.*
             FROM   loop,
                    LATERAL (
                      SELECT ...
                      WHERE  current_state.mode = '⚙'
                        UNION ALL
                      SELECT ...
                      WHERE  current_state.mode = '🪡'
                    ) AS update,
                    LATERAL (
                      SELECT ...
                      WHERE  update.direction = '➡'
                        UNION ALL
                      SELECT ...
                      WHERE  update.direction = '⬇'
                        UNION ALL
                      SELECT ...
                      WHERE  update.direction = '⬅'
                        UNION ALL
                      SELECT ...
                      WHERE  update.direction = '⬆'
                    ) AS move,
                    LATERAL (
                      SELECT compose_state(update, move)
                    ) AS next
             WHERE  NOT loop.done
           )
         SELECT current_state.result
         FROM   loop
         WHERE  loop.done
       ) AS _3(result)
```

Figure 7: Fully translated skeleton as depicted in Figure 2.

5 WRAPPING UP

Writing algorithms and programs in SQL that exhibit complex control flow can be difficult for a multitude of reasons. First and foremost, are the impedance mismatch between the imperative form many programs are written in and the relational declarative nature of SQL. Further, some classes of problems are notoriously difficult to express succinctly in SQL, like optimization problems.

In this paper, though through an unconventional example, we demonstrated that any imperative program, even Turing-complete ones, can be expressed in SQL. In the process, we discussed each of three distinct types of control flow imperative programs exhibit—straight-line, branching, and looping. And for each of these, we considered some options for translation to SQL.

REFERENCES

[1] Donald D. Chamberlin. 2012. Early History of SQL. *IEEE Annals of the History of Computing* 34, 4 (2012), 78–82. https://doi.org/10.1109/MAHC.2012.61

[2] Daniel B. Cristofani. 2020. *Universal Turing Machine in Brainfuck*. http://brainfuck.org/utm.b

[3] Matthew Fuller. 2008. *Software Studies: A Lexicon* (illustrated edition ed.). The MIT Press. https://mitpress.mit.edu/9780262062749/software-studies/

[4] Steven E. Ganz, Daniel P. Friedman, and Mitchell Wand. 1999. Trampolined Style. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming* (Paris, France) *(ICFP '99)*. Association for Computing Machinery, New York, NY, USA, 18–27. https://doi.org/10.1145/317636.317779

[5] Alan C. Kay. 1993. The Early History of Smalltalk. In *The Second ACM SIGPLAN Conference on History of Programming Languages* (Cambridge, Massachusetts, USA) *(HOPL-II)*. Association for Computing Machinery, New York, NY, USA, 69–95. https://doi.org/10.1145/154766.155364

[6] Michael Mateas and Nick Montfort. 2005. A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics. In *6th Digital Arts and Culture Conference*. 144–153. https://eis.ucsc.edu/papers/a_box_darkly.pdf

[7] Matus Goljer. 2009. *Brainfuck interpreter written in Befunge-98*. https://web.archive.org/web/20111108050613/http://fi.muni.cz/~xgoljer/bf.txt

[8] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. 1962. *LISP 1.5 Programmer's Manual*. The MIT Press. https://mitpress.mit.edu/9780262130110/lisp-1-5-programmers-manual/

[9] Urban Müller. 1993. Brainfuck. http://aminet.net/package/dev/lang/brainfuck-2

[10] MySQL Team. 2023. *My Server*. Oracle. https://www.mysql.com/de/

[11] PostgreSQL Developers. 2023. *PostgreSQL*. The PostgreSQL Global Development Group. https://www.postgresql.org/

[12] Chris Pressey. 2018. Befunge-93 Documentation. https://catseye.tc/view/Befunge-93/doc/Befunge-93.markdown

[13] Chris Pressey. 2018. Funge-98 Final Specification. https://catseye.tc/view/Funge-98/doc/funge98.markdown

[14] Mark Raasveldt and Hannes Muehleisen. 2023. *DuckDB*. DuckDB Foundation. https://duckdb.org/

[15] Eric S. Raymond. 2003. The INTERCAL Resources Page. http://www.catb.org/~esr/intercal/

[16] SQL Server Team. 2023. *SQL Server*. Microsoft. https://www.microsoft.com/de-de/sql-server/

[17] SQL:1999 1999. *SQL:1999 Standard. Database Languages–SQL–Part 2: Foundation*. ISO/IEC 9075-2:1999.

[18] Wouter van Oortmerssen. 2010. The FALSE Programming Language. https://strlen.com/false-language/