



Bachelorthesis Computer Science

How expressive is Flummi really? Can it run Advent of Code?

Fredo Hogen

31.07.2024

Examiner

Prof. Dr. Torsten Grust

Co-Examiner

-

Supervisor

Tim Fischer

Fredo Hogen:

How expressive is Flummi really? Can it run Advent of Code?

Bachelorthesis Computer Science

Eberhard Karls Universität

From 01.04.2024 to 31.07.2024

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorthesis selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorthesis wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum

Fredo Hogen

Acknowledgement

Abstract

Contents

Acknowledgement	v
Abstract	vii
Acronyms	xi
1 Introduction	1
2 Basics	3
2.1 Language	3
2.2 The used Database Management System	4
2.2.1 The Compilation of PL/Flummi to SQL	4
2.2.2 What features of DuckDB does PL/Flummi use	5
2.3 The Advent of Code	5
3 Implementation	7
3.1 Chosen Problems	7
3.1.1 High-Entropy Passphrases	7
3.1.2 Infinite Elves and Infinite Houses	8
3.1.3 Hill Climbing Algorithm	8
3.1.4 Pipe Maze	9
3.1.5 Firewall Rules	10
3.2 General Implementation	10
3.2.1 Common Patterns	10
3.2.2 Debugging	11
4 Evaluation	13
4.1 Time of Learning and starting to use PL/Flummi	13
4.1.1 The Learning Process	13
4.1.2 Features of PL/Flummi, comparing to other Languages	14
4.2 Implementing in PL/Flummi	15
4.2.1 Tools for Development	16
4.2.2 Ease of Development	17
4.3 Executing	18
4.3.1 High-Entropy Passphrases	18
4.3.2 Infinite Elves and Infinite Houses	19
4.3.3 Hill Climbing Algorithm	20
4.3.4 Pipe Maze	21
4.3.5 Firewall Rules	21
4.4 General	22
5 Conclusion	23
5.1 Conclusion	23
5.2 Future Work	23
Bibliography	25

Acronyms

AoC Advent of Code

BNF Backus-Naur-Form

DBMS Database Management System

IDE Integrated Development Environment

SQL Structured Query Language

Introduction

Provided the esoteric procedural programming language **PL/Flummi** which is aiming to be compatible with multiple Database Management System (**DBMS**), we are going to put it to use and evaluate it regarding its expressiveness and general usability.

To reach this goal we first got acquainted with the programming language itself. We learned the grammar of the language to be able to write **PL/Flummi** programs from scratch using all features provided. In order to be compatible with multiple **DBMS**, **PL/Flummi** is being compiled into their respective Structured Query Language (**SQL**) variant. In our case we use the **DuckDB DBMS** [1], which provides us with all its data types [2, 3] and functions [4] to be used in the **PL/Flummi** code. To leverage its features we are able to pass **DuckDB** statements through **PL/Flummi** statements. This allows us to use **DuckDB** types for variable declarations and functions to interact with variables of these types.

We implement solutions for a selection of problems from the Advent of Code (**AoC**) project [5]. This allows us to put **PL/Flummi** to practice on different kinds of algorithms, including string processing, path finding and more. To compare it to already established programming languages, we used solutions of different languages as reference. This gives us benchmarks to evaluate **PL/Flummi** upon and determine its strengths and weaknesses.

Once familiar with **PL/Flummi** and with a variety of **AoC** problems on our hands we started implementing the solutions. In the process we gained many impressions on what works well and what does not. Since the constructs for loops and if-else clauses require more boilerplate code to fully use, the **PL/Flummi** code ends up being far more bloated, than the code references we aim to mimick. Additionally we have to implement the loop logic ourselves, conventional loop constructs like while- or for-loops do not exist in **PL/Flummi**.

Still the features provided by **PL/Flummi** allowed us to write code resembling the referenced code. Meaning we were able to fully express the algorithmic solutions of all our selected **AoC** problems [6, 7, 8, 9, 10]. Leveraging the type and function toolkit of the underlying **DuckDB DBMS** allowed us to represent even complex data structures through **PL/Flummi**.

Compiling and running the **PL/Flummi** code works quite differently than the way we are used to from powerful Integrated Development Environment (**IDE**s). There is no tooling whatsoever to aid in **PL/Flummi** development. Running and executing the code requires us to always use the command line to firstly compile the code and secondly run the code for each development iteration. Debugging support does not exist at all, our only source of error feedback are the error outputs from the compiler and the **DBMS** running the compiled **SQL** code. The lack of tools slows down the programming speed significantly, but does not make it impossible by any means.

Executing all our solutions we ran into a few more problems though, that we were not able to

solve. The usage of complex data structures in the form of lists fill up the memory of the host machine fully. This happens due to faults in the memory management of lists in the underlying **DuckDB DBMS** [11] and lack of memory optimization, holding countless copies of the same data in memory. Another hindrance during runtime are extremely high execution times, especially on high iteration loops containing non mundane logic.

Basics

In the scope of databases procedural programming languages are common practice. These supply the **DBMS** with functionalities like conditionals or loops to be used for the creation of custom extension functions. Well-known proprietary examples are **PL/SQL** for the Oracle **DBMS**, **T-SQL** for the Microsoft **SQL** Server and **PL/pgSQL** for the PostgreSQL **DBMS** [12, 13, 14]. These programming languages, however, while being highly optimized on a rather low level, share one downside. Which is being coupled to only their respective singular **DBMS**.

PL/Flummi on the other hand strives to give more flexibility in the choice of the underlying **DBMS**. Leveraging their individual strengths and features through injected **SQL** expressions. We will present a more detailed look into how **PL/Flummi** intends to achieve this goal and what drawbacks are entailed in the following contents of this chapter.

To get an overview over what scope of algorithms we will implement in **PL/Flummi** to evaluate it, we will also take a look at the **AoC**.

2.1 The Language **PL/Flummi**

A **PL/Flummi** program always consists of only one function. This function is called for each entry in the target table. The function execution hereby is also parallelized. This means the output of the execution on the first entry is not available to be read in any way in the execution of the second entry. In the **CALL** statement, the columns of the table required for execution are specified in the form of expressions.

$$P := \text{CALL } E(, \dots, E) \text{ IN } F$$

The definition, which values of which type are required for the above **CALL** statement, as well as the type of the output value are defined in the following function statement.

$$F := \text{FUN } v(: \tau, \dots, v: \tau) \rightarrow \tau: S$$

The statements provided by the **PL/Flummi** language enable the programmer to write familiar-looking and readable code. Any variables need to first be explicitly declared and afterward can be assigned their value in one or more following statements.

Even while working in this restrained programming language, the most basic tools for **LOOP** control and conditionals are provided. As well as **EMIT** and **STOP** statements to output the result and terminate the program execution.

All statements can also be a sequence of statements enclosed by curly braces. This is primarily applicable to loops and conditionals. Another peculiarity of **PL/Flummi** lies in the lack of a return statement. As only one function is supported by the language, instead of returning a value to another calling function, the program instead outputs intermediate or final results.

```
S := v : τ
    | v <- E
    | EMIT E
    | IF E THEN S ELSE S
    | LOOP v S
    | CONTINUE v
    | BREAK v
    | STOP
    | NOOP
    | { S; ...; S }
```

A powerful perk of the **PL/Flummi** language is the ability to pass **SQL** statements for the used **DBMS** into the compiled **SQL** code through expressions. This enables the use of logical, numerical or any other arbitrary function or expression the **DBMS** provides. Similarly, the available types for the variables are specified through the underlying **DBMS**.

To be able to compile any **PL/Flummi** code into **SQL** code for different **DBMS**s these are the critical points to adjust, as they are the only DB-specific properties of the **PL/Flummi** code. Expressions and types have to match the **DBMS** in use.

```
E := §<\ac{sql} expression>§v[, ..., v]
τ := §<\ac{sql} type>§
v := <variable>
```

2.2 The used Database Management System

2.2.1 The Compilation of PL/Flummi to SQL

The focus of this work lies upon evaluating the expressiveness and functionality of **PL/Flummi** code. So it is not important to understand and follow the process of the compilation. Only the fact that **PL/Flummi** is being compiled to **SQL** code matters at this point. As this prescribes how to work with and use the compiled code.

Additionally this gives a foundation for a benchmark when comparing the compiled code with code written explicitly in **SQL** concerning efficiency and runtime.

2.2.2 What features of DuckDB does PL/Flummi use

As long as the the functionality of **PL/Flummi** is not being bypassed, **DuckDB** functions are available to be used [4]. To be able to use **PL/Flummi** at all, basic functions are required. Those include:

- Numeric Functions, to be able to control values to be used in conditionals [15]
- Comparison Operators, to be able to evaluate conditionals with given values [16]
- Text functions, to be able to parse the inputs of the problems [17]

Besides the most necessary functions, also more specific ones are being used. They are needed to work with the different data formats and requirements of the algorithm implementations. These are for example composite datatypes [3] to work with more complex data structures. But as the focus of this work lies upon evaluating the **PL/Flummi** language a more extensive use of **DuckDBs** features is renounced while it can be avoided.

2.3 The Advent of Code

To be able to put **PL/Flummi** to practice, we will implement solutions to problems of the **AoC** project [5]. The **AoC** is a yearly recurring advent calendar taking place from the first to the 25th december since 2015. The problems presented in this advent calendar are of varying complexity and are solvable in a short timeframe. The goal every day is to be of the fastest solvers after release to rank high in the top list each day and year.

The small scope of these problems makes them solvable in only a single function applying usually one algorithm. Given this manageable extent of required code makes these problems great candidates to be solved in **PL/Flummi**.

Implementation

Now that we have an understanding on what tools we are working with it is time to put them to practice. We will take a look at the algorithms that were implemented and highlight parts of code to understand the implementation process.

3.1 The Problems of Choice

As the motivation for this work is to evaluate the expressiveness of **PL/Flummi** fundamentally different problems to implement have been chosen. The algorithms cover very different levels of cyclomatic complexity and data structures required to represent the domain of the unique problems. Starting with basic string processing on small amounts of data, through numeric operations in a larger scope, up to more complex data structures and solving pathfinding problems. Considering the fact that this work is not about finding solutions for the AoC problems, but implementing solutions in **PL/Flummi**, pre-implemented solutions in other languages are, for the most part, used as reference for that matter.

3.1.1 High-Entropy Passphrases

To start off basic and to get to warm up using **PL/Flummi**, we chose the problem "High-Entropy Passphrases" [6]. To be able to lean upon a preexisting solution for implementation and comparison later one, we use an implementation in **Python** for reference [18]. Since this algorithm just requires basic string processing, the implementation only uses the standard library. This in turn makes it easy to implement mostly matching code in **PL/Flummi**.

It is quite a straightforward problem, that does not necessitate extensive use of complex data types or algorithmic implementations. The solution of this problem merely requires reading the input and processing strings iteratively through loops. In order to implement string processing and list operations the respective functions of **DuckDB** are being called through **PL/Flummi** expressions.

```

1 element_to_sort <- $string_split({0}[{1}], '')${elements, sorting_iterate_count};
2 element_to_sort <- $list_sort({0})${element_to_sort};
3 elements_sorted <- $list_append({0}, {1})${elements_sorted, element_to_sort};

```

The control of the iteration through **PL/Flummi**'s conditional and loop statements are of course natively implemented.

```

1 LOOP sorting_loop {
2     ...

```

```

3     IF ${0} = {0} ${sorting_iterate_count}
4     THEN BREAK sorting_loop
5     ELSE sorting_iterate_count <- ${0} - 1 ${sorting_iterate_count}
6 };

```

Work with this problem gives us a good understanding and impression on what the toolkit **PL/Flummi** presents. Additionally it made clear to us how and when to interact with the underlying **DBMS** to execute operations on data, that would be impossible to achieve otherwise.

3.1.2 Infinite Elves and Infinite Houses

As the problem "Infinite Elves and Infinite Houses" [7] is little bit more advanced an algorithmic standpoint, we fall back onto using code reference for the implementation in **PL/Flummi** again. The used code reference [19] is written in the C programming language. The basic nature of the C code without any libraries makes it quite clear in how all the constructs have to be translated to **PL/Flummi** equivalents.

This problem already presents different challenges for **PL/Flummi**. The algorithm only requires numeric operations, no handling of complex data. But on the other hand the number of loop iterations during execution has significantly increased. This may deliver new and different insights into how well **PL/Flummi** is able to tackle varying compilations of complexity of data structures and high numbers of required loop iterations. Regarding the used functions of the underlying **DBMS** the algorithm also looks a lot more basic. Only numeric and logical operations are required for the implementation. Like for example:

```

1     sqrt <- ${sqrt}({0}) ${house_no};

```

or

```

1     IF ${0} % {1} == 0 ${house_no, divisor}

```

Regarding other properties or peculiarities of **PL/Flummi** and the usage of the **DBMS**, this algorithm did not provide any new insights.

3.1.3 Hill Climbing Algorithm

In order to increase the complexity of a solution to be implemented we chose the "Hill Climbing Algorithm" [8]. This algorithm once again is based on a existing solution [20], which is implemented in Python, only using the standard library. This in turn allows the basic algorithm implementation to be easily translated into its **PL/Flummi** counterpart. All array and other needed functions in the python solution have an equivalent in **DuckDB**.

As this solution requires the implementation of the BFS pathfinding algorithm, the complexity of data as well as the cyclomatic complexity of the programm drastically increase. The complex data structures that are used in this case are made up of nested structs and lists. As in this variable for example:

```

1     level: ${struct}(pos int[], x int)[] ${};

```

This extensive use of complex data structures results in heavy reliance on **DuckDBs** functionality to work with this kind of data. Basically every interaction and manipulation of data has to be managed by passing **DuckDB** functions through **PL/Flummi** expressions. However, this is completely compliant to programming in **PL/Flummi**, as it only implements basic statements to control the flow of the program, as discussed in chapter **PL/Flummi**.

```

1 IF ${list_filter}({0}, x -> x.pos == {1}) == []${level, neighbor}
2 THEN {
3   new_level <- ${struct_pack}(pos := {0}, x := {1} + 1)${neighbor, current_elem_level};
4   level <- ${list_append}({0}, {1})${level, new_level};
5   elems_to_traverse <- ${list_append}({0}, {1})${elems_to_traverse, neighbor}
6 } ELSE NOOP;

```

Additionally it is important to be noted, that the number of unique variables containing data is significantly higher compared to previous algorithms. This fact coupled with the usage of nested loops to iterate over and manipulate these values, results in significantly higher memory workload. Unfortunately this high memory usage cannot be solved through spilling data to the disk [11]. In the **DuckDB DBMS** the aggregate functions, which the frequently used list is one of, are not managed by the buffer manager. The painful consequence is, that during runtime the memory will not be spilled to disk and consequentially overflow memory, which crashes the host machine.

3.1.4 Pipe Maze

To further diversify the implemented algorithms, we choose the "Pipe Maze" [9] problem. Which also is implemented in **PL/Flummi** based on reference code [21]. The reference code used once again is python code that only uses the standard library to ensure the solution can cleanly be translated into **PL/Flummi** code.

The solution to "Pipe Maze" is a step back from the "Hillclimb Algorithm" regarding complexity of data and algorithm, but retains high cyclomatic complexity. Nonetheless this algorithm faces similar issues regarding memory workload. At first glance the algorithmic solution needed to solve this problem merely requires one list besides primitive types and no nested loops. The one thing this algorithm implements though are a high number of conditionals and nested conditionals. These are necessary to control the movement of the cursor in the traversal loop depending on its state, direction and position.

```

1 IF ${0} == 'S'${direction}
2 THEN y <- ${0} - 1${y}
3 ELSE NOOP;
4
5 ...
6
7 IF ${0}[{2}][{1}] == 'L'${matrix, x, y}
8 THEN {
9   IF ${0} == 'N'${direction}
10  THEN direction <- 'E'${}
11  ELSE direction <- 'S'${}
12 } ELSE NOOP;

```

As this algorithm requires only one list besides primitively typed variables, the usage of **DuckDB** functions is very limited. Only few list functions need to be consulted to prepare data for the traversal loop. Besides that only basic numeric, logic and list reading operations are being used for the actual traversal algorithm.

3.1.5 Firewall Rules

Unlike all previous problems the "Firewall Rules" problem [10] presents us with whole new challenges. The requirements for this solution cover large numeric values and a long list of data to be processed efficiently. To have a foundation on how to implement the solution efficiently, we choose a implementation for the solution in python with only the standard library as reference once again [22].

In order to solve the problem, the data read from input first has to be strategically prepared for the executing algorithm itself. To do so all lines of the input are read, parsed from string to int and sorted ascendingly. The herefore needed **DuckDB** functions represent the only special functionality, as for the algorithm itself only basic numeric functions and reading from lists are required.

```
1 LOOP input_loop {
2   row <- $SELECT row FROM input WHERE rowid == {0}$[iterate_count];
3   record <- $string_split({0}, '-')$[row];
4   records <- $list_append({0}, [{1}[1], {1}[2]])$[records, record];
5
6   IF $1 = {0}$[iterate_count]
7     THEN BREAK input_loop
8     ELSE iterate_count <- ${0} - 1$[iterate_count]
9 };
10
11 records <- $list_sort({0})$[records];
```

This preparation allows the algorithm to skip big parts of the given input.

3.2 General Implementation

To close things up regarding the implementation process, we will address necessities and difficulties during development time, that all these algorithms and **PL/Flummi** in general share.

3.2.1 Common Patterns

All these programs share common building blocks. Regarding the structure of the program, they all for the most part follow the order of:

1. declaring all variables
2. reading and preparing the input
3. execute the core algorithm
4. output the result

The algorithms requiring a single input value will read it from the input table in a single `SELECT` statement. If the input however is split into multiple entries in the input table and intermediate results are required to be shared between algorithm iterations, it is necessary to iterate over all of them. We have done this fetching the total number of rows via:

```
1  iterate_count <- $SELECT MAX(rowid) FROM input$[];
```

and subsequently iterating over all table entries using the `iterate_count` as the limiting `row_id`.

```
1  LOOP input_loop {
2    row <- $SELECT row FROM input WHERE rowid == {0}$[iterator];
3
4    ...
5
6    IF ${0} >= {1}$[iterate_count, iterator]
7      THEN BREAK input_loop
8      ELSE iterator <- ${0} + 1$[iterator]
9  };
10
```

At this point we should note the discrepancy between the `row_id` and the indices of for example lists in **DuckDB**. The table rows are 0-based while the other indices are 1-based. This may lead to confusion when first reading **PL/Flummi** code, but is limited to this one exception of table rows being 0-based.

3.2.2 Debugging

One big challenge, when working with **PL/Flummi** is to find and resolve mistakes made. The compiler itself provides relatively precise output in which statement line or statement block compilation errors occur. Meaning mistakes that errors made, that result in code that is not compilable, are quite easy to spot and resolve after having some experience with the grammar of the [language](#). To spot errors in the **SQL** code after compilation is also straightforward, as we can just analyze the output of the execution as usual when working with **SQL** itself.

However things get tricky on the other hand, when trying to find errors in the algorithm. In case of for example wrong conditions in an **IF**-statement, there is no way of debugging the code step by step, as it is common in other languages. To be able to have any insight into intermediate values during runtime, the only way is to output the data via the **EMIT** statement. The drawback hereby is, that the program has to terminate to emit any values. So sometimes falling back to manually debugging the code step by step is necessary.

Evaluation

Now we have implemented a variety of algorithms in **PL/Flummi**. From basic string processing, over processing large amounts of numeric values, up to a path finding algorithm requiring complex data structures. The process of working with these problems gave us great insights into the workings and limitations of **PL/Flummi**. Using these insights and experiences we will evaluate different aspects of working with **PL/Flummi**.

4.1 Time of Learning and starting to use PL/Flummi

All things considered the learning process is not too hard. Granted the person learning to use the language is already proficient in working with other programming languages leveraging similar constructs.

4.1.1 The Learning Process

Given the fact **PL/Flummi** is a research language of the Database Systems chair in the University Tübingen, this is the only source of information regarding the **PL/Flummi** project. Hence our only learning resources are provided by them. To learn the language we were given a handful of sample programs additionally to the Backus-Naur-Form (**BNF**) grammar definition. These sample programs are ranging from a counter outputting one to ten, to a raytracing algorithm. In these programs all features of **PL/Flummi** were leveraged, providing us with enough information to write the code ourselves. Still there were cases, where the problems learning the language could not be resolved only using reference programs. In these cases a tutor of the Database Systems chair provided valuable advice.

This was the case with understanding how **PL/Flummi** works with multiple rows of the input table when using the **CALL** statement. As the initial understanding of how **CALL** works often times may be, that all rows will be successively be called by the program and evaluated one after the other.

```

1 CALL $($supkey[], $$orderkey[]) IN
2 FUN (supkey: $$int, orderkey: $$int) -> $$bool: {
3     ...
4 }
```

Listing 4.1: Table columns listed in **CALL** statement, function called for each input table row

And this entails another assumption to be made. That the output or in this case **EMIT** of the program of the previous table row is in any way accessible in the execution of the program for the

next table row. Which it is not. All rows are independently evaluated, even partially in parallel. So there is no way to share any computed data between the executions of the different table rows.

This discovery leads to the iteration over the input data to be done in the program itself. The `CALL` statement will therefore be left empty in the scope of implementing solutions to AoC problems. These almost exclusively require evaluating multiline input to solve the problem.

```

1  CALL () IN
2  FUN () -> $$vchar: {
3      iterate_count <- $$SELECT MAX(rowid) FROM $input[];
4      records <- $$[];
5
6      LOOP input_loop {
7          row <- $$SELECT row FROM input WHERE rowid == ${0}[iterate_count];
8          record <- $string_split({0}, '-')${row};
9          records <- $list_append({0}, [{1}[1], {1}[2]])${records, record};
10
11         IF $1 = ${0}[iterate_count]
12             THEN BREAK input_loop
13             ELSE iterate_count <- ${0} - $1[iterate_count]
14         };
15
16         ...
17     }

```

Listing 4.2: Empty `CALL` statement, input read through loop iterating over table

4.1.2 Features of PL/Flummi, comparing to other Languages

The language features provided by **PL/Flummi** are quite limited. Only very basic constructs are implemented. Firstly we only have basic `IF` and `ELSE` statements at our hands. If we for example want to only evaluate an `IF` statement, we have to still write the corresponding `ELSE` statement containing a `NOOP` statement.

```

1  IF $NOT {0} == $0[x_start]
2      THEN y_start <- $$${0}[iterator]
3      ELSE NOOP;

```

Listing 4.3: `IF` statement in PL/Flummi

Comparing this `IF` without `ELSE` statement in **PL/Flummi** with its **Python** equivalent shows how much more code to write this basic statement is required.

```

1  if x_start != 0:
2      y_start = iterator

```

Listing 4.4: `IF` statement in Python

This results in quite heavy boilerplate code in cases like this. Secondly we have basic `LOOP` implementation at our disposal. The `LOOP` will only stop iteration, when the `STOP` statement is executed. We manually have to place this `STOP` statement inside conditionals to control the flow of the `LOOP`.

```

1 iterate_count <- $SELECT MAX(rowid) FROM $input[];
2
3 LOOP input_loop {
4   ...
5
6   IF $1 = ${0}[iterate_count]
7   THEN BREAK input_loop
8   ELSE iterate_count <- ${0} - $1[iterate_count]
9 };

```

Listing 4.5: Input LOOP construction using BREAK statement in PL/Flummi

Taking a look at the equivalent implementation of this LOOP, we can once again see a big difference in required boilerplate code to achieve basic tasks like a simple loop.

```

1 for line in open("input.txt", "r").readlines():
2   ...

```

Listing 4.6: Input LOOP equivalent in Python

Us needing to build the loops by hand also means, there are no pre-built constructs like for- or do-while-loops. We have to implement the logic required for that functionality ourselves. This enables us to construct all common LOOP types, but once again entails boilerplate code to reach our goal of implementing a working LOOP. PL/Flummi also provides the CONTINUE statement to skip iterations or parts of an iteration, as it is common in many other languages as well.

We will, as this paragraph suggests, not find any syntactic sugar in PL/Flummi. All constructs have to be implemented by hand and no shortcuts are provided to do so. Also there is no shortcut around the separate declaration and initialization of variables. It is always necessary to declare the variables to be able to assign values later on. A way to do this, like in many modern languages, does not exist.

```

1 house_no: $$int;
2 ...
3 house_no <- $$1[];

```

Listing 4.7: Variable declaration and initialization in PL/Flummi

```

1 house_no = 1

```

Listing 4.8: Variable declaration and initialization in Python

4.2 Time of Implementing in PL/Flummi

Implementing the AoC solutions in PL/Flummi enabled us to gain impressions on where the capabilities and weaknesses of PL/Flummi lie. We were able to implement all algorithms for the problems we chose to solve. The process of implementing them also was, regarding the language itself, not too different from other languages. Only the already mentioned required boilerplate code bloats up the programs a bit. Where it got tricky though, were the tools for development, that we had at our disposal. This will be further discussed in the contents of this section.

4.2.1 Tools for Development

Being a very young language, the toolkit is highly limited. To develop in **PL/Flummi** we only have a code editor and a command line of our choice to use. As there exists no integration of **PL/Flummi** into existing development environments, we have to do all steps from compilation ...

```
1 py -m flummi compile path/to/program.fl
```

Listing 4.9: Shell statement to compile **PL/Flummi** code, while in compiler root directory

... to execution through the command line.

```
1 cat path/to/input.sql path/to/compiler-out.sql | duckdb
```

Listing 4.10: Shell statement to run **SQL** code compiled from **PL/Flummi** code

This makes the development process more tedious than we are used to using **IDEs** with integrated shortcuts for compiling and executing. For each development session we start on a freshly booted computer, we have to prepare the necessary statements for compilation and execution in the command line. To re-execute the statements we then can use the arrow keys. But we still have to switch to the console and execute two statements for each implementation iteration to execute the code, instead of being able to execute one shortcut in the **IDE** itself.

Another drawback of missing integration in any development tools is, that we don't have any immediate feedback at the time of writing code. There is no display of any syntax, compilation or logical errors or any warnings for that matter either. This makes it necessary to always go through the steps of compiling and executing, to get the command line output of the compiler or **DBMS** as our only source of feedback.

The feedback provided by the compiler allows us to find the cause of the errors reliably, after getting comfortable with the syntax and quirks of the language.

```
1 py -m flummi compile ../path/to/program.fl
2
3 Traceback (most recent call last):
4
5 ...
6
7 File "C:\path\to\compiler\flummi\parser.py", line 243, in parse_statement
8 raise self.error("Expected statement")
9 SyntaxError: Expected statement (at 30:2)
```

Listing 4.11: Example syntax error output of the compiler, providing line and column of error source

When running into errors when executing the compiled **SQL** script, we have to rely on the output of the **SQL** error to provide the necessary information. The information provided in the error output allows us to spot faulty **SQL** expressions, that we passed through using **PL/Flummi** expressions. Knowing which expression in the **SQL** script caused the error usually lets us trace back to the **PL/Flummi** expression passing through the broken **SQL** expression and fix it. A few steps are required to troubleshoot this way, but it is still possible.

```
1 cat path/to/testdata.sql path/to/compiler/out.sql | duckdb
2
```

```

3 Catalog Error: Scalar Function with name str_split_regexp does not exist!
4 Did you mean "str_split_regex"?
5 LINE 61:          SELECT CAST((str_split_regexp("%inputs%".current_phrase), ' ')) AS
          varchar[]) AS "elements",

```

Listing 4.12: Example SQL error output caused by passing incorrect expressions through **PL/Flummi** to **DuckDB**, providing information on error source

Once we have compilable and executable code, but do not yield the expected output, we have to start debugging the code. As we have encountered an algorithmic error we usually need to debug the code step by step. In another language having a debugger in their respective **IDE**, we would start to debug the code step by step in critical points, to observe variable values and program flow to see where the algorithm is faulty. But in **PL/Flummi** this functionality does not exist, due to the nature of **PL/Flummi** being compiled into **SQL** code and then executed. The result of all this is, that we have to debug the program code by hand, step by step. Which is obviously more time intense than using a built in debugger, but certainly not impossible. As **PL/Flummi** allows only one function per program and we were implementing manageable algorithms, the critical points in the code were not too numerous to be evaluated manually.

All in all the limited tools result in a less efficient development process. The time required is increased due to the need of having to do all tasks manually. But the development using **PL/Flummi** certainly is not impossible. Since the scope of its programs is limited by default keeping an overview is manageable. Additionally the most fundamental tools for compiling and executing provide us with the minimal required feedback on what is the cause of errors during development.

4.2.2 Ease of Development

PL/Flummi has allowed us to successfully implement all the algorithms into program code. While being Turing-complete, the **PL/Flummi** language provides us with a toolkit powerful enough to implement problems requiring single functions very effectively. The fundamental statements provided by **PL/Flummi** were able to allow us to write quite readable code to express the problem solutions. Our only tool in terms of programming paradigm at hand though is procedural programming. Trying to write code another way is not possible due to the restricted toolkit of **PL/Flummi**.

The most powerful tools to achieve expressive code were conditional and loop structures. They allowed us to implement the algorithm in a manner familiar to experienced programmers. Especially helpful is leveraging the very flexible construction of **LOOPS**. To be able to control the flow through the program by being able to place the **BREAK** wherever we want allows for powerful freedom in implementing algorithms. Though this may reduce readability when done in an unusual way, as most programmers will only be familiar with common loop variants like for, foreach, while and do-while loops. To maintain consistency and readability we chose to implement all loops as do-while loops. The do-while loop is applicable to all use cases in the **AoC** problem solutions, since they all require at least one iteration.

Since **PL/Flummi** is a procedural language, that only allows a single function, we are limited in different ways. For one we are limited in the vastness of the scope of the problem, that we are able to solve. Meaning we are not able to split off small functions, delegating tasks to them using expressive function names for the respective sub-task. This way we will end up more easily with deeply nested loops, high amounts of conditionals and bringing it all together in a single function. This highly reduces readability and maintainability the more extensive the implemented program gets. Though if we keep the scope small, the programs can be very readable and easily maintained and modified.

Given the fact, that we can comment the code, we can at least give extra context and information on critical parts of the code, that may not be as innately clear by themselves. A way to write documentation comments similarly to Javadoc comments does not exist, so it is needed to rely on basic line comments.

4.3 Time of Execution

Even though we had good experiences expressing our algorithms in **PL/Flummi**, problems emerged while executing some of the programs. The programs show different symptoms, from running fine over having high runtimes up to extremely high memory usage. We will take a look at what algorithms ran fine and which had problems. Also we will try and spot possible causes for the programs faults at execution time.

To get an impression on how performant the same algorithms are in other languages we will compare **PL/Flummi** program runtimes with their equivalent in other languages. To compare the **SQL** compiled from **PL/Flummi** to native **SQL** we searched for implementations of the problem solutions in **SQL**. Unfortunately those are rare and we could only find **SQL** code for two problems. Nonetheless we will compare these native **SQL** scripts with compiled ones.

4.3.1 High-Entropy Passphrases

To start of looking at our experiences while executing the programs we are not presented with any problems. As mentioned in Chapter [Implementation](#) this algorithm is very basic, not needing complex data structures or high cyclomatic complexity. Our runtimes are acceptable, the execution terminates and yields the correct results.

```
1  cat testdata.sql solver_1.sql | duckdb
2
3  | %result% |
4  | varchar  |
5  |          |
6  | 466      |
7  |          |
```

Listing 4.13: Execution statement and output of the compiled High-Entropy Passphrases solver

Though it should be mentioned, that the runtime, while being acceptable, is nowhere near a time normal for this algorithm. Looking at the **Python** implementation we used as reference and comparing their runtimes, we see significant differences.

	PL/Flummi	Python
Solver 1	1 593 ms	107 ms
Solver 2	19 574 ms	110 ms

Table 4.1: Average runtimes of five executions using **PL/Flummi** and **Python**

The **Python** equivalent achieves far better runtimes. The runtimes for the solver of the first part of the task being about 15 times longer in **PL/Flummi**. For the second part of the task **PL/Flummi** takes 1779 times longer than its equivalent. This of course is a huge difference in efficiency and leads us to get an impression on how high the runtimes will be for higher iterations and higher complexity.

Since **PL/Flummi** is compiled into **SQL** code, we also will take a look at a solution for this AoC problem, that is natively implemented in **SQL**. As the solutions for AoC problems in **SQL** are very scarce this solution [23] is written for the **PostgreSQL DBMS**. But still this can give us an insight on how efficient the solution can be implemented in **SQL**.

	PL/Flummi	PostgreSQL
Solver 1	1 593 ms	71 ms
Solver 2	19 574 ms	188 ms

Table 4.2: Average runtimes of five executions using **PL/Flummi** and **PostgreSQL**

Once again we spot a big difference in performance. Runtimes for the solver of the first part are 22 times faster in native **SQL** than in **PL/Flummi**, for the second part it is even 1042 times faster. This big of a difference in the solutions run in **SQL** are most likely caused through the nature of compiling the imperative code of **PL/Flummi** into declarative code in the form **SQL**. The sacrifices made during compilation lead to inefficient and bloated **SQL** code.

4.3.2 Infinite Elves and Infinite Houses

As we discussed in the Chapter [Implementation](#) already, this algorithm requires a lot more loop iterations. This impacts the runtime of the program significantly. It will not terminate in any realistic timeframe when given the original input provided by the AoC problem itself. Yet providing the compiled **SQL** script with a smaller input it outputs the correct result.

```

1  cat testdata-fast.sql solver_1.sql | duckdb
2
3  | %result% |
4  | int32   |

```

```

5  ┌───┐
6  │   │ 48 │
7  └───┘

```

Listing 4.14: Execution statement and output of the compiled Infinite Elves and Infinite Houses solver for input 1000

To get a feeling for how badly the number of loop iterations impacts the runtime of the program, we look at the development of the runtime with increasing inputs. The input for this algorithm directly correlates with the number of loop iterations executed inside the program.

	Input 100	Input 1 000	Input 10 000	Input 100 000
Solver 1	223 ms	658 ms	8 597 ms	266 196 ms
Solver 2	495 ms	846 ms	10 609 ms	300 132 ms

Table 4.3: Average runtimes of five executions using **PL/Flummi** with different inputs

Here we can quickly see, how fast the runtime grows when increasing the input. While reaching five minutes of runtime for an input of 100 000, we are still far off of the originally provided input of 36 000 000. This leaves us with the assumption, that **LOOP** iterations in **PL/Flummi** are highly inefficient and are a major contribution to high runtimes in **PL/Flummi** programs.

The reference code we used to implement our **PL/Flummi** program equivalent was able to achieve far better runtimes. It is written in **C** and implements the same algorithm as our **PL/Flummi** solution. While showing the runtimes for different inputs, because the **PL/Flummi** runtime is so high, we can get a strong impression in how fast this solution actually can be calculated.

	PL/Flummi	C
Solver 1	266 196 ms	2 430 ms
Solver 2	300 132 ms	2 920 ms

Table 4.4: Average runtimes of five executions using **PL/Flummi** with input 100 000 and **C** with input 36 000 000

As we have a runtime of around five minutes for the **PL/Flummi** program on an input of 100 000 versus the **C** program with the full input of 36 000 000, we can not fathom how long the runtime of the **PL/Flummi** version would be on the full input. Since the original input is 360 times the one given to the **PL/Flummi** program and our runtime increasing exponentially it is not realistically possible to terminate the execution in **PL/Flummi** for the full input.

4.3.3 Hill Climbing Algorithm

After being confronted with high runtimes, we now have to face completely new problems when executing the **PL/Flummi** program to solve the Hill Climbing Algorithm problem. This time around

the limiting factor is the memory required by the program. The memory used during execution is incredibly high and **DuckDB** and **PL/Flummi** for that matter don't provide us with any kind of protection or handling regarding excessive memory usage.

This is probably because the compiled **SQL** code make the **DBMS DuckDB** hold multiple copies of the same data in memory instead of efficiently managing it. Normally this should not result in the inability to finish executing. After the program exceeds the memory provided by the host machine, which in this case holds 16GB of RAM, **DuckDB** spills the excess data to the disk [11]. Unfortunately though this algorithm required complex data structures which are represented by lists and lists are not managed by **DuckDBs** buffer manager. Meaning lists will not be spilled to the disk when reaching memory limits. The moment the lists fill up the memory the host machine crashes. Even providing the program with a minimal input set does not circumvent this problem. The memory will always overflow and crash the host machine. The runtime is very long too, as the program is running multiple minutes before exceeding memory and crashing.

Despite not successfully executing the program in **PL/Flummi**, we still want to take a look at a solution for the problem written natively in **SQL** for **DuckDB**. Looking at natively written **SQL** code is interesting either way. It shows clearly, how efficient and concise **SQL** code can look like to solve these algorithmic problems. The compiled **SQL** code is unoptimized and unreadable. But the code implemented in **SQL** in comparison is readable and structured for the human eye. Also the lengths of the **SQL** scripts are very different. Our compiled script amounts to about 1300 lines of code, while the natively written script only requires 40 lines.

When executing the native **SQL** script, we can also make another observation. Its memory requirement is drastically lower than for the compiled **SQL** script. Only requiring 35-50 MB of memory throughout its 30s of execution time. This shows, the compiled code does not leverage tools and mechanisms to increase performance and optimize memory usage.

4.3.4 Pipe Maze

Once again we are presented with the same problems as before. Just like during the execution of the Hill Climbing Algorithm solver, we crash the host machine while executing the program solving the Pipe Maze problem. Facing this problem while working with the program solving Pipe Maze, we could not get any new insights over the ones gained working with the Hill Climbing Algorithm.

4.3.5 Firewall Rules

Executing the solver for the Firewall Rules problem, we are again confronted with a more basic algorithm not requiring complex data. The number of **LOOP** iterations, which has proven to impact the runtime significantly, is higher again though. During programming we were able to implement data preprocessing before the solving algorithm itself, this advances the high iteration count required to the beginning of the program. Only minimal logic is executed in each iteration. As a result of this effort the algorithm itself can skip over large amounts of the preprocessed input data.

```

1  cat testdata.sql solver_1.sql | duckdb
2
3  | %result% |
4  | varchar |
5
6  | 17348574 |
7

```

Listing 4.15: Execution statement and output of the compiled Firewall Rules solver

This basic optimization enabled us to achieve acceptable runtimes for the **PL/Flummi** program. Though the **Python** code equivalent is a lot more performant.

	PL/Flummi	Python
Solver 1	3 967 ms	101 ms
Solver 2	11 693 ms	101 ms

Table 4.5: Average runtimes of five executions using **PL/Flummi** and **Python**

Once again the **Python** equivalent achieves far better runtimes. For the solver of the first part of the task the runtimes are being about 40 times longer in **PL/Flummi** and about 117 times longer for the second part. Still this is a big discrepancy, but does not mean **PL/Flummi** is unusable for that matter.

4.4 General Thoughts

When looking at other use cases, that do not involve high cyclomatic complexity, high numbers of iteration or high amounts of complex data structures, other better potential use cases come to mind. As we discovered solving the AoC problems in **PL/Flummi**, the biggest hindrances are complex data structures followed by high number of **LOOP** iterations. If we find a problem domain, that does not require these constructs, but is able to leverage **PL/Flummi**'s properties and features, **PL/Flummi** can be of great value.

Considering **PL/Flummi** is compiled into **SQL** code, in our case for the **DuckDB DBMS**, we can leverage its OLAP [24] specialization. As **DuckDB** is designed to perform analytical query workloads, we can implement new queries as procedures in **PL/Flummi**. Possible applicable use cases could be analytical procedures on complex data to statistically evaluate business data. Another possible application could be to analyze time series data to spot trends in sales considering complex factors, for which a **PL/Flummi** program can merge and analyze relevant information. These programs created through **PL/Flummi** can then of course be executed in the **DuckDB DBMS** on business data.

PL/Flummi also provides us with good educational values. The **PL/Flummi** language is following an imperative approach, while the compiled code in the form of **SQL** is declarative. This opens up a good opportunity to look at how bridging the gap between the imperative and the declarative programming paradigms can look like.

Conclusion

5.1 Conclusion

The aim of this work lies upon determining the expressiveness and general usability of **PL/Flummi**. To achieve this we used the language to implement solutions for various **AoC** problems. We chose problems, that require very different kinds of approaches to be solved. From basic string processing, over large sets of data requiring many loop iterations, up to more advanced algorithms requiring complex data structures. This gave us good insights in how to represent these algorithms in code using **PL/Flummi**, which was very possible for all problems addressed. Though the lack of any tools aiding the development using **PL/Flummi** made the testing and debugging process quite tedious. When running the programs though, the weaknesses of **PL/Flummi** became apparent. The biggest hindrance being the excessive memory usage as discussed for the [Hill Climbing Algorithm](#), which for our hardware configuration crashes the host machine. Another problem are the high runtimes, mostly in the case of the [Infinite Elves and Infinite Houses](#) Problem, where it was not possible to run the program on the full original Input provided by **AoC**.

These discoveries lead us to believe **PL/Flummi** in its current state without major optimizations is not suited for handling complex data structures and for high amounts of loop iterations containing complex calculations. Yet there can be use cases suited for **PL/Flummi** involving developing procedures representing analytical queries leveraging the strength of the underlying **DuckDB DBMS**.

5.2 Future Work

We focused on putting **PL/Flummi** to practical use and evaluating its expressiveness and general usability. Therefore we gathered good insights where adjustments to improve the overall experience can be made.

Starting with the development process adding tooling to aid with compilation and execution processes would ease the development process. Another feature to improve the experience of writing **PL/Flummi** code would be the implementation of structures like **IF** without the **ELSE** clause only containing **NOOP** and different preconfigured **LOOP** configurations like for-loops or while-loops.

Looking at the difficulties we faced while executing our **PL/Flummi** programs, potential optimizations become apparent. Firstly analyzing the problems regarding the high runtimes when executing many **LOOP** iterations and the problem of excessive memory usage crashing the host ma-

chine. Given there are implementable solutions to these problems, these optimizations would greatly increase possible use cases of **PL/Flummi**.

Bibliography

- [1] DuckDB Foundation. *DuckDB - An in-process SQL OLAP database management system*. 2024. URL: <https://duckdb.org/> (visited on 07/23/2024).
- [2] DuckDB Foundation. *Data Types - General-Purpose Data Types*. 2024. URL: https://duckdb.org/docs/sql/data_types/overview#general-purpose-data-types (visited on 07/23/2024).
- [3] DuckDB Foundation. *Data Types - Nested / Composite Types*. 2024. URL: https://duckdb.org/docs/sql/data_types/overview#nested--composite-types (visited on 07/23/2024).
- [4] DuckDB Foundation. *Functions*. 2024. URL: <https://duckdb.org/docs/sql/functions/overview> (visited on 07/23/2024).
- [5] Eric Wastl. *Advent of Code*. 2015. URL: <https://adventofcode.com/> (visited on 07/23/2024).
- [6] Eric Wastl. *Day 4: High-Entropy Passphrases*. 2017. URL: <https://adventofcode.com/2017/day/4> (visited on 07/23/2024).
- [7] Eric Wastl. *Day 20: Infinite Elves and Infinite Houses*. 2015. URL: <https://adventofcode.com/2015/day/20> (visited on 07/23/2024).
- [8] Eric Wastl. *Day 12: Hill Climbing Algorithm*. 2022. URL: <https://adventofcode.com/2022/day/12> (visited on 07/23/2024).
- [9] Eric Wastl. *2023 Day 10: Pipe Maze*. 2023. URL: <https://adventofcode.com/2023/day/10> (visited on 07/23/2024).
- [10] Eric Wastl. *Day 20: Firewall Rules*. 2016. URL: <https://adventofcode.com/2016/day/20> (visited on 07/23/2024).
- [11] DuckDB Foundation. *Pragmas - Resource Management - Memory Limit*. 2024. URL: <https://duckdb.org/docs/configuration/pragmas#resource-management> (visited on 07/23/2024).
- [12] Oracle. *PL/SQL*. 2024. URL: <https://www.oracle.com/database/technologies/appdev/plsql.html> (visited on 07/23/2024).
- [13] Microsoft. *Transact-SQL reference*. 2024. URL: <https://learn.microsoft.com/en-us/sql/t-sql/language-reference?view=sql-server-ver16> (visited on 07/23/2024).
- [14] Postgres. *PL/pgSQL — SQL Procedural Language*. 2024. URL: <https://www.postgresql.org/docs/current/plpgsql-overview.html> (visited on 07/23/2024).
- [15] DuckDB Foundation. *Functions - Numeric Functions*. 2024. URL: <https://duckdb.org/docs/sql/functions/numeric> (visited on 07/23/2024).
- [16] DuckDB Foundation. *Comparisons - Comparison Operators*. 2024. URL: https://duckdb.org/docs/sql/expressions/comparison_operators (visited on 07/23/2024).
- [17] MultiMedia LLC. *Functions - Text Functions*. 2024. URL: <https://duckdb.org/docs/sql/functions/char> (visited on 07/23/2024).

- [18] <https://www.reddit.com/user/miran1/>. *2017 Day 4 Solutions - Python 3 solution*. 2017. URL: https://www.reddit.com/r/adventofcode/comments/7hf5xb/comment/dqqs04/?utm_source=share&utm_medium=web3x&utm_name=web3xcss&utm_term=1 (visited on 07/23/2024).
- [19] Deleted Reddit User. *2015 Day 20 Solutions - C solution*. 2015. URL: <https://www.reddit.com/r/adventofcode/comments/3xjpp2/comment/cy5buyt/> (visited on 07/23/2024).
- [20] <https://github.com/sudo-grue>. *2022 Day 22 Solutions - Python 3 solution*. 2022. URL: <https://github.com/sudo-grue/aoc22/blob/master/day12/combo.py> (visited on 07/23/2024).
- [21] <https://github.com/tmo1>. *2023 Day 22 Solutions - Python 3 solution*. 2023. URL: <https://github.com/tmo1/adventofcode/blob/main/2023/10.py> (visited on 07/23/2024).
- [22] <https://www.reddit.com/user/nullmove/>. *2016 Day 20 Solutions - Python 3 solution*. 2016. URL: https://www.reddit.com/r/adventofcode/comments/5jbeqo/comment/dbevcua/?utm_source=share&utm_medium=web3x&utm_name=web3xcss&utm_term=1 (visited on 07/23/2024).
- [23] <https://github.com/seltiko>. *2017 Day 4 Solutions - PostgreSQL solution*. 2017. URL: https://github.com/seltiko/sql_advent_2017/blob/master/Day%204.sql (visited on 07/23/2024).
- [24] Wikipedia. *Online analytical processing*. 2024. URL: https://en.wikipedia.org/wiki/Online_analytical_processing (visited on 07/23/2024).