Implementation of a High-level Set Theoretic Programming Language

Joel Luckett

September 8, 2025

Abstract

Set theory defines much of today's mathematics and should therefore define computer science in a similar way. Set-theoretic programming is one such way that fits computer science into a mathematical framework. This article introduces an implementation of a set-theoretical language (JMPL) as an efficient way to translate mathematical ideals to programming, while exploring relevant structure and design details.

Contents

1 Introduction

Set theory is the study of collections of objects known as sets. It is a fundamental concept that shaped the language of mathematics and forms the basis of many core topics in computer science. Yet, despite its importance, very few programming languages use a set-theoretic paradigm. Despite its age, the concept of 'set-theoretic programming' is still a fledgling field without a concrete definition. For this article, I define 'set-theoretic programming' as a style of programming based on set theory where first-order sets are the primary data structure. Set-theoretic programming is one of many paradigms — a style of programming that dictates the structure of code. The advantages of a set-theoretical language stem from its abstractions in computer science to fit a mathematical framework. Programmers do not need to choose and define data structures; rather, they write the abstraction of an algorithm, and the rest is handled for them. This style of programming lends itself well to rapid development of algorithms and high productivity. We call this amount of abstraction 'high-level' due to the disconnect between the programmer and the inner workings of their program. The problem is that no modern language encompasses the benefits of a truly set-theoretic programming language. This article provides

an introduction to JMPL, my own implementation of a high-level set-theoretic language, created to address these issues.

2 Set-theoretic Programming

The concept of sets is prevalent throughout computing. The set data structure is one of the most common and is widely used in many applications. A common example of this, which is becoming increasingly prevalent, is the dataset. Out of the ten most popular programming languages given by the TIOBE Index (TIOBE 2025), half have a built-in set data structure. This prompts the question, what research has gone into creating a set-based language? Research started in the late 1960s with the introduction of SETL, although other 'set-theoretically oriented languages' are known to have been in development at a similar time (Kennedy & Schwartz 1975, p.98). Ever since, SETL is the only notable set-theoretic language, dominating the sparse Wikipedia page on the topic. SETL was designed with set theory in mind to be an efficient way to write abstractions of algorithms without being highly technical. This made it effective for 'rapid prototyping of large software systems' and for the refinement of 'sophisticated algorithms' (Cantone et al. 2001, p.4). It was successful in this sense as researchers used it to create the first implementation of the popular Ada programming language (Dewar et al. 1980). However, this style of programming quickly fell out of fashion as the popular paradigms widely used today, such as functional and object-oriented, were being developed around the same time. Since then, no other mainstream programming languages have branded themselves as 'set-theoretic' despite the advancements in mathematical computing and programming language design. As a result, knowledge of set-theoretic programming is largely composed of the research done around SETL by its creator, Jacob T. Schwartz. Despite this, some modern programming languages are structured around set-theoretic ideas. An example would be SQL, a popular language designed to query databases. SQL uses queries in a way reminiscent of set notation to produce sets of database values.

In the summary of a paper on SETL, its authors (including Schwartz) predict that set-theoretic languages will be 'prime vehicles for algorithm specification, large system development, and one-shot programming projects' (Kennedy & Schwartz 1975). So why hasn't this come to fruition? Many reasons may be at fault, but the prospects of set-theoretic remain clear.

3 JMPL

Allow me to introduce JMPL (pronounced jump-OOL) (Luckett 2025), my implementation of a high-level, set-theoretic programming language. The primary goals when designing JMPL were to maintain a close resemblance to mathematical syntax and to ensure it is straightforward to use for both mathematicians and computer scientists. Aside from SETL, JMPL has multiple inspirations from modern languages such as Python and Haskell. JMPL has many applications as a programming language and a learning tool. JMPL code can be written in a style close to mathematical syntax, making it a great device for mathematicians to learn programming and programmers to learn mathematics. It offers an efficient way to quickly write solutions to problems and algorithms, especially those of a mathematical nature.

3.1 Syntax

To help understand JMPL's syntax, it is best to use an example. For this, we will use the recursive definition of a power set. In set theory, this is the collection of all subsets, given a set, denoted by $\mathcal{P}(S)$. A definition of $\mathcal{P}(S)$ for finite sets is as follows:

$$\text{If } S = \{\}\text{, then } \mathcal{P}(S) = \{\{\}\}$$
$$\text{else, let } e \in S \text{ and } T = S \setminus \{e\}\text{, then } \mathcal{P}(S) = \mathcal{P}(T) \cup \{t \cup \{e\} \mid t \in \mathcal{P}(T)\}$$

Converting this to a function is trivial, as JMPL has been designed for set-related problems. A standard definition for a function that calculates the power set of a given set is as follows:

```
// JMPL Code

func power_set(S) =
    if S == {} then
        {{}}
    else
        let e = arb S                // Get an arbitrary element from S
        let T = S \ {e}              // Let T be S without e
        let P_T = power_set(T)

        P_T ∪ {t ∪ {e} | t ∈ P_T} // Return the union of power_set(T)
                                  // and the set of all elements of
                                     power_set(T) expanded with e
```

As you can see, JMPL's syntax closely aligns with the mathematical syntax, allowing programmers to transfer ideas between mathematical writing and code easily. Now, take a popular programming language such as Java. If a beginner programmer were to try to write this code in Java, it would be verbose and almost unrecognisable from the original problem.

It should be known that Java and other languages often have built-in methods of creating a power set in an optimised way. This is just an example of a potential problem a beginner programmer might encounter.

```
// Java Code

public class PowerSet<T> {
    public Set<Set<T>> getPowerSet(Set<T> setS) {
        if (setS.isEmpty()) {
            Set<Set<T>> result = new HashSet<>();
            result.add(new HashSet<>());
            return result;
        } else {
            T e = setS.iterator().next();

            Set<T> setT = new HashSet<>(setS);
            setT.remove(e);

            Set<Set<T>> pSetT = getPowerSet(setT);

            Set<Set<T>> subsets = new HashSet<>();
            for (Set<T> t : pSetT) {
                Set<T> subset = new HashSet<>(t);
                subset.add(e);
                subsets.add(subset);
            }

            Set<Set<T>> result = new HashSet<>(pSetT);
            result.addAll(subsets);

            return result;
        }
    }
}
```

3.2 Features

Due to its more imperative structure rather than the more declarative nature of set theory (ideas we will discuss in Section 4), JMPL inherits some common features found in most imperative languages. This includes variables, functions, if and while statements, and operators. In this section, we will introduce several of JMPL's key features that distinguish it from contemporary languages and construct its set-theoretic paradigm.

3.2.1 Unicode

One way JMPL follows mathematical syntax is by allowing programmers to express operations with Unicode symbols. Unicode is an international standardised set of character encodings that computers use to recognise letters and symbols. Unicode characters are each assigned a unique code, such as U+2260 for the symbol '≠'. Symbols enable a program to be almost backwards compatible with the equivalent idea in mathematics, reducing the learning curve for mathematicians learning programming and programmers learning mathematics. Taking the phrase

```
t ∪ {e}
```

from the power set example, the symbol '∪' appears. In mathematics, this means the union (combination) of two sets, which is what the code here is doing, combining t with $\{e\}$. JMPL recognises certain words and Unicode symbols as the same idea, even if the symbol is not from standard keyboard input. When the usage of Unicode symbols is not possible or practical, we can freely substitute them with keywords, a special word reserved by the language that we cannot use to name a variable. In this case, '∪' would be substituted with 'union'. For the rest of this article, keywords will be used in favour of symbols for clarity. All symbol and keyword substitutions can be found in Appendix A.

3.2.2 Sets and Objects

In a program, JMPL passes data around as values. We can split values into two groups: atoms and objects. Atoms are values that we cannot subdivide, such as numbers and characters. Objects are the opposite — values composed of values. As with set theory, every JMPL object is immutable and first-class. An object is immutable if we cannot change it after its creation, and it is first-class when we treat it like any other value. The most important object is, of course, the set. Sets follow the mathematical definition of unordered collections of distinct entities and can be created in several ways: as literals, ranges, or set-builders. The first two are straightforward. A set literal creates a set by specifying its values, while ranges specify a start, step, and end value.

```
// JMPL Code

let S1 = {1, 2, 3, 4, 5} // Literal
let S2 = {1, 3 ... 9}    // Range for {1, 3, 5, 7, 9}
let S3 = {'a' ... 'e'}   // Works for characters too
```

The third way, set-builder notation, is an idea from set theory that builds a set by specifying the members' properties. This is one of JMPL's defining features, as it allows for the easy creation and manipulation of sets. As seen with the power set example, it is very powerful and substantially minimises the amount of code needed. Its syntax and function are borrowed directly from set theory, making it a practical, transferable knowledge. To create a set using set-builder notation, you describe its members using a predicate (a true or false function). The syntax for this in both set-theory and JMPL is

$$\{x \mid P(x)\}$$

where $P(x)$ is a predicate or rule for x. To define where we obtain x, you specify a domain using $x \in S$ where S is a set. The keyword 'in' can be used to replace '∈' if necessary. In JMPL, this is called a 'generator' - an idea used throughout the language. Multiple generators and predicates can be used in one set-builder, given they are on the right-hand side of the pipe and are delimited by commas. Some examples of this are:

```
// JMPL Code

let S = {1, 2, 3, 4, 5}

let S1 = {x in S | x mod 2 > 0}     // The odd members of S: {1, 3, 5}
let S2 = {x^2 | x in S}             // The members of S squared:
                                    // {1, 4, 9, 16, 25, 36}
let S3 = {x + y | x in S, y in S }  // Multiple generators:
                                    // {2, 3, 4, 5, 6, 7, 8, 9, 10}
let S4 = {x in S | x > 2, x < 5 }   // Multiple predicates: {3, 4}
```

A tuple object is secondary in importance but antithetical to sets. Where sets are unordered collections of unique elements, tuples are ordered and can store duplicate values. Tuples can be defined similarly to sets with literals and range notation, just with parentheses rather than braces.

```
// JMPL Code

let t1 = ('h', 'e', 'l', 'l', 'o') // Literal
let t2 = (9, 7 ... 1)              // Range for (9, 7, 5, 3, 1)
```

Both objects can be nested and combined in any way to form new objects. One beneficial combination is the idea of a map. In both mathematics and computer science, this is an object that 'maps' one value to another. In set theory, we can construct a map as a set of pairs (tuples of length 2) or as a kind of function. JMPL mirrors this:

```
// JMPL Code

// Map made from literals
let M1 = {(1, 1), (2, 4), (3, 9), (4, 16)}

// Function-style mapping
let M2 = func (x) -> x^2
```

In formal mathematics, a function or mapping is typically accompanied by an input set and an output set, known as the domain and codomain, respectively. As of the time of writing, this feature is not currently present in JMPL but is planned for inclusion in a future extension.

3.2.3 Quantification

As set theory is formulated within logic, it uses the idea of a quantifier. Given a formula, a quantifier tells us how many members in a set satisfy a predicate. JMPL offers three quantifiers that allow us to query sets in a declarative manner easily. The first two are the standard mathematical quantifiers: 'forall ($\forall$)' and 'exists ($\exists$)'. They use the syntax

$$\text{quantifier } x \ \in \ S \mid P(x)$$

where S is a set, x is a variable, and $P(x)$ is a predicate of x. If the quantifier is 'forall', then the statement is true if, for all x, $P(x)$ is true. Conversely, if the quantifier is 'exists', then the statement is true if there exists at least one x that makes $P(x)$ true.

```
// JMPL Code

let S = {1, 2, 3, 4, 5}

println(forall x in S | x < 6) // This will print true
println(forall x in S | x > 1) // This will print false

println(exists x in S | x < 3) // This will print true
```

```
println(exists x in S | x > 6) // This will print false
```

The third quantifier is called 'some' and returns a value from a set rather than a truth value. It works like 'exists', where it returns the first value in a set that satisfies the predicate. By extension, using a set-builder to get all the values that satisfy the predicate could work as a fourth quantifier.

```
// JMPL Code

let S = {1, 2, 3, 4, 5}

println(some x in S | x < 6) // This will print 1
println(some x in S | x < 0) // This will print 'null' as it doesn't
                             // exist
```

3.3 Implementation and Performance

After designing a programming language, we must construct a program that can execute code as a series of instructions on a computer's CPU. This is known as an implementation. The implementation of JMPL was created in the C programming language and follows a bytecode interpreter design. An interpreter is a program that, in this case, translates code into a more computable format (bytecode), which then executes directly.

In Section 3.1, we compared JMPL to Java. This is unfair as Java was created for completely different purposes than JMPL. A much fairer comparison would be to Python, one of JMPL's inspirations. Python's main implementation is also a bytecode interpreter, and its syntax is much more comparable to JMPL. Python even includes a set literal syntax. To measure JMPL's performance, I conducted a small benchmark comparison between JMPL and Python using the power set example. Like Java, Python has built-in functions that tackle problems like power sets much more efficiently than the JMPL example does. Instead, this benchmark was to test how effective JMPL's set operations are in contrast to a similar language.

For this benchmark, I translated the power set example into Python. The input to both languages' functions was a set containing the integers 0 to N, where N ranged from 1 to 22. Each program was run five times, and its average execution time in seconds was recorded. Results for $N = 1$ to $N = 17$ were very similar, with execution time below half a second. On the other hand, results for $N = 18$ to $N = 22$ diverged, showing Python to be considerably slower as the input set grew.

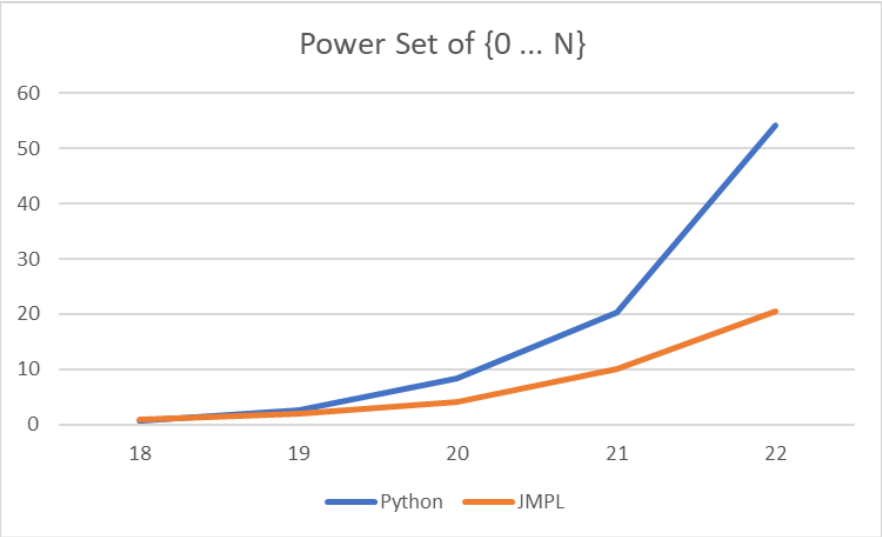

Figure 1: Benchmark results from $N = 18$ to $N = 22$

The results of this benchmark show that JMPL, even in its infancy, is a viable alternative to modern programming languages as a tool for set-based problem solving. The code, machine specifications, and full results can be found in Appendix B.

4 Design

When designing a set-theoretic language, a few problems quickly arise: (1) What set-theoretic framework should be used? (2) How purely set-theoretic should it be? and (3) Where should accuracy be sacrificed for computation?

To address (1), it was easy to pick a framework. The system known as Zermelo-Fraenkel set theory is the most well-known and studied, so it was a safe starting point. Zermelo-Fraenkel (ZF) is a collection of axioms where, in a highly simplistic manner, we construct everything out of sets. This quickly raises questions (2) and (3). In Section 3, we discussed the idea of the atom value, which in set theory is known as a urelement. ZF does not contain urelements, whereas JMPL does. It made sense to include atoms, as computers are designed for the computation of atomic values. This is just one necessary abstraction JMPL uses that causes it not to purely align with ZF. Another design problem was with tuples. Tuples do not have a standard definition in set theory, causing them to be difficult to encode as a data type within a set-theoretic language, as demonstrated in the article 'Definition of the concept "vector" in set theoretic programming languages' by H. S. Warren Jr. (Warren 1976). Instead of settling on a rigid set-based encoding, JMPL defines tuples as a distinct object. While defining everything as a set would be 'pure', the constant encoding and decoding would be a waste of computation. This is why JMPL uses an optimal data structure for each concept, a core philosophy shared by SETL (Kennedy & Schwartz 1975, p.97). However, it does not limit those who want to program in a purely set-theoretical style from doing so.

Another point to consider when designing JMPL was whether it should be imperative or declarative. In essence, imperative programs describe *how* a task should be performed while declarative programs specify *what* the result should look like (IONOS 2021). Set theory and mathematics are often written in a declarative style, while most popular programming languages are written in an imperative style. JMPL uses a combination of these paradigms to retain set-theoretical ideas while remaining accessible. Programs are written line-by-line in an imperative manner, while set-related ideas are often written declaratively. These design choices allow JMPL to be a convenient and flexible tool perfectly suited for a variety of applications.

5 Conclusion

In this article, we discussed the unutilised advantages of the set-theoretic paradigm and its potential as a vehicle for strong program design. As a solution to this, we explored JMPL and its design that makes it not only a proponent for set-theoretic programming, but a tool for learning. However, JMPL still has room to grow. Many potential features can make JMPL more set-theoretic and more useful. In the future, I hope to explore adding features such as the ability to handle infinite sets or allowing user-defined operations. Additionally, a more mature implementation would employ advanced compiler techniques. Such ideas would increase the efficiency of the language, giving a boost to its practicality. In summary, set-theoretic programming is a powerful paradigm left neglected by modern computing languages. However, with languages such as JMPL, there is hope that its ideas will be explored further to create sophisticated algorithms and rival the dominant paradigms.

References

Cantone, D., Omodeo, E. & Policriti, A. (2001), 'Set Theory for Computing, From Decision Procedures to Declarative Programming with Sets'.
URL: *https://doi.org/10.1007/978-1-4757-3452-2*

Dewar, R. B. K., Fisher, G. A., Schonberg, E., Froehlich, R., Bryant, S., Goss, C. F. & Burke, M. (1980), 'The NYU Ada translator and interpreter', *SIGPLAN Not.* **15**(11), 194–201.
URL: *https://doi.org/10.1145/947783.948659*

IONOS (2021), 'Imperative programming: Overview of the oldest programming paradigm', *IONOS Digitalguide* . Accessed on 06/09/2025.
URL: *https://www.ionos.com/digitalguide/websites/web-development/imperative-programming/*

Kennedy, K. & Schwartz, J. (1975), 'An introduction to the set theoretical language SETL', *Computers & Mathematics with Applications* **1**(1), 97–119.
URL: *https://www.sciencedirect.com/science/article/pii/0898122175900115*

Luckett, J. (2025), 'JMPL (version 0.2.2) [Computer software]'.
URL: *https://github.com/Jogll1/JMPL*

TIOBE (2025), 'TIOBE Programming Community Index'. Accessed on 03/09/2025.
URL: *https://www.tiobe.com/tiobe-index/*

Warren, H. S. (1976), 'Definition of the concept "vector" in set theoretic programming languages', *Computers & Mathematics with Applications* **2**, 73–83.
URL: *https://www.sciencedirect.com/science/article/pii/0898122175900115*

A Symbols

Symbol	Unicode	ASCII Substitution
¬	U+00AC	not
∀	U+2200	forall
∃	U+2203	exists
∈	U+2208	in
∧	U+2227	and
∨	U+2228	or
∩	U+2229	intersect
∪	U+222A	union
≠	U+2260	/=
≤	U+2264	<=
≥	U+2265	>=
⊂	U+2282	subset
⊆	U+2286	subseteq

Table 1: Symbol Substitutions

B Power Set Benchmark

Python (version 3.13.1) function used for the benchmark:

```python
import random

def power_set(S):
    if len(S) == 0:
        return {frozenset()}
    else:
        e = random.choice(list(S))
        T = S - {e}
        P_T = power_set(T)

        return P_T | {t | {e} for t in P_T}
```

JMPL function used for the benchmark:

```
func power_set(S) =
```

```
if S == {} then
    {{}}
else
    let e = arb S
    let T = S \ {e}
    let P_T = power_set(T)

    P_T ∪ {t ∪ {e} | t ∈ P_T}
```

Specifications of the device used for the benchmark:

System Model: HP Laptop 15-fc0xxx
System Type: x64-based PC
Processor: AMD Ryzen 7 7730U with Radeon Graphics, 2000 MHz, 8 Cores
Installed RAM: 16 GB
Operating System: Windows 11 (10.0.26100 Build 26100)

Power was constantly supplied via a 45W charging cable during the benchmark.

For the benchmark, a set containing N integers was used as input to the function. Each function was tested five , and an average was found. Data for $N = 1$ to $N = 9$ is omitted as it was too small to measure. Data is rounded to three significant figures. Full benchmark data:

	Execution Time (s)											
	Python						JMPL					
N	1	2	3	4	5	Avg.	1	2	3	4	5	Avg.
9	0.0005	0.0005	0.0005	0.0005	0.0005	0.001	0.001	0.001	0.001	0.001	0.001	0.001
10	0.00115	0.00109	0.00109	0.00107	0.00119	0.00112	0.00200	0.00200	0.00100	0.00200	0.00200	0.00180
11	0.00246	0.00228	0.00239	0.00242	0.00228	0.00237	0.00300	0.00400	0.00400	0.00400	0.00400	0.00380
12	0.00468	0.00489	0.00489	0.00474	0.00487	0.00481	0.00700	0.00800	0.00700	0.00700	0.00800	0.00740
13	0.0101	0.0104	0.00992	0.0116	0.0101	0.0104	0.0120	0.0140	0.0130	0.0130	0.0130	0.0130
14	0.0256	0.0262	0.0267	0.0263	0.0262	0.0240	0.0270	0.0260	0.0260	0.0260	0.0260	0.0258
15	0.0663	0.0676	0.0664	0.0662	0.0662	0.0665	0.0640	0.0650	0.0630	0.0620	0.0630	0.0634
16	0.140	0.141	0.141	0.140	0.141	0.141	0.152	0.152	0.155	0.154	0.151	0.153
17	0.471	0.455	0.456	0.452	0.454	0.458	0.392	0.391	0.391	0.391	0.394	0.392
18	0.793	0.787	0.787	0.792	0.787	0.789	0.833	0.845	0.834	0.841	0.838	0.838
19	2.67	2.74	2.64	2.63	2.67	2.67	2.03	2.01	2.02	2.00	2.04	2.02
20	8.39	8.41	8.21	8.27	8.19	8.29	4.09	4.12	4.32	4.32	4.14	4.20
21	20.2	19.8	21.1	20.0	20.0	20.2	10.1	9.92	9.99	10.3	10.1	10.1
22	55.0	54.4	53.8	54.5	53.2	54.2	21.0	21.3	20.1	20.1	20.1	20.52

Table 2: Benchmark data