The Lisp Programming Language and its Influence

ABSTRACT

The importance for a programmer to broaden their horizons when it comes to their most important tool – the language – can not be overstated. Without innovative minds that explore alternatives, we would perhaps still write all our software in an assembly language. One language that possesses the power to shift ones view is Lisp. In this paper we will show the influence of the programming language Lisp on the basis of its core features. We will gather how mainstream languages of today exhibit characteristics that Lisp introduced or popularized. By giving an insight to its early history and its current state, we attempt to answer the question of how Lisp itself eventually declined in usage. To achieve all this, we will give a brief introduction to its syntax and characteristics, from which we will be able to extrapolate to its deeper paradigms and concepts. We will demonstrate how these ideas work not in spite of the compactness of the core language but precisely *because of it.*

KEYWORDS

lisp, functional programming, metaprogramming, macro

1 INTRODUCTION

The topic of programming languages is a large and confusing one and learning all of them poses a near impossible task. Instead it is important for a programmer to look at languages that differ in their approaches to learn about new ways to express the solution to a problem. A common saying among programmers is that learning a functional programming language shows its advantages even when not directly writing in one. As the very first functional programming language [24], Lisp not only suits this purpose, but can also provide an insight to programming history that modern languages can not.

Lisp is a language that was conceived in the 1960s by John McCarthy as a purely theoretical foundation for solving algorithms. It gave birth to a family of programming languages —so called *Lisp dialects* or just *Lisps*— that try to stay true to its original ideas while extending it with new useful features [21]. As a result, the term Lisp by itself now implicates the original language as well as its successors that share fundamental ideas with the early design of McCarthy. Through its many dialects, Lisp now lives on as the second oldest programming language still in use (Fortran being the oldest) [4, 9, 14].

Considering this achievement, we will uncover how a now 60 year old language still finds its place in our modern landscape of programming languages. We identify the areas in which we can still see Lisp today, not just in the form of tangible dialects, but also in the appearance of Lisp's innovations in other (seemingly unrelated) programming languages.

We will do so by briefly talking about its history and giving an introduction to its syntax and language constructs. Using this knowledge, we will be able to understand what paradigms it follows and which unique concepts it offers. We will show how these possibilities are used in real-world programs and areas of research.

2 PROBLEM STATEMENT

In its 2019 statistics, *GitHub* published their "Top 10 primary languages over time, ranked by number" [1], giving a rough estimation as to which programming languages are the most widespread of our time.

Table 1: GitHub's 2019 top ten languages, ranked by number

#	Language	Interp.	G.C.	O.O.	Dyn. typ.	C Syntax
1	JavaScript	Yes	Yes	Yes	Yes	Yes
2	Python	Yes	Yes	Yes	Yes	No
3	Java	J.I.T.	Yes	Yes	No	Yes
4	PHP	Yes	Yes	Yes	Yes	Yes
5	C#	J.I.T.	Yes	Yes	Optional	Yes
6	C++	No	No	Yes	No	Yes
7	TypeScript	Yes	Yes	Yes	Yes	Yes
8	Shell	Yes	No	No	Yes	No
9	C	No	No	No	No	Yes
10	Ruby	Yes	Yes	Yes	Yes	No

In Table 1, we present these languages, showing (from left to right) which ones are interpreted languages, use garbage collections, have object-oriented features, are dynamically typed or use C-like syntax. We can see that six out of ten are interpreted. Seven out of ten make use of garbage collection. Eight support object-oriented programming in some form. Six are dynamically typed and seven of them employ C-like syntax.

Lisp is a language that spawned over 30 dialects[2], none of which are present on the figure above. Considering this, one could interpret the table as evidence that Lisp fails to deliver any advantages as a programming language. This begs the question of why languages took inspiration from Lisp in the first place and why we refer to these languages as mere *dialects* of Lisp. Furthermore, why the term "C dialect" is not used, when its influence is immediately apparent from the example. More broadly, we would like to know if Lisp in itself has any significance in the present time.

By exploring the answers to these questions, we will be able to illustrate Lisp's influence on modern programming languages by this very table.

3 HISTORY

Unlike conventional languages that are still in use today, Lisp has no clear predecessors and its roots instead lie primarily in theoretical

[1] https://octoverse.github.com/#top-languages accessed 20-05-20
[2] https://en.wikipedia.org/wiki/List_of_Lisp-family_programming_languages

mathematics. In this section, we will briefly cover the history of Lisp from its origin as a research paper to the current state of existing Lisp dialects.

3.1 Concept for Lisp

Lisp as a programming language was developed by John McCarthy in 1958 at the Massachusetts Institute of Technology (MIT) as part of the Artificial Intelligence Project. McCarthy's design was partly influenced by the *Information Processing Language* (*IPL*) [14].
He published his findings in 1960 in the paper *Recursive functions of symbolic expressions and their computation by machine, part I*[3] [13]. It is considered to be the original paper on Lisp. His motivation was to describe a language both as programming language and as a simple formalism to build a Turing-complete language for algorithms. He named this language LISP for "LISt Processor". His intention was to build a theoretical foundation for Lisp, thus he did not provide any implementation.

3.2 First Implementation

The original paper on Lisp included a definition of a Lisp function called `eval[e, a]`— the universal Lisp function. A function intended to compute the value of any valid Lisp expression `e` under variable assignments `a` [13].

Upon reading the paper, Steve Russell — one of McCarthy's colleagues at MIT — understood that this `eval` function could be the base of a working Lisp interpreter. He programmed it in machine code using punched cards, resulting in the foundation for running Lisp programs [14].

This first version of Lisp served as a proof of concept and motivated the development of an improved Lisp interpreter that was labeled *Lisp 1.5*. Following the publication of the *LISP 1.5 Programmer's Manual* in 1962, many implementations of Lisp were written for a variety of computers with different architectures [14].

3.3 Evolution of Lisp

Since there was no organization for standardizing the language and little to no communication between the programmers, versions of Lisp diverged more and more from each other. During this time —in an effort to further improve the original language— works on *LISP 2* began.

By 1967 however, the difficulties of replacing Lisp 1.5 were too great and problems in the development of Lisp 2 lead to the project eventually getting abandoned [21].

Consequently, the branching of Lisp increased in the 1960s and 70s as illustrated in Figure 1. Notable dialects of that time were *Maclisp*, *ZetaLisp*, *InterLisp* and *Scheme*. They eventually stopped seeing usage with the exception of Scheme – a Lisp dialect that was created by Guy L. Steele and Gerald Jay Sussman at MIT [21].

Attempts to unify the Lisp dialects into one language began in the 1980s and 1990s. The new language was to be named Common Lisp and aimed for rough compatibility with preceding dialects while focusing on portability and consistency [20]. As a result of this work the ANSI Common Lisp standard was published in 1994.

[3]Part II was never written, but "was intended to contain applications to computing with algebraic expressions"

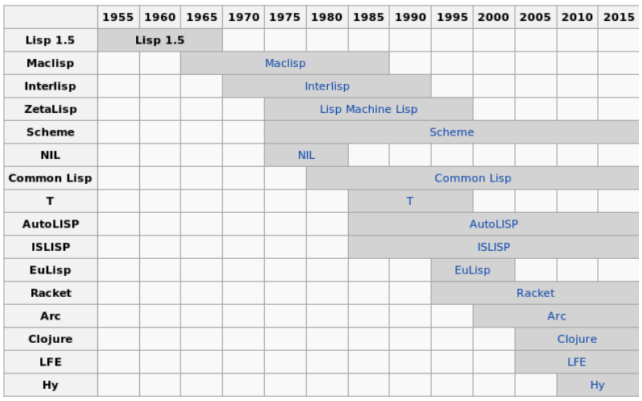

Figure 1: Timeline of Lisp Dialects [12]

4 SYNTAX AND CHARACTERISTICS

Naturally, dialects of the original Lisp took different approaches to the specificities. The various examples in this paper will be written in standardized *ANSI Common Lisp*. The goal of Common Lisp is to extend Lisp with everyday conveniences that are aimed to make the language more practical overall, while staying faithful to the original vision and core syntax. As such, the simple code examples in this section are valid code in all major Lisps.

An underlying characteristic of Common Lisp and other Lisps that deserves mention is the use of garbage collectors, eliminating the need for memory management. Furthermore, they are interpreted languages that commonly offer a *Read-Eval-Print-Loop* or *REPL* as a terminal application that evaluates any expression given to it.

4.1 Prefix-Notation and Expressions

There are two traits of the Lisp syntax that make any of its dialects immediately stand out: The use of a *prefix notation*[4] and a heavy reliance on pairs of parentheses that can signify the start and end of an expression.

An expression is either an atom, which is a word not surrounded by parentheses (e.g. `foo`), or a list of expressions surrounded with parentheses, separated by whitespace (e.g. `(foo bar)`) [10]. Expressions in Lisp are more precisely called *symbolic expressions*, or *s-expressions*. We will give a more in-depth insight to them in 5.1. To compute `3 + 2` in Lisp, one would write the following expression:

```
(+ 3 2)
```

In these expressions, Lisp expects the name of a function at the start (`+` in this case), followed by its arguments (`3` and `2`). These expressions can be nested to arbitrary depth, resulting in the arguments of the most inner expression getting evaluated first. A semi-colon denotes a comment.

```
(+ 2 (+ 3 4)) ; evaluates to 9
```

However, to compute `2 + 3 + 4`, we do not have to nest the additions, since functions (such as `+`) can have a variable number of arguments.

```
(+ 2 3 4) ; evaluates to 9
```

[4]also known as polish notation

There can be a limit for the number of arguments that can be passed to such functions. According to the Common Lisp specification, this limit is implementation dependent but must not be smaller than 50.

Together with the `print` function, a Hello-World program can now be written as expected:

```
(print "Hello World")
```

4.2 Lists

The key datastructure of Lisp is the list. Lists are defined using expressions like above, but if we were to define a list containing all prime numbers up to 7, writing `(2 3 5 7)` would cause Lisp to interpret 2 as a function and `3 5 7` as its parameters. In other words, the expression would be *treated as code* [10]. To stop the expression from being evaluated like a function and instead being seen as data, we need to *quote* the expression: `(quote (2 3 5 7))`. Using an apostrophe or single quote before an expression is shorthand for the same operation: `'(2 3 5 7)`.

To access and retrieve elements of a list, Lisp provides the functions `car` and `cdr`. The former retrieves the beginning of the list, while the latter returns *the rest of the list.*

```
(car '(first second third)) ; eval.: first
(cdr '(first second third)) ; eval.: (second third)
```

4.3 Variables and functions

Due to the simple syntax relying on just a few characters with special meaning, a variety of characters is available for the naming of variables, functions and symbols. For instance, following expression would declare a valid global variable named `1+!b*.-x?_=>Z` containing the value 42.

```
(defvar 1+!b*.-x?_=>Z 42)
```

Lisp programmers frequently make use of this freedom. For instance, global variables are conventionally surrounded by asterisks.

```
(defvar *pi* 3.141)
(defvar *language* "Common Lisp")
```

Local variables can be declared with the function `let`. These variables are only valid inside of the body of the `let` function, meaning the last expression after binding of the variables.

```
(let ((a 5)) (+ 1 2 a)) ; evaluates to 8
(let ((a 5) (b 2)) (+ 1 2 a b)) ; evaluates to 10
```

We can observe that variables do not need a type specification, since Lisp is a dynamically typed language. Unlike strictly typed languages, variables in Lisp thus can be assigned any type and lists can contain any type:

```
; create a list with integer, symbol, string, float
'(4 abc "Alice" 3.141)
```

Analogous to variables `defun` and `flet` can be used to define global and local functions. They both take the parameters of the function in an expression as the first argument, and the definition of the body in the second argument. The following function returns the square of a given parameter x.

```
(defun square (x) (* x x))
```

4.4 Symbols

We previously mentioned the *symbol* datatype. This datatype seems identical to a string, but they differ in their internal representation and how they are compared. Symbols in Lisp are unique, while the same string can be stored multiple times in memory. Symbols can have values *bound* to them (such as strings) by using `setf`. Evaluating a symbol then returns that value.

```
(setf foo 'test)
(setf bar 'test)
```

There are several functions to test for equality in Common Lisp. The two most important ones are `eq` and `equal`. The former compares two arguments on them being the same, identical object (i.e. they point to the same memory address). The latter returns true if the given arguments contain the same object. Knowing this, we can now show off the difference between symbols and strings in code:

```
(eq foo bar)     ; evaluates to true
(equal foo bar)  ; evaluates to true

(setf foo "hello")
(setf bar "hello")
(eq foo bar)     ; evaluates to nil (false)
(equal foo bar)  ; evaluates to nil (false)
```

4.5 Conditionals

The concept of true and false in Common Lisp is implemented via lists. An empty list evaluates to false, while a non-empty list evaluates to true. With this in mind, we can now demonstrate the usage of the special operator `if` that takes three expressions as its arguments. The first argument is an expression that either evaluates to false (the empty list) or to true (a list with at least one element). If it is the latter, the `if` expression will then evaluate the second expression, otherwise the third expression.

```
(if ()
    (print "true")   ; evaluated if true
    (print "false")) ; evaluated if false
```

The empty list `()` in above expression can be replaced with `'()` `()` `'nil` or `nil`, which are all synonymous in Common Lisp. It is worth mentioning that this is not a uniform approach throughout Lisp dialects: Scheme, for example, uses explicit symbols to represent true (`#t`) and false (`#f`) [1, 2].

To avoid unreadable chaining of `if` expressions in a program, Common Lisp provides the `cond` macro that can take a variable number of arguments and can check an arbitrary conditional expression to determine which expression should be evaluated.

```
(setf foo 3)
(cond ((< foo 3) (print "Smaller"))
      ((= foo 3) (print "Equal"))
      (t         (print "Default")))
; Output: "Equal"
```

This code snippet exemplifies Lisp's *lazy evaluation*: Every expression that is associated with a condition that appears after the first true condition will not be evaluated. We can specify a default case by providing the last condition with the "true" symbol `t`. So if none of the conditions above it are met, `cond` is guaranteed to evaluate the last expression.

Lisp introduced recursion, i.e. functions that can call themselves. With this knowledge, we can now implement a function that calculates the Fibonacci number of `n`.

```
(defun fib (n)
  (cond ((< n 2) n)
        (t (+ (fib (1- n)) (fib (- n 2))))))

(fib 4) ; evaluates to 3
```

4.6 First-class functions

Functions in Lisp are *first-class functions*. In particular, this means the language supports returning functions from other functions, assigning and storing functions like variables and passing functions to other functions —so called *higher-order functions* [1]. An example for this is the commonly used function `mapcar`, which takes a function as its first argument and a list to operate on as its second. When evaluating, it will then apply the given function on each element of the list.

```
(setf mylist (list '("hel" 24)
                   '("lowo" xyz "Bob")
                   '("rld")))

(mapcar #'car mylist) ; eval.: ("hel" "lowo" "rld")
```

The notation `#'car` is syntactic sugar for `(function car)`. This is a needed mechanism in Common Lisp because functions and variables do not share namespaces – unlike Scheme.

4.7 Macros

Macros are pattern specifications that can range from simple text replacements to defining new control constructs in a language. Macros in Lisp are defined with `defmacro` and look similar to function definitions. The difference is that macros in Lisp do not get evaluated, they get *expanded into* a Lisp expression.

A simple example for a macro is the following implementation of an increment operator:

```
(defmacro 1++ (value) `(+ 1 ,value))
(1++ 1) ; evaluates to 2
```

The backtick character ‘ in Lisp can used for so called *quasiquoting*: It returns the quoted list but evaluates all expressions that are preceded with a comma.

5 PARADIGMS AND CONCEPTS

Lisp is a multiparadigm language. In particular, this means programmers can bend and extend Lisp to their liking and program in a style that fits the desired paradigm [19]. For instance, Lisp allows writing programs in a functional paradigm, but can not be considered a pure functional language by default, since Lisp functions allow the occurrence of side-effects[5] [14]. Furthermore, Lisp can be extended with object-oriented programming features [9]. An example for this is CLOS: the *Common Lisp Object System* that presents an object-oriented extension to Common Lisp. CLOS introduces concepts like classes, (multiple) inheritance, methods and generic functions [9, 21].

[5] meaning a change of state in the program that is observable outside the local environment of the called function

The reason for Lisp mutability lie in it being a programmable programming language [8]. To be able to explain how the language accomplishes this feat, we must first introduce the underlying key concepts and ideas of Lisp.

5.1 Everything is a list

In Lisp, everything is composed of expressions as opposed to statements. More precisely, these expressions are called *symbolic expressions* or *s-expressions*. In his original paper, McCarthy defined them followingly [13]:

(1) *Atomic symbols are S-expressions.*
(2) *If e_1 and e_2 are S-expressions, so is $(e_1 \cdot e_2)$.*

With this knowledge, we can illustrate the s-expression `(* 3 2)` as a tree data structure:

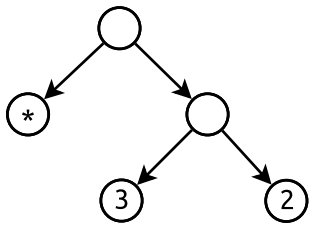

Figure 2: `(* 3 2)` as tree data structure

The whole language is structured as s-expressions, which ultimately culminates in the concept of the next section.

5.2 Code as Data

The building blocks of Lisp are s-expressions and lists. Since lists themselves however are defined as s-expressions and are structurally identical, Lisp knows no hard boundary between code and data. This leads to code and data both internally being represented as abstract syntax trees. The `eval` function as mentioned in 3.2 takes in Lisp expressions and evaluates them. To be more specific: it reads in Lisp programs expressed as Lisp data [14]. A most interesting property of this is that Lisp can be written in itself [10]. This concept is now referred to as *homoiconicity*.

5.3 Metaprogramming

Lisps lack of distinction between code and data explains its well-fitted application for *metaprogramming*, essentially meaning the technique of writing programs that write programs. Here is an example of a macro that implements the `cond` operation out of nested `if`'s and `when`'s:

```
(defmacro cond (&rest clauses)
  (if (eq (length clauses) 1)
      (if (eq (caar clauses) t)
          `(progn ,@(cdar clauses))
        `(when ,(caar clauses)
           ,@(cdar clauses)))
    `(if ,(caar clauses)
         (progn ,@(cdar clauses))
       (cond ,@(cdr clauses)))))
```

This technique is so ingrained in the language that Lisp programmers are known to write *domain-specific languages* or *DSL's* for their problem area. The result is a language that is specifically tailored to its area of application [5].

6 DEFINING LISP DIALECTS

We can now define what characterizes a Lisp dialect, and additionally explain how it is not only valid, but necessary, to refer to these languages as *dialects of Lisp*, as opposed to talking about them as languages that only draw inspiration from Lisp. We arrive at following requirements for a Lisp dialect:

- Support of macros that are treated as first class citizens of the language.
- Based around expressions as opposed to statements.
- A minimal syntax without special rules and edge cases.
- Functions as first-class citizens.
- Reliance on singly-linked lists as its primary datastructure.
- Usage of symbols.

By aiming to implement the above conditions —particularly the requirement of a minimal syntax—, a language naturally will start to resemble Lisp. Note that the use of a prefix-notation with delimiters for its expressions (like parentheses) is not a requirement, but rather a consequence.

Furthermore, it is now understandable why the concept of "C dialects" is not a widely spread one. For instance, aiming to support typical features of C in a language does not mean that such a language will arrive at a C-like syntax.

7 USAGE

7.1 Artificial Intelligence

Lisp has its root in the research of artificial intelligence and was the most popular language for A.I. programming [16]. The term *artificial intelligence* was in fact coined by Lisp inventor John McCarthy himself [15]. The success of Lisp in A.I. is explained by the field's unusual requirements that could not be met by conventional programming languages. The flexibility of Lisp turned out to be a right fit for the demands that researchers at the time had [16]. In fact, Lisp is still considered well-equipped for the task [3]. Another discipline that Lisp proved to be the adequate tool for was fast prototyping and experimentation [11] – in- as well as outside of Artificial Intelligence.

7.2 Embedded Lisp

The simplicity of Lisp makes the implementation of an interpreter an easy task and made —and continues to make— it an attractive language to embed in an application or use it to interact with the core mechanisms of a program.

The extendable, open-source text editor *GNU Emacs* uses its own Lisp dialect called *Emacs Lisp* to expose functionality of the software to the user. A small core of Emacs was written in C, but the defining trait is its extensibility that it owes to the use of its scripting language. Users of Emacs have embraced its free and customizable nature, making it possible to install extensions to use Emacs as a pdf reader[6], email client[7] or web browser[8], among others.

The CAD[9] software *AutoCad* uses its own Lisp dialect called *AutoLisp* as its scripting language and large parts of it are written in Lisp.

The *GNU Ubiquitous Intelligent Language for Extensions* or *GUILE* is a scheme-based extension that makes use of Lisp's aptitude as an embedded language and aims to allow easy integration into any application, thus eliminating the need to implement an own Lisp interpreter. Projects that use GUILE include *Guix* (a purely-function meta package manager), *GNU Cash* (a free financial management software) and the *GNU Project Debugger* —also known as *GDB* .

7.3 Education

Naturally Lisp was taught heavily in courses on artificial intelligence, but also outside of it: the popular textbook *Structure and Interpretation of Computer Programs* served as a companion book to a course at MIT with the same name starting in 1980 by Hal Abelson and Gerald Sussman [1] and found heavy usage outside of MIT upon release. The book uses the in-house developed Scheme as its language for demonstrating algorithms and programming practices.

In 2008, the course was stopped being taught using the textbook as well as Scheme[10]. Despite that, it is worth mentioning that teaching of the book at MIT resumed with another course in 2019[11] with the younger Lisp dialect *Racket*.

7.4 Modern Lisps

Racket: Racket is a scheme-based Dialect that began development under the name "PLT Scheme", until it was later renamed. It is used as a language for teaching, includes a graphical IDE and adds useful procedures to Scheme [5].
Clojure: A modern dialect that has gained traction in recent years is *Clojure*. Clojure is designed to run on the Java Virtual Machine and consequently supports interoperability with Java itself. It employs typical Lisp syntax and encourages functional programming [6].
Clasp: In spite of the many implementations of Common Lisp, there are still new ones being developed. *Clasp* is an implementation that is designed to interoperate with C++ using an LLVM backend [18].
Arc: *Arc* is a new dialect of Lisp that is designed for prototyping of software and is in early release [12].

8 INFLUENCE

Many of the languages features that we introduced were in fact inventions by Lisp. Lisp introduced first-class functions [4].

Garbage collection was an innovation by Lisp. It was described in McCarthy's original paper as "reclamation cycles" [13], although internally the process was already called garbage collection [14]. Recursion as a form of functions calling themselves first appeared in Lisp [4].

[6] https://github.com/politza/pdf-tools
[7] https://github.com/wanderlust/wanderlust
[8] GNU Emacs Web Wowser (EWW) package, by now a built-in
[9] computer-aided design
[10] https://mitadmissions.org/blogs/entry/the_end_of_an_era_1/ accessed 25-05-2020
[11] http://web.mit.edu/alexmv/6.S184/ accessed 25-05-2020
[12] http://arclanguage.org/

8.1 Conditional Expression

Lisp was the first language to introduce conditional expressions as the one shown in 4, while languages like Fortran still only offered conditional gotos [14]. Languages later started supporting this type of control flow with the `if-else` instructions that are now ubiquitous in programming languages [4].

8.2 Object-oriented Programming

We also trace back influences from Lisp to the now common paradigm of object-oriented programming. According to Alan Key, Lisp deeply influenced him during the design of Smalltalk [7]. The success of Smalltalk on the other hand fueled incorporation of object-oriented features into Lisp dialects. Thanks to its extensibility, Lisp turned out to be suitable for experiments for more advanced object-oriented concepts like multiple inheritance.

Additionally, an impact on the web was left by Lisp through the language *JavaScript*. Its creator, Brendan Eich, originally began work on a browser-embedded language intended to be modeled after Scheme [17]. Later on, a decision to make the syntax more Java-like was made. The aftermath is a language that resembles Java, but offers features of Scheme like closures, first-class functions and support for a functional style of programming [17].

9 DECLINE

Having shown the areas in which Lisp excels and its wide-ranged influence, we now need to ask the logical question of why Lisp does not see the same amount of usage as the languages it greatly influenced. We will explore three potential reasons and see whether they still apply today.

9.1 Performance

An argument against Lisp for early computer hardware is to be made in regards to performance. The resource requirements of running a Lisp program compared to languages that did not use garbage collection, dynamic typing or an interpreter like C or Fortran was significantly higher and thus those languages were preferred for performance critical tasks. The datastructure of Lisp is not helping in this cause, since a Lisp program will frequently access an element of a large list via linear searching [11] and early Lisp had a reputation of being inefficient for numerical computations [1].

Modern computers on the other hand, can accommodate these requirements. Python — like Lisp — is an interpreted, dynamically typed, garbage collected language that sees widespread usage.

9.2 Design

Through its self-contained design, Lisp was not well suited for systems programming, resulting in increased usage of languages like C and Fortran in that area [11].

However, the need for systems programming languages has declined since then.

9.3 Change in needs

As mentioned in 7.3, the MIT course for *Structure and Interpretation of Computer Programs* was stopped in 2008 and switched the used programming language from Scheme to Python. In a 2016 talk [22] — on being asked for the reasons behind the switch — Gerald Sussman remarked that the book originally was used to teach a language of engineering that lost relevancy for today's engineers: While engineers in the 1980s and 1990s used well-understood parts to build complex systems, the engineers of today more commonly combined complex systems with small parts. In particular, the practice of simply including and combining established libraries and frameworks to achieve the desired results arose and transformed the needed skills for software engineering [22].

10 RELATED WORK

In his article *How Lisp Became God's Own Programming Language* [23], Sinclair Target tries to explain how Lisp managed to stay relevant and earned the reputation of a language with almost mystical properties in programming culture. Throughout, he presents three theories. The first concerns Lisp's small, irreducible core language definition. Unlike other programming languages, Lisp is axiomatic and can be defined fully in theory with just a few special operations. The second is about the use of Lisp in the field of A.I. and how it contributed to giving Lisp a more futuristic appearance. The third theory attributes Lisp's success to the textbook *Structure and Interpretation of Computer Programs* [1] that we also mentioned throughout this paper.

11 CONCLUSION

We introduced the syntax of Lisp and were able to show how the simple structure of is not a hindrance but rather the key factor in the deeper ideas that enable its powerful features, making it multiparadigm in every sense of the word.

By pinpointing these features, we found that the term "Lisp dialect" is a valid one, because a language that tries to implement these concepts will eventually resemble Lisps foundation — either intentionally or by accident.

We showed where Lisp's capabilities found their application and that the usefulness of Lisp in the early days still is present today and that new developments are on the horizon.

We were able to trace Lisp's influence to the major languages that we mentioned in 2 and more by showing how its innovations are found in them and how it helped shape whole programming paradigms.

Finally, we took a critical look at its decline and concluded that its loss in use can be attributed to practical and historic developments and not a shortcoming of the core of the language itself. We determined that a language's inertia or momentum now plays a key role in determining the success of a language. This crucial momentum is now picking up again as recent developments come to fruition, fueling speculation of another rise of the Lisp programming language.

REFERENCES

[1] Harold Abelson and Gerald J. Sussman. 1996. *Structure and Interpretation of Computer Programs* (2nd ed.). MIT Press, Cambridge, MA, USA.

[2] Conrad Barski. 2011. *Land of Lisp - Learn to Program in Lisp, One Game at a Time!* No Starch Press. http://nostarch.com/lisp.htm

[3] Christian Betz and Lothar Hotz. 2011. Verwendung von Lisp in KI-Projekten. *KI - Künstliche Intelligenz* 26, 1 (Nov. 2011), 69–74. https://doi.org/10.1007/s13218-011-0150-7

[4] Pascal Costanza, Richard Gabriel, Robert Hirschfeld, and Guy Steele. 2008. Lisp50: The 50th birthday of lisp at OOPSLA 2008. 853–854. https://doi.org/10.1145/1449814.1449882

[5] Ryan Culpepper, Matthias Felleisen, Matthew Flatt, and Shriram Krishnamurthi. 2019. From Macros to DSLs: The Evolution of Racket. In *3rd Summit on Advances in Programming Languages (SNAPL 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 136. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 5:1–5:19. https://doi.org/10.4230/LIPIcs.SNAPL.2019.5

[6] Chas Emerick, Brian Carper, and Christophe Grand. 2012. *Clojure Programming - Practical Lisp for the Java World.* "O'Reilly Media, Inc.", Sebastopol.

[7] Stuart Feldman. 2004. A Conversation with Alan Kay. *Queue* 2, 9 (Dec. 2004), 20–30. https://doi.org/10.1145/1039511.1039523

[8] John Foderaro. 1991. LISP: Introduction. *Commun. ACM* 34, 9 (Sept. 1991), 27. https://doi.org/10.1145/114669.114670

[9] Paul Graham. 1993. *On LISP: Advanced Techniques for Common LISP.* Prentice-Hall, Inc., USA.

[10] Paul Graham. 2001. Roots of Lisp. (May 2001). http://www.paulgraham.com/rootsoflisp.html (retrieved on May 15, 2020).

[11] Gary D. Knott. 2017. *Interpreting LISP - Programming and Data Structures.* Apress, New York.

[12] Daniel Kochmanski. 2016. *Common Lisp ecosystem and the software distribution model.* http://turtleware.eu/static/talks/pkgsrcCon-2016-lisp.pdf (retrieved May 25, 2020).

[13] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (April 1960), 184–195. https://doi.org/10.1145/367177.367199

[14] John McCarthy. 1978. *History of LISP.* Association for Computing Machinery, New York, NY, USA, 173–185. https://doi.org/10.1145/800025.1198360

[15] John McCarthy. 1997. *Defending AI Research - A Collection of Essays and Reviews.* Cambridge University Press, Cambridge.

[16] Peter Norvig. 1992. *Paradigms of artificial intelligence programming: case studies in Common LISP.* Morgan Kaufmann.

[17] Casimir Saternos. 2014. *Client-Server Web Apps with JavaScript and Java - Rich, Scalable, and RESTful.* "O'Reilly Media, Inc.", Sebastopol.

[18] Christian A. Schafmeister and Alex Wood. 2018. Clasp Common Lisp Implementation and Optimization *(ELS2018).* European Lisp Scientific Activities Association, Article 8, 6 pages.

[19] Peter Seibel. 2005. *Practical Common Lisp.* Vol. 1. Apress.

[20] Guy L. Steele. 1982. An Overview of COMMON LISP *(LFP '82).* Association for Computing Machinery, New York, NY, USA, 98–107. https://doi.org/10.1145/800068.802140

[21] Guy L. Steele and Richard P. Gabriel. 1993. The Evolution of Lisp. *SIGPLAN Not.* 28, 3 (March 1993), 231–270. https://doi.org/10.1145/155360.155373

[22] Gerald Jay Sussman. 2016. *Flexible Systems, The Power of Generic Operations.* LispNYC. https://vimeo.com/151465912 (retrieved May 21, 2020).

[23] Sinclair Target. 2018. *How Lisp Became God's Own Programming Language.* https://twobithistory.org/2018/10/14/lisp.html

[24] D. A. Turner. 2013. Some History of Functional Programming Languages. In *Trends in Functional Programming*, Hans-Wolfgang Loidl and Ricardo Peña (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20.