Preface

This document is a collection of essays written by the students of the seminar course INF328B on *History of Programming Languages* that was given at the University of Bergen (Norway) in the Spring term 2021. Instructors of the course were Mikhail Barash and Anya Helene Bagge from Bergen Language Design Laboratory at the Department of Informatics.

We had 13 students who actively participated in the course (11 Master students, 1 Bachelor student, and 1 Ph.D. student). At each session (classes were held once a week), a student or a team of two students presented a paper from *The Fourth ACM SIGPLAN History of Programming Languages Conference* (a.k.a. HOPL IV, to be held in June 2021[1]).

The students of the course have presented HOPL IV papers on history of the following programming languages: **APL**, **Clojure**, **D**, **F#**, **Groovy**, **JavaScript**, **Logo**, **S & R**, and **Standard ML**.

The presentations on three languages from HOPL IV were given virtually at our course by the corresponding HOPL IV papers' (co)author(s): **Bjarne Stroustrup** talked about history of **C++**[2], **John Reid** presented history of **Fortran**[3], and **Peter Van Roy** gave a talk on history of **Oz**[4].

To document our students' learning experiences with this course, we asked them to prepare a short essay summarizing the HOPL IV paper they have read and presented. The structure of each essay is as follows[5]:

- a short discussion on why the student has chosen a particular HOPL IV paper to present,
- a brief overview of that HOPL IV paper,
- a brief overview of the programming language,
- a discussion of the student's most favourite feature of that language,
- related work on the language, both within the context of HOPL I–IV conferences and a wider context,
- a brief overview of tooling available for the language (compilers, IDEs),
- the student's personal experience with the language and the HOPL IV paper,
- questions that the student would ask the author(s) of the HOPL IV paper.

We see these essays as a valuable documentation of students' learning experiences, and we hope that this will be an exciting read for everyone.
In the meantime, we are looking forward to the next edition of our course—to be held in 2030s—based on papers from the next HOPL conference.

Bergen, 14th June 2021
Mikhail Barash (`mikhail.barash@uib.no`)
Anya Helene Bagge (`anya@ii.uib.no`)

[1] We are thankful to HOPL IV organizing committee, in particular, Guy L. Steele Jr., for offering a special rate for the students of our course to participate in the conference.
[2] The talk was organized in cooperation with University of Turku (Finland): `https://www.utu.fi/fi/ajankohtaista/uutinen/utu-tech-webinar-bjarne-stroustrup`.
[3] Announcement: `https://www.uib.no/ii/144128/history-coarrays-and-spmd-parallelism-fortran`.
[4] Announcement: `https://www.uib.no/en/ii/144917/history-oz-multiparadigm-language`.
[5] The LaTeXtemplate we used is an adapted version of `https://github.com/sylvain-kern/magazine`.

Contents

APL

Link to the HOPL IV paper:
https://dl.acm.org/doi/10.1145/3386319

Link to the students' presentation:
https://git.app.uib.no/uib-hopl-iv/slides/-/blob/master/APL.pdf

Essay on History of APL

Karl Henrik Elg Barlinn
University of Bergen

The paper "*APL since 1978*" [1] by Roger K.W. Hui and Morten J. Kromberg picks up the story where the HOPL I paper *The Evolution of APL* [2] left off. The paper describes the evolution of the APL umbrella of languages. It reflects on how not only APL, but the world as a whole, have been shaped by the great progression since 1978; and throws a few new ideas for how APL may yet be changed.

Brief Overview of the HOPL Paper

The paper starts off by presenting a historical perspective on how the world as a whole has changed since the previous HOPL paper on APL [2] was presented. Then it explores all new functions and primitives which have been added to the language.

I chose this particular HOPL paper to present because I have always been fascinated by interesting problems. APL is a language which I knew nothing about before preparing the presentation. During the initial elevator pitch lecture APL was called an "array processing language with special graphical notation" which does sound, to me at least, like nothing I have come across before and very interesting.
—Elg

It also explores what it means to be primitive, why there are so few of them, and what this means for the language as a whole. New concepts such as leading axis (major cell) and trains are explained with special regard to how they drive the continuing development of the language. Branching languages, such as J and K, are also mentioned in how they influence APL with their own concepts and ideas. The paper also explores the the famous symbols of APL, and why there are so few of them. APL was, in its infancy, known for the easy and natural way a non-technical user could create user interfaces. How this evolved with the arrival of non-text based user interfaces is also explored in the paper. The paper goes on to explore how compilers, and interpreters have changed throughout the years, why APL does not really need a compiler, and why it has some anyway. Finally, the paper explores subjects that seem to interest the authors of the paper: what's in a name, backwards compatibility, index origin, among other more niche and minor topics. This is not a paper strictly about the history of APL, it is a celebration on how far APL have come since 1978.

Brief Overview of APL

APL was created by Kenneth Iverson during the 1960s to address the shortcomings of conventional mathematical notation, and to make it executable. As such APL is designed for executing mathematical notation rather than a programming language. This is one of the reasons it used glyphs rather than the more normal ASCII names we are used to in other languages like C. Curiously it has no operator precedent, everything is associated to the right. This first version of APL is known as *APL\360* was published in 1966 [1]. The main targets of APL are domain experts without a need to have a background in computer science. This drives the implementers of the language to try and accommodate this lack of formal education in programming by for example, by making it easy to create user interfaces.

The main features of APL is the concise way complex algorithms can be created. This is done by having *functions* and *operators* perform commonly needed tasks. A function is applied to an array of arguments to produce array results. There is also the notion of the major cell within the design of functions. As explained by the paper "A major cell of an array is a subarray with rank one less than the rank of an array arranged along its leading axis: an item of a vector, a row of a matrix, a plane of a 3-d array, etc. Operating on the leading axis is analogous to treating an array

as a (conceptual) vector and with the major cells as its (conceptual) items" [1]. Some functions mentioned in the paper are tally, which returns the number of major cells in an array, and grade, which when used monadically produces the indices needed to sorts its argument. There are of course others, but those are left to the reader to explore.

> In a way I always thought programming languages should use a C style syntax; but the more I learn about non-C-like languages, the more I realize how wrong I was.
> —Elg

Operators are higher order functions, meaning they take functions as arguments and not arrays. An array can either take a single argument (monadic) or two arguments (dyadic), but not either one or two (ambivalent). Operators are what distinguishes modern APL from APL in 1978. Back then there were only seven operators, while now there are over twenty. The place of honor, regarding operators, goes to the rank operator which are in detail described in the paper [1]. Other operators which might not be known to the reader are *power, key, stencil,* and *each.* I implore the reader to take a look at these fascinating operators.

APL have since influenced many modern and widely used programming languages, such as MATLAB, C++, Python, J, and K [5]. In conclusion, APL is not a dead, ancient, language archaeologists are exploring, but rather a continuously developing language which have and will be influencing new programming languages for years to come.

User Interfaces with APL: An Overview

Due to the non-technical nature of the target audience, APL have from the very start simplified created user interfaces for their developers. During the early days, when APL ran on time-shared mainframes, the terminal were more simple to code for. Often it only had a 28 rows and 80 columns text display, sometimes with more than two colors! APL is very well suited for this kind of text matrix display, after all the screen can be looked at as another, rather special, matrix. Using quad notation, namely □*win*, a user could create state-of-the-arts character-based interfaces.

However, as the glory days of mainframes dwindled, financial institutions drifted towards Microsoft Windows. Due to the fierce competitive APL market at the time there was no standardisation causing APL from different vendors not being executable. This was the time where Object Oriented Programming (OOP) also took off, Dyalog exploited this together with namespaces to integrate the Win32 objects directly into the workspace. The HOPL IV paper, written by employees at Dyalog, argues that this makes APL significantly more powerful and easier to use.

The "C++ API" era was a frustrating time for APL users and vendors alike. The APIs changed rapidly, forcing vendors to rapidly innovate to keep up with the ever changing API market. As mentioned, APL developers have little interest in the engineering efforts with behind the scenes development and just wanted the applications to keep working. Some vendors leaned too heavily on deprecated frameworks, which ultimately made their dialects to be withdrawn from the market. The *.NET* framework and other more modern frameworks are much more usable for both users and implementers as they use the same design patterns. It also supports features such as *reflection* and *dynamic code emission* which makes it easier to create a generic bridge between APL and "normal" *.NET* classes. With this APL can use the same APIs as other *.NET* languages to create user interfaces. Another alternative presentation technology, namely Jupyther notebooks are usable by less technical users.

Related Work on APL

As mentioned, the history of APL until 1978 can be found in *The Evolution of APL* [2]. This paper is worth reading for the earliest history of APL. APL is an influencing language which have been an important influence of the rise of functional programming languages[6]. APL has also

influenced other HOPL-relevant languages such as C++ with individual functions and ideas [7]. MATLAB, another language represented in HOPL IV, have called APL a predecessor of MATLAB [8]. APL has reached a point where it no longer directly influences languages, but rather influences languages which are influenced by APL. For example APL influenced FP [6] which again influenced Haskell, a language presented at HOPL III. Pper [1] notes a couple of other history papers in *appendix E. A History of APL in 50 Functions* [3] celebrates APLs 50th year of existence. It, unsurprisingly, shows, explains and highlights 50 functions written in APL. For example the *Halting*, *Monty Hall*, *pascals triangle*, and 47 more fascinating functions. The paper *APL Quotations and Anecdotes* [4] gives a lighthearted look into the personalities of the creators behind APL. It features numerous puns, funny stories, interesting facts, and of course quotes from the people who created APL. The best place for learning to program in APL is described in the next section.

Tool Support for APL

Perhaps the most useful tool for people new to APL is `https://tryapl.org/`. It is an in-web-browser Dyalog APL interpreter which works on all modern web-browsers. You can download Dyalog for free for non-commercial usages. If you prefer a FOSS implementation there is `https://www.gnu.org/software/apl/`, this however does not have an online interpreter.

Personal Experience

APL is a niche, beautiful, powerful, "read-only" programming language. As for someone who comes from a computer engineering background, APL was never designed for me. It looks like something written on *The One Ring* from *Lord of the Rings*. This was in a way what was intrigued by how different it is from what I have learned in school. In a way I always thought programming languages should use a C style syntax; but the more I learn about non-C-like languages, the more I realize how wrong I was. This paper opened my eyes, so to speak, in that it introduced a whole new world of (to me) strange and wonderful new ways of looking at problem solving.

For the authors of the paper I wish to ask a couple of questions which naturally were not addressed in the paper proper due to their non-history nature. Firstly, I wish to know if there are plans to try and popularize APL within the wider community of programming languages. I ask this because I see how it can be very useful for the mathematical community to write papers *and* be able to execute the notation.

The second question I have is, given unlimited influence, where do you wish to see APL be used? If your answer is everywhere, does that mean APL is fit to do everything? On the other hand if your answer is *not* everywhere, where is not fit to be used and why?

Finally, do you think APL with its glyphs is more fit to be taught in school than J, as special equipment is no longer an issue with UTF being widely adopted?

References

[1] Roger K. W. Hui, Morten J. Kromberg, *APL since 1978*, Proc. ACM Program. Lang., Vol. 4, No. HOPL, Article 69.

[2] Adin D. Falkoff and Kenneth E. Iverson. 1978. *The Evolution of APL*. Available at: `http://www.jsoftware.com/papers/APLEvol.htm`.

[3] Roger K.W. Hui. 2016. *A History of APL in 50 Functions*. Available at: `https://www.jsoftware.com/papers/50/`.

[4] Roger K.W. Hui. 2020. *APL Quotations and Anecdotes*. Available at: `https://www.jsoftware.com/papers/APLQA.htm`.

[5] *Wikipedia page on "APL (programming language)"*, link: `https://en.wikipedia.org/wiki/APL_(programming_language)`.

[6] Awards.acm.org. 1977. *ACM Award Citation – John Backus* Archived from the original on February 12, 2008. `https://web.archive.org/web/20080212043802/https://awards.acm.org/citation.cfm?id=0703524&srt=all&aw=140&ao=AMTURING`.

[7] CPP Reference, *std::iota*, link: `https://en.cppreference.com/w/cpp/algorithm/iota`.

[8] Computer History Museum, *An interview with CLEVE MOLER Conducted by Thomas Haigh On 8 and 9 March, 2004 Santa Barbara, California.*

Avaliable at: https://web.archive.org/web/20141227140938/http://archive.computerhistory.org/resources/access/text/2013/12/102746804-05-01-acc.pdf.

[9] Python 3 Documentation: *itertools — Functions creating iterators for efficient looping*. Link: https://docs.python.org/3/library/itertools.html.

Essay on History of APL

Sondre Nilsen
University of Bergen

This essay presents the paper "APL since 1978" [1], written by Roger K. W. Hui and Morten J. Kromberg published in Proceedings of the ACM on Programming Languages, volume 4 and will be presented at the HOPL IV conference. This paper, as the name suggests, is about the history of APL from 1978; though not a history of the language but more a history of the evolution of the language itself, how it has changed, how influences from software and hardware has changed it to where it is today. The paper builds upon the history in the paper "The Evolution of APL" [4], presented at the inaugural HOPL conference in 1978.

Brief Overview of the HOPL Paper

I chose to present this paper mostly because APL is a language that I think most computer scientists / students have seen at some point and thought to themselves; "what on earth is this?" I've come across it at various times, mostly through social media whenever the famous video of an employee at Dyalog implementing the Game of Life in APL [10]. It is a language unlike most other languages, including esoteric programming languages,——

The paper walks through how changes in hardware has changed what's possible in APL and how different dialect and implementations have inspired and changed how APL works; from the early days of mainframes to the home computer to Software-as-a-Service (SaaS). The paper talks a bit about the actual history of APL as a language before getting into the specifics on the evolution of core functions and operators, how the underlying data structures have changed and how we use computers have forced APL to change alongside the rise and fall of computing.

Much of the focus of the paper is on a few select functions and operators, as well as the underlying data structures used and their implementation. It also covers how idiomatic APL has changed over the years, how to parse and implement compilers and interpreters and touches upon the usage of glyphs in the language and how user interfaces could be used.

The selected functions and operators are chosen to highlight specific discoveries or insights that led the design of APL forward, for example how the tally operator ≢ came into being after having evolved from similar but more basic functions like ρ (shape). Much of the time spent on functions and operators focus on how different implementations or authors came up with initial solutions to problem, how they evolved and how we ended up with what we have now; for example how the rank operator ⍤ is a generalization of the each operator ¨.

The paper also goes into the underlying data structures used in different implementations, from sparse arrays to arrays of objects and more.

APL has a rich history that spans a long time, especially for a language that has been mostly in a niche of its own since nearly its inception.

Brief Overview of APL

APL, "A Programming Language", is an array programming language. You might make a case for the name being a backronym, but the book published by Iverson in 1963 is where the name is from [6]. Core to any APL dialect is the array, be it a single vector, a 2D matrix or a 3D plane, as well as the use of glyphs instead of ASCII characters to represent functions and operators (though some dialects like J and K do use ASCII [1, p. 62]).

If you come from more conventional languages based on object-oriented programming, it is sure to give you a mind-bending experience when seeing actual APL code being written and executed. I'm sure it's comparable to students' experience when they see someone writing in a functional programming language for the first time: how

can so few characters do so much?
Part of the beauty of APL is in its notation, which was developed by Iverson from his time at Harvard and became the basis manipulating arrays in what would later become APL. Functions and operators in APL have a range of tricks up their sleeve. A function or operator can be

1. *monadic*: it takes a single argument on the right.
2. *dyadic*: it takes two arguments, one on the left and one on the right.
3. *ambivalent*: either monadic or dyadic.

——where instead of regular ASCII characters you use a specialized alphabet of glyphs and symbols to denote what you want to do. How can anyone actually remember and use that? When I saw that there was a paper this year available for APL, I figured this was my chance to finally be able to attempt to get a better grasp of what APL is.
—Sondre

For example, in its monadic form the ⌈ function means `ceil`, while in dyadic form it is `max`. So ⌈`3.2` returns `4` while `5.2`⌈`1.2` returns `5.2`.
Furthermore, since APL is based on array programming, it also means that functions and operators also work on any array, ⌈`1.2 5.3 2.3 8.0` will apply ⌈ to each element along the leading axis of the array, in this case returning `1 5 2 8`. It also works dyadically on two arrays, `1.2 5.2 3.2`⌈`1 3 4` will pairwise compare the arrays and return the largest of the pair: `1.2 5.2 4`. This does require the arrays to have the same shape.
Shape and leading axis are two important concepts in APL, the *shape* is simply the array shape, e.g., a 2×4 matrix while *leading axis* is the main axis of an array. Closely related to the leading axis is the concept of major cells which are items of rank one less than the array containing it arranged along its leading axis [1, p. 26]: an item in a vector, a row in a matrix and so on.
Compared to most conventional languages, the functions and operator may make you scratch your head slightly: what on earth could $\alpha(\ominus \ddot{\star}1)\omega$ possibly mean? Built-in primitive functions and operators all use their own symbols, while more can be accessed through quad names: ⎕`A` for the alphabet, for example. Variables and functions can be named using the assign ← function: `x ←` ⍳`5`.
Finally, we need to explain the difference between functions and operators, if you have previous experience in functional programming explaining operators as higher-order functions should get the point across. Functions apply to an argument (mostly arrays) and return an array, while operators apply functions to arrays and therefore return derived functions.

My favorite APL feature: arrays

Now, this might feel a bit like cheating given that it's not really a single feature in the sense of a single function or operator, but given how central arrays are to APL and how foreign the concept is to most programmers I think it is my favorite feature from it (similar to how function composition is my favorite thing in Haskell, which—in a similar vein—is not strictly a feature).
In most of the programming languages I use (Rust, TypeScript, Kotlin and Python) is in some sense built on objects, you create structures that contain the data matching some phenomenon and then add some methods that make sense for that *thing* to have. In a language like APL you really need to radically change how you think about the problem. As an example I will show how to solve the Two Sum [15] problem in both Python and APL.
In short, you are given an array of numbers and a target value, and you need to return the indicies of the numbers that sum to the target value. So for `[2,7,11,15]` with a target of `9`, the result should be `[0,1]`. For conciseness I am omitting imports, and these do not give the correct answer for inputs like `[3,3], 6`.

```
def two_sum(nums: List[int],
                target: int) -> List[int]:

     c = list(combinations(nums, 2))

     (x, y) = [(x, y) for (x, y)
               in c if x + y == target][0]

     return [nums.index(x), nums.index(y)]
```

The Python implementation is fairly straightforward, create all combinations of numbers in the list, iterate through these tuples and find the ones that sum to our target, then we find the indices in our input that matches our numbers.
The APL solution looks like Egyptian hieroglyps that somehow find the correct solution:

```
twoSum ← {1↑⍸⍺=(⍵∘.+⍵)}
```

Here is how these functions can be called in Python and APL, respectively:

```
     two_sum([2, 7, 11, 15], 9)
[0, 1]
     9 twoSum 2 7 11 15
0 1
```

> APL as a language is truly something unique, my brain melts when you try to think about problems as arrays and not objects with state. Very similar to how I felt when I first learnt Haskell: awe at how easy it makes things seem but also very different and strange compared to previous experiences.
> —Sondre

A couple of things to note here:

- α is the left argument,
- ω is the right argument.

This means that our `twoSum` function is dyadic, the target goes on the left and the list on the right. I will try to explain in words, though for clarity I have included a explanation with code in the appendix.

1. $(\omega\circ.+\omega)$: first, we take the outer product of our list and sum each cell.
2. $\alpha=$: we then find which cells contain our target value.
3. $\underline{\iota}$: we then find the indices of our target values.
4. $1\uparrow$: finally, we take the first element.

This to me is completely magical and awesome, in just a few symbols we have done what the Python solution required multiple lines to do, and all without state, just passing the result from one function and operator to the next. Yes, unless you know APL the solution is utterly incomprehensible, but just from reading the paper and playing around a bit on TryAPL [14] I was able to implement this solution in a few minutes.

Related Work on APL

This paper is the successor of the previous "The Evolution of APL" [4] paper presented at HOPL I. This is as far as I am aware the only paper that is directly related to APL. Array programming languages are few and far between, with APL being the only "mainstream" dialect of its kind. In my opinion, outside of the two submitted papers there aren't many that related directly to APL, though two worth mentioning are the papers on MATLAB [2] and R [3], where the paper on MATLAB actually mentions bearing a slight resemblance of APL [2, p. 5].
R, being an array language, in some sense builds upon the foundations laid by APL, which can be seen with for example Dyalog having a guide to interface with R from Dyalog APL [7], and there is a paper on how to write R as if it was an APL dialect [5]. I have however been unable to find direct mentions of APL as an inspiration for R.

Tool Support for APL

There are many different dialect and descendants of APL, each one slightly different and some very different, so there is no single unified editor or toolchain one can use for it. The most feature-rich implementation with the most support is Dyalog APL, where one can use TryAPL [14], or their new IDE RIDE [11]. The APL Wiki [9] maintains a list of text editor extensions [8] that one can use to develop with APL. As an example of incompatibilities between dialect, for Emacs there are two different modes for either GNU APL [12] or Dyalog APL, which are mutually incompatible.

Coming from more mainstream languages the tooling support for APL could be described as poor, though the RIDE IDE works as one would expect from an editor though it is missing most bells and whistles compared to IDEs like those created by JetBrains [13].
If you wanted to try out APL, I can think of no better resource than TryAPL [2]. It has help, tutorials, explanations for most functions and operators and interactive explanations of, for example, implementing Game of Life in APL. A distant second is RIDE, but I actually prefer TryAPL.

Personal experience

I had a blast learning about the history of APL, and also learning how to write and (very barely) read it. APL is a very interesting language, and one that I has piqued my interest a few times here and there but this was the first time I actually made an effort to understand it. The paper did explain and introduce APL very well, though it felt like you should have some familiarity with array languages beforehand because some terms or explanations were very difficult to grok.
The paper mixes discussion of implementation, history and how things work very well, though in my opinion the sections on parsing APL and compilers felt a bit superflous; probably mostly because they were the least interesting parts of the paper in my opinion. APL as a language is truly something unique, my brain melts when you try to think about problems as arrays and not objects with state. Very similar to how I felt when I first learnt Haskell: awe at how easy it makes things seem but also very different and strange compared to previous experiences.
I don't really have many questions for the authors, I think the paper was very well balanced between subject expertise, beginner explanation and thorough introductions to the history of APL and implementations and so on. If I would have some feedback it'd be to include an "array programming languages for dummies" appendix that could be used to look up foreign concepts, words and phrases that unfamiliar aspiring APL developers may not know.
I highly recommend anyone with a passing interest in programming languages and new ways of approaching problems to read the paper and try out APL for yourself. I hope that it was as enlightening and exhilarating for you as it was for me, even though it still feels like writing and reading Egyptian hieroglyphs. The fact that the language has been going strong for 50 years with many different implementations and dialects just shows that there is a niche for these languages to occupy, regardless of how far away from the mainstream they are.

References

[1] R. K. W. Hui, M. J. Kromberg, *APL since 1978*. Proc. ACM Program. Lang. HOPL IV. 2020.

[2] C. Moler, J. Little, *A History of MATLAB*, Proc. ACM Program. Lang. HOPL IV. 2020.

[3] J. M. Chambers, *S, R, and Data Science*, Proc. ACM Program. Lang. HOPL IV. 2020.

[4] A. D. Falkoff, K. E. Iverson, *The Evolution of APL*, ACM HOPL I. 1978.

[5] J. De Leeuw, M. Yajima, *APL in R*. Available at: `10.13140/RG.2.1.2372.0724`.

[6] K. E. Iverson, *A Programming Language*. John Wiley & Sons, Inc. 1962.

[7] *Dyalog APL R Interface Guide*. Available at: `http://docs.dyalog.com/14.1/Dyalog%20APL%20R%20Interface%20Guide.pdf`.

[8] *APL Wiki page on "Text editors"*. Link: `https://aplwiki.com/wiki/Text_editors`.

[9] *APL Wiki*. Link: `https://aplwiki.com/wiki/Main_Page`.

[10] *Conway's Game Of Life in APL* (video recording). Link: `https://www.youtube.com/watch?v=a9xAKttWgP4&t=17s`.

[11] *RIDE*, Dyalog Ltd. Link: `https://github.com/Dyalog/ride/`.

[12] *GNU APL*. Link: `https://www.gnu.org/software/apl/`.

[13] *JetBrains*. Link: `https://www.jetbrains.com/`.

[14] *TryAPL*. Link: `https://tryapl.org/`.

[15] *Two Sum*, Leet Code. Link: `https://leetcode.com/problems/two-sum/`.

Appendix: Explanation of Two Sum Function in APL

1. First, we create our input list and print it:

```
      ⍝ x ← 2 7 11 15
```

2. The outer product takes a left and right argument and a binary function. Our binary function here is the dyadic variant of ",": catenate. This creates a matrix of each possible pairing of elements in our array:

```
     x∘.,x
2 2   2 7   2 11   2 15
7 2   7 7   7 11   7 15
11 2  11 7  11 11  11 15
15 2  15 7  15 11  15 15
```

3. Instead of using catenate, we can sum each cell; in our case we are looking for 9:

```
       x∘.+x
 4  9 13 17
 9 14 18 22
13 18 22 26
17 22 26 30
```

4. Then we simply check which cells equal 9. Due to symmetry from the outer product, there are two cells that equals 9: those in positions (0, 1) and (1, 0).

```
       9=x∘.+x
0 1 0 0
1 0 0 0
0 0 0 0
0 0 0 0
```

5. Using the *where* function we then find the indices where the cell equals 1:

```
       ⍸9=x∘.+x
 0 1   1 0
```

6. We can then take the first element in our array:

```
       1↑⍸9=x∘.+x
 0 1
```

7. To convert this code into a function, we wrap it in braces and replace the target with α and the list with ω:

```
      twoSum ← {1↑⍸⍺=(⍵∘.+
         ⍵)}
      9 twoSum x
 0 1
```

CLOJURE

Link to the HOPL IV paper:
https://dl.acm.org/doi/10.1145/3386321

Link to the students' presentation:
https://git.app.uib.no/uib-hopl-iv/slides/-/blob/master/Clojure.pdf

Essay on History of Clojure

Daniel Berge
University of Bergen

This essay summarizes the paper "A history of Clojure" by Rich Hickey [1] presented at the fourth HOPL conference. The paper discusses the history of the language starting from its initial design to 2019.

Brief Overview of the HOPL Paper

This paper covers the background for Clojure, and importantly why Rich Hickey decided to make Clojure.

I chose this paper because I am interested in functional programming, and I have some background in Java programming. Therefore, a functional programming language that runs on JVM caught my interest. I had some experience with functional programming, due to courses taken in my bachelors degree at the university, but the only functional language that was introduced was Haskell. In addition, I had no prior experience or knowledge about Lisp or dialects of Lisp, and wanted to learn a bit about the topic.
—Daniel

It starts with the motivation behind Clojure, where Hickey addresses his experience with object-oriented languages, and his first impressions with functional languages. He has professional experience with C++, Java and C# and categorizes himself as a information systems programmer.

The next part of the paper is Clojure's initial design and implementation phase. He started working on Clojure in 2005 and did all of the early work by himself, he later open-sourced the project and released its first version in 2007. The paper also covers his design choices for the language according to Lisp, including the similarities and differences.

The evolution of Clojure is a big part of the paper, as this is the history of how it turned out to be the language it is today. The author goes through each of the versions released, and covers his choices of features and implementation details. He includes everything from its first release of Clojure 1.0 to 1.10 which was released in 2019, and his focus into the future.

In the retrospective the author reflects on the language, its use by others, and feedback from the community, which helped Hickey form the language. The community helped with support, bug fixes and some implementation work in the later years. Big parts of the community comes from object-oriented languages, many from java, as Clojure is hosted on JVM. They found that Clojure is a great language for them to code better applications and systems.

Lastly, the author covers the adoption and user success, there are several success stories of great success with Clojure. It has become a respectable language and it is being used in hundreds of companies across the world. Not only Clojure itself, but also bi-products such as ClojureScript, ClojureCLR and Datomic.

Brief Overview of Clojure

Clojure is a dialect of Lisp, it is a functional language suitable for professional use. The objective of Clojure is to be a programming language that is as acceptable as popular languages as C# and Java, but functional and with a simpler programming model [1]. It has features like pure functions, immutable data structures, concurrency-safe state management and more.

As Clojure is a Lisp, it has benefits like code is data: "*a Lisp program is defined in terms of the interpretation of its core data structures, rather than by a syntax over characters*" [1], like other languages. It also has a small core language, with syntax familiar to Lisp.

All of Clojure's data structures are persistent and immutable, since it is normally slow to make data structures this way, the author used hash array mapped tries to make them performant and usable. This is a path copying strategy that made fast data structures possible. Data structures also

have meta-data, that make it possible to add context of information to each data structure made, without impacting equality.

Clojure is hosted, which means that it runs on the JVM. With JVM you can interact with Java and Java libraries, which was a big factor in Clojure's adoption, as there are loads of libraries written in Java. Also debugging tools that work with Java work with Clojure. This means that Clojure had all the benefits of JVM from the start, such as running on many different machines and not having to write its own runtime model.

> What I liked most about the language, is the fact that it is hosted, and that it is possible to use JVM libraries. This makes the language a lot more appealing for many people and also for me.
> —Daniel

State management in Clojure is meant to be simple, Hickey did not want to use complex locks and mutex approaches, but rather something that embraced and worked efficiently with persistent immutable data. The idea was to have a variety of references that were concurrency-safe [1], so user locking was not needed, with update semantics in synchronization and coordination. So the author designed and built an STM around MVC.

There is support for polymorphism in Clojure, this is done by protocols, which is named sets of polymorphic functions: `defprotocol` defines the signature and documentation of the functions, without implementation; with `defrecord` programmers can define new maps that include the necessary internal type tagging to support protocols.

The host: An Overview

"*Libraries are power*" [1]: they are a very important part of programming languages, it complements the core language with a lot of extra features, and is the most important reason for Clojure being hosted.

JVM was a good fit for Clojure as the runtime needed to be garbage collected and have high performance mechanism like optimization for runtime polymorphic dispatch [1]. The author started out by making Clojure compatible with CLR as well, but this way he had to implement the same features twice, instead of making twice the features. Therefore, he had to choose only one of them. He ended up with JVM instead of CLR because of the open source library ecosystem of Java and the more dynamic nature of the JVM.

To maximize interopability and performance, Clojure's types are alike host types, Clojure strings are `java.lang.Strings`, `nil` is Java's `null`, Clojure vectors are Java lists, and so on. This means that Clojure can use `java.collection` methods with its own array types in Clojure, as they are the same type in the host.

```
;; Java method requiring Java List,
;; passing Clojure vector
=> (java.util.Collections/binarySearch
     [2 4 7 9 43 76] 7)
2

=> (class [2 4 7 9 43 76])
clojure.lang.PersistentVector
```

The biggest host construct that works directly with Clojure is exceptions. The host implements error handling with try-throw-catch style exceptions with stack unwinding. It is optimized, has tooling and security support, which makes it great for Clojure.

Hickey's objective was to make Clojure portable, so that programmers could use Clojure where it suited each programmer. Therefore, it was important to recognize the portability of the host itself. JVM has been extensively ported to different operating systems and architectures [1].

In addition to the benefits for end users of the language, it also had a lot of implementation benefits. Not introducing a new runtime model helped both. Implementing a new runtime model is a lot of work; in addition, programmers need intuition about the languages runtime, memory and how garbage collection works to be effective in a language. As JVM is already a familiar runtime for many people, it helps with adoption.

Related Work on Clojure

There is more focus on functional programming languages in the last two HOPL conferences, this is because there many new functional languages in the recent years, and their popularity has grown. The papers that are most connected to this paper are about Lisp at both HOPL I and HOPL II, as Clojure is a newer dialect of Lisp.

A good resource to learn more about the topics of this paper, is by watching Rich Hickey's talks. He has some interesting and popular talks, where he talks about e.g. concurrency and state. One of his most popular talks is one from JVM Languages Summit in 2009 called *Are we there yet?* [4], where he talks about object oriented languages, functional programming, types, state and more.

Another important talk is *Simple Made Easy* from the Strange Loop Conference in 2011 [3], which is a talk that still attracts people to Clojure today. This is a talk about simplicity and the benefits of making programming languages and software simple.

There are many other great talks from Hickey, and they can be found on YouTube. He has a YouTube channel called ClojureTV that have many videos about Clojure [5], including Clojure talks.

Clojure has an official website `clojure.org`, that consists of tutorials, API reference, guides and news. It is a great place to start to learn about Clojure and getting started.

Tool Support for Clojure

Personally I have not used Clojure myself, but the community have recommended some editors and dependency managers [2]. My favorite editors are IntelliJ, Vim and Visual Studio Code, and there are Clojure tools for all of these and many other editors. There is also an official CLI tool made by the Clojure core team called `clj`, which is a tool for managing dependencies, running a REPL, and running Clojure programs. In addition, tools made by the community work as well.

Personal Experience

I liked the evolution part of the paper a lot, it was interesting to see how the language evolved over the years. From the initial design when there were few users to the last couple of years where there are hundreds of companies using it, Rich Hickey is still the one that does most of the implementation work.

It was also interesting to see the extra products that were made because of the language, like Datomic and ClojureScript. What I liked most about the language, is the fact that it is hosted, and that it is possible to use JVM libraries. This makes the language a lot more appealing for many people and also for me.

Survey results showed that one that one of Clojure's weaknesses was "poor error messages and incomprehensible stack traces" [1]. I wonder if the author would do this differently today: while the approach he did it was using the host implementation of exceptions, maybe it would be better to have implemented a better custom system for the language? Another question I have, is if he sees the language growing in the next few years? I had never heard of this language before this paper, and it feels like a language many more programmers could adopt to.

References

[1] Rich Hickey, *A History of Clojure*, Proc. ACM Program. Lang. 4, HOPL, Article 71 (June 2020)

[2] Community, *Clojure Tools*, available at: `https://clojure.org/community/tools`.

[3] Rich Hickey, *Strange Loop Conference*, available at: `https://www.infoq.com/presentations/Simple-Made-Easy/`.

[4] Rich Hickey, *Are We There Yet?*, available at: `https://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey/`.

[5] Rich Hickey, *YouTube channel ClojureTV*, available at: `https://www.youtube.com/channel/UCaLlzGqiPE2QRj6sSOawJRg`.

Essay on History of Clojure

Andreas Garvik
University of Bergen

This essay summarizes the paper [1] by Rick Hickey presented at HOPL Volume 4. The paper provides insight about the motivation behind the development of Clojure and discusses various design decisions and language constructs. It also covers the evolution of the language, before the initial release and later versions, adoption and community.

Brief Overview of the HOPL Paper

The paper starts with Rick Hickey recounting his background and movitation.

I chose this paper for multiple reasons. During the fall 2020, I got introduced to functional programming with Haskell and I found this programming style quite interesting. I had previous experience with lambda functions in Java and map/filter/reduce in JavaScript, but Haskell introduced me to a whole different way of thinking and reasoning about programs. It was strict and enforced a functional style. I started to look more into functional programming languages and Lisp and its dialects came to my attention. At first, I thought the syntax was very odd and unreadable,——

He gives an introduction about projects he has done in the past and his experience in programming.

He characterizes his work as "information systems" programming, making systems that do something with information about the world, using mainly C++/Java/C#. After spending time with Common Lisp, and experiencing a revelation, he attempts bridging Common Lisp to the JVM/CLR and ultimately decides to make his own language hosted on the JVM/CLR.

Different characteristics and features of the language in Clojure's initial release is presented and rationalized. The paper lays out its evolution from versions 1.0 through 1.10, where features and concepts were added or revisited, in variable amount and significance (notably, polymorphism in version 1.2 and "transducers" in version 1.7). Clojure evolved with attention to retention of code, as Rick Hickey wanted Clojure to be a stable language to be used professionally with minimal deprecations and breaking changes.

In retrospective, Rich Hickey mentions that the problem most new Clojure developers migrating from Java, JavaScript and other object-oriented languages faced, was not as maybe anticipated "dealing with all those parentheses", but rather learning functional programming style in general. Neither did he anticipate, after releasing Clojure, all the invitations to give talks nor the responsibility for stewarding the community.

The paper closes off with an overview of Clojure's adoption and success stories. Surveys show a large increase of developers using Clojure professionally over the years.

Walmart and Netflix are mentioned as large businesses that develop programs written in Clojure.

He concludes that, to the extent that Clojure was designed to allow programmers to write simpler, functional programs professionally, while developing systems for a wide variety of domains where Java and Javascript is prominent, it has certainly succeeded.

Brief Overview of Clojure

Clojure is a Lisp, meaning that it shares a lot of characteristics, concepts and ideas from the family of Lisp languages. Code is data, where the syntax you write in the program is essentially just the data structures itself.

For instance, a function call and its parameters is a list. The data read / print system is separated from compilation / evaluation, essentially the use of data structure literals instead of language instructions to construct the same data. Lisp languages can be very small, and Clojure has a short syntax and few keywords.

Clojure differs from Lisp in that it has a different basis then traditionally Lisp languages have,

——being mostly familiar with C-like syntax, but I wanted to learn more. Since Lisp or functional programming in general is not very widespread, I didn't get to spend time with it an use it in school or at work. I also came across one of Rick Hickey's talks about state and he argued that it was done wrong in object-oriented programming. He argued that encapsulation and mutability of object was a total mess especially in a concurrent and/or parallel context. These combined made me choose Clojure. I wanted to learn more about Lisp in general, Rick Hickey's thoughts and ideas and what Clojure had to offer. Recalling, I read about Clojure one time in a Stack Overflow Survey, where it was ranked as the highest payed programming language. I was intrigued.
—Andreas

`cons` cell with mutable `car` and `cdr`, instead it uses an abstraction *seq* as its basis, which is an abstraction over all collections. Clojure is also strict and the core objective of the language to ease programming.

This is achieved by a desire to support and ensure programs written in Clojure consist of immutable values and a set of pure functions to apply.

Clojure is hosted on the JVM, which means it can take advantage and comsume of all the well-developed libraries from Java. Clojure programs are Java libraries and bytecode when running on the JVM. It inherits its features such as garbage collection and runtime polymorphic dispatch, but differs greatly in state management. Clojure's idea was to have references being concurrency-safe, without requiring any locks. It archives this with a software transactional memory build [6] around multiversion concurrency control [7]. Data structure in Clojure is immutable, persistent and fast, thanks to hash array mapped tries [5].

Here is a code example written in Clojure: a factorial function.

```
(defn factorial [n]
  (loop [cnt n, acc 1]
    (if (zero? cnt)
      acc
      (recur (dec cnt) (* acc cnt)))))
```

Evaluation of programs: An Overview

Clojure as other languages in the Lisp family shares the same basic structure of programs as data. The basic structure is a data structure, or a series of data structures, no other syntax. The data structures *are* the code. This means it doesn't matter if it comes from written in a file or another program. When you want to turn your text representation of a data structure on file you invoke something that's called the *Reader*. This Reader turns the text into the respective data structures. The data structures are in turn handed to the compiler. This makes it very easy for a program to provide the data structures for a program to the compiler. One great benefit of this that a programmer could change a function definition in a running program without down time or loss of program state. In addition, since the compiler is just dealing with data scructures, the possibility to extend the evaluation capabilities of the compiler with *macros* is enabled. Macros give the programmer syntactic extensibility, where if you want something in the language syntax, all you have to do is to define a macro and when the compiler encounters it, it will look for your definition. This is in stark contrast to other languages that do not support macros—one might have to wait years before a desired language feature is added to the language. Actually, a lot of things you normally find "built-in" to other languages are defined as macros in Clojure.

Related Work on Clojure

Before starting the work on Clojure, Rich Hickey was wishing for a better alternative to imperative programming on .NET and briefly mentioning F# and how he found it insufficiently expressive. He initially targeted both JVM and CLR, but decided quite early to just work on the JVM to accomplish twice as much rather then do everything twice. Later David Miller started a port of Clojure back to the CLR, porting Java code to C#, named ClojureCLR [3]. Other related work is ClojureScript [4] which is a compiler that targets JavaScript. A paper on Lisp and its evolution was presented at the second HOPL Conference.

Tool Support for Clojure

Clojure has CLI with REPL support which make it easy to try out, interact and play with. It also has a VS Code plug in.

I think the things that fascinated me about Lisp was its concise and minimal syntax and how one essentially just wrote data structure literals, and had a choice whether to evaluate the code right away or store the code as a data structure to be used in the code.
—Andreas

Apart from the tool I have tried out, the paper states that Clojure had sophisticated tooling from the start. Because it was hosted on the JVM, it got a lot for free. Rick Hickey implemented just after the release, in 2007, emission of the needed debug information, which made all the Java breakpoint/set debugging and profiling software tools work on Clojure programs. This was huge for such a brand-new programming language.

Personal Experience

I enjoyed the most learning about Lisp concepts and ideas. Clojure inherits a lot of the same characteristics which I found very interesting.

I think the things that fascinated me about Lisp was its concise and minimal syntax and how one essentially just wrote data structure literals, and had a choice whether to evaluate the code right away or store the code as a data structure to be used in the code. This opens up a suite of options and flexibility where you could easily create a program to write a program or with macros even tell the compiler how a part of the code should be evaluated.

I would like to ask Rick Hickey why he might think that Lisp family languages, in particular Clojure, or functional programming style is not more common or the standard way of programming. I would also like to ask if we had more time developing, if he would spend it getting rid of the JVM underneath and create a compiler or if we would create more libraries and abstractions on top. I find it interesting to question if one today should not spend more time reinventing the wheel by creating compilers or if there still is many more aspects to research and enhance and we have just scratched the surface. I wonder if abstractions an reuse of existing compilers and libraries leads to more progress in language design. This was the route Rick Hickey chose with Clojure at least.

References

[1] Rick Hickey *A history of Clojure*, Proc. ACM Program. Lang., Vol. 4, No. HOPL, Article 71.

[2] Guy L. Steele Jr, Richard P. Gabriel *The evolution of Lisp*, Proc. ACM Program. Lang., Vol. 2, No. HOPL, Chapter VI.

[3] *ClojureCLR*, available at: `https://clojure.org/about/clojureclr`.

[4] *ClojureScript*, available at: `https://clojurescript.org/`.

[5] *HAMT*, available at: `https://en.wikipedia.org/wiki/Hash_array_mapped_trie`.

[6] *Wikipedia page on "Software transactional memory"*, available at: `https://en.wikipedia.org/wiki/Software_transactional_memory`.

[7] *MVCC*, available at: `https://en.wikipedia.org/wiki/Multiversion_concurrency_control`.

D

Link to the HOPL IV paper:
https://dl.acm.org/doi/10.1145/3386323

Link to the student's presentation:
https://git.app.uib.no/uib-hopl-iv/slides/-/blob/master/D.pdf

Essay on History of the D Programming Language

Marius Kleppe Larnøy
University of Bergen

This essay summarizes the paper “Origins of The D Programming Language” [1] by Walter Bright, Andrei Alexandrescu and Michael Parker, presented at HOPL IV. The paper tells the story of Walter Bright’s work in the compiler industry from the late 70s up to the inception of the D programming language in 1999. It then covers the development of the language leading up to its 1.00 release in 2007.

Brief Overview of the HOPL Paper

The Origins of the D Programming Language covers Walter Bright’s two decade long journey in the compiler industry, as well as the first seven years of development on the D language.

The reason I chose this paper is because I was fascinated by the bold decision to attempt to create a successor to two of the most widely used programming languages in history, namely C and C++. Drawing users from two mature languages with codebases decades old is no easy task, and I was interested in learning about what features the D language could offer that would make a reasonably conservative user base of systems developers jump ship.
—Marius

The paper uses its first few pages to lay down the background for Walter Bright’s decision to develop his own language. How he started out as a mechanical engineer at Boeing with game development as a hobby, who wrote his own C compiler to improve performance of his games, to working full time developing and maintaining industry grade compilers for C, C++ and later Java and JavaScript[1].

The main sections of the paper details the first few years of development on the Digital Mars D compiler, Digital Mars being the company Walter founded for the project. How it started out as a one man project and how it grew into a sizable group of people with community driven projects. It tells the story of how Walter Bright as the sole developer interacted with his growing (and vocal) community, and how his vision of the D specification not always matched what his target demographic wanted out of a systems programming language.

The Digital Mars D compiler version 1.00 was released on January 2nd 2007, and with that the paper concludes: “Any tale of *origins* of the D programming language must reasonably end with the release of version 1.00” [1].

Brief Overview of D

D is a systems and application programming language. It has support for multiple programming paradigms, such as procedural, object-oriented and metaprogramming. In its current form (D2) D has added support for functional programming, as well as improved metaprogramming as part of the core language[1].

D follows C’s "Algol-like" syntax closely, with the intention of making it easy for C and C++ programmers to make the transition over to D. To further cater to C-programmers D is interface compatible with C, one of the core design goals of D was that *"a syntactical construction should have the same semantics in D as in C, such as the integer promotion rules, or fail to compile"*[1].

Perhaps the feature that separates D from C and C++ the most is its inclusion of a garbage collector. A controversial addition to a systems programming language aimed at C developers, and heavily debated on the Dlang forums[1]. Until the addition of the `@nogc` function attribute in D2, automatic memory management was a fully embedded part of D.

D is a statically typed language[12], such that every expression has a type and typing errors are resolved at compile time. The language has its own list of basic built in data types, such as numeric types. In addition to the basic types, there are also the derived data types, such as point-

ers, functions and arrays. D also supports user-defined types such as enums, classes, structs and interfaces (full overview can be found in the language specification at `dlang.org`).

Template mixins: An Overview

Template mixins are an extension to D's templates. According to the paper, mixins were first suggested to solve issues surrounding using macros in C++ to circumvent issues with the C preprocessor in the codebase of the first Unreal game[1]. Semantically, mixins were designed to avoid the problems of the preprocessor, while allowing exposure of metaclass information, i.e., passing around a `classref*` which exposes static functions and constructors [1]. When the `mixin` keyword is prefixed to a template instantiation, the body of the template is inserted into that location and takes on the scope in which it is initialized. This is the opposite of what happens when you initialize a template normally, where the body takes on the scope in which it was implemented [1].

> At first glance, the motivation behind developing a successor to two languages such as C and C++ can be a bit hard to grasp. The paper does a good job in inviting the reader into the mindset of Walter Bright, and getting to know his motivations behind this decision.
> —Marius

According to the specification, "*a `TemplateMixin` takes an arbitrary set of declarations from the body of a `TemplateDeclaration` and inserts them into the current context*" [13].

A mixin template comes wrapped in its own scope, with internal symbols aliased to the external scope[1]. This allows for multiple mixins of the same template in the same module, or mixins with several templates with the same internal symbols or possibly conflicting symbol names already in scope.

The example below from the paper highlights the use of two mixins of the same template, with identifiers `v1` and `v2` to disambiguate between them.

```
template addVars(T) {
    T x;
    T y;
    T z;
}

mixin addVars!float v1;
mixin addVars!double v2;

void main() {
    v1.x = 10; v2.x = 20;
}
```

Related Work on D

Related HOPL papers. Out of all the papers presented at HOPL IV, the paper on D is mainly connected to Bjarne Stroustrup's paper *Thriving in a crowded and changing world: C++ 2006-2020* [2]. As D was intended as an improvement upon C and C++ with similar syntax to draw users already fluent in those languages, D will inherently be connected to the history and development of C++.

There is one other paper from HOPL IV that has some connection to D, and that is *JavaScript: the first 20 years* [3]. The garbage collector originally employed in the D runtime library was a repurposed garbage collector that Walter Bright had written for the Chromium JavaScript implementation in 2000 [1].

Jumping back in time to the previous HOPL conferences, there are some papers in particular that stand out in regards to relevance for the D language. For HOPL-III, Bjarne Stroustrup wrote a paper on the history of C++ from 1991-2006, *Evolving a language in and for the real world: C++ 1991-2006* [4]. This period of time is particularly connected to the development of D as it was during this time D was initially conceived, and the D language's feature set was strongly influenced by the versions of C++ that emerged during this era. Out of the HOPL II papers, there are three that can be connected to D in some way. HOPL-II was the first installment that had C++ on its list of papers. In *A history of C++: 1978-1991*[5], Bjarne Stroustrup goes into detail about the first 13 years of C++ existence. This initial era of C++ was when Walter Bright began working on compilers for C

and C++, most famously the Zortech C++ compiler [1], and it played a crucial role in his later work on the D compiler. Dennis Richie also presented a paper on C at the conference: *The development of the C language* [6].

The last papers that have some relevance to the paper on D are the papers on ALGOL 60 [9, 8] and ALGOL 68 [7]. This is more of an "honorable mention" as the connection is transitive. The term "Algol-like syntax" spread from ALGOL to C, to C++, and finally to D, and so beyond that direct impact ALGOL had on the development of D is neglicable.

Related work and further reading. The full language specification for D, as well as the documentation for the standard runtime library are available at `dlang.org` [10][11].

Andrei Alexandrescu's book *The D Programming Language*[15] is recommended by the D Language Foundation as "the definitive book on D"[14]. One drawback of the book is that at the time of the release in 2010 the current iteration of D (D2) was not yet feature-complete [1], and thus it does not cover the complete specification of modern D.

A more recent book by Ali Çehreli published in 2016 [16] is also recommended by the D Language Foundation. It is a comprehensive introduction to the language aimed at both beginners and more experienced programmers. It also has the benefit of having a online version available which receives updates as the language evolves.

Tool Support for D

The tools I decided on using for D programming was VSCode with the `code-d` extension [17]. This extension requires you to have a D compiler installed beforehand, but offers a nice suite of helpful tools like syntax highlighting, auto-complete and linting. Compiling can be done from the VSCode editor or by running the compiler from a terminal window. I am also aware of that there exists a D-mode for Emacs and a D Plugin for IntelliJ IDEA, but I have not personally tested these out.

There is also a choice between different compilers to use. DMD is the official reference compiler, which features the latest version of D. GDC is a GCC based D compiler which offers lots of optimizations and support for a great variety of architectures. Lastly there is LDC, a LLVM-based D compiler, which also offers lots of optimizations. For working on this essay I decided to use DMD for the ease of use, but all compilers are viable based on your needs.

Personal Experience

At first glance, the motivation behind developing a successor to two languages such as C and C++ can be a bit hard to grasp. The paper does a good job in inviting the reader into the mindset of Walter Bright, and getting to know his motivations behind this decision. Once the paper reaches the beginning of D development in the late 90s, the daunting task seemed more like a natural step in the evolution of C-like languages.

True to its intentions, with some knowledge about C and C++ both reading and writing D code is in my opinion a close to seamless transition. It showcases a lot of well-thought-out—sometimes subtle—changes that can take some time getting used to, but overall an enjoyable experience.

Here are my questions to the authors of the paper:

- Part of the initial motivation behind D was that C++ was becoming exceedingly large. With D becoming a more mature language with support for multiple paradigms, how to avoid D running into the same issues?
- Where do you see D in 10-20 years?

References

[1] W. Bright, A. Alexandrescu, M. Parker, *Origins of the D Programming Language,* Proc. ACM Program. Lang., Vol. 4, No. HOPL, Article 73.

[2] B. Stroustrup, *Thriving in a crowded and changing world: C++ 2006-2020,* Proc. ACM Program. Lang., Vol. 4, No. HOPL, Article 70.

[3] A. Wirfs-Brock, B. Eich, *JavaScript: the first 20 years*, Proc. ACM Program. Lang., Vol. 4, No. HOPL, Article 77.

[4] B. Stroustrup, *Evolving a language in and for the real world: C++ 1991-2006*, HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages, Pages 4-1–4-59.

[5] B. Stroustrup, *A history of C++: 1979–1991*, HOPL-II: The second ACM SIGPLAN conference on History of programming languages, Pages 271–297.

[6] D. Richie, *The development of the C language*, HOPL-II: The second ACM SIGPLAN conference on History of programming languages, Pages 201-208.

[7] C. H. Lindsey, *A History of ALGOL 68*, HOPL-II: The second ACM SIGPLAN conference on History of programming languages, Pages 97-132.

[8] A. J. Perlis, *The American side of the development of Algol*, ACM SIGPLAN Not. 13, 8, pp 3-14.

[9] P. Naur, *The European side of the last phase of the development of ALGOL 60*, ACM SIGPLAN Not. 13, 8, pp 15-44.

[10] D Lang Foundation, *The D Language specification*, available at: `https://dlang.org/spec/spec.html`.

[11] D Lang Foundation, *Phobos Runtime Library*, available at: `https://dlang.org/phobos/index.html`.

[12] D Lang Foundation, *Types*, available at: `https://dlang.org/spec/type.html`.

[13] D Lang Foundation, *Template Mixins*, available at: `https://dlang.org/spec/template-mixin.html`.

[14] D Lang Foundation, *Books*, available at: `https://wiki.dlang.org/Books`.

[15] A. Alexandrescu, *The D Programming Language*, Addison-Wesley Professional, 2010. ISBN-13: 978-0321635365.

[16] A. Çehreli, *Programming in D*, Ali Çehreli, 2016. ISBN-13: 978-0692599433. Available at: `http://ddili.org/ders/d.en/index.html`.

[17] Webfreak, *D Programming Language(code-d)*, `https://github.com/Pure-D/code-d/wiki`.

F#

Link to the HOPL IV paper:
https://dl.acm.org/doi/10.1145/3386325

Link to the student's presentation:
https://git.app.uib.no/uib-hopl-iv/slides/-/blob/master/F-sharp.pdf

Essay on The Early History of F#

Kenneth Fossen
University of Bergen

This essay summarizes the paper "The Early History of F#" [3] by Don Syme [4] presented at the HOPL IV. The paper discusses the historical summary for how the F# language was created, and how the early days functional programming (FP), and Don Syme's personal experience with FP influenced F#'s design and implementation.

Brief Overview of the HOPL Paper

I choose this paper for a few reasons. I've been experiencing functional programming through other courses at University in Bergen [5, 7], and have been enjoying the new way of thinking and the Hindley-Milner type-inference magic [6]. I've also been curious to transfer this knowledge from my courses where I've been using functional programming, into my part-time job, where we mainly work in the imperative language C# and .NET environment. This HOPL paper would give me a good introduction to the language, enable learn it by using it,——

The HOPL paper "The early history of F#" [3] is journey of strongly typed functional programming languages in the early 80's to the creation of F# in late 2001 and inclusion of .NET Generics in 2005.

In this period, it covers briefly, what the basis of functional programming is, and how this inspired the paper *Why no one uses functional programming* by Philip Wadler [8]. As Syme mentions, "*the paper was central to my understanding of the programming language landscape*" [3, p. 9] when he entered Microsoft Research in 1998. The rise of object-oriented programming (OO) that lead to Wadler and Odersky with a mission to *integrate specific technical features associated with strongly typed functional languages into "mainstream" OO languages* [3, p. 9] then formed the creation of Pizza programming language. This later led to generics being incorporated in Java, and later influenced Syme's work on .NET Generics [9]. This was an important inclusion in .NET as it shaped parts of F#.

In 2003, 0.5 version of F# was released, but it didn't get much attention before the version 1.0 in 2005. From 2001 to 1.0, there were added a many features and the language matured. Many of the early inclusions in F# 1.0 are today the most beloved features of the language. The Pipeline operator is one of the popular features that was added early in the language. Later Units-of-Measure and Type Providers were added, to mention some.

In the period 2005 → 2020, the paper covers the evolution of F#, and how the industry changed and how Microsoft had to adapt to this. F# got a new dawn with becoming open source, and cloud computing was on the rise where it played a central part.

Syme also takes us through some language mistakes [3, p. 51] that in retrospective should not have been added to the language. Here he mentions, for example, the Back piping operator and he discusses his early F# decision about not including, an oft-requested feature in F# at present time, type classes that originates from Haskell.

Brief Overview of F#

F# is a strongly typed functional programming language that has been heavily inspired by OCaml [16]. It started out as a project getting OCaml to run on .NET, but there was resistance in the Microsoft Research community: "*So, why do we really need a .NET port of OCaml? OCaml is working fine on Windows, and on many other OS*" [3, p. 16]. But the author decided to write F# from scratch and port the core of OCaml to target .NET. It is strongly typed through their own implementation of Hindley-Milner that works together with .NET Generics. It comes as a part of .NET SDK since 2010, and has a rich REPL through `dotnet fsi` that supports F# Scripting. Other tools are mentioned in section *Tool Support for*

F# of this essay.

Popular features of F# is Units-of-Measure [17], Active Patterns [18], Type Providers and Quotations [19].

Units-of-Measure is type-safety for metrics. If two measures are incompatible, you can't sum them, for example. This also removes any doubt about what metric this number is representing in your code. A popular example for this is NASA's Mars Climate Orbiter that failed due to a confusion between newtons and pound force [20].

Active Patterns allow for partial, complete and multi-case patterns, and make it easy to create new patterns and use them. Syme's example is parsing `Int` and `Bool`: `let (|Int|_) str = ...` [3, p. 29]. Quotation is used for metaprogramming [22] in F# with Abstract-Syntax Tree's (AST) [21] and can be used to generate code or work with language creation in F#.

Type providers is a feature that gives strongly typed data sources, e.g., for working with JSON or Databases.

——and get to understand how functional and imperative programming can interoperate in the same code base in the .NET SDK [15]. If I were to get to use functional programming in my current workplace that does .NET, this interop between C#, F# and .NET seems like the likely way of letting this happen, and hopefully we could harvest the best of both worlds in the software development team.
—Kenneth

NuGet [23] is the package manager for .NET projects for C# and F#, and since F# is delivered through .NET SDK, you can add any new NuGet packet and start using these libraries in your code; it doesn't matter if it is written in C# or F#. This goes also the other way.

I look forward to try to incorporate this language into my work, and here I believe Units-of-Measure is a big thing we could benefit from.
—Kenneth

Pipe operator: An Overview

It became a favorite instantly as it made me able to chain together several commands, letting it pass on the parameters from one to the next, and at the same time increased the readability. The operator is is just a triangle symbol `|>`, that has the definition, where **f** is a function and **x** is a parameter.

```
let (▷) x f = f x
```

From my experience with Haskell, I've come accustomed to reading from right to left in these cases, but this swap, made it intuitive and easy to read from left to right and following the logic. It's also extendable to `||>` (two) and `|||>` (tree parameters) to pass on to the next. Here is a an example from the paper comparing F# style chaining with `|>` with Haskell style using F#.

```
[1..10]
   ▷ List.map(fun x → x*x)
   ▷ List.filter(fun x → x % 2 = 0)

-- instead of
List.filter (fun x → x % 2 = 0)
   (List.map (fun x → x * x) [1..10])
```

The pipe operator also comes with a backward pipe operators, `<|, <|| and <|||`. It is not recommended to use the backward operator, since it doesn't add readability for the user. Here comes an example we still use the backward pipe operator: in this case we cannot chain more backward operators, due to the nature of left-to-right associating for the operator that is inherited from OCaml.

```
let antagonist a str = str
        + " " + a + " your "

let joinMe str1 str2 = str1 + str2
let protagonist = "Luke,"

let plot_twist =
   protagonist
    ▷ antagonist "I'm"
    ▷ joinMe ◁ "father"
(* See a TIE fighter, so cool *)

printfn "%s" plot_twist
(* Luke, I'm your father *)
```

Related Work on F#

From the first HOPL in 1978, to the HOPL IV in 2020, there has been a few contributions that link to this paper. At HOPL I, the *History of LISP* [27] is

of interest as F# builds in these ideas that was incorporate into LISP. At the HOPL II, there was also a new contribution with regards to LISP, called *The evolution of LISP* [26]. After this we can mention *A history of Haskell: being lazy with class* [11] that has influenced F# design choices and is a very popular functional programming language in academia. Among HOPL IV papers, we can mention *The History of Standard ML* [25] that was a direct competitor to F# running on .NET with SML.NET port but never became anything of, and lastly *A History of Clojure* [24] that is a functional dialect of LISP interacting with Java that is sharing some of the same ideas of F#.
Related work that paved the road for F# to integrate with .NET is the authors work on .NET Generics. [9]. Besides this, `fsharp.org` is a good starting place for more information about the language itself, community and many other resources. One part of this website is the *Academic Papers* page that contains a list of academic publications that could be of interest for the reader to study up on. Furthermore, you can follow their work on the GitHub account `github.com/fsharp`.

Tool Support for F#

The great thing about F# is that is runs everywhere the .NET SDK [15] is installed. This means you can use `dotnet` too that comes with the .NET SDK to create projects. But this is not the only way to use F#. It is easy to follow F# guides on Microsoft's web pages [12] with F# Scripting (*.fsx*) and use either VSCode [13] or your favorite editor. VSCode supports the community created Ionide extention [10] that gives good language support for F#. The REPL that follows .NET SDK, I found not useful and I would stay away from it, due to its cumbersome addition of double ; ; for commands and other quirks from the OCaml notation. I mainly used VSCode with Ionide, but I will also highly recommend JetBrains Rider [14] with .NET development.

Personal Experience

In the paper, I really enjoyed how Don Syme took us through the history from the early 70's of FP to the inspiration for how and why F# was created. It was exciting to see what inspired and what other languages influenced Syme before he created this language. My limited use of the language has shown me a positive side of the F# language, though I struggle some with things, when moving from Haskell world. I look forward to try to incorporate this language into my work, and here I believe Units-of-Measure is a big thing we could benefit from.
One of the things I struggle with in F# was getting going, F# Scripting is one thing, modules and projects are another. Why can't we just use them straight up as we can in Haskell? Since C# puts a lot of restrictions on how they did things in F#, I'm curious to ask, how the author would have seen F#'s if it wasn't so tied to C# and their development? Also, the author mentions that `Span` helped iron out some minor problems since F# 2.0. Are there other aspects of C# and/or .NET that is hindering F# move in the direction that would make F# more adaptable and usable?

References

[1] S. Ratnakumar, *A brief F# exploration*. Available at: `https://notes.srid.ca/fsharp-exploration`.

[2] *Bolero: F# in WebAssembly*. Available at: `https://fsbolero.io/`.

[3] D. Syme, *The early history of F#*, Proceedings of ACM on programming languages: 4. 2020.

[4] *Wikipedia page on "Don Syme"*. Available at: `https://en.wikipedia.org/w/index.php?title=Don_Syme&oldid=1003588951`.

[5] *INF222: Programmeringsspråk / Programming Languages*, University of Bergen. Link: `https://www.uib.no/en/course/INF222`.

[6] *Wikipedia page on "Hindley–Milner type system"*. Available at: `https://en.wikipedia.org/w/index.php?title=Hindley%E2%80%93Milner_type_system&oldid=1016004296`.

[7] *INF122: Funksjonell programmering / Functional Programming*, University of Bergen. Link: `https://www.uib.no/en/course/INF122`.

[8] P. Wadler, *Why no one uses functional languages*, SIGPLAN Notices 33:8. 23–27. 1998.

[9] D. Syme, *ILX: Extending the .NET Common IL for Functional Language Interoperability*. Electronic Notes in Theoretical Computer Science 59:1. 53–72. 2001.

[10] *Ionide*. Link: `https://ionide.io/`.

[11] P. Hudak, J. Hughes, S. Peyton Jones, P. Wadler, *A history of Haskell: being lazy with class*. ACM SIGPLAN HOPL III. 2007.

[12] P. Carter, *Get started with F#*. Available at: `https://docs.microsoft.com/en-us/dotnet/fsharp/get-started/`.

[13] *Visual Studio Code - Code Editing. Redefined*. Available at: `https://code.visualstudio.com/`.

[14] *Rider: The Cross-Platform .NET IDE from JetBrains*. Available at: `https://www.jetbrains.com/rider/`.

[15] T. Dykstra, *.NET SDK overview*. Available at: `https://docs.microsoft.com/en-us/dotnet/core/sdk`.

[16] *OCaml – OCaml*. Link: `https://ocaml.org/`.

[17] P. Carter, *Units of Measure - F#*. Available at: `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/units-of-measure`.

[18] P. Carter, *Active Patterns - F#*. Available at: `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/active-patterns`.

[19] P. Carter, *Code Quotations - F#*. Available at: `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/code-quotations`.

[20] *Wikipedia page on "Mars Climate Orbiter"*. Available at: `https://en.wikipedia.org/w/index.php?title=Mars_Climate_Orbiter&oldid=1016599571`.

[21] *Wikipedia page on "Abstract syntax tree"*. Available at: `https://en.wikipedia.org/w/index.php?title=Abstract_syntax_tree&oldid=1016693387`.

[22] *Wikipedia page on "Metaprogramming"*. Available at: `https://en.wikipedia.org/w/index.php?title=Metaprogramming&oldid=1017521092`.

[23] J. Douglas, *What is NuGet and what does it do?*. Available at: `https://docs.microsoft.com/en-us/nuget/what-is-nuget`.

[24] R. Hickey, *A history of Clojure*. Proceedings of ACM on programming languages: 4. 2020.

[25] D. MacQueen, R. Harper, J. Reppy, *The history of Standard ML*. Proceedings of ACM on programming languages: 4. 2020.

[26] G. L. Steele, R. P. Gabriel, *The evolution of Lisp*. SIGPLAN Notices: 28. 231–270. 1993.

[27] J. McCarthy, *History of LISP*. SIGPLAN Notices: 13. 217–223. 1978.

GROOVY

Link to the HOPL IV paper:
https://dl.acm.org/doi/10.1145/3386326

Link to the students' presentation:
https://git.app.uib.no/uib-hopl-iv/slides/-/blob/master/Groovy.pdf

Essay on History of Groovy

Jenny Strømmen
University of Bergen

This essay summarizes the paper "A history of the Groovy programming language" [1] by Paul King presented at HOPL IV. The paper discusses the history of how Groovy was made, its important features and how the language have evolved.

Brief Overview of the HOPL Paper

The first section of the paper covers the language vision and who it is meant for.

Section 2 describes how it got to the first version, how it got inspired by other languages and the similarities and differences between Groovy and Java.

Furthermore, it looks at some interesting aspects of the language at section 3, describing different features like operator overloading, command chains, how you can customise the compiler and so forth. In section 4 we get to know how Groovy was hosted throughout the years, how the repository technology was changed and how it affected code contribution. Here also sponsors and contributors are mentioned.

Section 5 talks about Groovy's main changes and changes in the library during the development. It starts at Groovy 1.5 and ends at Groovy 4.0. At the end of the section the author mentions some lessons learned, and what could be done better. In this section, the author discusses the new features in the different versions, like abstract syntax tree transformation (ASTT) and command chains.

Further on, in Section 6, it is described how the GDK has evolved, and in the following section the paper goes more into detail about the ASTTs. It describes the motivation of the implementation, what effect it has on the language and how it strengthens the vision of having an extendable language.

In section 8, the author discusses Java compatibility, how it has evolved and lessons learned.

In the next section 9, the paper goes into detail about the static vs. dynamic nature of Groovy, and discusses going from a dynamically typed language to a statically typed language. In section 10, it points out the importance of testing, and in section 11, we get to know how Groovy is doing today. The last section discusses how Groovy influenced other languages.

In a previous job I used Groovy to write tests in, and specifically I used Groovy with Spock. This allows you to write tests with method names as a string, and create easier behaviour tests with a built in "given, when, then". [2] This makes the tests much clearer and easier to read. Though I did not use Groovy that much, it was easier an less rigid to create objects to test because of the dynamic typing. Having developed a lot in Java and felt the frustration that some things has very complicated code in comparison to, for example, Python, I wanted to learn more about the Groovy language.
—Jenny

Brief Overview of Groovy

In August 2003, James Strachan made an announcement about what would be the start of the Groovy language. Here he mentions that the Java platform should have its own dynamic language, and that it will be inspired by Python and Jython. Furthermore, he says he wants to use elements from Ruby, and that it should be a concise language. However he is very unclear about what the actual result will be, but he calls it a "... Groovy new language ..." [1, p. 9].

In section 1 of the paper it is stated that Groovy is supposed to be a better Java and should be easy for Java developers to learn. It should reduce some of the complexity in Java, and make it compatible with Java. Groovy was meant to be a complementary language, and not a replacement for Java. You should be able to write both Java and Groovy in the same project, and choose Groovy when you do not, for example, want static typing. It is also meant to be better for script

writers, and there is an example of how Groovy can write the script much more concise then in Java [1, Sect.1.1]. It is also argued that Groovy is a good choice for data scientists, and creating domain specific languages.
The language is influenced by Python as mentioned, and we see that strings, map and lists are taken from Python [1, Sect.2.2]. Named parameters and collection methods are taken from Smalltalk, and Ruby influenced the metaprogramming.

In my opinion, the combination of removal of boilerplate code and the easy creation of DSLs improves code readability, especially for people who have little coding experience.
—Jenny

To begin with, Groovy had a dynamic nature, but as it developed, there were requests that Groovy also had static typing. How and if this should be done is widely discussed in section 9 in the paper, but the conclusion is that you can use annotations (ASTTs) in your code to signify if you want static typing, with or without optimisation.
In section 3 we go into some of the features of the language, where, among others, command chains are described. Command chains let you write in a more natural language, and the expression `take 3 then 6` is parsed as `take(3).then(6)`. Another important feature is the ASTTs, and the paper goes into more depth about it in section 7. I will give a detailed overview of this feature in the next section of this essay.

Abtract Syntax Tree Transformations: An Overview

I chose to go into more depth about the ASTTs because this feature removes a lot of boilerplate code by creating an annotation to inject code into the compiler, hence I believe it increases code readability.
In [1, Sect. 7] you can read about these tranformations. Here they write that when Groovy is compiled, the source code is transformed to an AST, and it is here where the transformations happen. The main reason for the implementation of this feature is to make the language extendable: the idea is to make the programmer to be able to extend the language to their needs. Additionally, one wants to be able to add features without actually having to change the grammar of the language and thereby risk to make the language overly complex.
With AST transforms you can remove boilerplate code, for example getters and setters in a class. If you use `@Property` the compiler generates these methods for you automatically. The annotation that I think is most interesting is the `@Builder` annotation, that lets you take use of the Builder pattern without the boilerplate code that comes with it.
Later in the section, they represent macros for creating annotations, to make it easier to create them. Before macros, when you were to create an ASTT, you had to know how Groovy code was mapped into an AST. When using a macro however, you can customise annotations easier. Also, there is an AST matcher that lets you write a pattern match in Groovy code, matches on the AST and can replace the match with some other code.
At the end of the section, the paper highlights some problems related to the ASTTs. For example, if you have `@Trace` that logs every method in a class, together with `@ToString` that injects a `toString()` method for this class. Then the question is: does the `toString()` method get traced as well? The answer depends: if `@ToString` is before `@Trace` then `toString()` is not traced, and the other way around, though this is not a guaranteed behaviour.

Related Work on Groovy

As mentioned, Groovy got some inspiration from other languages. According to the paper this was Java, Python, Smalltalk, Ruby and some Lisp. Also, it affected other languages, mostly Kotlin. Smalltalk gave inspiration to Groovy about the collection method names and the named parameters. HOPL II features a paper on Smalltalk [3], where they discuss the early history of Smalltalk, but the features are not discussed there as far as

I can see. Also at the HOPL II conference, there was a paper about Lisp [5], but I could not find any more relevant articles from HOPL.
The Groovy documentation provides information about how to getting started using Groovy, information about the language and tools [6]. If you want to try to use the ASTTs I would look at the documentation found at [7]. If you want to have a look at the domain specific languages in Groovy, you should read [8].

Tool Support for Groovy

I have not tried using Groovy other than with IntelliJ, you can read more about support for Groovy at [9]. Integrating Groovy with an excisting Java 11 project in IntelliJ is described, for example, in [10]. If you are using Maven as a build tool you could look at [11]. I would strongly recommend to try to write tests in Groovy with Spock, you can find a tutorial [2].

Personal Experience

I liked best the features concerning ASTTs and command chains. Looking at the documentation of domain specific languages (DSL) [8], you also see that you can give properties to numbers. This lets you write, for example, `5.hours.from.now`, which I think really enhances the understanding of the code.
In my opinion, the combination of removal of boilerplate code and the easy creation of DSLs improves code readability, especially for people who have little coding experience. An example of this is given in [8, Sect.1], where they transform a complicated and hard to read processing of a string into a clean and concise piece of code. The `@Builder` ASTT is lowering the threshold to implement builders, which I also believe increases the readability of code.
If I could ask any questions to the author of the paper, I would ask him about the problem related to conflicting ASTTs, `@Trace` and `@ToString`. What would it take to make a guarantee that the annotations are injected in the way that they appear in the source code? Further on I would ask how they decided on the `a b c d` equals `a(b).c(d)` in command chains, and if they have discussed other possible patterns for this. At last, I would ask if adding the static features to the language in any way changed their relationship to Java, if they were now seen as a more competing language rather than just complementary to Java.

References

[1] P. King, *A History of the Groovy Programming Language*. ACM SIGPLAN HOPL IV. 2020.

[2] *Introduction to Testing with Spock and Groovy | Baeldung*. Link: `https://www.baeldung.com/groovy-spock`.

[3] A. C. Kay, *The Early History of Smalltalk*. ACM SIGPLAN HOPL III. 1996.

[4] G. L. Steele, R. P. Gabriel, *The Evolution of Lisp*, ACM SIGPLAN HOPL II. 1993.

[5] G. L. Steele, R. P. Gabriel, *The Evolution of Lisp*, SIGPLAN Notices 28:3. 231–270. 1993.

[6] *The Apache Groovy programming language - Documentation*. Link: `https://groovy-lang.org/documentation.html`.

[7] *The Apache Groovy programming language - Runtime and compile-time metaprogramming*. Link: `http://groovy-lang.org/metaprogramming.html#_available_ast_transformations`.

[8] *Domain-Specific Languages*. Link: `http://docs.groovy-lang.org/docs/latest/html/documentation/core-domain-specific-languages.html`.

[9] *Groovy | IntelliJ IDEA*. Link: `https://www.jetbrains.com/help/idea/2021.1/groovy.html`.

[10] *Getting started with Groovy and Java 11 project*. Link: `https://www.jetbrains.com/help/idea/getting-started-with-groovy-java-9-project.html`.

[11] *Integrating Groovy into Java Applications | Baeldung*. Link: `https://www.baeldung.com/groovy-java-applications`.

JAVASCRIPT

Link to the HOPL IV paper:
https://dl.acm.org/doi/10.1145/3386327

Link to the students' presentation:
https://git.app.uib.no/uib-hopl-iv/slides/-/blob/master/JavaScript.pdf

Essay on History of JavaScript

Kathryn Frid
University of Bergen

This essay summarizes the paper [1] by Allen Wirfs-Brock and Brendan Eich presented at the HOPL IV conference. The paper discusses the history and evolution of JavaScript over its first 20 years. It goes over its birth and initial standardization. The two attempts to evolve the language to ECMAScript 4, the process after both of these attempts fail, and how the language moved past these failed attempts.

Brief Overview of the HOPL Paper

The paper goes over the birth of JavaScript, its early evolution, the numerous attempts around the 2000s at evolving the language, and where the language eventually went ten years later.

I wanted to cover JavaScript as it is a language I already knew had an interesting story. It is also a language I find interesting because of how dynamic and free form it is. There is rarely a need to add an entirely new concept to JavaScript, as the concepts it posses are flexible enough to encode new ideas using old ones. I had already over the past year been writing more and more JavaScript when I chose this topic, so it felt like a good and exciting choice.
—Kathryn

It starts with why Netscape created JavaScript and some of its early evolutions. After this, a large part of the paper goes over the complicated process of creating ES4, which never succeeded. It goes into detail about what factions and companies wanted at the time and why. It then goes over many of the language changes that eventually landed in ES5 and ES6/ES2015. Included here are the thoughts that went into them, potential discussions over how the features would work, and problems uncovered as they were implementing the features.

The paper goes into extra detail about what ES4 was intended to be and why many TC39 members did or did not want it, how the different members like Mozilla, Microsoft, Adobe, IBM, Yahoo, Apple, and others viewed the proposal, what they liked, and what they did not like. After ES4 failed, it shows what work was done on ES3.1, which would later become ES5. From ES5, the paper also goes into extensive detail about the thoughts and internal changes that had to be done for many of the bigger changes, like classes, arrow functions and proxies.

Brief Overview of JavaScript

JavaScript is often called the language of the web. It is the language that runs on almost all websites we navigate to nowadays.

As a language, JavaScript is a scripting language focused on objects and the properties of these objects. Properties here refer to values bound to the object with a name or some other symbol. Functions in JavaScript are also first-class values, and developers can pass them around like all other values.

New and this. Objects can be made in JavaScript through object literals or the `new` operator. `new Object` gives developers a fresh object that can then be assigned properties as they want. One can also get more traditional objects some might be familiar with from a more object-oriented background through a constructor function. A constructor function is like any other function, but it is usually capitalized and assigns values to `this` in its body. Using this constructor function with `new` then gives a new object with the function called on the object. For example, `new Foo(4, 3)` creates a new object using a function `Foo` and passing 4 and 3 to `Foo` as arguments.

Functions and this. Functions can also be set as properties on an object. When they are properties and an object like this, and are called as part of accessing the object (for example, `foo.bar()`), they have access to the object they

are a part of through `this`. Through this mechanism, JavaScript can encode methods on an object.

JavaScript has always been more than just a thing browsers use, even though that is where we find it most often.
—Kathryn

Prototypes. JavaScript also has a way to share properties between many instances from the same constructor function using the prototype property on the constructor function. All instances from a constructor function share all properties set on the prototype of the constructor function. This is how JavaScript achieves a form of inheritance.

Object literals and destructuring: An Overview

One place where JavaScript shines is creating and breaking down structured data from and to smaller pieces. It does this through object literals and destructuring.

Object literals. Object literals let the develpoper create a new JavaScript object from existing values in a declarative way. If the name of a property in the object literal and the name variable the value comes from are the same, then specifying the property's name is optional. There is also a more concise syntax for defining a function in an object literal, where the `function` keyword is removed, and the arguments are added to the name binding itself.

```
const bv = 2, c = 3

const obj1 = {
     a: 1,
     b: bv,
     c: c,
     d: function(a, b) {return a + b}
}

const obj2 = {
     a: 1,
     b: bv,
     c,
     d(a, b) {return a + b}
} // also allowed
```

Destructuring. Destructuring lets you break down an object into smaller pieces, taking only what you need. It is in many ways similar to a pattern match seen in functional languages, although it doesn't have an as defined failed state. If the pattern doesn't match, the values are just `undefined`.

Destructuring uses a syntax similar to object literals but on the opposite side of the equals sign. Values can be explicitly bound to a different identifier by using `objProp: myVarName`, where `objProp` is the property's name in the object, and `myVarName` is the name one desires to bind it to. Leaving the property name out binds the variable to the property of the same name. A default value can be given to a binding in case the object does not have a property with the given name using `prop = default` [3].

```
const obj = {
     a: 1,
     b: 2,
     nc: {x: 10, y: 20}
}

const {
     a,
     b,
     c = 3,
     nc: {x: nx, y: ny, z: nz = 30}
}
```

Putting these two features together, developers can write concise and understandable code that transforms values in many different ways.

Related Work on JavaScript

There are not many other HOPL papers with a definitive link to the paper covered in this essay. However, one which might pique some interest is the HOPL III paper on the Self programming language [2]. Self was an inspiration to the JavaScript prototype object model.

For more information about JavaScript as a language as it exists today, then check out the MDN Web Docs. Here web developers can find information on JavaScript features and the language itself and many other nice things to know when doing web development.

To get more involved in the development of JavaScript as a language today, check out the

TC39 Discourse forum where the language and its specification is discussed. You can also check out the `https://github.com/tc39`, where much of the work of maintaining and developing the specification is located. The most recently updated specification of the language, with any finished proposals included, can be found at `https://tc39.es/ecma262/`.

Tool Support for JavaScript

There are many ways to try JavaScript. The easiest way is to open up the browser console (often F12) and write stuff. JavaScript comes with all modern browsers, so there is no need to install anything for small experimentation. Browsers come with many features you would often find in an IDE, like a debugger. For more extensive development, I personally find WebStorm to be a nice IDE, but many also use VS Code. For not running code in the browser, installing Node.js is the most common approach. It allows both running JavaScript from the command line and serves as a REPL.

For more extensive development in JavaScript, a package manager like `npm` or `Yarn` is recommended. A package manager handles downloading packages as independent modules that developers can include in scripts. Having a bundler like WebPack can also be helpful to make working with different modules easier. A bundler also helps with working with different web technologies, like different languages (SCSS, TypeScript), process images, handle frameworks, linting, and more.

Personal Experience

There are two big things I really found interesting that I learned in the paper. The first is how what we often tell each other through "cultural osmosis" about JavaScript is not quite what is actually true. One big example of this is server-side JavaScript. Before this paper, I always thought of that as both a modern thing and a Node.js thing. In truth, server-side JavaScript has always been a thing from the very beginning. JavaScript has always been more than just a thing browsers use, even though that is where we find it most often.

The second thing is how revolutionary JavaScript seems to have always wanted to be. It might not always have managed to reach that height, but it never stopped trying. Many of the ideas around ECMAScript 4, for example, are things TypeScript is still trying to figure out now 20 years later.

One thing I think would be interesting to hear a bit more about from the paper is how stuff has progressed after ECMAScript 2015 was released. How is the process still going, and are there other interesting stories to be heard? What would be JavaScript's future in the next five years? What role will WebAssembly play in all of this? Will it be the same as back when JavaScript was first conceived of, where it would serve as a glue language between Java applets?

References

[1] A. Wirfs-Brock, B. Eich, *JavaScript: The First 20 Years*. Proc. ACM Program. Lang. HOPL IV. 2020.

[2] D. Ungar, R. B. Smith, *Self*. Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III). 2007.

[3] MDN Web Docs, *Destructuring assignment*. Available at: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment`.

Essay on History of JavaScript

Åsmund Aqissiaq Arild Kløvstad
University of Bergen

This essay summarizes the paper *JavaScript: The First 20 Years* [1] by Brendan Eich and Allen Wirfs-Brock presented at HOPL IV. The paper covers the history of JavaScript from 1995 to 2015 and tells the story of how Netscape's browser scripting language became one of the world's most widely adopted programming languages. This essay covers my motivation for choosing Eich and Wirfs-Brock's paper, an overview of the history, language features, related work, and tooling, as well as my experience reading and presenting the paper.

Brief Overview of the HOPL Paper

The paper is split into four parts. *The Origins of JavaScript* covers some early web history, the development of JS 1.0 and 1.1, and the Mocha and SpiderMonkey engines.

I was especially interested in JavaScript because it is a monumentally successful language by all metrics, and yet has a reputation for being weird, messy and "the worst invention ever" [2]. I wanted to know more about the history that lead to this contradiction—and perhaps learn to love JavaScript in the process.
—Åsmund

Creating a Standard deals with the formation of the TC39 technical committee and the creation of the first JavaScript standards up to ECMAScript 3. *Failed Reformations* covers (roughly) the period 2000-2008 when several attempts were made to radically change JavaScript, ultimately resulting in the Harmony proposal. Finally, *Modernizing JavaScript* tells the story of ES5 and ES2015—in many ways the results of the Harmony work. This brief summary follows the same structure.

Origins. In the early to mid 1990's the World Wide Web was becoming increasingly important, and several companies were making web browsers for the public. One of these companies was Netscape who in 1995 recruited Brendan Eich to create a scripting language that would enable interactive web pages. At the same time Sun was marketing Java and it was decided that Netscape would create a "little language" that could be used to glue together larger programs written in Java.

In 10 days in May of 1995 Eich created Mocha—the first prototype of this little language—and demonstrated it for Netscape. The demo was successful and in March of 1996 Netscape Navigator 2.0 shipped with JavaScript 1.0. A year later, Navigator 3.0 shipped with JS 1.1. While 1.0 was essentially just the working features from the Mocha prototype, 1.1 completed the initial definition and development of the JavaScript language.

Around this time Microsoft were working to include JavaScript support in their Internet Explorer browser. The result was JScript – a JavaScript implementation based on the behavior of Netscape browsers. The lack of a proper specification was a problem.

Netscape was sensitive to this criticism and wanted to work on a specification of JavaScript. However, Eich felt it was more important to create a new engine since JS 1.1 was still running on Mocha. The new engine was called SpiderMonkey and was the basis for JS 1.2 which shipped in 1997.

The first standard. At the same time work started on a JavaScript standard. Netscape and Sun had initially wanted to work with the W3 consortium or Internet Engineering Task Force, but neither was interested in standardizing a programming language. Instead, they landed on Ecma International (formerly the European Computer Manufacturer's Association) which was recognized by ISO and perceived to be sufficiently democratic. The creation of a standard committee was approved, and in November 1996 Ecma Technical Committee 39 (TC39) held its first meeting.

The group started work on a specification of JavaScript without host-specific libraries or extensions. The specification was designated ECMA-262 and in September of 1997 *ECMA-262, 1st Edition* was released for publication.

Sun had licensed the name JavaScript to Netscape and were supportive of standardization efforts, but were protective of the Java trademark and would not allow JavaScript as a name for the standard. Options like LiveScript, Espresso and EZScript were suggested, but the committee eventually decided on ECMAScript as the permanent name.

After a fast-track ISO process, a revised specification was approved and published as *ECMA-262, 2nd Edition* in 1998.

Meanwhile the TC39 working group had moved on to the extensions of the language included in JavaScript 1.2 and JScript 3.0. Unifying and standardizing these new elements took time, and *ECMA-262, 3rd Edition* (ES3) was approved in December 1999.

Failed Reformations. This longer process might have been a boon, because it took nearly a decade before ES3 was replaced by a new standard.

As early as 1998 Dave Raggett (a W3C Fellow) had proposed sweeping changes to JavaScript with "Spice". This proposal included changes to integrate JavaScript better with HTML and CSS, as well as a classes, modules and types. At first the proposal was not well received, but after ES3 there was interest in extending the language and make drastic changes.

Netscape was also working on a "JavaScript 2.0" with similar sweeping changes, and in late 1999 TC39 started work in earnest on Edition 4 (called $ES4_1$ in the paper for reasons that shall soon become apparent). This was intended to be a big change that made ECMAScript suited for "programming in the large" and was not necessarily backwards compatible.

However, work was slow and concerns for browser interoperability made difficult to implement changes.

At the same time, Microsoft announced the .NET framework and TC39 was rechartered to work on "Programming Environments" with ECMAScript demoted to a Task Group. Microsoft had also won the "browser wars" and Netscape laid off most of its staff. In July of 2003 the working group decided to suspend work on $ES4_1$. In hindsight "[$ES4_1$] was too sweeping and broad for completion or adoption" [3].

A few years later Macromedia (developers of Flash and ActionScript) had become Ecma members, Brendan Eich was CTO of the Mozilla Foundation and the use of JavaScript on the web was booming. In late 2005 work resumed on Edition 4 ($ES4_2$ in the paper). The goal was still to support programming in the large with type annotations, modules and improvements, but this time backwards compatibility was prioritized.

Not everyone was on board with this development. As it started to gain traction Microsoft and Yahoo! realized these changes would be a significant change to the web, and might be in direct competition with larger languages like C#. Together they proposed ES3.1 - a much more incremental approach.

The changes of $ES4_1$ turned out to be difficult to integrate and the project gradually lost support. In a 2008 Oslo meeting it was decided that TC39 should focus on completing ES3.1 while also working on "Harmony" – a future edition making larger changes.

Modernizing JS. Throughout the development of ES3.1 it became apparent that even these modest update constituted a new standard. In order to avoid confusion with the many years of work on alternate proposals, ES3.1 was renamed and published in 2009 as *ES5*.

The Harmony continued work with ES5 as a baseline, slowly incorporating versions of features from the previous ES4 attempts in a way that was compatible with the new direction of ECMAScript. The committee adopted a "Champions" model were one person was responsible for championing each new feature proposal. In fitting these proposals together, they were re-

worked to be "orthogonal" [5].
The result was ES6/2015 published in 2015. This is the final edition discussed in Eich and Wirfs-Brock's article and has many of the $ES4_{1/2}$ proposed features including classes, modules and iterators.

Brief Overview of JavaScript

JavaScript is one of the world's most used programming languages. In the 2020 Stack Overflow Developer Survey [13] JavaScript ranked first with 69% of professional developers reporting they use it.

Reading the paper, I greatly enjoyed all the little anecdotes and personal touches that turn out to have so much influence on a language as large as JavaScript.
—Åsmund

It is a dynamically typed, object-oriented language with first-class functions. The most common use of JavaScript is as a web scripting language, where it runs interpreted or JIT-compiled in browsers. However, it is also used as a server-side scripting language.
The syntax of JavaScript is intentionally C/Java-like with `//` or `/**/` for comments, curly braces for blocks and semicolons to terminate statements. Unlike C and Java, whitespace can impact JavaScript semantics because of automatic semicolon insertion.

Function Expressions: An Overview

This section gives an overview of function expressions as they are presented by Eich and Wirfs-Brock.
Function expressions were introduced in JavaScript 1.2 as a way to define anonymous functions. They were included in the standard as of ES3. Brilliantly this is done by simply making the name optional in function declarations.

```
// a regular function definition
function double (x) { return x + x };

// a function expression assigned to a var.
var quadruple = function (x) {
     return double(double(x))
};

quadruple(2); // returns 8
```

Function expressions are useful for defining functions as arguments for use as callbacks or in map. However, they can also be combined with object literals to define classless objects that have methods.

```
function Vector2D(x,y) {
  return { x: x, y: y,
           dot: function (other) {
             return (other.x * this.x)
                    +
                    (other.y * this.y);
             }
         }
}

var v1 = new Vector2D(1,0);
// creates a vector object

v1.dot(new Vector2D(0,1));
// returns 0
```

In this example we define a 2D vector object that contains a method for calculating its dot product with another vector. This method is provided as a function expression assigned to the `dot` attribute of the object.
Function expressions differ from the arrow functions introduced in ES2015 in that the latter does not bind `this` or `new`. As a result arrow functions are useful for mapping and callbacks that do not use `this`, while function expressions work well for object declarations and constructors.
The ES3 specification actually enables function expressions to be recursive by using the `callee` property of function objects, but this is no longer supported in ES5 (and is forbidden in strict mode).

```
function (n) {
  return n==1 ? 1
              : n * arguments.callee(n-1)
};
```

Related Work

This section presents some starting points for further reading on selected topics.

Earlier HOPL papers. For more on pre-JS scripting languages there is William Cook's HOPL III paper on AppleScript [4].

For more on the concept of "orthogonal features" there is the HOPL II paper on ALGOL 68 [5].

As a part of the ES5 specification work, property attributes were described using Statechart, which was presented at by David Harel at HOPL III [6].

Mocha/early JS. The very early history of JavaScript and the development of the Mocha prototype have been described by Brendan Eich in blog posts [9, 10] and a podcast [11].

Browser wars. For more information about the "Browser wars" of the early 2000s there is the browser wars article cited by Eich and Wirfs-Brock [7].

For a contemporary prediction of what was to come there is this PC week article [8].

Finally the wikipedia article gives a good overview [12].

Active Community. TC39 is still active and welcomes contributions. To see what they are working on or start contributing to the ECMAScript standard there is the ECMA262 github `https://github.com/tc39/ecma262`.

Tool Support for JavaScript

JavaScript has probably the most widely distributed REPL of any programming language since every browser comes equipped with a JS interpreter. By selecting the *Inspect* command form the browser's context menu, or pressing F12 and navigating to *Console*, it is very easy to access an interactive console and start experimenting.

For more visual or artistic experimentation there are options like p5.js (`https://editor.p5js.org`) that include a simple library for drawing shapes, colors and animations.

Additionally JavaScript is extremely prevalent and your IDE of choice probably has a JS plugin.

Personal Experience

In this section I share some notes about the experience of working with the paper and the history of JavaScript. Finally, I present some questions for the authors.

Reading the paper, I greatly enjoyed all the little anecdotes and personal touches that turn out to have so much influence on a language as large as JS. It was also interesting to follow the development of what I would consider very "modern" features of JavaScript all the way from the 1998 Spice proposal, through the turmoil of the oughts and into ES5 (or not).

Additionally, I was introduced to the active work TC39 is still doing through this paper and a guest lecture by Yulia Startsev. The open and collaborative nature of contemporary ECMA-262 development is very exciting.

My questions are largely posed by the authors themselves. In their conclusion they list a number of *"what if"*s. I am especially interested in speculations on the wider web environment. What if Microsoft had pursued Visual Basic instead of JScript? What if Macromedia/Adobe had pushed for ActionScript instead of participating in $ES4_2$? What if the browser wars had gone differently?

All of these scenarios could have had huge impacts on the web of 2021, and I would be curious to hear the speculations of the authors. Could a different language than JavaScript have served its purpose? What were the best- and worst-case scenarios for the interactive web?

References

[1] A. Wirfs-Brock, B. Eich, *JavaScript: The First 20 Years*. Proc. ACM Program. Lang. HOPL IV. 2020.

[2] Bert Bos, *JavaScript, the worst invention ever*. Available at: `http://www.phonk.net/Gedachten/JavaScript`.

[3] William A. Schulze, *TG1 Convener's Report to TC39s*. Available at: `http://archives.ecma-international.org/2004/TG1/tc39-tg1-2004-006.pdf`.

[4] W. R. Cook, *AppleScript*. Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III). 2007.

[5] C. H. Lindsey, *A History of ALGOL 68*. The Second ACM SIGPLAN Conference on History of Programming Languages (HOPL II). 1993.

[6] D. Harel, *Statecharts in the Making: A Personal Account*. Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III). 2007.

[7] J. Borland, *Browser wars: High price, huge rewards*. Available at: http://www.zdnet.com/article/browser-wars-high-price-huge-rewards/.

[8] J. Berst, *Web-Wars*. Available at: https://web.archive.org/web/20110718025042/http://www.well.com/user/oink/oinkxweb/snippets/g7-bxls/webwar3.htm.

[9] B. Eich, *Popularity* (blog post). Available at: https://brendaneich.com/2008/04/.

[10] B. Eich, *New JavaScript Engine Module Owner* (blog post). Available at: https://brendaneich.com/2011/06/.

[11] *The Origin of Javascript with Brendan Eich*. Link: https://devchat.tv/js-jabber/124-jsj-the-origin-of-javascript-with-brendan-eich/.

[12] *Wikipedia page on "Browser Wars: First Browser War (1995-2001)"*. Link: https://en.wikipedia.org/wiki/Browser_wars#First_Browser_War_(1995-2001).

[13] *2020 Developer Survey*. Link: https://insights.stackoverflow.com/survey/2020/#technology-programming-scripting-and-markup-languages-professional-developers.

LOGO

Link to the HOPL IV paper:
https://dl.acm.org/doi/10.1145/3386329

Link to the students' presentation:
https://git.app.uib.no/uib-hopl-iv/slides/-/blob/master/Logo.pdf

Essay on History of Logo

Simen André Lien
University of Bergen

This essay summarizes the paper History of Logo [1] by Cynthia Solomon, Brian Harvey, Ken Kahn, Henry Lieberman, Mark L. Miller, Margaret Minsky, Artemis Papert and Brian Silverman presented at the HOPL IV conference. The paper discusses the history of the programming language Logo, its technical characteristics, powerful ideas and influences throughout the years. Cynthia Solomon, one of the authors of this paper, is also one of the creators of Logo.

Brief Overview of the HOPL Paper

The paper on the history of Logo covers the early influences of the language, such as the people, places, and its time in history.

I chose to present this particular HOPL paper because the idea of having a programming language created specifically for children sparked my interest. I can only imagine how difficult it must have been for children to learn to program with the existing programming languages at the time since none of them had been designed with children in mind.
—Simen

It also covers the technical characteristics of Logo to great detail, such as the changed and unchanged aspects from Lisp, which is the language that Logo is based on. It also contains the creators' reasoning for the design choices they made when creating Logo.

Furthermore, the paper talks about Logo before and after personal computers. It includes reflections from one of the creators of the language, Cynthia Solomon, and it talks about some of the powerful ideas that came from the early days of Logo, such as antropomorphization and body syntonics.

The paper goes on to talk about the 80's and forward, mentioning some of the hundreds of Logo dialects that have been created. It also talks about the design for Logo Microworlds, which are spaces that you can think about a particular problem, and have goals that are related to them. The most well known of Microworlds is the turtle graphics. Additionally, the paper talks about object-oriented programming versions of Logo, as well as terms such as localization, which is about translating Logo into other languages. Localization was hugely important for Logo's success around the world because Logo had to be accessible in the children's native language in order for them to use it.

Brief Overview of Logo

Logo is a programming language that was designed for children to learn programming. It was invented in 1966 by Seymour Papert, Wallace Feurzig, Daniel Bobrow and Cynthia Solomon at Bolt, Beranek and Newman, Inc in the USA. The Russians had managed to send the first artificial satellite into orbit on October 4, 1957, and this caused USA to improve their funding for schools all around the country, focusing especially on education around math and science.

It was when visiting a children's classroom that Papert saw the need for a programming language for children, because the programming languages the students were using at the time were far too complex for them. This inspired him to create a programming language designed specifically for children, and so the start of Logo begun. Logo was the first programming language created for children, and with it came a new way of thinking about children and the way they learn. Logo was influenced by Marvin Minsky's AI research at MIT, and by psychologist Jean Piaget's constructivism: the idea that people learn by doing, and that they form meaning from their experiences [4].

Logo's competitors at the time, Pascal and BASIC, were compile-then-run, while Logo had an interactive development model. This meant that you could type an instruction and immediately have

it be executed, which they argued was an essential feature for children [1, p. 15]. Logo also had automatic memory management, because they didn't want children to have to worry about freeing up memory. They argued that Logo should be easy and intuitive to use, and that children shouldn't have to worry about such specifics [1, p. 18].

Furthermore, Logo had great attention to wording. Its primitives for working on lists such as `car` and `cdr` from Lisp were named `first` and `butfirst` to be more descriptive in order to make it easier for children to use. Another feature that was given careful attention was error messages. The creators of Logo didn't want children to be afraid of getting error messages, and so they focused on making them helpful and intuitive for children to work with. Instead of getting error messages such as "RangeError" or "Invalid code point NaN", the children got more descriptive messages such as "I don't know how to *function name*" when calling a function that had not yet been defined [1, p. 28].

The part that I enjoyed the most about the paper was the environment that they created around children's learning.
—Simen

Today there are hundreds of dialects of Logo, and Logo has inspired many other programming languages with its features and ways of thinking. One of the most well known features that came from Logo is the turtle graphics, which allow you to draw on the screen by moving a turtle. Turtle graphics has since been implemented as a library for most programming languages today.

Turtle graphics: An Overview

Historically speaking, the turtle graphics is arguably the most important feature built for Logo. It allows the user to draw on the screen by using a turtle that has attributes such as a position, a direction and a pen. To move the turtle, you would call commands that move the turtle relative to it's current position and direction. The six primitives for the turtle are `FORWARD`, `BACK`, `RIGHT`, `LEFT`, `PENUP` and `PENDOWN` [1, p. 36]. For instance, one could say `move forward 100`, or `turn right 90 degrees` and the turtle would draw while moving if the pen was down. The creators of Logo also created real life robots called floor turtles which the students could program to walk around on the floor, and these robots were very popular in the early days of Logo [1, p.45]

The creators of Logo also came up with some powerful ideas about how the students could think about the turtle and how it would move. Telling the turtle to `move forward 100` was not always clear for the students. What is forward? Is it north? Is it forward in the direction the turtle is facing? Something else? To make it intuitive for the students, they were taught to imagine that they themselves were the turtle. Telling the turtle to move forward was now clear to the students, because they themselves imagined being the turtle, and now understood what forward, right, left and back meant. Papert coined this the term Body Syntonics [1, p. 41].

The students were also taught to think about the turtle as someone that they could try to help, as opposed to the turtle just being a machine that is fully controlled by the programmer. When a bug occurred, it allowed the students to have the mindset of them trying to help the turtle solve the bug, instead of putting the blame on the student for the bug being there. All in all, turtle graphics was a great way for children to learn and get engaged with programming at the time, and today turtle graphics is a common library for most programming languages and is used by people of all ages, all around the world.

Related Work on Logo

There are four different types of papers that can be submitted to the HOPL IV conference, namely papers that talk about one of these categories: 1) Early History of a language, 2) Later Evolution of a language, usually one that has already been in a previous HOPL conference, 3) a cross-language examination of a feature or concept, or 4) a consideration of a class of languages that have a common theme or purpose [2].

The History of Logo paper talks about the early history of Logo, and is connected to the other papers that fall into the same category. The paper talks about the language from when it was created through to present day, and it mentions why certain language choices were made over others. Some of the HOPL IV papers that are similar to the History of Logo paper are "A history of the Groovy programming language" and "The Early History of F#", which both are accepted papers to the HOPL IV conference. [3].
Here are some links to related sources if you would like to learn more about Logo:

- Website for the Logo Foundation, which offers workshops for teachers and more: `https://el.media.mit.edu/logo-foundation/`.
- Learn more about Logo on this website for the Logo foundation: `https://el.media.mit.edu/logo-foundation/what_is_logo/logo_programming.html`.
- Wikipedia article about the Logo programming language: `https://en.wikipedia.org/wiki/Logo_(programming_language)`.
- Here is an article from 2014 that was written about Logo. It also contains links to other resources about Logo: `https://www.kidscodecs.com/logo-programming-language/`.

There are also links to many resources in the History of Logo HOPL IV paper. It has its own section called "Logo-Related Web Sites", and it also mentions multiple book publications about Logo: `https://dl.acm.org/doi/10.1145/3386329`.

Tool Support for Logo

Here are some tools that I would recommend checking out if you would like to try Logo.

- If you would like to download Logo, then check out Logo Foundation's website: `https://el.media.mit.edu/logo-foundation/resources/software_hardware.html`.
- If you would like to try Logo in an online browser, then check out this link: `https://www.calormen.com/jslogo/#`.
- Here is a great way to learn more about turtle graphics in Logo: `https://turtleacademy.com/`.

Personal Experience

The part that I enjoyed the most about the paper was the environment that they created around children's learning. Revolutionizing the thinking about children's ability to learn, and showing that given the right tools, and pedagogical support, children were capable of much more than previously thought. I think it is great that they opened up the possibilities for many more children to learn programming and start enjoying the fields of mathematics and science.
Some of my questions to the authors of the paper are: Did you ever think that turtle graphics would have become such a popular feature? A question for Cynthia Solomon that I have is, looking back, is there something that you would have done differently with Logo? If so, what? Another question I have is what was some of the most interesting things you found when gathering information to write the paper?

References

[1] Cynthia Solomon, Brian Harvey, Ken Kahn, Henry Lieberman, Mark L. Miller, Margaret Minsky, Artemis Papert, Brian Silverman, *History of Logo*, Proc. ACM Program. Lang., Vol. 4, No. HOPL, Article 79.

[2] *HOPL IV Papers*, available at: `https://hopl4.sigplan.org/track/hopl-4-papers#Content-Guidelines-for-Authors`.

[3] *HOPL IV Papers Overview*, available at: `https://hopl4.sigplan.org/track/hopl-4-papers#event-overview`.

[4] *Piaget's Theory on Constructivism*, available at: `https://www.teach-nology.com/currenttrends/constructivism/piaget/`.

Essay on History of Logo

Emily Mi L. Nguyen
University of Bergen

This essay summarizes the paper *History of Logo* [1] by Cynthia Solomon, Brian Harvey, Ken Kahn, Henry Lieberman, Mark L. Miller, Margaret Minsky, Artemis Papert and Brian Silverman presented at the HOPL IV. The paper discusses Logo's background, the technical characteristics of Logo, the historical events in Logo's timeline and the different perspectives on Logo.

Brief Overview of the HOPL Paper

This HOPL paper starts with an introduction and an overview of the early influences on Logo. In section 2 we meet Wallace (Wally) Feurzeig, Seymour Papert, Jean Piaget, Marvin Minsky, Cynthia Solomon and other early contributors. This section also covers how great Logo fit in with the time and place it was invented in.

I chose this particular HOPL paper to present because I share the same interest as the creators of Logo when it comes to the psychological and cognitive theory of a child's mind and way of thinking, as well as the evolutionary epistemology.
—Emily

Further on in section 3, this paper dives into the technical characteristics of Logo. It starts with the design process, and continues to show how Logo differs from Lisp. Some aspects of Logo are taken unchanged from Lisp, while other aspects are totally new and do not inherit from Lisp. Logo has no standard, however it has some special forms known as the Terrapin and the LCSI Split. The creators also pay a great amount of attention to how they word themselves, on dynamic scoping and the practice of debugging, which in the early days was a central part of Logo's culture.

The history of Logo starts in section 4 till section 6, presented in a chronological order. First, in section 4 we see how Logo did before personal computers were available for the public. Second, in section 5 the '80s hit and the usage of Logo increased drastically now that personal computers were available for the public. And at last, section 6 covers the idea of a more visual programming language for children in the present time.

The last three sections cover the different perspectives on Logo. Section 7 mentions the critical perspectives and critiques on Logo and also how these critiques affected negatively on the development. Section 8 enfolds Logo's influence on Artificial Intelligence and computer science. We get to see these influences through Henry Lieberman's, Mark Miller's and Ken Kahn's reflections. Section 9 is about the relationship between Logo and school, and as well as how it has changed the views on learning and teaching for both children and teachers.

Brief Overview of Logo

Logo is known to be the first programming language that explicitly is designed for educating children. It was invented in 1966 and already released to the public in the following year. This initially took place at the Bolt, Beranek and Newman, Inc (BBN) in Cambridge, Massachusetts, United States.

Danny Bobrow had been a doctoral student of Marvin Minsky and Seymour Papert, and became head of the Artificial Intelligence Group at BNN. Bobrow was the one introducing Papert to Wallace Feurzeig, and Feurzeig was in charge of a couple educational projects. Papert then began to consult on Feurzeig's educational projects, and Cynthia Solomon was at the time a member of Feurzeig's group as well. Papert visited several of these educational classes and saw the need for a better programming language designed for children, thus the idea of designing Logo was born. This is how all the creators of Logo met and started their journey together.

Logo's design is based on two theoretical frameworks. The first one is Jean Piaget's theory of constructivism, which is mainly about creating knowledge through experiences and interactions. The second one is Marvin Minsky's Artificial Intelligence research at MIT. Additionally, Logic started off basing on the early Lisp, and therefore became a dialect of Lisp.

Lisp was a great model for a language like Logo. It had an interactive development model, making Logo interactive, and not a compiled language. This was beneficial for children when they programmed, since they could see it being executed immediately after typing an instruction. Logo also inherited Lisp's recursive functions, dynamic scoping of variables, symbolic computations, and operations on linked lists.

I enjoy reading the paper knowing one of the creators of Logo has directly taken part of the paper and acknowledged it.
—Emily

However, Logo did not inherit all aspects from Lisp unchanged. Logo was not fully parenthesized, as well as not every procedure returned a value like Lisp did. Furthermore, Logo was designed around the idea of *microworlds*, and one of its microworlds is the famous *turtle geometry*. There were lots of features and concepts developed in Logo that Lisp did not have.

Paying Attention to Wording: An Overview

Usually paying attention to how things are worded in a programming language is not the highest priority, however it is in Logo. We will look into how they do this with assignment operators, predicates and error messages.

Different programming languages have different ways of saving a value with a name to it. For Logo there are at least two ways of doing it. One way is to save a constant value, which is a value that will not change throughout the program. As mentioned in the paper [1], this can be done by giving a name to the constant value like for instance this:

```
name 3.141592654 "pi
```

Another way is to save a value that changes during the program. These values could be, for instance, the value of an index in a loop that changes for each iteration. These types of assignments have a value that has to be put in a box, and this is how it is done [1]:

```
make "index :index+1
```

Predicate functions, the ones that return a Boolean value, use some special notations in all Lisp dialects. An operation that is supposed to make a list containing some values is a `list` operation. However, to check if a value is a list one could use the `listp` operation in traditional Lisps, although that later on became `list?` with a question mark in Scheme. There was an intense debate on how they should design the predicate function and which of the ideas were the most optimal ones. In short, when read out loud, the `listp` with the notation "p" for "predicate" is more audible noticeable to distinguish from `list`, however other words that end with "p" might be mistaken as the same type of notation like for instance `stop` should not be pronounced as "stow pee". In Scheme the `list?` was a great design since it is visually unique and a question mark has a lower risk of coming at the end of a word adventitiously. The first Logo dialect even designed a predicate function with a "q" which naturally stood for "queue", this could be the best design since it had both benefits from the `listp` and `list?`, but it did not remain part of the standard language design.

Since Logo was built to become a programming language for children, the developers have put a lot of thoughts into the error messages that children would receive. In general, people with no to little experience in programming tend to associate errors with bad things, especially for children this can be very discouraging. In order to prevent that, developers reconstructed the error messages to make them more user-friendly and understandable for the young students. Already in the 1960s Logo worked with this type of

language design. Although not all programming languages have yet adapted this design, a few languages, for instance, Elm, Go, C++ have made awareness and attempts to improve their error messages.

Related Work on Logo

This paper, *History of Logo*, has many things in common with the other HOPL IV papers. First of all, HOPL IV accepts four types of papers. The paper has to be either an early history of a programming language, a later evolution of a programming language, a close look into a cross-language that focuses on a specific feature or concept, or a whole class of languages with a common theme or motive [2]. This paper falls under the category as an early history paper, therefore will also have some similarities in content to the other papers in the same category, like for instance *A history of MATLAB* [3] or *The history of Standard ML* [4]. Second, Lisp is among one of the oldest high-level programming languages, thus has served as a model for many programming languages later on. This HOPL paper explains how Lisp has influenced Logo and initially shaped Logo in the early stages. Besides, Logo is not the only dialect of Lisp. There is a HOPL IV paper about the programming language Emacs Lisp. Emacs Lisp got invented in 1985, and can be argued to be one of the most widespread language of Lisp dialects [5]. Another Lisp dialect that is part of the HOPL IV papers is Clojure. Clojure did not appear before in 2007 and interoperates with Java [6].

There are several Logo-related web sites and a good deal of Logo books for further reading. *The Daily Papert* [7] is a web site dedicated to Seymour Papert's work run by Gary Stager. Another web site is the *Logo Foundation* [8] that offers educational support and resources for teachers, parents, students and anyone interested in Logo. *Mindstorms* is among Papert's most noteworthy books [9]. Other great books from Papert are *The Children's Machine* [10] and *The Connected Family* [11].

Tool Support for Logo

There are many tools available online for trying Logo. Personally, the few I have encountered that I recommend are the *Logo Interpreter* (`https://www.calormen.com/jslogo/`), *Online Logo* (`https://www.transum.org/software/Logo/`) and a web site that teaches you how to get started with Logo and about programming principles in general called *Turtle Academy* (`https://turtleacademy.com`).

Personal Experience

This paper has given me much insights into Logo's History. Personally, what I liked the most is the fact that Cynthia Solomon is one of the authors of this paper. I enjoy reading the paper knowing one of the creators of Logo has directly taken part of the paper and acknowledged it. What I also find extremely interesting is how the creators of Logo focus on children's knowledge and development. Seymour Papert was heavily influenced by Jean Piaget, and they both believed that children were more than just empty minds ready to be filled with knowledge [1]. Children are small builders of knowledge, they just think differently, and the creators wanted to enhance that and take children's way of thinking more seriously.

I do not have much questions for the authors of the paper, though there are some. The children that spent time with Logo instead of normal curriculum in school, how are they now? Did they develop better computational thinking skills than the children who did not participate in the experiment? It would be great if teaching children programming early on proved to turn out more skilled than children who were not exposed to programming.

References

[1] Cynthia Solomon, Brian Harvey, Ken Kahn, Henry Lieberman, Mark L. Miller, Margaret Minsky, Artemis Papert, and Brian Silverman, *History*

of Logo, Proc. ACM Program. Lang., Vol. 4, No. HOPL, Article 79.

[2] The Program Committee, *HOPL IV*, available at: `https://hopl4.sigplan.org/track/hopl-4-papers#Content-Guidelines-for-Authors`.

[3] Moler Cleve and Little Jack, *A History of MATLAB*, Proc. ACM Program. Lang., Vol. 4, No. HOPL, Article 81.

[4] MacQueen David, Harper Robert, Reppy John, *The History of Standard ML*, Proc. ACM Program. Lang., Vol. 4, No. HOPL, Article 86.

[5] Monnier Stefan and Sperber Michael, *Evolution of Emacs Lisp*, Proc. ACM Program. Lang., Vol. 4, No. HOPL, Article 74.

[6] Hickey Rich, *A History of Clojure*, Proc. ACM Program. Lang., Vol. 4, No. HOPL, Article 71.

[7] Gary Stager, *The Daily Papert*, available at: `http://dailypapert.com`.

[8] Michael Tempel, *Logo Foundation*, available at: `https://el.media.mit.edu/logo-foundation`.

[9] Seymour Papert, *Mindstorms: Children, Computers, and Powerful Ideas*, 1980, Basic Books, Inc., New York, NY, USA.

[10] Seymour Papert, *The Children's Machine: Rethinking School In The Age Of The Computer*, 1994, BasicBooks.

[11] Seymour Papert, *The Connected Family: Bridging the Digital Generation Gap*, 1996, Longstreet Press.

S & R

Link to the HOPL IV paper:
https://dl.acm.org/doi/10.1145/3386334

Link to the students' presentation:
https://git.app.uib.no/uib-hopl-iv/slides/-/blob/master/S-R.pdf

Essay on History of S and R

Janne Hauglid
University of Bergen

This essay summarizes the paper *S, R, and Data Science* [1] by John Chambers presented at The Fourth ACM SIGPLAN HOPL conference. The paper discusses the creation and development of the *R* programming language as a domain-specific language for the domain of data science, and the language's historical connections to both the programming language *S* and the field of data science.

Brief Overview of the HOPL Paper

S, R, and Data Science covers the evolution of both languages, *S* and *R*, from the early start in the 1960s to present day.

I choose this particular HOPL paper because I wanted to learn more about the R language. I personally do not have any experience with the language myself, but I have been fascinated about the increased popularity it has in statistical computing and data analysis. Therefore, I thought choosing this paper would be a good opportunity to learn more about both S *and* R *and their connection to data science.*
—Janne

John Chambers, the author, emphasizes the importance of understanding the connection the languages has to data science, and how this has strongly influenced the creation and the development of both languages.

The paper is divided into three parts. The first part includes the evolution of *S* from 1965-1985, and the importance of the culture and research atmosphere found at Bell Labs and the data analysis that was done there.

The second part continues from the year 1985 until year 2000. This part describes the transition from *S* to *R*, and the connection between those two languages. The structure of *R* as it is derived from *S* is described, as well as the relevant new features of *R*.

R was officially launched in the year 2000. In the last part, the author talks about the growth and evolution of the language since the launch, and how it has continued to influence the research done in the data science community.

Brief Overview of S and R

The history of S, R, and Data Science begins at AT&T's Bell Telephone Laboratories in New Jersey in the decade of the 1960s. Bell Labs was the birthplace of a great deal of important scientific advances in the years following the second world war. Among these advances were the programming language *S*. It is the research in data analysis at Bell Labs that was done during the evolution of *S* that today can be recognized as the precursor to data science. This research can be seen as the driver behind the creation of *S*.

In the 1960s, powerful software for data analysis existed in the form of an extensive subroutine library written in Fortran. However, the details needed to perform this analysis were tricky and extensive, especially for analysis involving serious data in terms of size or complexity. All research teams therefore required programmers to help with this.

John Chambers and Rick Beckers wanted to make it easier for the researchers themselves to make these changes, instead of having to go through programmers. Chambers and Beckers decided they wanted to create an "interactive statistical system" that would continue to support data analysis, while giving the analyst a convenient, direct interface.

Chambers and Beckers had three requirements for this new software: *convenience*, *completeness*, and *extensibility*. They found that the solution that would fulfill these requirements were to build the system around an interface to Fortran. By the time when *S* was distributed generally, the interface mechanism was provided to the users as an interface language that was precompiled into Fortran.

Chambers wrote in a 2016 book on R [2] that its design can be summarized by three principles:

- *objects*: everything that exists in *R* is an object;
- *functions*: everything that happens in *R* is a function call;
- *interfaces*: interfaces to other languages are a part of *R*.

These principles are broadly visible from the first version of *S*, and they explain a number of detailed decisions made during the development.

I found it very fascinating to read about the culture and research atmosphere at AT&T's Bell Telephone laboratories, and how this was very unique for this place.
—Janne

In the 1990s, work on *R* was started by Ross Ihaka and Robert Gentleman. The goal of the project was that *R* would be "a free immplementation of something 'close to' version 3 of the *S* language". Since the launch of *R* in 2000, the language has gradually become the main software to use for statistics and data science, and the popularity is still increasing today.

CRAN: An Overview

The paper does not talk much in detail about specific programming features of the language regarding syntax or semantics. However, it does mention some features of the language itself, one of them being the wide array of packages and libraries that can be installed and used.

The Comprehensive *R* Archive Network, also known as CRAN, is mentioned several times in the paper. According to the author, CRAN is by far the largest and most used site associated with the *R* language.

CRAN consists of the *R* distributions, documentation for *R*, contributed extensions and binaries [3]. The CRAN package repository currently contains more than 17600 packages [4].

According to Chambers, different repositories of contributed packages, and then in particular CRAN, have become a major factor in the growth and extension of *R*, particular within scientific disciplines. The contributed packages have played a central role in both the popularity and usefullness of the language. Data science as a field is constantly evolving, and will continuously need a wide range of software in order to fulfill the needs of its specialized audience. *R* and its package structure have become a favoured way to bring specialized techniques to their respective users.

Since *R* was officially launched in 2000, the language has expanded in every area, including the number of users, contributors and citations, as well as in public awareness. In addition, CRAN has received an exponential growth in the number of packages added.

Related Work on S and R

Connection to other HOPL papers. This HOPL paper has a few "connections" to other HOPL papers from all earlier HOPL-conferences, from the first conference held in 1978 to the current one.

One connection that the language *S* has to other programming languages is having the same "birth place" of Bell Laboratories. As previously mentioned, Bell Labs was home to a great deal of scientific advances. Among those were several programming languages, such as C (HOPL II), C++ (HOPL II–IV) and SNOBOL (HOPL I). All of these three languages have been presented at previous HOPL-conferences.

In addition, both *S* and *R* were created on interfaces to both Fortran (HOPL I) and C (HOPL II). Furthermore, *R* has also been influenced by Lisp (HOPL I–II). For example, the datatype `pairlist` in *R* is the traditional Lisp list.

Related work and further reading. The R-project web page [5] contains a great deal of information for readers who are interested in learning more about the language. In addition, the "*R* FAQ" [3] found on the R project's page contains answers to some of the most commonly asked questions regarding *R*.

Online tutorials. For readers who are interested in learning the language, there are several tutori-

als available online at popular tutorial sites such as Udemy, Codecademy and Coursera.

Books. *R for Data Science* by Hadley Wickham and Garrett Grolemund gives an introduction on how to do data science with *R*. Throughout the book, they present the skills needed to import data into *R*, how to structure it in the most useful way, as well as how to transform, visualize and model the data. The book is available in hard copy as well as a free-to-use website [6].

Tool Support for R

As I do not have any experience with the language myself, I do not have any personal recommendations for tools to use with the language. However, the paper does bring up the importance that IDEs have had for the language, in particular for the creation and revision of packages and other *R* software. An IDE that is specifically mentioned and recommended is RStudio. According to the paper, RStudio has become well-liked for teaching in data science courses. In addition, this IDE significantly simplifies both editing and installation of *R* packages.

Personal Experience

The part I enjoyed the most about the paper was reading about the history and situation that led to the creation of *S*. I found it very fascinating to read about the culture and research atmosphere at AT&T's Bell Telephone Laboratories, and how this was very unique for this place. The culture and management style allowed for the researchers to explore their own ideas while giving them the freedom to pursue their dreams.

If I had the chance to ask the author any questions at all, I would mainly focus on two different topics: the economics and budget behind the project, and mistakes made during the development. My questions would be:

1. Did the first project of developing *S* receive a specific budget, and if so, was it adequate for the project's specific needs?
2. How did the budget affect the development process and design decisions of the language?
3. Were any "mistakes" made during the development, and if so, what would the author have liked to do differently if he had that opportunity today?

References

[1] John M. Chambers, *S, R, and Data Science*, Proc. ACM Program. Lang., Vol. 4, No. HOPL, Article 84.

[2] John M. Chambers, *Extending R*, Chapman & Hall/CRC. 2016.

[3] Kurt Hornik, *The R FAQ*, available at: `https://CRAN.R-project.org/doc/FAQ/R-FAQ.html`. 2020.

[4] R-Project, *Contributed Packages*, available at: `https://cran.r-project.org/web/packages/`.

[5] R-Project, *The R Project for Statistical Computing*, available at: `https://www.r-project.org/`.

[6] Hadley Wickham, Garrett Grolemund, *R for Data Science*, available at: `https://r4ds.had.co.nz/`.

STANDARD ML

Link to the HOPL IV paper:
https://dl.acm.org/doi/10.1145/3386336

Link to the student's presentation:
https://git.app.uib.no/uib-hopl-iv/slides/-/blob/master/Standard-ML.pdf

Essay on History of Standard ML

Knut Anders Stokke
University of Bergen

This essay presents the paper *The History of Standard ML* by David MacQueen, Robert Harper and John Reppy [1]. The paper reviews the background and early history of ML, the effort of standardizing and formally defining the language, the development of some of Standard ML's major features, and the impact the language had on the field of computer science.

Brief Overview of the HOPL Paper

The introduction of the paper gives a review of Standard ML with a brief summary of its history, an explanation of the language's key features, and a short description of the importance of the language in computer science.

Personally, I have no previous experience with ML, but I am familiar with ML's type system from programming with Haskell. Both languages share a strong emphasis on type safety and formal specification. While many programmers find such rigorous programming languages limiting, I find them intriguing and worthwhile; programming admittedly requires more effort up front to satisfy the type checker, which usually pays off when one later maintains and expands the code base. ML has been influential in research,——

The following section covers the background of the language and gives insights into the British programming language research in the 1960's and 1970's, the programming research at the University of Edinburgh and the Edinburgh LCF project, and the programming languages that influenced Robin Milner's development of LCF/ML (the first ML language) and those that later influenced Standard ML. Among these programming languages are Lisp for its symbolic computations, ISWIM for using lambda calculus as a basis, POP-2 which had function closures, GEDANKEN and its principle of completeness, HOPE with its datatypes and pattern-matching, and Cardelli's ML, which had datatypes and modules. As several implementations and variants of ML appeared, there was a motivation for developing a Standard ML. The paper covers this process of standardising and formalising ML, which led to the published definition of SML in 1990 and the revised definition that was published in 1997. The following sections of the paper thoroughly describe five important aspects of SML. The first aspect is SML's Hindley-Milner type system, for which the paper covers the previous research that led to SML's type system and explains the system's basic features: the type inference algorithm (Algorithm W), some of the system's concepts (e.g., type constructors, type abbreviations, data types), LCF/ML's ad-hoc solution to combining side effects and polymorphism, the polymorphic equality operation, and the type errors. The next aspect is the module system of SML, which did not exist in the first ML language, but was introduced in Cardelli's ML; the paper gives an explanation of the module system and the involved design choices. The definition of the Standard ML is the third language aspect covered by the paper; the authors describe the early work on formalisation of semantics and the semantic model used to define the language, and they explain the content of the two published definitions in short. The last two language aspects covered by the paper are the type system used in the language's definition and the SML Basis Library.

The paper concludes with some of the mistakes in the definition, the impact that Standard ML had in the world of programming languages, and a comment on the SML's non-evolution that led to the language's declined popularity in the recent years.

Brief Overview of Standard ML

Standard ML is a statically typed, functional programming language. In functional programming,

programs are constructed by defining, applying and composing functions. Functions are first-class citizens, which means that they can be treated as any other values in the language: they can be passed as arguments to other (higher-order) functions, returned from functions, and assigned to variables.

——as well as in teaching and software applications, and this paper was a good opportunity to learn the basics and history of this important language.
—Knut Anders

The polymorphic type system used by Standard ML is the Hindley-Milner system (by Roger Hindley and Robin Milner). The language uses static types, which enables SML code to be type checked before it is executed and is therefore considered safer than dynamically typed code. It supports parametric polymorphic types, which enables the same code to be applicable for several types.

Programmers can construct new types in Standard ML using type abbreviations and algebraic data types, where a datatype can have several constructors, each containing a number of values. The language supports pattern matching for extracting these values from a datatype and case expressions that uses a datatype to create branches of an expression based on the constructor of the datatype.

The paper enabled me to get a better understanding of the origins of statically typed functional programming.
—Knut Anders

SML expressions are evaluated immediately, which makes the language strictly evaluated. Since the runtime computes values that might never be needed, larger computations may take more time then it would in lazy evaluated languages. On the other hand, a runtime that uses lazy evaluation, such as the Haskell runtime, must store code thunks that may later be evaluated, which can lead to high memory usage and, as a consequence, slow execution.

While Standard ML is a functional language, it also supports some imperative features, such as variable assignment and mutation of data structures. By pointing a reference of type `ref` to a variable, one can later modify the variable either through recursion or sequential composition.

The language has a safe escape mechanism that enables programmers to fail the evaluation of an expression by raising an exception. If a function can raise an exception, any caller of that function must provide an expression that handles the exception if it is raised.

Lastly, Standard ML uses a module system to divide a program into smaller, composable units. The details of the module system is covered in the following section of this essay.

Modules: An Overview

Standard ML uses a module system to constitute programs from smaller, composable units. The module system has a notion of *structures* and *signatures*. A structure is a collection of named components, where each component is typically a value, a function, or a type. A signature is a static description that specifies the components of a structure, and is thus similar to interfaces in object-oriented programming. Structures and signatures are declared individually and have a many-to-many relationship: one signature can ascribe several structures and vice versa. By ascribing a signature to a structure, the type checker will verify that the declared components in the signature exist and have the right types in the structure, and it will also hide the components in the structure which is not declared in the signature. A structure can be a component of another structure, which makes structures hierarchical.

The following code represents an example of a signature and structure for an ordering.

```
signature ORD = sig
    type elem
    val le : elem * elem -> bool
end

structure IntOrd : ORD = struct
    type elem = int
    val le = Int.<=
    val x = 5
end
```

Structure `IntOrd` declares type `elem` to be `int`

and value `le` to be the integer relation "less than or equal to". Signature `ORD` verifies that `IntOrd` has components `elem` and `le`, and since value `x` is not declared in the signature, the value is not visible from outside the structure. Components inside the structure, however, can still use the hidden value.

A structure can further take another structure as an argument and use the components of that argument. Such structures are called *functors*, since they behave as functions from one structure to another. To give an example, consider a structure `Sort` for sorting a list of integers, with an insertion sort algorithm that uses function `Int.<=` to compare any two elements in the list. The algorithm uses only the ordering of integers to sort the list, and can therefore be used to sort any lists over elements with an ordering; if we change `Sort` into a functor that takes `(X : ORD)` as an argument, the algorithm can sort lists of type `X.elem list` by using `X.le` to compare any two elements.

Related Work on Standard ML

Standard ML is an interesting language from the historical aspect, as it is both influenced by and has influenced many languages. The first ML-dialect, LCF/ML, was implemented in the programming language Lisp, which is covered in the proceedings of the two first HOPL conferences [2, 3]. The second volume of HOPL reviews Pascal [4], one of the first programming language to incorporate SML's record types. F# [5] is another dialect of ML, and is thus closely related to Standard ML. A language that is significantly influenced by ML is the purely functional programming language Haskell, which was one of the languages covered in the third HOPL conference [6].

As previously mentioned, ML touches on several aspects of computer science research. The LCF proof system and ML as a meta-language for interactive proofs are reviewed in the paper *A metalanguage for interactive proof in LCF* [7] by Milner et al. The original, formal definition of SML from 1990 is found in *Definition of Standard ML* [8] and was later revised in 1997 [9]. There exist several implementations based on the language's definition, such as *Standard ML of New Jersey* (SML/NJ) [10] and *The ML Kit* [11], and an overview of compilers and variations of the language can be found at SML's website [12]. SML is an often used language in teaching of programming concepts, ranging from introductory programming [13, 14] to more advanced topics as compilers [15] and concurrent programming [16]. One of the books on introductory programming, *Programming in Standard ML* [17], is written by Robert Harper, who is one of the authors of the HOPL paper. The Hindley-Milner type system and the type inference algorithm used in Standard ML are reviewed in *A Theory of Type Polymorphism in Programming* [18].

Tool Support for Standard ML

To test out Standard ML, one can use the open source compiler SML/NJ [10]. It implements the revised definition of SML and extends the basis library with, among other features, functions for getting system information and performing lazy suspension. The compiler claims to be incremental, which means that it only compiles the modified code for effeciency reasons. It also features a REPL (short for "read-eval-print loop"), which enables programmers to easily get type information of expressions, as well as evaluate them. The text editor Visual Studio Code has an extension *SML Environment*, developed by V. Julião [19], that provides IDE support for SML with features such as syntax highlighting and code suggestions. It also features evaluation of a selected part of the code, for which it uses SML/NJ behind the scenes.

Personal Experience

The paper enabled me to get a better understanding of the origins of statically-typed, functional programming. I learnt about the early efforts of designing proof assistants and the need for a pro-

gramming language that is both type-safe and convenient in its use. I also got insights into the history of the Hindley-Milner type system, and I learnt about the programming language ISWIM, which was the first to use lambda-calculus as a basis for programming.

The end of paper explains how the evolution of SML was prevented by Robin Milner and Mads Tofte. They wrote to the implementers that revisions would no longer be accepted, and that one should design new languages based on SML instead of changing the language itself. I would like to know more about the process that led to this decision, whether the other researchers involved with SML agreed, and if a standard committee would reach the same conclusion.

References

[1] D. MacQueen, R. Harper, and J. Reppy, "The history of Standard ML," *Proc. ACM Program. Lang.*, vol. 4, June 2020.

[2] J. McCarthy, *History of LISP*, p. 173–185. New York, NY, USA: Association for Computing Machinery, 1978.

[3] G. L. Steele and R. P. Gabriel, *The Evolution of Lisp*, p. 233–330. New York, NY, USA: Association for Computing Machinery, 1996.

[4] N. Wirth, *Recollections about the Development of Pascal*, p. 97–120. New York, NY, USA: Association for Computing Machinery, 1996.

[5] D. Syme, "The early history of F#," *Proc. ACM Program. Lang.*, vol. 4, June 2020.

[6] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler, "A history of Haskell: Being lazy with class," in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, (New York, NY, USA), p. 12–1–12–55, Association for Computing Machinery, 2007.

[7] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth, "A metalanguage for interactive proof in LCF," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 119–130, 1978.

[8] R. Milner and M. Tofte, *The definition of Standard ML*. Citeseer, 1990.

[9] R. Milner, M. Tofte, R. Harper, and D. MacQueen, *The Definition of Standard ML: Revised*. MIT Press, 1997.

[10] A. W. Appel and D. B. MacQueen, "Standard ML of New Jersey," in *International Symposium on Programming Language Implementation and Logic Programming*, pp. 1–13, Springer, 1991.

[11] L. Birkedal, N. Rothwell, M. Tofte, and D. N. Turner, *The ML Kit: Version 1*. DIKU, 1993.

[12] "Standard ML family GitHub project." `https://smlfamily.github.io/`, 2015. Accessed: 2021-05-20.

[13] M. Felleisen, D. P. Friedman, and D. Bibby, *The little MLer*. MIT Press, 1998.

[14] J. D. Ullman, *Elements of ML Programming (ML97 Ed.)*. USA: Prentice-Hall, Inc., 1998.

[15] Z. Shao and A. W. Appel, "A type-based compiler for Standard ML," *ACM SIGPLAN Notices*, vol. 30, no. 6, pp. 116–129, 1995.

[16] J. H. Reppy, *Concurrent programming in ML*. Cambridge University Press, 2007.

[17] R. Harper, *Programming in Standard ML*. Citeseer, 2001.

[18] R. Milner, "A theory of type polymorphism in programming," *Journal of Computer and System Sciences*, vol. 17, no. 3, pp. 348–375, 1978.

[19] V. Julião, "SML enviroment." `https://github.com/vrjuliao/sml-vscode-extension`. Accessed: 2021-05-20.