Getting Up and Running the λ-Calculus

GEORGE(S) ZAKHOUR

Abstract (Spoiler Alert) — Programmers, developers, and coders suffer from a myriad of issues pertaining to their health. These can vary from eye redness to repetitive strain injuries and a diminished life expectancy. A common exercise that software practitioners can engage in to reduce these health averse conditions is to be in the outdoors and to move—by means of walking or running—longer. Alas, professionals are hesitant to pursue their physical and mental well-being as that sacrifices productivity and programming-derived joy.

In this paper, we address this problem through the lens of programming languages and provide a solution that greases the friction between outdoor activities and programming. Through the insight that humans leave a trace while moving and the observation that apparatuses to record such traces are ubiquitous, we formalize an encoding of the λ-calculus in those traces. We develop an alternative front-end to Church's language which we call Poololoop and we provide a reference compiler that produces Haskell and Scheme code. We evaluate Poololoop on two use-cases that we ran and show that the compilation runs in a few milliseconds.

CCS Concepts: • **Social and professional topics** → Computing occupations; • **Human-centered computing** → *Interaction techniques*; • **Theory of computation** → **Formalisms**; *Grammars and context-free languages.*

Additional Key Words and Phrases: λ-calculus, Programmer Health, Runtimes

1 INTRODUCTION

The attentive reader would have noted that computers and the Internet are ubiquitous. For the clueless reader: in the 2010s, 3.04 billion personal computers were shipped [33] in part to the 5.35 billion Internet users [20]. Those—the PCs—are being comandeered, in 2024, by an estimated 28.7 million software developers [19], showing that they too—the software developers—are ubiquitous.

Popular Programming Plagues. It is not too uncommon to witness a plethora of software engineers and programmers shuffle this Earth bemoaning their bad health. Luria [48] collected the death notices published in *Science* between 1958–1968 and found that the mean age at death of male engineers is 71.1 (N=192) and that of women was 82 (N=1). For both reported genders it was found that archeologists survive engineers: 76.7 for men (N=12) and 84 for women (N=1). In their longitudinal study on life expectancy by occupation, Luy et. al [49] found that in the 1990s the probability of German men in technical occupations such as engineering and maths aged between 40 and 60 of surviving is 89.5% (N=364) while that of German women in the same occupation is 91.7% (N=56). And similarly to the previous study, men in Social service and education's probability of survival is 90.9% (N=159) and that of women is 93.9% (N=172). While these numbers include software engineers they also include other professions that require their practitioners to wither a lifetime on a desk. Nonetheless these numbers show that professions that require being outdoors, weathering the elements, have their practitioners live longer. For instance mucking about in the mud [84] will grant the mucker 2–5 more years as well as the opportunity to uncover ancient teeth, rusty Victorian trash, Roman garbage, and large feathered reptiles[1].

Statistics reporting on the life expectancy of software engineers are scarce. Nonetheless, Postamate reported that "software engineers have a life expectancy of only 55 years, compared to 78 years for the general population" [27]. The website, whose slogan is "Home of Satire and Sarcasm",[2] proceeds to ask why software engineers die so early without delivering a satisfying answer.

[1]The reader is recommended the article by Gartley [32]—by which we mean the article written by Gartley and recommended by the author(s), and not recommended by Gartley, although the author(s) find it hard to believe that Gartley would not recommend Gartley's article for it is a good article—for some dank dinosaur memes.

[2]A note to the editors: commas and generally any punctuation sign, will go outside quotations and parenthesised sentences.

Yet, because programming, coding,[3] and software engineering are sedentary jobs they come with a myriad of related health issues. Chief among the Repetitive Stress Injuries that programmers must deal with is Carpal Tunnel Syndrome: the professionals suffering from CTS are dominantly programmers, system administrators,[4] and IT professionals [1, 80, 81]. Other musculoskeletal problems include pain and stiffness in the neck among 48.6% of computer professionals, in the lower back (35.6%), and in the shoulders (15.7%) [81].

Sedentary jobs present an uncountable number of other issues: (1) vision blurring (13.2%), (2) irritation in the eyes (18.6%), (3) watering of eyes (23.2%), (4) pain in the eye (25.7%), (5) burning in the eye (29.8%), and (6) headaches (29.2%) [81].[5]

In summary, it is surprising that nerds—software developers—suffer from so many illnesses that can be avoided if they could just go outside and have a walk.

Problem Statement. The main activity of software developers is to develop software [68] through software development languages [68]. With the exception of very few languages discussed in Section 7, these are primarily developed to be written using a so-called full-size keyboard consisting of somewhere between 101 and 105 keys resting on a desk and read—the programs expressed in the programming languages that is and not the keys—on a computer monitor beaming every character onto the reader's cornea at a generous 120Hz. The author(s) believe that this overly constrained development environment is the direct cause of the software developer's overly restrained physical posture. We address this problem at its very core by designing a programming language that will not have negative health effects on its user.

Solution. By rethinking how programs are expressed and by deconstructing the syntax used to express these programs the author(s) present a language whose syntax is the path left behind by a technophile walker, hiker, runner, or cyclist[6]. These activities are predominantly done in the outdoors and are often recorded passively through a smartwatch or a smartphone. The programming language, Poololoop, leverages the twists and turns in the recorded path to express programs.

The main benefits of using Poololoop are thus:

(1) The eyes are free to observe Nature and wildlife, eliminating the need for the 20-20-20 rule[7],
(2) The hands and fingers are free to be relaxed,
(3) The body is exposed to natural sources of vitamins such as the Sun—being an example of a source and not a vitamin—,
(4) The user's partner is free to believe that the user took time off work to hang around.

By developing a programming language to solve this problem the author(s), expert(s) in the domain of programming languages, abide by Maslow's principle: if the only tool you have is a hammer, it is tempting to treat everything as if it were a nail [51].

Paper Structure. In Section 1 we motivate the problem. So if you have not been motivated already then reread Section 1 until you are. In Section 2 we present the necessary background on programming languages and running. In Section 3 we describe the language; its high-level ideas and its formalism. In

There is nothing you can say that will convince me otherwise. The compromise I offer is to typeset the comma, or period, directly under the quotation signs. Something like "this", or "that".

[3]Similarly to Footnote 2, the Oxford comma is another hill I am willing to die on.

[4]The American editors are seething and malding right now.

[5]You know that feeling you get after reading a long list of symptoms? The burning in the eye, the tingling in the fingers, the shooting headache, the slight dizziness... That feeling that these symptoms creep up on you one by one and the conclusion your mind draws is that you must be suffering from those symptoms? The author(s) feel that this is happening to them as they are typing. But let's not forget that the author(s) are computer professionals and they might be exhibiting actual symptoms.

[6]While the techniques presented here apply to all four activities the paper will only focus on the runners and walkers.

[7]Every 20 lines-of-code take 20 minutes to update 20 dependencies.

Section 4 we provide code examples from the Poololoop standard library. In Section 5 we discuss the implementation of the compiler. In Section 6 we showcase two use cases where Poololoop was used in real-life. In Section 7 we discuss the related work. And in Section **??** we do not conclude in solidarity with McCann [53] who recommends that *SIGBOVIK* bans conclusions.

2 BACKGROUND

In this section we describe the necessary background that we assume the reader is ignorant of.

2.1 Running

Running is not only an action that programs do. Running is an action that many things do, probably too many things, as it has the most number of meanings in the Oxford English Dictionary [89]. The first definition, *I.i.1.a*, in the dictionary states that running, when applied to mammals—which humans are—is the act of moving rapidly on alternating feet—otherwise it's just hopping—while never having all appendages simultaneously on the ground. The second definition, *V.79.d.i* is the one familiar to most serious programmers. In this paper we adopt both definitions and disambiguate them where needed by explicitly mentioning whether a program is to be ran, or a human—author(s) included—is to be doing the running.

Scientifically, running has been the object of study as old as *Science*[8]. Famously, in 2010, Keller, a scientific-mathematical human runner [50], solved the long standing problem of the *Jogger's Ponytail* [42]: a phenomenon observed by the running community where a jogger's ponytail sways from side-to-side while her head bobs up-and-down. And in 2002 the answer to whether one should run or walk in the rain has been proposed by Bailey [7].

Technologically, running has been commoditized under *Sport Business* [71]. Unsurprisingly for readers in the first quarter of the twenty-first century, social networks[9] for runners exist. The leading platform is Strava [78] where runners connect with other runners. Each runner's run is traced on a geographic map and their bio-stats plotted on colorful graphs. Runners can give each other *kudos*—a digital signal meant to deliver a rush of serotonin in the receiving runner's brain—for runs that they have done. They can—optionally—comment—optionally—motivating messages on runner's runs, and they can tag other runners who ran with them a run.

Digitally, runs and the act of running is encoded in multiple format. The most popular format is the open GPS Exchange Format (GPX) [85] which is an extension of the XML file format [72] that is meant to be human-readable[10]. Most electronic tracking devices, colloquially smart devices, that record runs do so in a GPX format. The GPX file format consist of multiple tracks (`<trk>`) each with its own name. Each track is composed of track segments (`<trkseg>`) defined by a sequence of points (`<trkpt>`) defined by their geo-coordinates (`lat` and `lon`) attributes and optional fields such as the elevation (`<ele>`). Below is a small example demonstrating the GPX file format:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<gpx creator="" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.topografix.com
    /GPX/1/1 http://www.topografix.com/GPX/1/1/gpx.xsd" version="1.1" xmlns="http://www.topografix.com/GPX/1/1">
  <trk>
    <name>Run, rabbit, run. Dig that hole, forget the sun.</name>
    <type>running</type>
    <trkseg>
      <trkpt lat="51.53707" lon="-0.18343"><ele>37.0</ele></trkpt>
      <trkpt lat="51.53712" lon="-0.18333"><ele>37.0</ele></trkpt>
    </trkseg>
```

[8]Not to be confused with the prestigious *Science* scientific journal
[9]Readers in the last quarter of the twentieth century may be more familiar with that social phenomenon where every activity has its own glossy magazine.
[10]Where human is not well defined.

```
  </trk>
</gpx>
```

2.2 λ-calculus

The λ-calculus has been called the "smallest programming language" by some[whom?]. It was introduced by Alonzo Church in 1932 [14] as a new foundational theory of logic and mathematics. Concurrently, Alan Turing introduced in 1936 the Turing Machine [87][11] to show that it is not possible to resolve Hilbert's 1928 *Entscheidungsproblem* that asks whether a general algorithm exists to prove any mathematical proposition. While Turing has broken into the mainstream, having his own movie—meaning a movie where he is portrayed rather than written or directed or produced by Turing[12]—published in 2014 starring the movie heartthrob and English sweetheart Benedict Cumberbatch as Turing himself [55], Church saw no such treatment.

Besides the λ-calculus, Church is commonly recognized among the nerd community through the Church-Turing thesis. Luckily for Church his name came first in the thesis' name because C, which happened to be the first letter of Church, comes earlier in the alphabet than T, which also happened to be the first letter in Turing [25]. Church was also first[13] in showing that the *Entscheidungsproblem* is not possible to solve [15].

2.2.1 Syntax. The λ-calculus is the mother of all functional languages. Its one and only feature are functions. What do we do with functions? (1) We create them, (2) we give their arguments name, and (3) we apply them to something, i.e. we replace names with other things.

So all *expressions* in the λ-calculus have one of the following three shapes.

1. *Abstraction*: Take an expression, find your favorite sub-expression, keep it in your pocket and replace it with a variable, then wrap the whole expression in a syntactic form that says that variable should stand for something.
2. *Variable*: That thing you replace your favorite sub-expression with,
3. *Application*: The way you say that a variable stands for your favorite sub-expression.

Formally though, expressions are denoted with the symbol $\mathtt{e}$ and they are described by a *grammar*, which is just the type of the syntax tree. Anyways, the λ-calculus expressions are described by this self-explanatory grammar:

$$\mathtt{e} ::= \lambda x.(\mathtt{e}) \mid x \mid (\mathtt{e}_1\,\mathtt{e}_2)$$

In practice, programmers are familiar with the λ-calculus if they have written in a Lisp-like dialect, Haskell, Scala, F#, or some other functional language. Below we address three common criticism against the λ-calculus.

ThErE aRe ToO mAnY pArEnThEsEs. No. You're used to calling functions like this, `f(x, g(y))`, right? The λ-calculus not only eliminates those useless commas, it gets rid of one pair of parenthesis! It simply moves the parentheses one token to the left to become $(\mathtt{f}\,\mathtt{x}\,(\mathtt{g}\,\mathtt{y}))$ and removes the outermost ones to become $\mathtt{f}\,\mathtt{x}\,(\mathtt{g}\,\mathtt{y})$. But wait, there's more. To eliminate more of these pesky parentheses we adopt two conventions: λ associate to the right and applications to the left. What does this mean? It means that $\lambda x.(\lambda y.\mathtt{e})$ becomes $\lambda x.\lambda y.\mathtt{e}$ and $(\mathtt{e}_1\,\mathtt{e}_2)\,\mathtt{e}_3$ becomes $\mathtt{e}_1\,\mathtt{e}_2\,\mathtt{e}_3$. For example: $\lambda x.\lambda y.x\,x\,y\,y$ stands for $\lambda x.(\lambda y.(((x\,x)\,y)\,y))$. Neat, no?

[11]Turing did not call the Turing Machine the Turing Machine because Turing was humble, it was Church who dubbed Turing's machine: Turing Machine.

[12]The author(s) are not aware of any movie that Turing wrote, directed, or produced

[13]The author(s) choose to conveniently sweep the Gödel debacle under the footnote line. Kurt Gödel, the archetypical Austrian—archetypical not being a qualifier for Austrian—nerd proved that the *Entscheidungsproblem* cannot be resolved in 1931 [34]. Everyone was aware of Gödel's result and it is acknowledged by Church and Turing.

I cAn'T pRoGrAm WiThOuT nUmBeRs, ClAsSeS, aNd ObJeCtS. You don't need them bro. You just need these three syntactic forms to write any program. Booleans? They're functions. Numbers? They're functions. Classes and objects? They're the poor man's closure [16]. Closures? They're just functions. In Section 4 we'll show you how you can do it too.

WhErE ArE ThE mUlTiVaRiAtE fUnCtIoNs? They don't exist. You know why? Cause they're useless. Just Schönfinkel them [67, 70] my dude. A function that takes two things is a function that takes the first thing and returns a function that expects the second thing. This fact can be recalled with this mnemonic rhyme:

Many arguments   are mere ornaments.
With higher-order   languages, ordure
as these can be teased   out with such an ease:
Make λs take one,   which when asked to run,
returns another   expecting the other,
until no more can arguments dwindle.
Congrats. Now you know how to Schönfinkel.

Or you could bundle your arguments in a pair. But guess what? Pairs are also just functions.

2.2.2 Semantics: What does it all mean? So now you know how to write λ programs. How do you run them? You use the relation $\mathsf{e} \longrightarrow \mathsf{e}'$ which can be read as "e becomes e' after a single step". This style is conveniently called small-step semantics because it invites the evaluator to behave like a CPU and to operate in discrete units commonly known as clock-cycles.

The $\longrightarrow$ relation is defined through a single rule: the β rule. That's the only[14] rule[15] you will ever need. Do you see now why it's the "smallest programming language"? The β-rule looks like this:

$$(\lambda x.\mathsf{e}_1)\ \mathsf{e}_2 \longrightarrow \mathsf{e}_1|^x_{\mathsf{e}_2} \tag{β}$$

Where the notation $\mathsf{e}_1|^x_{\mathsf{e}_2}$ is the capture-avoiding substitution [77]. We could define it, or we could do what every other programming language does and show you examples.

$$(add\ x\ d)|^x_{add\ a\ d} = add\ (add\ a\ d)\ d$$

$$(\lambda y.sub\ x\ b)|^x_{sub\ u\ b} = \lambda y.sub\ (sub\ u\ b)\ b$$

$$(\lambda x.foo\ x)|^x_{\chi} = \lambda x.foo\ x$$

This again, is all you need to do any computation. In other words, this model: the three syntactic forms and the β rule is all you need to write Linux and Clang which you can then use to program a simulator of a Turing Machine. However unlike Linux, the λ-calculus does not require files, and unlike Clang it's not developed by a group, and unlike Turing Machines there are no tapes. What's common across them? The state. Church, by inventing the λ-calculus, stated computationally and constructively the separation between Church and state[16].

[14]That's only one if you consider—which the author(s) do—the weak-head normal form evaluation strategy. You can ignore these words, it's beyond the scope of this paper.
[15]That's not really true. The first giveaway is the rule's name. You would think that if there's a β, then there's an α. This is also not entirely true as there is also an η. The α rule just renames some variables, which you might need to do in some cases when free—as in freedom, not as in free beer—variables are involved. The η rule is at best a compiler optimization.
[16]This joke is paraphrased from Guy Steele: "And some people prefer not to commingle the functional, lambda-calculus part of a language with the parts that do side effects. It seems they believe in the separation of Church and state. :-) :-) :-)" [76]

Variables Begone! In 1972, the iconic Dutch composer Louis Andriessen composed de Volharding [4, 8][17] which lead to the creation of the eponymous minimal jazz group de Volharding [24]. This event signaled the climax of the minimalism movement in Europe [9]. In 1972, the Dutch computer scientist Edsger W. Dijkstra delivered EWD340 as his Turing Award acceptance speech [26] in which he announced that he was the first Dutch to register as programmer. In 1972, the Dutch mathematician Nicolaas Govert de Bruijn published *Lambda Calculus Notation with Nameless Dummies* [21] which combined both minimalism, programming, and signaled the end of logical minimalism and computer design [23]. With his seminal paper, the visionary de Bruijn foresaw and solved half of the problem now attributed to Phil Karlton: "There are only two hard things in Computer Science: cache invalidation and naming things." [22, 30, 40]. By completely removing the need for variables in the λ-calculus de Bruijn resolved the latter problem [3] back in the 70s.

How did he do it? de Bruijn observed that in programs without free—as in freedom—variables variables are just "pointers" to wherever they were bound or declared. For example, $\lambda x.\lambda y.x$ can be said to be the lambda that returns a lambda which returns the variable bound by the lambda *two* levels before. Coincidentally, $\lambda a.\lambda b.a$ is the same lambda that returns a lambda which returns the variable bound by the lambda two levels before. So in de Bruijn notation we express them as $\lambda\lambda 2$. This notation is dubbed de Bruijn indices[18] and is the one we will use throughout the paper.

To get the reader used to this notation, we will list some lambda expressions and their de Bruijn indices in the following. The identity function $\lambda x.x$ becomes $\lambda.1$. The constant function $\lambda x.\lambda y.x$ becomes $\lambda\lambda.2$. The application function $\lambda f.\lambda x.f\ x$ becomes $\lambda\lambda.2\ 1$. This function $\lambda x.(\lambda f.x)\ x$ becomes $\lambda.(\lambda.2)\ 1$.

3 THE DESIGN OF POOLOLOOP

Poololoop, pronounced [puːləʊluːp], is a portmanteau [88] of "Pool of loop".

The main construct that Poololoop exploits for its syntax is the *loop*, or that which is colloquially called a *circle* by the general population who failed their basic geometry class and forgot that a circle has a constant radius. This choice is motivated by three reasons.

First, humans naturally walk around in circles [74], thus the user does not need to concentrate on the syntax and can rather spend their energy on the problem at hand. A feature that hardly any modern-day programming language enjoys.

Second, The programs naturally become small in diameter. The user does not need to stray far away from home and venture into foreign environments. This has the positive outcome of avoiding taking any unneeded risk that may trigger separation anxiety in users.

Third, loops make Poololoop future proof. Mastroianni et. al [52] and others [47] proved that the past was better. It thus follows that the function describing the quality of time is a monotonously decreasing function. The trivial corollary states that the future will be worse. Of the multiple proposed models describing the future, two dominate [44]: the Orwellian [59] and the Huxlerian [37]. In what follows we argue that Poololoop fits snugly in both models.

The Orwellian model predicts a general increase in user monitoring and language moderation by lifting the principle of least privilege [69] from software development into efficient social organization. In this realm, a successful language must cater not only to its users but also to Big Brother. Proponents of the Orwellian model justify it by offering the following argument. In 2023, 31.17% (N=59,[19]336) of software developers reported working for companies with a workforce of larger than 500 employees [75]. The law

[17]Dutch for perseverance. The author(s) recommend that the reader play this composition and read the remainder of the paper while listening to it.
[18]Not to be confused with de Bruijn levels
[19]For the editors from continental western Europe: this comma is a thousands separator and not a decimal separator. I'm sorry that your language is not so relevant scientifically anymore.

of large numbers—and one may not need to invoke this law to make a strong argument—implies that many programmers will have to work under surveillance-friendly conditions [2, 10, 38]. In this setting Big Brother subordinates shall not employ Poololoop as prescribed as that may give the wrong impression that the now-healthy workforce, coming back from a hard day's work in Nature[20] [41, 57, 58, 61] using Poololoop, showing signs of happiness and good mental health may be confused with an idle workforce loitering around the coffee machine all day. Thus a more consistent employment of Poololoop in the Orwellian model is one where the programmers are lead in file to the underground parking lots, away from sunlight and under the infrared glow of night-vision-enabled surveillance cameras, to write their programs in the oppressing underground stale air. In such a space, without exploiting loops, the space of programs that can be expressed becomes too limited to be useful.

The Huxlerian model predicts an intoxicating increase in developer tooling with excellent user experience and a deluge of mind-altering technologies[21] that will boost the productivity of programmers. In that model, for programmers to use a tool, it must be an addictive one. Utilizing Poololoop in the great outdoor will help greatly in improving the mental state of the programmer. Thus, an excess use of Poololoop has the benefit of increasing the user engagement. Moreover, it has been shown that runners and walkers in Nature do feel a sense of connection with natural entities and a sense of yearning to revisit them [12, 29, 65]. Hence, a use of Poololoop naturally leads to an excessive use which leads to unprecedented levels of programmer satisfaction, which according to the Huxlerian model, must lead to an increase in output and productivity.

3.1 Syntax

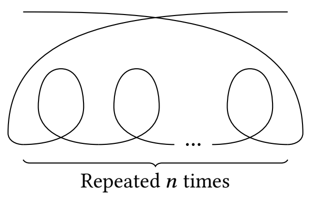
Poololoop is an alternative front-end to the λ-calculus with de Bruijn indeces as presented in Section 2.2. To that extend, we present the front-end to the three syntactic forms of the untyped λ-calculus: variables, functions, and function applications.

We start by distinguishing between the language Poololoop and the formal mathematical system ♉ underlying the language. The equivalent of the expressions e of the λ-calculus are the loops ɬ of ♉. To define the language we define in Section 3.1 the loop-encoding of e, $(\!|e|\!) : \lambda \longrightarrow$ ♉, which translates expressions into loops. The actual definition is by cases on the λ-calculus syntactic forms and is split in Definitions 3.1 to 3.3. And in Section 3.2 we define the untangling of ɬ, $[\![$ɬ$]\!]$: ♉ $\longrightarrow \lambda$, particularly in Definition 3.4, which translates loops back into expressions.

3.1.1 Variables. When we use de Bruijn indeces, variables will always be represented as natural numbers[22]. It is worth noting that we do not consider zero to be a natural number. Therefore quite naturally we denote a variable n by n consecutive empty loops.

Definition 3.1 (Variables). A variable n is loop-encoded as the loop wrapping n empty loops as follows:

$(\!|n|\!) \quad =$



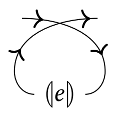
[20]Not to be confused with the prestigious *Nature* scientific journal.
[21]Exemplified by conversing with computers that pass the Turing test.
[22]In 1889 Giuseppe Peano published *Arithmetices Principa* [62] in which he presents the defacto agreed-upon axiomatization of the natural numbers. The first axiom is verbatim $1 \in N$ (sic). Implying that zero is not a natural number. However, in his 1901 *Formulario mathematico* [63] he realizes his mistake and includes zero as a natural number and the first axiom of his formalism

3.1.2 Functions and Applications; Introduction and Elimination; Yin and Yang. Functions are non-empty loops that contain the loop-encoding of their body and applications are non-empty loops that contain the components of the application. This creates a lovely correspondence between the syntactic class of an expression and the number of elements in its outer loop: nothing is a variable, one thing is a function, and many things is an application. Sadly we can't have nice things, so we introduce the directionality of the loop. The direction of the loop is formally the direction in which the loop turns. In practice this corresponds to time, and since loop-encoding is not a distributed system then time is a well-defined pre-order lattice with a complete total ordering [43]. Since introduction, or building, generally has a positive connotation and elimination, or destruction, has a negative one then a functions' loop is positive and an application's loop is negative.

To that end, functions and applications are formally defined as follows:

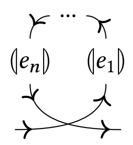
Definition 3.2 (Function). A function whose body is e is defined to be a positive (clock-wise) loop wrapping the loop-encoding of e.

$$(\!|\lambda.e|\!) \quad =$$



Definition 3.3 (Function Applications). A function applications whose function is e_1 and arguments are $e_2 \cdots e_n$ is a negative (clock-wise) loop wrapping the loop-encoding of all its components.

$$(\!|e_1 \cdots e_n|\!) \quad =$$



3.1.3 A Note on the Choice of Direction. The directionality of loops came to the author(s) as they showered [18, 39, 60]. While thinking about the loop-encoding of lambdas, the author(s) were observing the little water tornadoes that the water did as it swirled down the drain and after a Eureka moment they assigned the positive direction to lambdas. As the author(s) live in the northern hemisphere and wish to avoid exhibiting any north-south bias, we flip the directionality of functions and applications based on whether the program was ran in the northern hemisphere or the southern one. This hemispherical distinction in directionality has been introduced by the French mathematician Gaspard-Gustave de Coriolis [17] and popularized by Archer, Oakley and Weinstein [83]. As runs could start, end, and cross the equator, we leave the question of deciding the choice of directionality for future work.

3.2 Formal Semantics

Formal semantics in the theory of programming languages are split into two parts: static semantics, fancy for semantics at compile-time, think type systems, and dynamic semantics or runtime behavior. As Poololoop is untyped then static semantics are not relevant here. And since Poololoop is a front-end for the untyped λ-calculus then its dynamic semantics are exactly those of the λ-calculus with de Bruijn indeces, i.e. the weak-head normal form evaluation strategy with the β rule in Equation (β). This evaluation strategy is functional, i.e. given a reducible expression e there is a unique e' such that $e \longrightarrow e'$.

This observation motivates the colimit commutative diagram[23] definition in Figure 1 for the functional evalution $\rightsquigarrow$ at the ➰-level and the untangling $[\![⸸]\!]$ function, which in practice is the compilation relation.

becomes $0 \in N$. To those who define the natural numbers informally as the counting numbers: how many fingers am I holding up if I hold up my fist? It has been 123 years already since Peano's correction, so of course zero is a natural number.

[23]Which is otherwise quite useless.

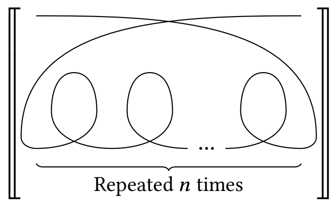
$$\begin{tikzcd}
℧ \arrow[r, dashed, "\rightsquigarrow"] & ℧ \arrow[d, dashed, "⟦\cdot⟧"'] \\
\lambda \arrow[u, "(\!|\cdot|\!)"'] \arrow[r, "\rightarrow"'] & \lambda
\end{tikzcd}$$

Fig. 1. The diagram describing the ℧ semantics with respect to λ semantics

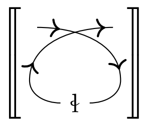
For the readers who are not versed in diagram notations, we simply mean the following: given the encoding function that "decompiles" and the underlying runtime that makes one step then it is possible to define (uniquely) the compiler and the interpreter (higher-level runtime) such that $e \longrightarrow ⟦(\!|e|\!) \rightsquigarrow⟧$.

In Definition 3.4 we define the compilation function of a loop ɬ into a λ-calculus expression.

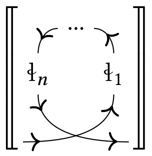
Definition 3.4 (Compilation). We define the compilation of a loop ɬ in ℧ by cases:



$= \quad n$



$= \quad \lambda.\, ⟦ɬ⟧$



$= \quad ⟦ɬ_1⟧ \cdots ⟦ɬ_n⟧$

With Definitions 3.1 to 3.4 we can formulate Theorem 3.5 which expresses the expected fact that untangling and loop-encoding are inverse operations.

THEOREM 3.5. *Loop-encoding and untangling are inverse operations. In other words, for every loop* ɬ *in* ℧ *then* $ɬ = (\!|⟦ɬ⟧|\!)$, *and for every λ-calculus expression e then* $e = ⟦(\!|e|\!)⟧$.

PROOF. Like most theorems in the domain of programming languages [5, 13, 45], the proof is a trivial application of structural induction. □

Defining the interpreter $\rightsquigarrow$ is also not difficult but requires the author(s) to typeset many complicated diagrams.

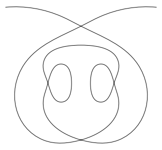
Definition 3.6 (The $\rightsquigarrow$ interpreter). The definition is left as an exercise to the reader.

4 PROGRAMMING IN POOLOLOOP

4.1 Church Booleans

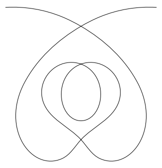
To construct a boolean one of the two boolean constructors must be used: true or false. Thus the church encodings of booleans will always have two outer lambdas, one for each constructor.

True. The boolean true value is encoded as $\lambda\lambda.2$. Informally, the first lambda asks its user for what is meant by true, and the second asks for what is meant by false. The expression then returns the true value provided by the user. Its loop-encoding in ℧ is the following:



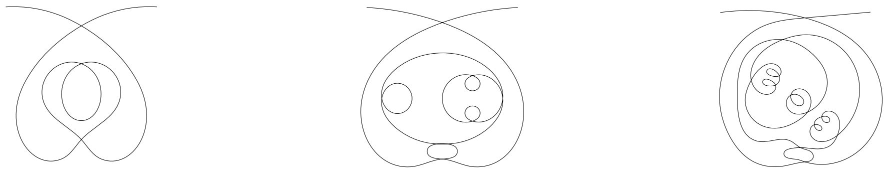
All loop examples are to be read from the top-left corner, traveling along the entire loop continuously without breaking smoothness, until the top-right corner is reached. The inhabitants of the southern-hemisphere are instead expected to read the diagram from the top-right corner towards the top-left corner in a similar fashion.

False. Dually, the boolean false will return the false that the user provided, i.e. false is $\lambda\lambda.1$. Its loop-encoding is the following:



4.2 Church Numerals

4.2.1 Numbers. The Peano encoding of the natural numbers [63] assume two constructors: the zero[24] and the successor function. The zero is then encoded as $\lambda\lambda.1$[25], one is encoded as $\lambda\lambda.2\;1$, two as $\lambda\lambda.2\;(2\;1)$, three as $\lambda\lambda.2\;(2\;(2\;1))$, etc. In ℧ we illustrate 0, 1, and 2 as the following diagrams in order from left-to-right:



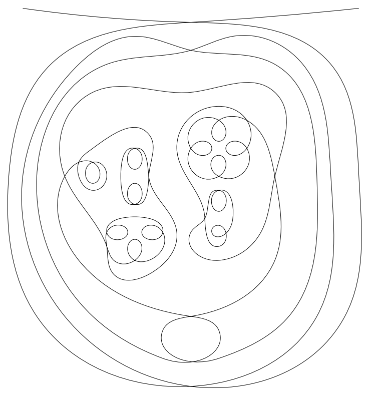
4.2.2 Addition. Addition is a fundamental operation [28]. The operation takes two Church numbers and produces a Church number. Therefore it has two outer-most lambdas for the given numbers, and two other lambdas for the zero and the successor function. A number *n* is encoded as the application of the given successor function *n* times to the given zero. Thus, addition of *n* and *m* is encoded as the application of the given successor function *n* times to *m* such that the given zero and successor functions are passed along to *m*. In other words, the de Bruijn encoding of addition is

$$\mathsf{add} = \lambda\lambda\lambda\lambda.4\;2\;(3\;2\;1) \tag{1}$$

Whose loop encoding is the following diagram:

[24]Here zero is a natural number. This is not be confused with the definition of the natural numbers at the meta-level.

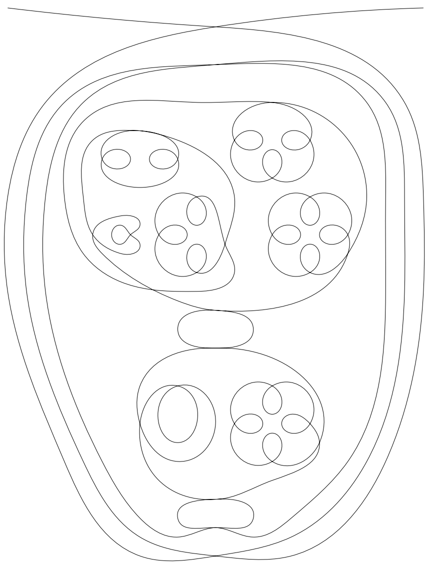
[25]The attentive reader would have noticed that the encoding of zero and false are the same. This feature has been added to Church encodings in order to attract C and Javascript programmers into functional programming.



4.2.3 Multiplication. Just as we re-interpreted the meaning of zero in the addition function to be the second number to be added, for addition we re-interpret the successor function to addition. Therefore multiplying *n* and *m* becomes *n* additions of *m* on a given zero. Thus, by inlining Equation (1), we obtain the following definition of multiplication

$$\texttt{mult} = \lambda\lambda\lambda\lambda.4\ (\lambda.4\ 3\ (1\ 3\ 2))\ 1 \tag{2}$$

Its loop encoding is the following diagram:

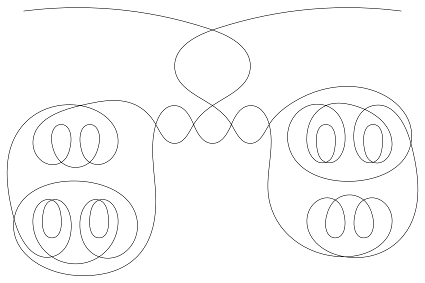


4.3 Fixed Point Operators

The Y-combinator allows recursive and diverging programs to be expressed. It is defined as follows:

$$\texttt{Y} = \lambda.(\lambda.2\ (1\ 1))\ (\lambda.2\ (1\ 1)) \tag{3}$$

and its loop encoding is the following:



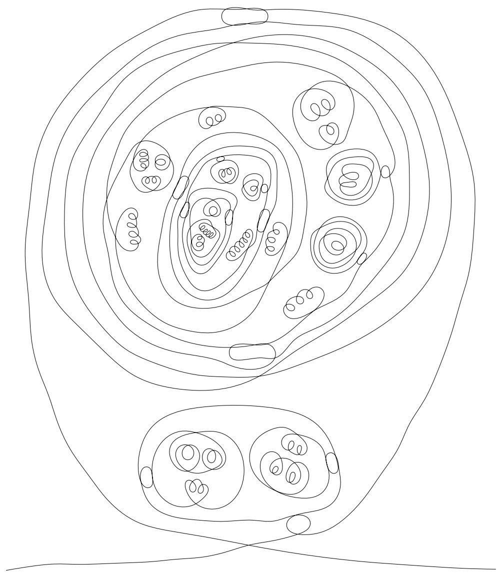
4.3.1 Recursive Programs: factorial. Now we demonstrate that Poololoop is not a toy programming language by implementing the factorial function. We implement it using the Y-combinator defined in Equation (3) and the church numerals. Below is the program in the λ-calculus without de Bruijn indices:

```
fact = (\f. (\x. f (x x)) (\x. f (x x)))               -- Y-Combinator Equation (3)
       \recurse.\n.\s.\z.                               -- First argument is the recursive call
         n (\_.\T.\F. F) (\T.\F. T)                     -- is true if n is zero and false otherwise
           (s z)                                        -- 0! = 1
           (recurse                                     -- recursive case
             (n \s.\z. n (\g.\h. h (g s)) (\u. z) (\u. u))  -- MAGIC: predecessor of n
             s (n s z))                                 -- the last argument recalls Equation (1)
```

Using de Bruijn incides, `fact` becomes:

$$(\lambda.(\lambda.2\,(1\,1))\,(\lambda.2\,(1\,1)))\,(\lambda\lambda\lambda\lambda.3\,(\lambda\lambda\lambda.1)\,(\lambda\lambda.2)\,(2\,1)\,(4\,(3\,\lambda\lambda.5\,(\lambda\lambda.1\,(2\,4))\,(\lambda.2)\,(\lambda.1))\,2\,(3\,2\,1)))$$

And its loop encoding is:



This particular diagram is meant to be read from the bottom-left strand, all the way throughout the path, until the end of the bottom-right strand. Godspeed.

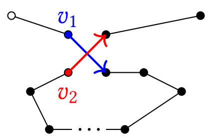


Fig. 2. A loop is defined by the intersection point of two vectors v_1 (blue) and v_2 (red) and its—the loop's—direction is defined by the same two vectors. The starting point is the empty point in the top-left corner.

5 IMPLEMENTATION

We implemented a compiler for Poololoop in the C programming language[26]. The source code, which is relatively tiny at 147 lines long is provided in Appendix A[27]. The compiler takes two command line arguments, and an optional third. The first argument is the path to a GPX file and the second is an identifier that the compiled code should be assigned to. The optional third argument specifies the target language to compile to. By default the target language is the Scheme programming language. The other alternative language is Haskell. The two targets are specified with `scm` and `hs` respectively.

The semantics implemented by the compiler is as described in Section 3. The implementation decides on the directionality based on whether the starting point has positive or negative latitude, i.e. is in the northern or southern hemisphere respectively.

The implementation abides by Postel's Robustness law [66], it accepts file formats that supersede GPX. Informally, Poololoop's compiler accepts any file that contains a sequence of latitude and longitude coordinates specified respectively with `lat="ieee_float"` and `lon="ieee_float"`. The syntactic class `ieee_float` is the class of IEEE floats.

Compiling the compiler is as easy as passing it to `gcc` and linking it with the math library using the `-lm` flag. When the compiler is compiled with the `-O3` optimization flag, then compiling the two case studies presented in Section 6 took less than 3 milliseconds on the author(s)' machine which is just an everyday laptop that one takes on a holiday. The GPX file of the two case studies contains 509 and 519 GPS points respectively. Section 6 provides the executed commands and their output.

The main observation is that every loop is defined by an intersection point. Thus, find the intersection point and you will find the loop. The direction of the loop is computed from two vectors: (1) the vector whose tail and tip are defined by the point just before the intersection point and the one just after, respectively, and (2) the vector whose tail and tip are defined by the point just before the intersection point and the one just after, respectively. Figure 2 shows an example of these two vectors, v_1 and v_2, respectively.

The compiler exploits the key idea that intersection points act as parentheses. Every intersection point is traversed twice by the runner, on entering the loop and on exiting it. Thus, the compiler finds the intersection points, sorts them by the time traversed, and treats each pair as a parenthesis: the first is equivalent to an open parenthesis and the second is equivalent to a closed one. Once this Intermediate Representation (IR) is generated, then compiling to a de Bruijn indexed λ-calculus IR is equally trivial, and compiling to Scheme or Haskell is just a boring task at this point.

The compiler uses the naïve $O(n^2)$ algorithm to find intersections in a piece-wise linear path: it checks segments two-by-two for intersections. When tried on a GPX file with 13,714 GPS points the compiler took 291 milliseconds to produce a syntax error[28].

[26]C11, probably.
[27]And also as a Gitlab Snippet: https://gitlab.com/-/snippets/3688034.
[28]Which is only raised after finding all intersections.

6 EVALUATION

In this section we show that using Poololoop in real-life is possible. We have chosen two programs from Section 4 and we ran one program on a large-scale and walked the other on a small-scale. We report on these two case studies in the following paragraphs.

The first author went on run along the *true* path on the night of Saturday 9th of March 2024. The GPX track recorded by the author's smart watch is shown in Figure 3a. The running distance was 4.48 kilometers long [29] at the respectable pace of 5 minutes and 35 seconds per kilometer[30] and the running time was around 25 minutes and 5 seconds, which is almost exactly the same length as de Volharding[31]. The engine running the program was indeed exposed to that composition as they were running the program. Therefore one thread of validity to this case study is the choice of music as it might have affected the pace of the runner and thus the measured run time [11].

The last author's *fix* was also done on the night of Saturday 9th March 2024. The path is 2.66 kilometers long. The author took 46 minutes and 42 seconds to walk it at the shameful pace of 17 minutes and 33 seconds per kilometer. The GPX track recorded by the author's smart watch is shown in Figure 3b. The slow pace can be explained by the author's report that conducting this experiment was tedious and awkward. They have in fact tried to walk the path in daylight but gave up soon after the first loop as they have reported feeling an uncomfortable level of awareness exhibiting itself through emotions of self-consciousness and fear of walking into small and developing humans engaging in a communal and recreational activity consisting of striking, repeatedly, with a single foot, a round orb thus carrying it off the ground and potentially into the faces of others—author included. While the author was not disturbed during the experiment they have received inquisitive looks—even under the cloak of darkness—from strangers. Thus one thread of validity to this case study are other humans.

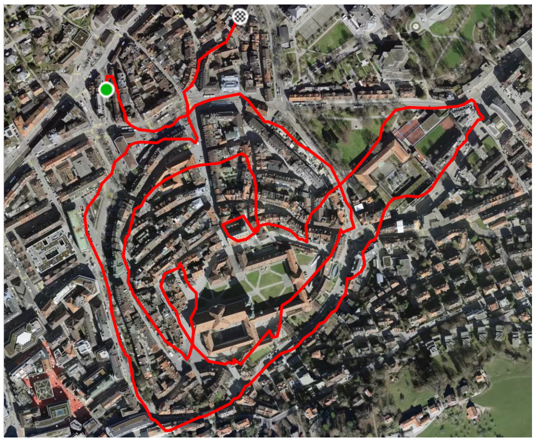
Both case studies have been recorded and archived on Strava [31, 78], a popular website where athletes preserve their artifacts. The tracks are publically accessible[32] alongside multiple plots and a GPX file which can be fed into the Poololoop compiler.

Below are the benchmarks of the four compilations that the author(s) conduced on the GPX files of both case studies as produced by the `hyperfine` benchmark utility [64].

```
> hyperfine "./poololoop gpx/fix.gpx f hs" "./poololoop gpx/fix.gpx f scm"  \
            "./poololoop gpx/true.gpx t hs" "./poololoop gpx/true.gpx t scm"

Benchmark 1: ./poololoop gpx/fix.gpx f hs
  Time (mean ± σ):      2.0 ms ±   0.7 ms    [User: 1.8 ms, System: 0.4 ms]

Benchmark 2: ./poololoop gpx/fix.gpx f scm
  Time (mean ± σ):      1.5 ms ±   1.0 ms    [User: 1.5 ms, System: 0.2 ms]

Benchmark 3: ./poololoop gpx/true.gpx t hs
  Time (mean ± σ):      1.7 ms ±   0.8 ms    [User: 1.6 ms, System: 0.3 ms]

Benchmark 4: ./poololoop gpx/true.gpx t scm
  Time (mean ± σ):      1.7 ms ±   0.8 ms    [User: 1.6 ms, System: 0.3 ms]
```

The output of each command is the following:

[29]Or 2.78 miles in freedom units.
[30]Or 8 minutes 57 seconds per mile in freedom units.
[31]Which should be just about done if you started it per Footnote 17.
[32]*λλ2 - SIGBOVIK'24 CS#2*: https://www.strava.com/activities/10926312841, and *λ(λ2(1 1))(λ2(1 1)) - SIGBOVIK'24 CS#1*: https://www.strava.com/activities/10925302747. By appending `/export_gpx` to the URL, the artifact evaluators can download the GPX file.


(a) The *true* program as described in Section 4.1

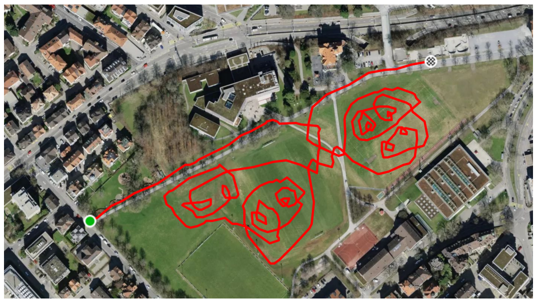

(b) The Y-combinator as described in Section 4.3

Fig. 3. A satellite image of the area where the two programs of the two case studies were ran and walked with the path superposed on top in a red line. The starting point of the run and walk is indicated by a green dot. The ending point of the run and walk is indicated by a checkered flag in a white circle.

```
> ./poololoop gpx/fix.gpx f hs
f = (\ x0 -> ((\ x1 -> (x0 (x1 x1))) (\ x1 -> (x0 (x1 x1)))))

> ./poololoop gpx/fix.gpx f scm
(define f (lambda (x0) ((lambda (x1) (x0 (x1 x1))) (lambda (x1) (x0 (x1 x1))))))

> ./poololoop gpx/true.gpx t hs
t = (\ x0 -> (\ x1 -> x0))

> ./poololoop gpx/true.gpx t scm
(define t (lambda (x0) (lambda (x1) x0)))
```

We would like to remind the artifact evaluators that the generated Haskell code of the Y-combinator cannot be used even though it is syntactically correct. That is because Haskell is strongly typed and the Y-Combinator cannot be typed in the Simply Typed λ-calculus, nor in System-F. That is no fault of ours. The generated Scheme code on the other hand is perfectly fine.

7 RELATED WORKS

As with most novel work such as ours, not much truly related related work exists. Two line of works nonetheless can be identified. The author(s) hope that by the end of this section the reader would have realized that none of these completely satisfy Poololoop's design choices from Section 3.

7.1 Alternative or Assistive Hardware

Augmented Reality (AR) devices which intend to superpose virtual objects on top of real-world objectives have been around commercially for more than a decade. These could be used to assist programmers in writing their programs in the great outdoors just as Poololoop does.

The first device that broke into the mainstream is the Google Glass in the early 2010s. It is a wearable device, just a pair of eyeglasses, with two small transparent glass rectangles covering a part of the user's field of vision. By projecting pixels onto these transparent rectangles the user can switch their focus from real-life into the virtual, and vice versa. But damn do they look dorky [90].

More recent devices, such as Facebook's Quest and Apple's VisionPro offer better image quality and hand-gesture detection. Yet these have two downsides. First, the real-world is seen through a screen looping back the view from a front-camera. Meaning these devices are virtual reality devices that happen to mirror the real-world—for now[33]. And second, they sure do look more obnoxious than the Google Glass.

7.2 Gestural Programming Languages

Gestural Programming is the domain of computer vision and artificial intelligence research where one teaches a robot how to accomplish tasks by demonstrating to the robot, visually, through the means of a human, how they are accomplished. This line of work has been explored by Soratana et al. [73] and Cabrera et al. [54]. But it's easy to conclude that this is unrelated to Poololoop and is not what we mean by "Gestural Programming".

Let's try again. Gestural Programming is the domain of programming pedagogy where researchers explore the use of input devices other than the keyboard and the mouse for programming. For example in Streeter's PhD thesis [79] the author[34] applied multiple gesture matching algorithms to data recorded from students programming in Google Blockly with the Microsoft Xbox Kinect. Similarly Toro-Guajardo et al. [86] reported on young people programming in Scratch using the Nintendo Switch Joy-Cons and found that they have more fun if they, and their hands, move. While this line of work is closer to Poololoop than the previous one, it is still unrelated.

One more time. Gestural Programming is the domain of programming language research which produced bodyfuck [35, 36, 82], a language in which programmers input programs through moving their bodies. Bodyfuck is an alternative front-end for the popular programming language brainfuck [56] where the eight brainfuck actions are mapped into eight bodily gestures the programmer performs facing a visual recording apparatus. For example, if one wishes to increment the register in focus then one must jump, and if one wishes to decrement it then one must duck. Bodyfuck has been birthed to be performative art. Fifteen programs were performed and put on display in the Things That Are Possible MFA Show [6]. It aims to separate the act of software performance, i.e. software inscription[35] from the computational context in which it happens in. Bodyfuck aims to demonstrate that software inscription can be done completely outside computers[36] in the surrounding cultural space. Similarly, Poololoop demonstrates that this inscription can be done completely outside.

This shows that we're on the right track of identifying actually related related work. Sadly, the author(s) could not find any other related work in that style. However, the early readers of the *SIGBOVIK* publication proceedings may recall the work of Leffert entitled "Harnessing Human Computation: β-reduction hero" [46] in the 2010 Technical Report track. Alas, the author(s) were unable to find the Flash application which was reported on nor the report nor its source code in order to learn what the work is about. Nonetheless we conjecture, based on the name, that it presents a λ-calculus evaluation technique that is gamified á-la Guitar Hero. Nevertheless if that is the case, then unlike Poololoop, programming can only be done indoors while staring at a screen.

[33]Seriously, imagine the horrors of having that feedback camera attacked.

[34]Being the author of Streeter's PhD thesis and not the author(s) of this very paper you are almost done reading.

[35]As the author(s) of this almost-finished paper have no background in academic art they have found the essay cited earlier to be extremely difficult to read, and they report here their best guess at what it could mean.

[36]Also around the year 2010, the first author of this almost-done paper that you are reading recalls writing a whole PHP program on a piece of paper during a biology class in their senior highschool year as they had no interest in biology and no access to a computer. This footnote tells the anecdote to show that this separation is natural and to brag that the first author was able to write a whole PHP application by hand on a piece of paper.

REFERENCES

[1] K. Mohamed Ali and B.W.C. Sathiyasekaran. 2006. Computer Professionals and Carpal Tunnel Syndrome (CTS). *International Journal of Occupational Safety and Ergonomics* 12, 3 (2006), 319–325. https://doi.org/10.1080/10803548.2006.11076691 arXiv:https://doi.org/10.1080/10803548.2006.11076691 PMID: 16984790.

[2] Seth Allcorn. 2022. Micromanagement in the workplace. *Organisational and Social Dynamics* 22, 1 (2022), 83–98.

[3] Reem Alsuhaibani, Christian Newman, Michael Decker, Michael Collard, and Jonathan Maletic. 2021. On the Naming of Methods: A Survey of Professional Developers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 587–599. https://doi.org/10.1109/ICSE43902.2021.00061

[4] Louis Andriessen. 1972. de Volharding. Composition.

[5] Carlo Angiuli. 2017. The Next 700 Type Systems. *SIGBOVIK* (2017), 169–171. https://sigbovik.org/2017/proceedings.pdf

[6] The Digital Arts and New Media MFA Program at UC Santa Cruz. 2010. *2010 MFA Exhibition :: Things That Are Possible.* https://danm.ucsc.edu/news_events/2010-mfa-exhibition Accessed: March 3, 2024.

[7] Herb Bailey. 2002. On running in the rain. *The College Mathematics Journal* 33, 2 (2002), 88–92.

[8] World Association For Symphonic Bands and Ensembles. 2021. *DE VORHALDING for Piano and Winds (1972) by Louis Andriessen (The Netherlands, 1939-2021).* https://wasbe.org/de-vorhalding-for-piano-and-winds-1972-by-louis-andriessen-the-netherlands-1939-2021 Accessed: March 3 2024.

[9] Maarten Beirens. 2016. European Minimalism and the Modernist Problem. In *The Ashgate Research Companion to Minimalist and Postminimalist Music*. Routledge, 61–85.

[10] Clive R Boddy. 2017. Psychopathic leadership a case study of a corporate psychopath CEO. *Journal of Business Ethics* 145, 1 (2017), 141–156.

[11] Robert Jan Bood, Marijn Nijssen, John Van Der Kamp, and Melvyn Roerdink. 2013. The power of auditory-motor synchronization in sports: enhancing running performance by coupling cadence with the right beats. *PloS one* 8, 8 (2013), e70758.

[12] Stefan Brené, Astrid Bjørnebekk, Elin Åberg, Aleksander A Mathé, Lars Olson, and Martin Werme. 2007. Running is rewarding and antidepressive. *Physiology & behavior* 92, 1-2 (2007), 136–140.

[13] Robert Chatley, Alastair Donaldson, and Alan Mycroft. 2019. *The Next 7000 Programming Languages*. Springer International Publishing, Cham, 250–282. https://doi.org/10.1007/978-3-319-91908-9_15

[14] Alonzo Church. 1932. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics* 33, 2 (1932), 346–366. http://www.jstor.org/stable/1968337

[15] Alonzo Church. 1936. A note on the Entscheidungsproblem. *The journal of symbolic logic* 1, 1 (1936), 40–41.

[16] C2 Wiki Community. 2022. *Closures and Objects Are Equivalent.* http://wiki.c2.com/?ClosuresAndObjectsAreEquivalent Accessed: March 3, 2024.

[17] Gaspard Gustave Coriolis. 1835. *Mémoire sur les équations du mouvement relatif des systèmes de corps.* Bachelier.

[18] Rebecca M Currano, Martin Steinert, Larry J Leifer, et al. 2011. Characterizing reflective practice in design–what about those ideas you get in the shower?. In *DS 68-7: Proceedings of the 18th International Conference on Engineering Design (ICED 11), Impacting Society through Engineering Design, Vol. 7: Human Behaviour in Design, Lyngby/Copenhagen, Denmark, 15.-19.08. 2011.* 374–383.

[19] Evans Data. 2023. *Worldwide Developer Population from 2016 to 2023.* https://www.statista.com/statistics/627312/worldwide-developer-population/ Accessed: March 3 2024.

[20] DataReportal, Meltwater, and We Are Social. 2024. *Internet and Social Media Users in the World 2024.* https://www.statista.com/statistics/617136/digital-population-worldwide/ Accessed: March 3 2024.

[21] Nicolaas Govert De Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes mathematicae (proceedings)*, Vol. 75. Elsevier, 381–392.

[22] Vitor De Mario, Golden Cuy, David Karlton, and Murven. 2014. Has Phil Karlton ever said "There are only two hard things in Computer Science: cache invalidation and naming things"? https://skeptics.stackexchange.com/questions/19836/has-phil-karlton-ever-said-there-are-only-two-hard-things-in-computer-science. Accessed March 3 2024.

[23] Liesbeth De Mol, Bullynck Maarten, and Edgar G Daylight. 2018. Less is more in the fifties: Encounters between logical minimalism and computer design during the 1950s. *IEEE Annals of the History of Computing* 40, 1 (2018), 19–45.

[24] Mark Delaere, Maarten Beirens, and Hilary Staples. 2004. Minimal music in the Low countries. *Tijdschrift van de Koninklijke Vereniging voor Nederlandse Muziekgeschiedenis* 1 (2004), 31–78.

[25] Oxford English Dictionary. 2022. *Dictionary.* Oxford University Press.

[26] Edsger W Dijkstra. 1972. The humble programmer. *Commun. ACM* 15, 10 (1972), 859–866.

[27] Postamate Editor. 2023. *Why Software Engineers Have Short Life Expectancy.* https://postamate.com/2023/08/why-software-engineers-have-short-life-expectancy/

[28] Mimi Engel, Amy Claessens, and Maida A Finch. 2013. Teaching students what they already know? The (mis) alignment between mathematics instructional content and student knowledge in kindergarten. *Educational Evaluation and Policy Analysis* 35, 2 (2013), 157–178.

[29] S Forster et al. 2009. The forest for leisure activities and tourism: a yearning for nature or sustainable development?(essay). *Schweizerische Zeitschrift für Forstwesen* 160, 7 (2009), 189–194.

[30] Martin Fowler. 2009. Two Hard Things. https://martinfowler.com/bliki/TwoHardThings.html. Accessed March 3 2024.

[31] Rob Franken, Hidde Bekhuis, and Jochem Tolsma. 2023. Kudos make you run! How runners influence each other on the online social network Strava. *Social Networks* 72 (2023), 151–164.

[32] Luke-Elizabeth Gartley. 2022. CLADISTICS ruined my life: intersections of fandom, internet memes, and public engagement with science. *Journal of Science Communication* 21, 5 (2022), Y01. https://doi.org/10.22323/2.21050401

[33] Gartner. 2021. *Global Shipments of Personal Computers from 2006 to 2021.* https://www.statista.com/statistics/273495/global-shipments-of-personal-computers-since-2006/ Accessed: March 3 2024.

[34] Kurt Gödel. 1931. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik* 38–38, 1 (Dec. 1931), 173–198. https://doi.org/10.1007/bf01700692

[35] Nik Hanselmann. 2010. *bodyfuck - gestural brainfuck interpreter (2010).* https://youtu.be/watch?v=ekjtZ85mA3I Accessed: March 3, 2024.

[36] Nik Hanselmann. 2010. *There is no hardware.* http://web.archive.org/web/20141205200435/http://www.nikhanselmann.com/public/etc/thesis/ Accessed: March 3 2024, Archived: December 5 2014.

[37] Aldous Huxley. 1932. *Brave New World.* Chatto & Windus.

[38] Feruzan Irani-Williams, Lori Tribble, Paige S Rutner, Constance Campbell, D Harrison McKnight, and Bill C Hardgrave. 2021. Just Let Me Do My Job! Exploring the Impact of Micromanagement on IT Professionals. *ACM SIGMIS Database: the DATABASE for Advances in Information Systems* 52, 3 (2021), 77–95.

[39] Zachary C Irving, Catherine McGrath, Lauren Flynn, Aaron Glasser, and Caitlin Mills. 2022. The shower effect: Mind wandering facilitates creative incubation during moderately engaging activities. *Psychology of Aesthetics, Creativity, and the Arts* (2022).

[40] David Karlton. 2017. Naming things is hard. https://www.karlton.org/2017/12/naming-things-hard/. Accessed March 3 2024.

[41] Mia Keinänen. 2016. Taking your mind for a walk: a qualitative investigation of walking and thinking among nine Norwegian academics. *Higher Education* 71 (2016), 593–605.

[42] Joseph B Keller. 2010. Ponytail motion. *SIAM J. Appl. Math.* 70, 7 (2010), 2667–2672.

[43] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. 179–196.

[44] John Lanchester. 2019. *Orwell v Huxley: whose dystopia are we living in today?* https://www.ft.com/content/aa8ac620-1818-11e9-b93e-f4351a53f1c3 Accessed: March 3, 2024.

[45] P. J. Landin. 1966. The next 700 programming languages. *Commun. ACM* 9, 3 (mar 1966), 157–166. https://doi.org/10.1145/365230.365257

[46] Akiva Leffert. 2010. Harnessing Human Computation: β-Reduction Hero. *SIGBOVIK Technical Report* (2010). https://sigbovik.org/tr/2010-001.html

[47] Luis Lugo, Sandra Stencel, John Green, Timothy S Shah, Brian J Grim, Gregory Smith, Robert Ruby, Allison Pond, Andrew Kohut, Paul Taylor, et al. 2006. Spirit and power: A 10-country survey of Pentecostals. In *The Pew Forum on Religion and Public Life*.

[48] Saul M Luria. 1969. Average age at death of scientists in various specialties. *Public Health Reports* 84, 7 (1969), 661.

[49] Marc Luy, Christian Wegner-Siegmundt, Angela Wiedemann, and Jeroen Spijker. 2015. Life Expectancy by Education, Income and Occupation in Germany: Estimations Using the Longitudinal Survival Method. *Comparative Population Studies* 40, 4 (Dec. 2015). https://doi.org/10.12765/CPoS-2015-16

[50] L Mahadevan. 2012. And the Ig Nobel Goes to... Joseph B. Keller. *SIAM News* 45, 10 (2012).

[51] Abraham Harold Maslow. 1966. The psychology of science: A reconnaissance. (1966).

[52] Adam M Mastroianni and Daniel T Gilbert. 2023. The illusion of moral decline. *Nature* 618, 7966 (2023), 782–789.

[53] Jim McCann. 2015. Comment: SIGBOVIK Should Ban Conclusions. *SIGBOVIK* (2015), 83–84. https://sigbovik.org/2015/proceedings.pdf

[54] Cabrera M.E., Sanchez-Tamayo N., R. Voyles, and J.P. Wachs. 2017. One-Shot Gesture Recognition: One Step Towards Adaptive Learning. *12th IEEE International Conference on Automatic Face & Gesture Recognition* (2017).

[55] Benedict Cumberbatch Morten Tyldum. 2014. The Imitation Game. Film.

[56] Urban Müller. 1993. *Brainfuck*. http://esoteric.voxelperfect.net/wiki/Brainfuck Accessed: March 3, 2024.

[57] Gunnthora Olafsdottir, Paul Cloke, André Schulz, Zoé Van Dyck, Thor Eysteinsson, Björg Thorleifsdottir, and Claus Vögele. 2020. Health benefits of walking in nature: A randomized controlled study under conditions of real-life stress. *Environment and Behavior* 52, 3 (2020), 248–274.

[58] Marily Oppezzo and Daniel L Schwartz. 2014. Give your ideas some legs: the positive effect of walking on creative thinking. *Journal of experimental psychology: learning, memory, and cognition* 40, 4 (2014), 1142.

[59] George Orwell. 1949. *Nineteen Eighty-Four*. Secker & Warburg.

[60] Linda A Ovington, Anthony J Saliba, Carmen C Moran, Jeremy Goldring, and Jasmine B MacDonald. 2018. Do people really have insights in the shower? The when, where and who of the Aha! Moment. *The Journal of Creative Behavior* 52, 1 (2018), 21–34.

[61] James Patience, Ka Sing Paris Lai, Elizabeth Russell, Akshya Vasudev, Manuel Montero-Odasso, and Amer M Burhan. 2019. Relationship between mood, thinking, and walking: a systematic review examining depressive symptoms, executive function, and gait. *The American Journal of Geriatric Psychiatry* 27, 12 (2019), 1375–1383.

[62] Giuseppe Peano. 1889. *Arithmetices principia: Nova methodo exposita*. Fratres Bocca.

[63] Giuseppe Peano. 1901. Formulario mathematico. *Revue de Métaphysique et de Morale* 14, 3 (1901).

[64] David Peter. 2023. *hyperfine*. https://github.com/sharkdp/hyperfine

[65] Darcy C Plymire. 2004. Positive addiction: running and human potential in the 1970s. *Journal of Sport History* 31, 3 (2004), 297–315.

[66] J. Postel. 1980. *DoD standard Transmission Control Protocol*. https://doi.org/10.17487/rfc0761

[67] John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, Massachusetts, USA) *(ACM '72)*. Association for Computing Machinery, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

[68] S Pressman Roger and R Maxin Bruce. 2015. *Software engineering: a practitioner's approach*. McGraw-Hill Education.

[69] Fred B Schneider. 2003. Least privilege and more [computer security]. *IEEE Security & Privacy* 1, 5 (2003), 55–59.

[70] M. Schönfinkel. 1924. Über die Bausteine der mathematischen Logik. *Math. Ann.* 92, 3–4 (Sept. 1924), 305–316. https://doi.org/10.1007/bf01448013

[71] David J Shonk and James F Weiner. 2021. *Sales and revenue generation in sport business*. Human Kinetics.

[72] Jérôme Siméon and Philip Wadler. 2003. The essence of XML. *ACM Sigplan Notices* 38, 1 (2003), 1–13.

[73] T. Soratana, M.V.S.M. Balakuntala, P. Abbaraju, R. Voyles, J. Wachs, and M. Mahoor. 2018. Glovebox Handling of High-Consequence Materials with Super Baxter and Gesture-Based Programming. *44th International Symposium on Waste Management* (2018).

[74] Jan L. Souman, Ilja Frissen, Manish N. Sreenivasa, and Marc O. Ernst. 2009. Walking Straight into Circles. *Current Biology* 19, 18 (2009), 1538–1542. https://doi.org/10.1016/j.cub.2009.07.053

[75] Stack Overflow. 2023. *Stack Overflow Developer Survey 2023*. Stack Overflow. https://survey.stackoverflow.co/2023/#work-company-info Accessed: March 3, 2024.

[76] Guy L Steele Jr. 2001. *Re: need for macros (was Re: Icon)*. https://people.csail.mit.edu/gregs/ll1-discuss-archive-html/msg01134.html Accessed: March 3, 2024.

[77] Guy L Steele Jr. 2017. It's Time for a New Old Language.. In *PPoPP*.

[78] Strava. 2024. *Strava*. https://www.strava.com/ Accessed: March 3 2024.

[79] Lora Streeter. 2019. Teaching Introductory Programming Concepts through a Gesture-Based Interface. *Theses and Dissertations* (2019).

[80] K Suparna, AK Sharma, and J Khandekar. 2005. Occupational health problems and role of ergonomics in information technology professionals in national capital region. *Indian Journal of Occupational and Environmental Medicine* 9, 3 (2005), 111–114.

[81] Richa Talwar, Rohit Kapoor, Karan Puri, Kapil Bansal, and Saudan Singh. 2009. A study of visual and musculoskeletal health disorders among computer professionals in NCR Delhi. *Indian J. Community Med.* 34, 4 (Oct. 2009), 326–328.

[82] Daniel Temkin. 2015. *BodyFuck - esoteric.codes*. https://esoteric.codes/blog/bodyfuck-gestural-code Accessed: March 3, 2024.

[83] The Simpsons. 1995. Bart vs. Australia. Television series. https://simpsons.fandom.com/wiki/Bart_vs._Australia Season 6, Episode 16.

[84] Time Team Official. 2011. *Phil's Pub Review*. https://youtu.be/watch?v=61lfmiAMC84 Accessed: March 3 2024.

[85] Topografix. 2004. *GPX (GPS Exchange Format) Version 1.1*. http://www.topografix.com/GPX/1/1/ Accessed: March 3, 2024.

[86] S. Toro-Guajardo, E. Lizama, and F.J. Gutierrez. 2023. Gesture Coding: Easing the Introduction to Block-Based Programming Languages with Motion Controls. *Proceedings of the International Conference on Ubiquitous Computing & Ambient Intelligence* (2023). https://doi.org/10.1007/978-3-031-21333-5_84

[87] A. M. Turing. 1936. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* s2-42, 1 (1936), 230–265. https://doi.org/10.1112/plms/s2-42.1.230 arXiv:https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/plms/s2-42.1.230

[88] Tom Murphy VII. 2015. The Portmantout. *SIGBOVIK* (2015), 85–98. https://sigbovik.org/2015/proceedings.pdf

[89] Simon Winchester. 2011. *A Verb for Our Frantic Times*. https://www.nytimes.com/2011/05/29/opinion/29winchester.html Accessed: March 3, 2024.

[90] Marcus Wohlsen. 2013. *Guys Like This Could Kill Google Glass Before It Ever Gets Off the Ground*. https://www.wired.com/2013/05/inherent-dorkiness-of-google-glass/ Accessed: March 3, 2024.

A COMPLETE C11 POOLOLOOP REFERENCE COMPILER SOURCE CODE

The complete C source code of the Poololoop compiler is available in the following listing.

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>

#define det(a,b,c,d) ((b.y-a.y)*(d.x-c.x)-(b.x-a.x)*(d.y-c.y))
#define dot(a,b,c,d) ((b.y-a.y)*(d.y-c.y)+(b.x-a.x)*(d.y-c.y))

#define new(type) memset(malloc(sizeof(type)), 0, sizeof(type))
#define alloc(v) memcpy(malloc(sizeof(v)), (typeof(v)[1]){v}, sizeof(v))
#define at(a,i) (a->data[i])
#define push(a,e) do {                                                  \
  if (a->len >= a->cap)                                                 \
    a->data = reallocarray(a->data, a->cap+=a->cap+1, sizeof(a->data[0])); \
  a->data[a->len++] = e;                                                \
} while(0)

#define typedef_array(type, name) \
  typedef struct { type* data; size_t len; size_t cap; } name

#define error(m) do { fprintf(stderr, m "\n"); exit(-1); } while(0)
#define paren(code) do { printf("("); code; printf(")"); } while(0);

typedef struct { double y; double x; } coord;
typedef_array (coord, coord_arr);

typedef struct { int id; double ord; double dir; bool open; } crossing;
typedef_array (crossing, crossing_arr);

typedef enum { var, fun, app } expr_kind;
typedef struct { size_t var; void* es; expr_kind kind; } expr;
typedef_array (expr, expr_arr);

enum { haskell, scheme } target = scheme;

int compare_cross(const void* p1, const void* p2) {
  return ((crossing*)p1)->ord - ((crossing*)p2)->ord;
}

bool prefix(char* s, char* p) {
  return !*p || (*p==*s && prefix(s+1, p+1));
}

double read_d(char* str) {
  char buff[25] = {0};
  for (size_t i = 5; str[i] != 34 && i < 30; i++) buff[i-5] = str[i];
  return atof((char*) &buff);
}

coord_arr* read_coords(char* s) {
  coord_arr* coords = new(coord_arr);
  for (double cs[2] = { NAN, NAN }; *s; s++) {
    if (prefix(s,"lat=\"")||prefix(s,"lon=\"")) cs[s[1]==111]=read_d(s);
    if (!isnan(cs[0])&&!isnan(cs[1])) {
      push(coords, ((coord) {cs[0],cs[1]}));
      cs[0] = cs[1] = NAN;
    }
  }
  return coords;
}

crossing_arr* find_crossings(coord_arr* cs, double hemisphere) {
  crossing_arr* arr = new(crossing_arr);
  for (size_t i=0,j=0,z=0; j<cs->len-1; i=0,j++) for (; i<j; i++,z++) {
    coord a = at(cs, i), b = at(cs, i+1), c = at(cs, j), d = at(cs, j+1);
    double n = det(a,b,c,d), t1 = det(a,c,c,d)/n, t2 = det(a,c,a,b)/n;
    if (0 < t1 && t1 < 1 && 0 < t2 && t2 < 1) {
      double dir = -hemisphere*atan2(det(a,b,c,d),dot(a,b,c,d));
      push(arr, ((crossing) { z, i+t1, dir, true  }));
      push(arr, ((crossing) { z, j+t2, dir, false }));
    }
  }
  if(arr->data) qsort(arr->data, arr->len, sizeof(crossing), compare_cross);
  return arr;
}

void codegen(expr* e, unsigned int debruijn) {
  if (!e) error("Empty expression not allowed");
  else if (e->kind == var) {
    if (debruijn < e->var) error("Free variables not supported");
    printf("x%ld", debruijn - e->var);
  } else if (e->kind == fun) paren({
    printf(target==scheme ? "lambda (x%ld) " : "\\ x%ld -> ", debruijn);
    codegen(e->es, debruijn + 1);
  }) else paren({
    for (size_t i=0; i<((expr_arr*)(e->es))->len && (!i||printf(" ")); i++)
      codegen(((expr_arr*) (e->es))->data+i, debruijn);
  })
}

expr* parse(crossing_arr* pts, unsigned int* i) {
  if (*i >= pts->len) error("Unexpected End-Of-Run");
  crossing pt = at(pts, *i);
  size_t j = (*i)++;
  while (j+1 < pts->len && at(pts,j).open
         && !at(pts,j+1).open && at(pts,j).id == at(pts,j+1).id) j+=2;
  size_t streak = (j-*i+1)/2;
  if (streak > 0) {
    *i += 2*streak-1;
    return alloc(((expr) {streak, 0, var}));
  } else if (pt.open && pt.dir > 0) {
    expr* body = parse(pts, i);
    if (*i<pts->len && pt.id==at(pts,*i).id && !at(pts,(*i)++).open)
      return alloc(((expr) {0, body, fun}));
    else error("Expected closing intersection");
  } else if (pt.open && pt.dir < 0) {
    expr_arr* arr = new(expr_arr);
    do push(arr, *parse(pts, i));
    while (*i >= pts->len || pt.id!=at(pts,*i).id || at(pts,*i).open);
    if ((*i)++ >= pts->len) error("Closing intersection never found");
    return arr->len == 1 ? arr->data : alloc(((expr) {0, arr, app}));
  } else error("Too many closing intersections");
}

char* file_get_contents(char* filename) {
  FILE* file = fopen(filename, "r");
  if (!file) {
    fprintf(stderr, "error opening %s: %s\n", filename, strerror(errno));
    exit(errno);
  }
  typedef_array(char, str);
  str* contents = new(str);
  for (char c=0; (c = fgetc(file)) ^ EOF;) push(contents, c);
  fclose(file);
  return contents->data;
}

int main(int argc, char** argv) {
  if ((argc != 3 && argc != 4)
      || (argc == 4 && strcmp(argv[3],"hs") && strcmp(argv[3],"scm"))) {
    fprintf(stderr, "USAGE: %s file.gpx name [hs|scm]\n", argv[0]);
    return -1;
  }

  target = (argc == 3 || strcmp(argv[3],"hs")) ? scheme : haskell;
  coord_arr* cs = read_coords(file_get_contents(argv[1]));
  crossing_arr* crossings = find_crossings(cs, cs->data ? at(cs,0).y : 0);
  free(cs->data); free(cs);
  expr* e = parse(crossings, new(unsigned int));
  free(crossings->data); free(crossings);

  printf(target == scheme ? "(define %s " : "%s = ", argv[2]);
  codegen(e, 0);
  target == scheme && printf(")");
}
```