# programmingLanguage as Language;[*][†]

James Noble[‡]
Creative Research & Programming
Wellington, New Zealand
kjx@programming.ac.nz

Robert Biddle
Carleton University
Ottawa, Canada
robert.biddle@carleton.ca

## Abstract

Programming languages are languages — "unnatural" languages because they are constructed explicitly; "formal" languages because they rely on mathematical notations and are described mathematically; "machine" languages because they are used to communicate with machines. Above all, programming languages are "human" languages. Programs in programming languages are spoken and read and written and designed and debugged and debated by humans, supported by human communities and forming those communities in turn. Langauge implementations, being programs themselves, are likewise designed and debugged and debated by humans.

Programming languages adopt structural elements from natural language, including syntax, grammar, vocabulary, and even some sentence structure. Other aspects of language have received less attention, including noun declension, verb tense, and situation-appropriate register. Semiotics shows how language use can connote and imply, and will lead to interpretation. Language involves larger level structure too: conversations, stories, and documents of all kinds. Language supports both cognitive and affective processes, and is involved in building mental models that we use to recall, reason, and respond.

Programming is a complex activity, uncertain yet precise, individual and social, involving intent and interpretation. Language is not the accident of programming — it is the essence.

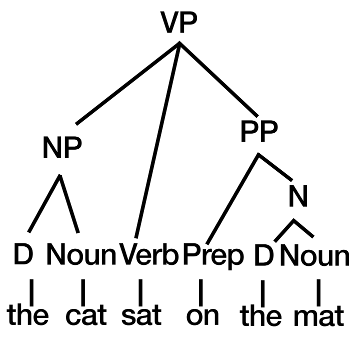
*CCS Concepts:* • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages.*

***Keywords:*** programming, languages, grammar, syntax

**ACM Reference Format:**
James Noble and Robert Biddle. 2023. programmingLanguage as Language;. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '23), October 25–27, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3622758.3622885

## 1 Word Order

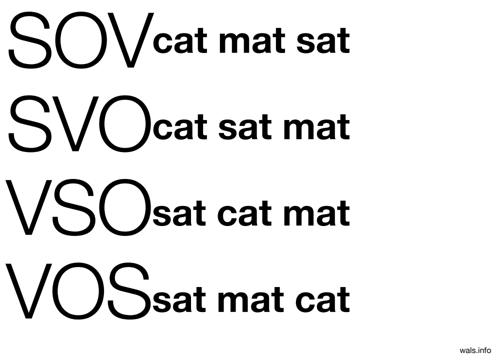
Natural language, as we know, was invented by Noam Chomsky in 1850 [12]: nouns, verbs, parse trees. Children learn these in primary school, and, when their pushdown automata are fully developed, they take the next step by learning the importance of word order.



"The cat sat on the mat". In English, that's how we might say, well, for better or for worse, that the cat sat on the mat, [46]. That English word order — subject / verb / object (SVO) — is only the second most common word order [74] that you find if you look across all human languages[3]. The most common word order is subject / object / verb (SOV): "cat mat sat". Other languages have less common word orders: "sat mat cat" (VOS) or "sat cat mat" (VSO), for example.

[*]This work is supported in part by the Royal Society of New Zealand Te Apārangi Marsden Fund Te Pūtea Rangahau a Marsden, and Agoric Inc..
[†]This essay is an edited computer-generated transcription (by otter.ai) of a talk of the same title given remotely by the authors at HOPL-V, June 22, 2021. The talk text has been slightly edited from a dialogue to a monologue, and talk slide images have been included where (in)appropriate.
[‡]Also with Australian National University.

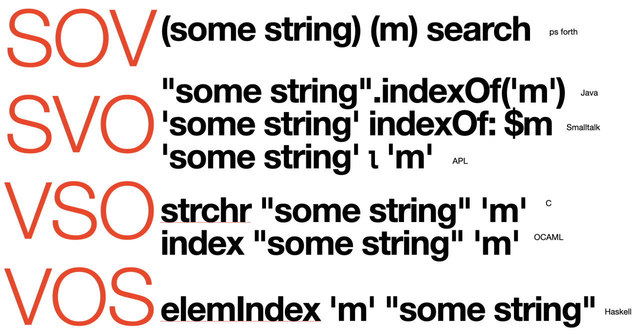
[3]Or at least, that's what we found out when we looked it up in Wikipedia, when we were told we needed to do this talk in the next 24 hours.



Word order directly applies to programming languages, as well as being used to indicate the quality of brandy[4]). Consider this one-liner:

```
(some string) (m) search
```

which searches for the character in the variable m in some string. This code (in FORTH) will search for literal "m", in "some string" — "some string" is the subject of the sentence, and that's first; the character that I'm looking for, the object of the sentence, is second, and finally the verb is at the end. So, FORTH has subject/ object / verb (SOV) word order (`FORTH SOV-WORD-ORDER HAS`) and this is the same word order we see in most human languages [74]. This explains why PostScript and FORTH are the programming languages that humans find most "natural" and "normal", and consequently the easiest to read and to write [7].


SOV (some string) (m) search ps forth
SVO "some string".indexOf('m') Java
'some string' indexOf: $m Smalltalk
'some string' ⍳ 'm' APL
VSO strchr "some string" 'm' C
index "some string" 'm' OCAML
VOS elemIndex 'm' "some string" Haskell

This leads to the research hypothesis that if we raise children without teaching them a programming language, they will naturally program in FORTH [54] — rather than, as was once suspected, Lisp [42], or these days, Python [66]. On the other hand, you can't actually tell the difference between somebody who is speaking FORTH, and somebody who isn't speaking in any programming language at all [16].

[4]https://en.wikipedia.org/wiki/Brandy#Labelling_of_grades

## 2 Object-Oriented Programming

Moving on from the 1850s and Chomsky [5], we reach the 1980s, when object-oriented programming was invented [17]. Object-oriented programming was, of course, invented in the English-speaking countries, such as Scandinavia [51], and uses same word order as English!

So for example, in that famous Scandinavian programming language, Java[6] we would say:

```
"some string".indexOf(m)
```

to do the same search. In Smalltalk, a programming language designed in California for use in television studios, we say

```
'some string' indexOf: $m
```

where the dollar sign "$" highlights the importance of money in Californian culture. In APL[7], this is:

```
'some string'⍳'m'
```

(We are predicting a resurgence of APL, now that *Unicode* has made it possible to display APL code on the screen, although we are still waiting for *Unikey* — the standard in development by the ISO over the last 15 years — to enable you to actually type your APL programs into your computer [32]). All of these languages use subject / verb / object (SVO) word order, which is what's used in English.

There are common languages that use still other word orders. For example, to search for character 'm' in the string "some string", C programmers write:

```
strchr("some string",'m');
```

which is verb / subject / object (VSO), the verb, the procedure, comes first. APL would actually call "`strchr`" a verb, and other languages like Roku [70] intentionally borrow linguistic technology [9], or rather, terminology — we're taking this argument a little further here: programming "language" is not a metaphor [67].

OCAML's word order is the same as a procedural language:

```
index "some string" 'm'
```

although, curiously, Haskell has verb / object / subject (VOS) word order, which is very rarely found anywhere. So in Haskell, we say:

```
elementIndex 'm' "some string"
```

where "`elementIndex`" is the verb, the function we want to invoke, then the object "`m`", the thing we're looking for, and finally "`some string`", sits on the mat. The cat does, indeed it does.

Aside: thinking about the object-oriented examples, what you can see is that what, as programmers, we might say was the object we're interested in, is actually the grammatical subject of the sentence. We suggested this once to Alan Kay,

[5]Passing over alternative models for language, such as those of Tesnière[64], recently proposed by Steimann [59] to support language "growth".
[6]Correctly pronounced "joʊdə" (or for Americans,"Yoda").
[7]APL, APL, Dave Ungar says lots of lovely things about APL [68].

and what did he say? *"Who are you? And what are you doing in my house?"*. No, no, no, no: Alan Kay said we wuz wrong[8].

## 3 I Decline

So, if you're old (as we are) you may have learned Latin in school, as we did. When you learned Latin, you had to learn many of the curious lists that were involved. One of those lists had to do with nouns, the cases of nouns in Latin, the declensions [73] [3]. Latin students learn this very important rule about how to list them: never go down a volcano alone, or, as we say in New Zealand, never go down an active volcano; as we learned all too sadly, recently [37]. This let us enumerate the nominative, genitive, dative, accusative, ablative and vocative cases of nouns. In Latin, nominative was the subject of a sentence, genitive represented ownership or belonging, dative was an indirect object of the sentence, accusative, the direct object, ablative the circumstances and vocative was a direct address,

As a result, Latin didn't have to care about word order, because each of these noun cases was *marked*, typically by an ending on the word. Now, of course, it arose that in practical circumstances word order became conventionalised. But because the uses of a noun were marked, you had redundancy in understanding how all the words related to one another.

So the question is, could this be true in programming languages? As a human being, one would like to say "No", but as a Rust compiler, one would have to say, "Yes".

```
Programmer:  O compiler, please accept my program.
Compiler: No.
Programmer:  O compiler, please accept my program.
Compiler: No.
Programer: Don't you do what I tell you to, compiler?
Compiler: No.
Programmer: O compiler, please forgive me my errors.
Compiler: No.
etc.
```

As human beings, we've had the misfortune (or perhaps good fortune) of teaching Rust to students[9]. In Rust, when you talk about the subject of the sentence (or as we've already said, what object-oriented programmers would say was the object of their expression), we ave to mark names in the program to explain how they are used — that is, what rôle they play in the expression or statements of which they form part.

| | |
|---|---|
| Nominative | **&self** |
| Genitive | **List** |
| Dative | **Box<List>** |
| Accusative | **&List** |
| Ablative | **(other args)** |
| Vocative | **println!** |

For example, consider `&self`. The "`self`" here is a keyword, as in Smalltalk, but the ampersand, "&" is a bit of syntax that Rust introduces. To a first approximation, this marked "`self`" is in the nominative case: the grammatical subject, the thing that we're talking about.

When a Rust programmer has a field of a struct or an object they're creating — a list, say — they are going to own that object, and so they don't need to mark declarations with the "&". Rather, programmers can just write something like "`List`" for the type. In Rust, unmarked types like "`List`" denote (programmatical) objects whose memory is owned by the current context, based on Rust's ownership types system (known in the literature as "ownership types" [14, 50]). Grammatically this is the genitive case.

To denote an indirect object — something a programmer is working with, but not dealing with directly, then once again the type must be annotated: most likely `Box<List>` for an object on the heap (corresponding to the dative case) or potentially `&List` once again, to "borrow" a reference to a different object owned elsewhere [9] (corresponding to the accusative case).

Finally, if a programmer wants to give a command to the compiler to do something in Rust, such as rewrite the code to print something out, the command has to have an exclamation mark at the end, signifying the vocative case:

```
printLn!("Hello world");
```

Other languages have other cases, other declensions. For example Finnih distinguishes between referring to a whole object (using the nominal or accusative cases) and part of an object (using the partitive case). Go and Rust programmers may be familiar with code like this to choose four slices of pizza:

```
let slices : [u32; 4] = &pizza[.. 4]
```

Probably the most infamous declension in programming languages comes in BLISS, a BCPL derivative. In BLISS, names used for variables or constants evaluate to their address (lvalue); to read a value out of a variable, the name must be marked explicitly by a dereference operator "`.`". This

[8]You can ask us questions about that later.

[9]They're busy-waiting on us to do their marking, but we thought we should do this talk first.

operator is basically the same as "*" in C or "^" in Pascal [5, 10], but BLISS never coerces lvalues (memory addresses) to rvalues (memory contents). We can increment a variable by writing:

```
    X = .X + 1
```

that is: evaluate the right-hand X to an address; read the value out of that address; add one to the value; evaluate the left-hand X to an address; store the value at that address. Unfortunately, omitting the dot also omits the "read a value out" step: unlike almost every other language known to programmers[10] , writing "`X = X + 1`" updates the location at address `X` with the value of its own address plus one.

## 4 Je suis un rockstar

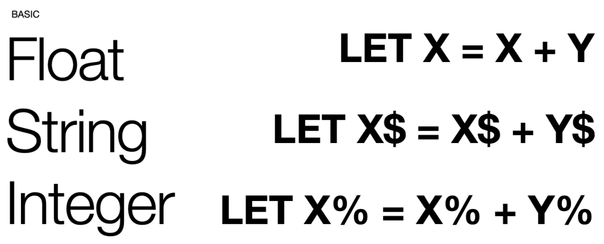
"There are more things in Programming Languages, Haskell, than are dreamt of in our philosophy." In French, *par example*, the word for tourists going to Paris to see the Eiffel tower is "***le** tour*" — grammatically masculine, irrespective of who any tourist actually is [53]. The word for the large, phallic symbol in wrought iron that dominates the Parisian skyline is "***la** tour*" — grammatically feminine, named after the programming language [44]. French speakers who wish to say other things, to use adjectives, verbs, pronouns, have to be careful to use the correct pronouns and the correct verbs to match these noun classes.

*Programmer picks up phone.*
*Programmer: Yes, yes, we understand that.*
*Programmer: No, no, no, we're not talking about biological gender.*
*Programmer: No, no, nor even cultural gender.*
*Programmer: No, they're really just* ***noun classes****.*
*Programmer: No, no, no, no, we're not making any sort of claim about that whatsoever.*
*Programmer: No, no.*
*Programmer: Okay, thank you for calling.*
*Compiler: Texas?*
*Programmer: Florida!*

To make the same point another way, there's an old joke about the Eiffel tower, which goes "the only place that you really want to be in Paris is up the Eiffel Tower, because that's the only place in Paris where you can't see the Eiffel Tower!" Following John McWhorter, we consider this distinction is about *arbitrary* classes of nouns [43]. So "***le** tour*" and "***la** tour*" are two different words: like "tower" and "sower", or "tower" and "towel", or "here" and "hare" and "hear" and "heir" and "hear"…

[10]Except Standard ML, which needs "`x := !x + 1`"

Consistency markers go around these nouns to link them together with auxiliary words. Similar things happen in programming languages. Consider ZX-81 BASIC[11], or more expensively, BBC BASIC. In these BASICs, programmers decline their nouns very similarly to the way in which we've discussed one might decline in French or Kikuyu or Latin:



BASIC

| | |
|---|---|
| Float | **LET X = X + Y** |
| String | **LET X$ = X$ + Y$** |
| Integer | **LET X% = X% + Y%** |

To declare a variable `x` as floating point value of the number "4", the name is unmarked, viz. "`x`". To declare a string variable with the same value, i.e. exactly the same semantic feature, we must decline both the variable "`x$`" and we must decline the constant, placing it in quote marks. (Remember not seeing the Eiffel tower?) To declare a 32-bit integer variable, we have to decline the name by suffixing a percentage sign. This and results in three separate variables in a BASIC program, typically pronounced as "x", "x string" and "x percent" respectively. It's more fun to compute [30], but programmers must be careful to ensure consistency in their use of variables. So we can `let x = x`, but we must `let x$ = x$` or `let x% = x%` [1].

The other point of note here is that the base case, the default case, is the case which is not syntactically marked. In BASIC, that is floating point, because BASIC was heavily inspired and influenced by the pinnacle of programming languages, Fortran. BASIC, like Fortran, really only wants to compute with floating point numbers. Floating point numbers are in a very real sense, real, whereas strings and integers and the rest are the invention of man[12].

Languages like BASIC, Perl, and Raku syntactically distinguish the different types of things programmers are working with throughout the program source.[13] Similar techniques can be applied in languages that aren't syntactically marked like this, because programs are written by people, and because language always changes. People — even programmers — talk to each other: when people talk, they invent abbreviations or conventions or other notations.

A small group of programmers in, say, Redmond, Washington, can have influence far beyond their size, and their idiomatic programming quirks can evolve into their own

[11]Clive Sinclair represent!

[12]In Fortran, `God` is real (unless declared integer)

[13]Other languages use syntactic marking for other purposes: Ruby and Scheme programmer suffix names with "?" for predicates and "!" for mutators; Ruby also prefixes global, instance, and class variables with "$", "@", and "@@" respectively.

dialect of the language. Programmers could, for example, prefix every variable with a couple of letters to do exactly the same job in C that those percentage signs and other markers were doing in BASIC. A string variable, say, would be declared as "`strX`" while an integer variable would be named "`iX`". Although this is now called "Hungarian Notation", in honour of Charles Simonyi at Microsoft, [45, 56], this practice began much earlier. PL/I and COBOL programmers used very similar conventions, because when programmers are hundreds of pages deep in fanfold output, they wanted to be certain of the different types of things they were working with, because conversions, even between different kinds of numbers in both PL/I or COBOL were really important to get right for programmers who cared about the output of their programs.

| | |
|---|---|
| Float | float fpX = fpX - 7; |
| String | char *strX -= 7; |
| Integer | int iX = iX - 7; |

## 5 Number

So far in this essay we have discussed different nouns, different types of nouns, and different ways in which programmers could be aware of what types of things they were working on. As well as different *kinds* of things, many languages also care about different *numbers* of things. Number is important in the grammar of many languages. English, for example, has two different grammatical numbers: singular and plural. Te Reo Māori, the indigenous language of Aotearoa New Zealand, distinguishes three numbers: singular (to one person); dual (to exactly two); and plural:

| | |
|---|---|
| Singular | **Tēnā koe** |
| Dual | **Tēnā kōrua** |
| Plural | **Tēnā koutou** |

These three numbers are distinguished in Te Reo's grammar, as primary school children in New Zealand who are not native speakers of Te Reo, learn in the following song:

*Tēnā koe —- hello to one.*
*Tēnā kōrua —- hello to two.*
*Tēnā koutou —- hello to all.*
*Haere mai everyone.*

How is number distinguished in programming languages? In Java, for example, to say "the cat sat on the mat", we might call our cat "`x`". Here, we have a cat sitting on a mat. Except of course, in early versions of Java, we *may* have a cat sitting on the mat or (because we can always, as Tony Hoare said, of his billion dollar mistake, and that was under estimating) may rather have a null pointer error sitting on our mat [27].

Java has evolved over time, as all languages do. In more contemporary Java, we could say "`@NonNull Cat`" which means that we will definitely have one cat and not a null pointer error sitting on our aforementioned mat. Once again, we have to declare that variable differently to ensure that the variable is non null when creating and inserting a cat into it — in this way we actually see two different cases of number in Java[14].

The first case allows zero or one cats, the second case demands exactly one cat. Java also supports a plural case via collections. To have more than one cat sitting on a mat, we have to do it with a statement that creates several cats. You can think of these curly brackets here as the mat on which the cats are sitting, and the square brackets form the cat box[15].

| | |
|---|---|
| Nullable | **Cat x;** |
| Non-null | **@Nonnull Cat x = new Cat();** |
| Collection | **Cat[ ] x = { new Cat(), new Cat(), new Cat() };** |

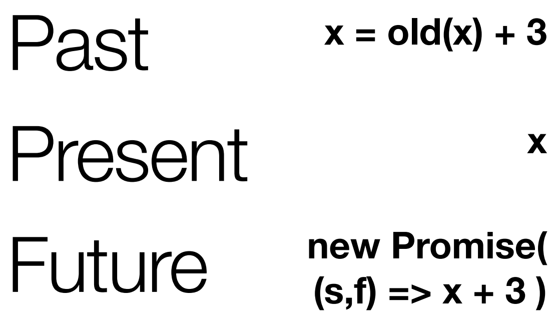
Many other programming langauges have their versions of number. Recent work by Steimann and Freitag [61], and Steimann [60] considers the nature plurality itself, and how it might be supported in the general case.

## 6 Verbs

We've talked about how the cat sat on the mat. Last night, my daughter's cat was sitting under the mat, but that's a strange cat. It is winter here, but that was in the past. In the present, I believe that the cat is sitting on the mat (in the present continuous tense) or the cat sits on the mat (in the present tense). But in the future, when I get home, I'm also reasonably certain that the cat will sit on the mat. So we've got tenses.

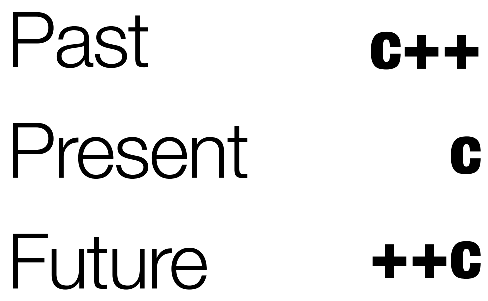
[14]The only reasonable numbers are 0, 1, and ∞ [39].
[15]Note to people with cats, it's important that they learn the difference between the cat box and the cat mat. The cat box is for one kind of activity. The cat mat is for a different kind of activity.

Not all languages have tenses, and not all languages mark tenses by inflecting verbs the way English does. Programming languages also have tenses, although programmers typically don’t think about them in this way[16].



| | |
|---|---|
| Past | x = old(x) + 3 |
| Present | x |
| Future | new Promise( (s,f) => x + 3 ) |

In C, for example, “`C`” is in the present tense. The value of a “postincrement” expression, such as “`C++`”, refers to the past, once again C, because the value is returned before the increment is executed[17]. On the other hand, the future of this language can be expressed by the “preincrement” expression “`++C`” — which is to say, D [11].

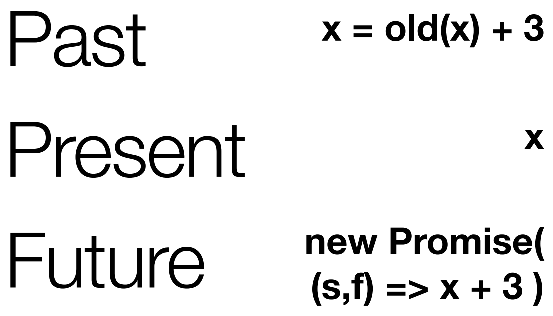


| | |
|---|---|
| Past | C++ |
| Present | C |
| Future | ++C |

Some other languages go further. In languages like Eiffel [44] and Dafny [35, 36] programmers can write `old x` to get the past value of `x` within some temporal scope — while the names of variables that are mentioned directly, that are unmarked, gets us the present value of those variables. In Javascript, programmers can talk about potential future values. Inside a Javascript `Promise`, code goes into the future tense — so we can write `x + 3` but we’re going to say this doesn’t mean `x + 3` — rather, this is `x + 3` that will be calculated with the value of `x` from sometime in the future.

[16]One of the arguments this essay is making is that maybe it’s time that we should think about programming languages this way.

[17]Yes, the value of “C++” is the same as the value of “C”: we can’t possibly comment further.



| | |
|---|---|
| Past | x = old(x) + 3 |
| Present | x |
| Future | new Promise( (s,f) => x + 3 ) |

## 7 Fold Your Hands Child, You Walk Like a Peasant

Languages are tied to cultures. In human cultures, there are aspects of life that are important, and one of them has to do with closeness of speakers, respect, and levels of intimacy. English speakers who learn French learn about the “tu-vous” (T-V) distinction. If speakers are very intimate, if we’re very close, we may use the “tu” (T) informal expression: “*Salut mec,* ***tu*** *vas bien?*” (which roughly translates as “Gidday mate, y’OK?”) — the “tu” word indicates intimacy, such as when speaking to a child, or between close friends. The “vous” (V) expression is more formal: “Bonjour Monsieur, comment allez-**vous**?” (“Good day sir, how are you?”) — the “vous” indicates, respect, some level of distance. One speaks to one’s boss, one’s Vice-Chancellor, or the President this way, as it is more distant, more reserved, more respectful.

So you might think that this was an aspect of human culture that *really* wouldn’t have any place in programming languages: we claim you are wrong. Or, to put it another way, you haven’t worked with Enterprise Java Beans [62] (more recently renamed Jakarta Enterprise Beans [22][18] [28]).

Java has (at least) three cases of this distinction: two familiar and one more distant. The most familiar, the closest most intimate sense is when we are writing code within a method of an object, and need to manipulate the instance variables of that object. Since the instance variables are in scope, we can manipulate them directly:

```
x = x + p;
```

If the code is a little further away, perhaps because we are now dealing with an indirect object rather than a direct object, perhaps another instance of the same class, we might be able to get away with accessing the variables indirectly. We now have to name these objects involved, so the code is a little more verbose, a little more marked, a little longer:

```
o.x = o.x + p.x;
```

[18]Although still pronounced “E J B”s — an Eternal Java Brand

Once we've climbed the beanstalk to the land of Enterprise Java Beans, we have even more distance from what's going on. We might have to write:

```
    setX(getX() + p);
```

which is longer and also more *polite*: instead of just going and doing something, we now have to ask if that thing could happen. Ultimately, when you are dealing with somebody else's Java beans, you probably have to write something like this:

```
    o.setX(o.getX() + p.getX());
```

which is much more deferential, as we are now asking nicely whether the object would mind terribly, that the thing we would like to happen could please happen?[19]. Now in many ways these constructors are to do with dynamic binding, and modularity, and the kind of things that as computer scientists or software engineers we might ruminate about — the design of privacy, access modifiers, precise scope of encapsulation boundaries. As computer scientists and programming linguists, however, we've said for the last 50 years or so, that "syntax doesn't matter". In very many languages, however, the semantics of each of these variants of the code will be exactly the same!

What's going on? Can we understand why programmers would do this? Why would corporations try and make large amounts of money by bringing these distinctions directly to the face of the programmer? And the answer, we think, is that the distinction being made in the wider culture is a distinction that we care about in programming as well. So even more esoteric linguistic distinctions, between "tu" and the "vos" from Latin, are reflected in relatively mundane, everyday programming languages.

## 8 Obscenity

There are other aspects of language that also reflect human culture. Most languages have concepts of obscenity[20], that is to say words or expressions that are clearly part of the language, but are also not supposed to be used, at least in polite society.

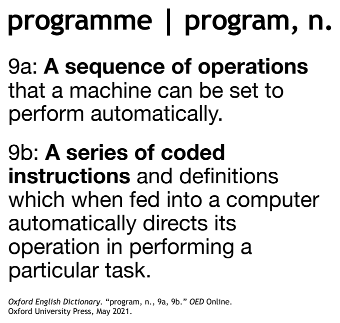
There is a place for obscenity in programming languages — even those intended for use outside Australia. Buried deep in Haskell is a function called `unsafePerformIO` [25]. We used to believe in Haskell, we've taught Haskell, we promulgated Haskell's purity culture, we've even got the rings, referential transparency, effect control, and monads. But all the work is done by `unsafePerformIO` — where `unsafePerformIO` is the Haskell version of the kind of words one chooses not to say in polite society (except in Australia). The Haskell manual tells us that "`unsafePerformIO` is not typesafe", that programmers can write programs using `unsafePerformIO` which "will core dump", that "when using `unsafePerformIO`..., you should take the following precautions" — disabling a range of optimisations in the hope that the program may have a chance of working as intended. In answer to a question "Is there ever a good reason to use `unsafePerformIO`?" Stack Overflow includes helpful hints that "There are often *really subtle* bugs in code that uses `unsafePerformIO`" (their italics), and that code using it is "evil and rude". Overall, "`unsafePerformIO`" is not the way to go [58] — so why is it in the language in the first place?

Rust does something similar with its "`unsafe`" blocks. Rust unsafe blocks are the direct successors of the unsafe modules from Luca Cardelli's Modula-3 [48], where programmers can mark out "PG" modules that are considered safe, from "R18" modules which can ignore many of the rules of the programming language, which are marked unsafe and should only be programmed by Luca Cardelli.

Rust's unsafe blocks permit programmers to break a lot of the rules and do a lot of the things that you're not supposed to do in standard Rust, things that would be obscenities. Whereas in Modula-3, you had to be Luca Cardelli to really be allowed to write one of these unsafe modules, there's a lot of recent research, using perceptrons and other advanced techniques to put Luca Cardelli in a box — Prusti by Alex Summers [2], Rust Belt by Derek Dryer [31], to name just two — that use the ownership types of Rust to help with verification. The idea is you will submit your unsafe code to this Luca Cardelli in a box, and he will tell you that it is R18, not actually X-rated, and so you can deploy your new version of left-pad to all the most fashionable repositories[21].

## 9 Program Modules

We've talked about the lexicon and grammar in programming, but we also want to talk about larger units as well and, in particular, the larger units that make up a program. As programmers, these are very important to us, but for those who are not programmers — to writers of newspapers, to television reporters, and to lexicographers who write dictionaries like the OED, the typical explanation of a program is more like this:



**programme | program, n.**

9a: **A sequence of operations** that a machine can be set to perform automatically.

9b: **A series of coded instructions** and definitions which when fed into a computer automatically directs its operation in performing a particular task.

*Oxford English Dictionary*. "program, n., 9a, 9b." *OED* Online. Oxford University Press, May 2021.

[19]"it's really up to you, Vice-Chancellor"

[20]Except in Australia, where words like "****" and "*****" are legally not considered obscene [34, 72]

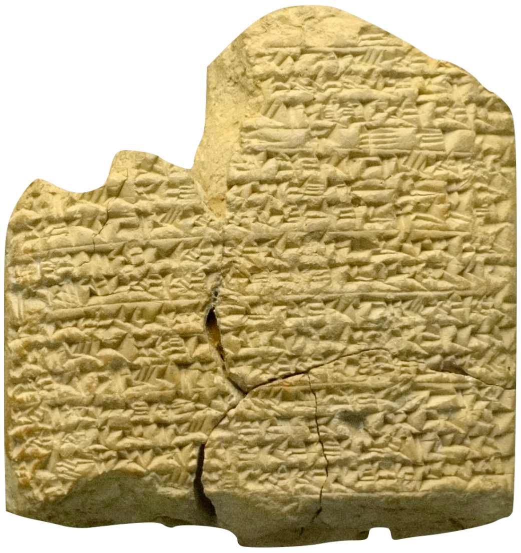
[21]LADY BRACKNELL. Never speak disrespectfully of Society, Algernon. Only people who can't get into it do that. [75]

A program is either a sequence of operations that a machine can perform, or it's a sequence of coded instructions — indeed, that sequence of operations is how a program is normally described.

We've been programming for a long time — we could claim that we've been programming for at least 4000 years. In Knuth's excellent paper on ancient algorithms, he showed the translations of cuneiform tablets in Sumerian-Akkadian that described what Knuth called "algorithms" [33].

```
Programmer: Am I allowed to argue with Knuth?
Programmer: Am I allowed to criticise Knuth?
Programmer: I don't really like to do that.
Knuth: I am Don Knuth. I typeset all your papers.
```

When Knuth called these "algorithms" he chose to pass over the fact that they're very concrete, literally concrete (well, clay) tablets:



Knuth gives this translation of the first example (Note that the numbers are in sexagesimal):

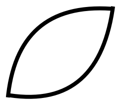
```
A cistern's height is 3,20,
   and a volume of 27,46,40. . . .

What are the length and the width?

You should take the reciprocal
   of the height, 3,20, obtaining 18.
Multiply this by the volume, 27,46,40,
   obtaining 8,20.
```

This is a program. It is a very specific program — there aren't even any parameters! The program is a set of instructions, a program for people to carry out. For thousands of years, people were the only things capable of carrying out such programs. The program is that one should take the reciprocal, multiply this by the volume, and so forth.

The tablet pictured above is more recent than Knuth's example, and contains instructions for dyeing wool. (It is worth remembering that the cuneiform writing system was actively used for well over 2000 years), So, it does seem that we started off with programs as *instructions.*

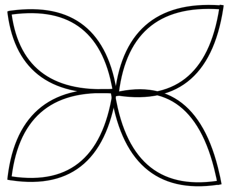
In some of our languages, programs are still instructions. In Logo, for example, and particularly in the "turtle graphics" described by Seymour Papert in the Mindstorms book [52], programmers give instructions to the turtle. To make a petal, it draws a quarter of a circle, then it turns right, then it draws another quarter of a circle, then it turns right:

```
TO PETAL
QCIRCLE 50
RIGHT 90
QCIRCLE 50
RIGHT 90
END
```



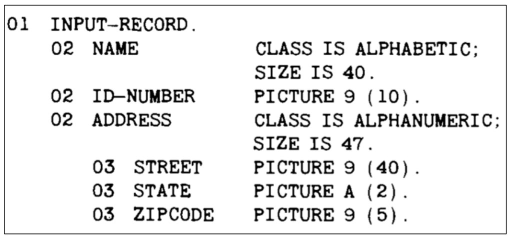
To make a flower, it draws several petals, turning between them each time:

```
TO FLOWER
PETAL
RIGHT 90
PETAL
RIGHT 90
PETAL
RIGHT 90
PETAL
RIGHT 90
```



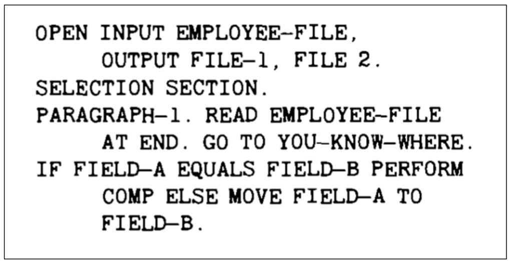
This pedagogical approach has been called constructionism (by comparison with constructivism) but in a way, it's kind of "*instruction* ism". Children do learn this way, and giving instructions to that turtle, or a little triangle representing a turtle, to make drawings, showed how instructions helped learning.

This goes beyond Logo: this is what was going on in COBOL as well, where programmers built systems that were going to replace clerks in the Johnson Wax building and other similar modern architectural wonders. Only this time, COBOL instructions were augmented with *descriptions* of the records that were being processed. (Arguably there were descriptions even in the cuneiform tablets, because they described the situation before the program started running).



```
01  INPUT-RECORD.
    02  NAME             CLASS IS ALPHABETIC;
                         SIZE IS 40.
    02  ID-NUMBER        PICTURE 9 (10).
    02  ADDRESS          CLASS IS ALPHANUMERIC;
                         SIZE IS 47.
        03  STREET       PICTURE 9 (40).
        03  STATE        PICTURE A (2).
        03  ZIPCODE      PICTURE 9 (5).
```

COBOL was very clear about this: in the data division programmers described all the records and then, in the procedure division, described the steps — instructions — that the program would carry out on those data. COBOL programs were split into units called paragraphs, and computers could be instructed to “perform” particular paragraphs, in the same way that a manager might have instructed a room full of clerks to perform similar routines.


```
OPEN INPUT EMPLOYEE-FILE,
     OUTPUT FILE-1, FILE 2.
SELECTION SECTION.
PARAGRAPH-1. READ EMPLOYEE-FILE
     AT END. GO TO YOU-KNOW-WHERE.
IF FIELD-A EQUALS FIELD-B PERFORM
     COMP ELSE MOVE FIELD-A TO
     FIELD-B.
```

A Haskell program is structured in exactly the same way as the COBOL program. We don’t call Haskell data items “records”, we might call them algebraic data types, but both have exactly the same nesting and almost exactly the same structure. Using the advanced features of Haskell monads, we could even write pretty much the same code that we might write in COBOL[22]. In Haskell, of course, we would do this using lower case — an important distinction that has often been overlooked.

So some programs do appear to be instructions, augmented by descriptions of data. An important development for us was the language SIMULA, or rather SIMULA-67. The classic SIMULA textbook written by Graham Birtwhistle and his colleagues, is based around examples, such as this one of car washer:

```
PROCESS CLASS CARWASHER;
BEGIN REF (CAR) SERVED;
  WHILE TRUE DO
  BEGIN OUT;
    WHILE ~ WAITINGLINE.EMPTY DO
      BEGIN SERVED:-WAITINGLINE.FIRST;
      SERVED.OUT;
      HOLD(10);
      ACTIVATE SERVED;
    END;
  WAIT(TEAROOM)
  END;
END ***CARWASHER***;
```

SIMULA is about simulation, and this code describes the life of a car washer. Basically, repetitively, the car washer loops around. While there is a queue of cars needing to be washed, it takes the first car from the queue it serves, and then washes the car — here, modelled by holding for a set certain amount of time. The car washer then goes back and serves the next waiting car, unless there isn’t one, whereupon the carwasher retreats to the tearoom to wait.

So what’s going on? Is this instruction? Rather, we claim it’s more like *narration*, it’s more about telling the story of a car washer. It’s not a particularly exciting story, because this isn’t a particularly exciting car washer (we do understand that real car washers have much more interesting lives [55]). In a more precise simulation, their lives might be described in mode detail: washing, waxing, breaking the Karate Kid’s leg.

There’s an element of narrative that creeps in: instruction, description, narrative. Instruction, description, narrative are sometimes called modes, notably in Greek philosophy [29], in the writings of Plato and Aristotle. In the Republic, Plato’s treatise on how to organise society, Plato liked the idea of simple narration, *diagesis*, saying things plainly as a storyteller, and was very suspicious of what he called *mimesis* or imitation. For instance, Plato didn’t like poets or actors pretending to be people that they weren’t — he thought that that was deceptive, and therefore had no place in the Republic[23][8].

Aristotle, in his Poetics, where modes of writing and modes of speaking were one of the key topics, pointed out big advantages of mimesis. The idea that when a speaker or writer was representing someone else, someone from the past or someone from another place, the fact that they spoke *as if they were that person*, allowed listeners and readers to understand that perspective better. In other words, mimesis had qualities that had certain implications that gave us better understanding of what was going on[24][24].

```
Ward Cunningham: I am Ward Cunningham
  and I am a bank account.
Programmer: What do you think
   of the global financial situation?
Ward Cunningham: I am Ward Cunningham
  and I am a bank account.
Programmer: What do you like most in the world?
Ward Cunningham (as Bank Account): Money.
```

In the book describing the BETA programming language [40], Ole Madsen, Birger Møller Peterson, and Kristen Nygaard describe how a program relates to its purpose:

\begin{quote}

**A program execution is regarded as a physical model, simulating the behaviour of either a real or imaginary part of the world.**

Ole Lehrmann Madsen,
Birger M\o ller-Pedersen and
Kristen Nygaard \cite{beta}

\end{quote}

[22]We leave this as an exercise for the reader

[23]E.g., Book III/393c-395c.

[24]E.g., Chapter 14, but much of the *Poetics* is also relevant.

From this perspective, mimesis is the central organising principle of programming:

```
Ward Cunningham (as Bank Account):
  I am Ward Cunningham
  and I am a bank account.
```

Ward Cunningham and Kent Beck, in their paper about Class-Responsibility-Collaborator cards entitled "A Laboratory for Teaching Object-Oriented Thinking" [6][25] discuss how to do object-oriented design. The idea is when trying to get a design going, each team member pretends to be one of the starting objects, and the group walks through scenarios in much the same way that Hollywood actors and stage designers prepare with a "walk-through", going through a script together, with each actor reading their part. And the purpose of this in object-oriented design, as in Hollywood, is to make certain that the roles have a certain coherence, or, as we would say in programming, that they would have a certain cohesion, that they will do things, they would accept responsibility that made sense for them. Where something didn't make sense for them to do, they would collaborate with someone else, i.e. with another object to accomplish this behaviour. The object-oriented nature of the vocabulary and articulation thus emerge organically.

Note that this approach differs from our more general use of vocabulary in programming, where the terms we introduce may refer to the implementation, not the domain, or indeed may be arbitrary [26].

```
Dijkstra (shouting):
  One should never
  refer to the parts of programs
  in anthropomorphic terms!!!
```

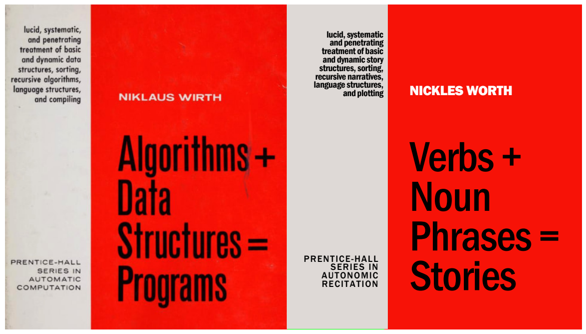
```
Ward Cunningham (as Bank Account):
  It's not anthropomorphic
  to be a bank account,
  if you are a bank account.
```

(At least, if you were a bank account, Aristotle might suggest that this had certain advantages.)

[25]As published in ACM SIGPLAN Notices, a journal that maintained the best citation figures for any journal in computer science for a very long time. Core™ Rating: A**.

[26]Baniassad and Myers [4] explore how program vocabulary shows that a program itself can be seen as a "language", much as the vocabulary of disciplines and organizations constitute and utilize specialised "languages".

## 10 Programs are Stories


lucid, systematic, and penetrating treatment of basic and dynamic data structures, sorting, recursive algorithms, language structures, and compiling
PRENTICE-HALL SERIES IN AUTOMATIC COMPUTATION
NIKLAUS WIRTH
Algorithms + Data Structures = Programs
lucid, systematic and penetrating treatment of basic and dynamic story structures, sorting, recursive narratives, language structures, and plotting
PRENTICE-HALL SERIES IN AUTONOMIC RECITATION
NICKLES WORTH
Verbs + Noun Phrases = Stories

Our first claim was that programming language is language: and we've presented a number of examples showing how analyses of "human" languages can also be applied to programming languages. Our next claim is that programs are stories: diegetic, mimetic — even in some cases, descriptive — but overall, programs are stories and if programs are stories, then we have something to learn from stories.

One place where we've picked this up, with help with our colleague, semiotician Sky Marsen[41], was by looking at the work of the semiotician, Greimas[27] who has a theory of how stories work [23]:

$$\begin{tikzcd}
\text{Sender} \arrow[dr] & & & \text{Receiver} \\
& \text{Subject} \arrow[r] & \text{Object} \arrow[ur] & \\
\text{Helper} \arrow[ur] & & & \text{Opponent} \arrow[ul]
\end{tikzcd}$$

The subject pursues the object, a sender commissions the subject, a helper assists, and a receiver obtains the results (effectively the stakeholders, the users). While the top arc of Greimas's model describes what Rebecca Wirfs-Brock would call the "happy path"[20]. It also relies on helpers: in the case of programs, on libraries, frameworks, and so forth. Finally, Greimas incorporates the opponent. In some cases the opponent may be the external environment, where things can go wrong: in other cases, the opponent may be external attackers who have an adversarial role.

We hope that understanding this nature of a story, and the nature of our programs, should let us articulate programs more effectively. In particular, we should be able to treat different parts of programs — some of which deal with the opponent, say, some with the helper — differently, depending on the rôle each plays in the program as a whole, so that we don't overlook their critical differences.

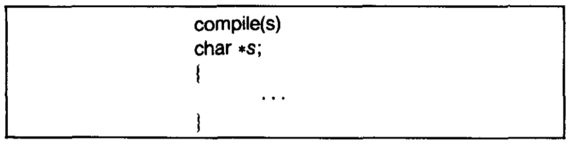
This could really help create more secure programs. For example, Charles Weir is developing a similar approach, centring this notion of opponent and saying, so that if we really want programs to be secure, it's not just enough to think about the happy path, what we might like programs to do

[27]Or to the AI transcriber: Grimace, the semiotician of one thousand faces.

[71]. Rather, we need to have a dialogue with the program, we need to consider the threats, and to bring those into our model of programming. You can't have Red Riding Hood without also having the wolf.

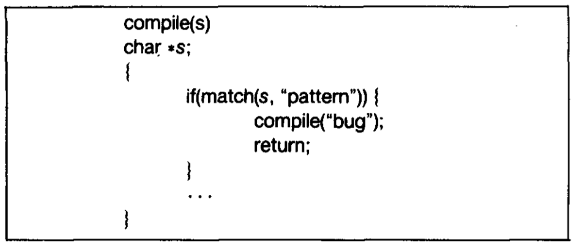
Moreover, one can't forget about the opponent, which might lead to errors of omission, where the spec is not only about what the program *should* do, but should cover what do the program *shouldn't* do. This kind of error of omission is behind very many vulnerabilities in code that lead to all kinds of problems. We think that this model from stories could improve our programs, and could be supported by programming or specification languages.

Here we can distinguish between traditional specifications that deal with *sufficient* conditions, versus holistic specifications that deal with *necessary* conditions [21, 38]. For example: a traditional specification for a "`login:`" method would say that a user will be logged in **if** they supply a valid username and password — that is: a valid username and password are *sufficient* to log in. The **if** here is important — or rather, what's important is that it's not **iff** (that it's not iff and only iff). To see why, consider that the "`ssh`" command, combined with public keys, in some circumstance means that "*the user can log in without giving the password*" (to quote the man page [77]). A holistic specification could help programmers make clear that a valid username and password are *necessary* to log in: thus excluding the use of `ssh` — or more accurately, excluding the use of `ssh` that does not require a valid username and password pair from a correct system. Of course, in real systems, vulnerabilities can be rather more subtle than ssh configuration files: Ken Thompson famously showed how a subverted compiler (producing the code captioned **FIGURE 3.2** below) could lead to a subverted system without any evidence appearing in the program's source code (**FIGURE 3.1**) [65]:



```
compile(s)
char *s;
{
        ...
}
```

**FIGURE 3.1.**

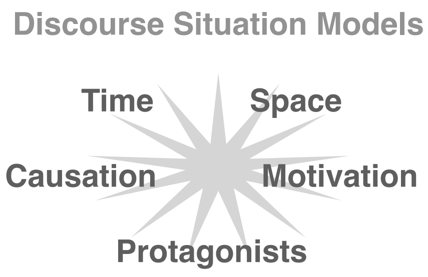


```
compile(s)
char *s;
{
        if(match(s, "pattern")) {
                compile("bug");
                return;
        }
        ...
}
```

**FIGURE 3.2.**

As in natural language, concealment and misdirection may be intentional or accidental, yet still have effect, and not always benevolent to the reader [26]. Even when the intention *is* benevolent, unexpected problems may occur. In User Interface design, concealment and re-representation of detail is critical to usable design, and this can obscure dangers that arise within the hidden detail[57]. The same issues need to be considered with encapsulation.

If we look at more recent work in discourse analysis, a lot of work in recent decades have focused on what Van Dijk and Kintsch called "situation models" [69]. These are the mental models that someone develops when they hear or read a story. Long after the text is gone, we can reflect on a story, we can learn from it, we can reason on that basis, or even translate and write the story in a different language — which is what translators do.



Psychologists suggest that these models might represent the cognitive processes involved in, engaging with, or understanding, stories [78] — and so they talk about the dimensions of a text that can lead to effective situation models: time, the sequence of events; the space in which the story takes place; causation, why things happen; motivation, why people are trying to make things happen; and the protagonists themselves, who embody agency in stories. Although this is still an ongoing research effort in psycholinguistics, we think this work holds insights and lessons for programming and programming language design. If programs are stories, then the cognitive apparatus that allows us to work with stories will help us work with with programs.

## 11 Conversation

"Conversation is the fundamental site of language use. For many people, even for whole societies, it is the only site, and it is the primary one for children acquiring language.
*— Clark and Wilkes-Gibbs*

Where do stories come from?
What comes first, writing or speaking?

Clark and Deanna Wilkes-Gibbs argue that stories come from conversation [13]. In much the same way, if programs are stories, then programs will also come from conversation — or in our case, the *act* of programming. Although not obvious in the text-obsessed West [67] — closely reading everything from the Bible to the Magna Carta to the Constitution to the Treaty of Waitangi to the ssh manual page — conversation is the *primary* site of language use[28]. When children acquire language, they begin in conversation. People don't learn to read and then learn to speak. People don't make up presentations or write essays all in one go. They talk it over, and over and over again, including during the essay presentation itself[29].

Programming is a conversation with a machine, as well as with other programmers, as well as with oneself. The next example is a excerpt of a conversation with Rust. This uses just a command line — a conversation doesn't require a sophisticated IDE. Even the one bit of information you get back from a compiler or interpreter (either "yes, I have run this", or "No, I haven't.") is just enough to sustain a conversation — albeit a conversation that's not terribly interesting, and indeed rather frustrating if you're the person who's only getting one bit back from your interlocutor.

```
warning: unnecessary parentheses around `while` condition
   --> regex1.rs:167:11
    |
167 |       while (expressions.peek().is_some() &&
    |  ___________^
168 | |                targets.peek().is_some())
    | |________________________________________^
    |
    = note: `#[warn(unused_parens)]` on by default
help: remove these parentheses
    |
167 |     while expressions.peek().is_some() &&
168 |              targets.peek().is_some()
    |
```

The point of this example, though, is Rust complaining at the programmer. Rust is upset, whinging to the programmer that "*you've put parentheses around your while loops*". [30][31]

All those languages put parenthesis around while loops: when scanning code, we don't just look for "`while`", we search for the combination of the keyword, whitespace, and a condition *delimited by parentheses* — "`while ( . . . )`". Language is always changing: programming languages as much as other languages (and it's because programming language *is* language that it makes sense to enthoerise[32] programming language *as* language). Rust is evolving away from that tradition, and so Rust `while` loops no longer require parentheses. The sweet Rust compiler is trying to be helpful. Rust is telling the programmer that they've got unnecessary parenthesis around their while condition (this is on by default)…and while they may have written a very good program, they would write an even better program if they remove those parentheses[33]. Rust is like the teacher who says "*you're in the classroom now! You're not in the playground, talking to your friends, Java and C++. Here in the classroom, you will speak proper!*"

The ultimate goal of all computer science is the program [49]. The ultimate goal of all Scheme programmers is the REPL [63] — the Read / Eval / Print / Loop:

```
(define (REPL env)
  (print (eval env (read)))
  (REPL env) )
```

If we look at this Scheme REPL — written by Guy Steele personally, because the ultimate goal of all Scheme code is to *become* written by Guy Steele personally — and we know this is Scheme (thus Guy Steele) rather than Lisp (thus Dick Gabriel) because the code says "`define`" instead of defun "`defun`"[34] — GOTO the REPL, thou sluggard; consider her ways, and be wise. The REPL, the Silurian hypothesis, the Dark Brandon, the reptilian brain of programming. With a REPL, programmers can type things and see how they turn out. With a REPL, programmers can:

> *"see with eye serene/*
> *the very pulse of the machine;/*
> *A Being breathing thoughtful breath/*
> *A Traveller between life and death"* [76].

All programming is conversation; programs are stories that come from conversation; stories we liked so much that we bought the company. This essay is a conversation of a conversation of conversation. When you have a conversation, there are many things to say. When you have a conversation, you can never be sure what will be said, and what will not. When you have a conversation, you can never be sure how it will turn out.

## 12 Time to Face the Strange Changes

The last rule we consider about natural languages (if there are rules about natural languages) is that they are continually changing. This insight goes back to Sassure, the founder of structural linguistics [18, 19], who talked about the difference between a diachronic and a synchronic analysis: a

[28]This essay is a semi-automatic transcription of a virtual conversation.
[29]We're easy to find: Let's talk about programming as conversation.
[30]We've been programming in C-syntax derived languages such as Java, Javascript, and C for *several decades*. Whose while loops are they, anyway? Mine. My precious!
[31]Where "programming" really means "teaching undergraduates to program."
[32]entheta-ize?
[33]Thus not be quite so Australian in the programs they are writing.
[34]Scheme is thus less fun that Lisp; luckily there's a Racket that adds any amount of fun back into Scheme [15]

synchronic analysis is about the relationships between language elements at one particular time, while a diachronic analysis studies the way a language changes over time.

So, mashups, creoles, new languages coming into being — this is perfectly normal: this is part of what makes language. Sometimes we want to make new distinctions. Sometimes we want to get rid of distinctions which turned out not to matter, or which no longer matter. Sometimes we're changing language intentionally, explicitly, for technical reasons, reasons of correctness, reasons of comprehension. Sometimes we're doing it for business reasons, for reasons of prestige, sometimes we're doing it as a way of fighting back or resisting, and sometimes even as a way of reconciling. Sometimes language seems to change of its own accord, or rather we're doing it implicitly, accidentally, for reasons of which we are unaware, or for no reason at all. Over time, language elements get deprecated, languages become endangered, become extinct.

## 13 Conclusion

When we presented a version of this work to some Americans, they asked "what are the takeaways?" Of course the "takeaways" must depend on each reader's perspective: programmer? end-user? researcher? or language designer? We leave these for later conversations. We acknowledge our focus on similarities with natural languages, and also leave for later the issue of differences.

In spite of the extraneous detail[35], we hope this essay, like the talk it was based upon, has managed to communicate a few key ideas: that programming is a conversation between programmers, and increasingly between the programmer and the machine; that programs, the outcomes of those conversations, are stories about how a domain is described and simulated; and that the language we used to write those stories — programming language — is language, as much as any other [7].

Language is not the accident of programming — it is the essence.

## References

[1] Laurie Anderson. 1982. Let X = X. In *Big Science*. Warner.

[2] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust types for modular specification and verification. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 147:1–147:30.

[3] Maurice George Balme and James Morwood. 1997. *Oxford Latin Course.* Oxford University Press.

[4] Elisa Baniassad and Clayton Myers. 2009. An Exploration of Program as Language. *SIGPLAN Not.* 44, 10 (oct 2009), 547–556. https://doi.org/10.1145/1639949.1640132

[5] D. W. Barron, J. N. Buxton, D. F. Hartley, E. Nixon, and C. Strachey. 1963. The Main Features of CPL. *Comput. J.* 6, 2 (Aug. 1963), 134–143.

[6] Kent Beck and Ward Cunningham. 1989. A laboratory for teaching object oriented thinking. *ACM Sigplan Notices* 24, 10 (1989), 1–6.

[35] and the bad jokes in the footnotes [47].

[7] Robert Biddle and James Noble. 2002. *Studying the Language of Programming.* Technical Report CS-TR-02-5. Victoria University of Wellington. This paper was presented at the Feyerabend Workshop at the European Conference on Object-Oriented Programming. Malaga, Spain, 2002. Available from http://www.mcs.vuw.ac.nz/comp/Publications/archive/CS-TR-02/CS-TR-02-5.pdf.

[8] Alan Bloom. 1991. *The Republic of Plato, Second Edition.* Basic Books.

[9] John Boyland, James Noble, and William Retert. 2001. Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In *ECOOP*. 2–27.

[10] Ronald F. Brender. 2002. The BLISS programming language: a history. *S–P&E* 32 (2002), 955–981.

[11] Walter Bright, Andrei Alexandrescu, and Michael Parker. 2020. Origins of the D programming language. *Proc. ACM Program. Lang.* 4, HOPL (2020), 73:1–73:38.

[12] Noam Chomsky. 1965. *Aspects of the Theory of Syntax.* MIT Press.

[13] Herbert H. Clark and Deanna Wilkes-Gibbs. 1986. Referring as a collaborative process. *Cognition* 22, 1 (1986), 1–39.

[14] David G. Clarke, John M. Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *OOPSLA*. 48–64. https://doi.org/10.1145/286936.286947

[15] Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. 2007. Advanced Macrology and the Implementation of Typed Scheme. In *ICFP workshop on Scheme and Functional Programming.*

[16] Ward Cunningham. 2014. FORTH Reusability. https://wiki.c2.com/?ForthReadability.

[17] Ole-Johan Dahl and C. A. R. Hoare. 1972. Hierarchical Program Structures. In *Structured Programming*, Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare (Eds.). Academic Press.

[18] Ferdinand de Saussure. 1916. *Cours de linguistique générale.* V.C. Bally and A. Sechehaye (eds.), Paris/Lausanne.

[19] Ferdinand de Saussure. 1966. *Course in General Linguistics.* McGraw-Hill.

[20] What Is Responsibility-Driven Design. 2006. A Brief Tour of Responsibility-Driven Design. (2006). https://www.wirfs-brock.com/PDFs/A_Brief-Tour-of-RDD.pdf.

[21] Sophia Drossopoulou, James Noble, Julian Mackay, and Susan Eisenbach. 2020. Holistic Specifications for Robust Programs. In *Fundamental Approaches to Software Engineering (FASE).* 420–440.

[22] Eclipse Foundation 2020. *Jakarta Enterprise Beans Specification version 4.0.* Eclipse Foundation.

[23] A. J. Greimas. 1983. *Structural Semantics.* University of Nebraska Press.

[24] Stephen Halliwell. 1998. *Aristotle's poetics.* University of Chicago Press.

[25] Haskell Contributors. 2021. `System.IO.Unsafe`. https://hackage.haskell.org/package/base-4.18.0.0/docs/System-IO-Unsafe.html. [Online; accessed 28-April-2023].

[26] Zahra Hassanzadeh, Robert Biddle, and Sky Marsen. 2021. User perception of data breaches. *IEEE Transactions on Professional Communication* 64, 4 (2021), 374–389.

[27] C.A.R. Hoare. 2009. Null References: The Billion Dollar Mistake. (2009).

[28] Douglas H. Hofstader. 1979. *Gödel, Escher, Bach: An Eternal Golden Braid.* Penguin Books.

[29] Ted Honderich (Ed.). 1995. *The Oxford Companion to Philosophy.* Oxford University Press.

[30] Ralf Hütter, Florian Schneider, and Karl Bartos. 1981. It's More Fun to Compute. In *Kraftwerk: Computer World.* Kling Klang.

[31] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *PACMPL* 2, POPL, Article 66 (Jan. 2017), 66:1–66:34 pages.

[32] Poul-Henning Kamp. 2010. Sir, Please Step Away from the ASR-33! To Move Forward with Programming Languages we Need to Break Free from the Tyranny of ASCII. *Queue* 8, 10 (oct 2010), 40–42.

[33] Donald E. Knuth. 1972. Ancient Babylonian Algorithms. *CACM* 15, 7 (1972), 671–677.

[34] Stephen Lawrence, Fellcity Graham, and Christian Hearn. 2016. "You FUCKING BEAUTY" "Fuck Fred Nile"' and other inoffensive comments. https://criminalcpd.net.au/wp-content/uploads/2016/11/You-Fucking-Beauty-Fuck-Fred-Nile-Stephen-Lawrence-Feliciy-Graham-Christian-Hearn.pdf.

[35] K Rustan M Leino. 2013. Developing verified programs with Dafny. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 1488–1490.

[36] K. Rustan M. Leino. 2020. *Program Proofs*. Available from Lulu.com.

[37] Dyani Lewis. 2021. Science agency on trial following deadly White Island volcano eruption. *Nature* 598 (2021), 243–244.

[38] Julian Mackay, Susan Eisenbach, James Noble, and Sophia Drossopoulou. 2022. *Necessity* specifications for robustness. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 811–840.

[39] Bruce MacLennan. 1983. *Principles of Programming Languages: Design, Evaluation, and Implementation*. Holt, Rinehart, and Winston.

[40] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kirsten Nygaard. 1993. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley.

[41] Sky Marsen, Robert Biddle, and James Noble. 2003. Use case analysis with narrative semiotics. *ACIS 2003 Proceedings* (2003), 86. https://aisel.aisnet.org/acis2003/86.

[42] John McCarthy. 1959. LISP: a programming system for symbolic manipulations. In *Preprints of papers presented at the 14th National Meeting of the Association for Computing Machinery*. 1:1–1:4.

[43] John McWhorter. 2023. Why Do Languages Have Gender? slate.com-podcasts-lexicon-valley/2021/01/language-gender-noun-classes.

[44] Bertrand Meyer. 1992. *Eiffel: The Language*. Prentice Hall.

[45] Microsoft Corp. 2020. Coding Style Conventions. learn.microsoft.com.

[46] Eric Mottram. 1959. The Cat Sat on the Mat. *Poetry* (Feb. 1959).

[47] Preet J. Nedginn and Trebor L. Bworn. 1984. CLOG:6,SM,©,21 an Ada® Package for Automatic Footnote Generation in UnixTM. *Commun. ACM* 27, 4 (apr 1984), 351. https://doi.org/10.1145/358027.358044

[48] Greg Nelson (Ed.). 1991. *Systems Programming with Modula-3*. Prentice-Hall.

[49] James Noble and Robert Biddle. 2002. Notes on Postmodern Programming. In *Onward! Onward! Ever Onward!*

[50] James Noble, Jan Vitek, and John Potter. 1998. Flexible Alias Protection. In *ECOOP*. 158–185.

[51] Kristen Nygaard and Ole-Johan Dahl. 1978. The Development of the SIMULA Languages. *SIGPLAN Not.* 13, 8, 245–272. https://doi.org/10.1145/960118.808391

[52] Seymour Papert. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, Inc.

[53] Terry Pratchett. 1983. *The Colour of Magic*. Colin Smythe.

[54] Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore. 1996. The Evolution of Forth. In *History of Programming Languages—II*. Association for Computing Machinery, New York, NY, USA, 625–670. https://doi.org/10.1145/234286.1057832

[55] Rose Royce. 1976. Car Wash. MCA Records.

[56] C. Simonyi. 1976. Meta-programming: a software production model. PARC Technical Report CSL-76-7.

[57] Eric Spero and Robert Biddle. 2021. Out of Sight, Out of Mind: UI Design and the Inhibition of Mental Models of Security *(New Security Paradigms '20)*. Association for Computing Machinery, New York, NY, USA, 127–143.

[58] StackOverflow Contributors. 2021. Is there ever a good reason to use unsafePerformIO? stackoverflow.com/questions/10529284/is-there-ever-a-good-reason-to-use-unsafeperformio. [Online; accessed 28-April-2023].

[59] Friedrich Steimann. 2017. Replacing Phrase Structure Grammar with Dependency Grammar in the Design and Implementation of Programming Languages *(Onward! 2017)*. Association for Computing Machinery, New York, NY, USA, 30–43.

[60] Friedrich Steimann. 2023. A Simply Numbered Lambda Calculus. In *Eelco Visser Commemorative Symposium (EVCS 2023) (Open Access Series in Informatics (OASIcs), Vol. 109)*, Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 24:1–24:12. https://doi.org/10.4230/OASIcs.EVCS.2023.24

[61] Friedrich Steimann and Marius Freitag. 2022. The Semantics of Plurals. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering* (Auckland, New Zealand) *(SLE 2022)*. Association for Computing Machinery, New York, NY, USA, 36–54.

[62] Sun Microsystems 2002. *Enterprise JavaBeans Specification version 2.3*. Sun Microsystems.

[63] Gerald Sussman and Guy Steele. 1975. *SCHEME: An Interpreter for Extended Lambda Calculus*. Technical Report AI Memo 349. MIT Artificial Intelligence Laboratory.

[64] Lucien Tesnière. 2015. *Elements of structural syntax*. John Benjamins Publishing Company. Translated by Timothy Osborne and Sylvain Kahane.

[65] Ken Thompson. 1984. Reflections on Trusting Trust. *Commun. ACM* 27, 8 (aug 1984), 761–763.

[66] TIOBE 2022. TIOBE Index for June 2022. https://www.tiobe.com/tiobe-index.

[67] Eve Tuck and K. Wayne Yang. 2012. Decolonization is not a Metaphor. *Decolonization: Indigeneity, Education & Society* 1, 1 (2012), 1–40.

[68] David Ungar. 2003. Seven Paradoxes of Object-Oriented Programming Languages. www.oopsla.org/oopsla2003/files/key-4.html. Invited Keynote presentation to OOPSLA 2003..

[69] Teun Adrianus Van Dijk and Walter Kintsch. 1983. *Strategies of Discourse Comprehension*. Academic Press.

[70] Larry Wall. 1999. Perl, the first postmodern computer language. (Spring 1999). `http://www.wall.org/ larry/pm.html`.

[71] Charles Weir, Awais Rashid, and James Noble. 2020. Challenging software developers: dialectic as a foundation for security assurance techniques. *J. Cybersecur.* 6, 1 (2020).

[72] Michaela Whitbourn. 2016. Court finds 'f— Fred Nile' not offensive language at marriage equality rally. *Sydney Morning Herald* (Oct. 2016). https://www.smh.com.au/national/nsw/court-finds-f-fred-nile-not-offensive-language-at-marriage-equality-rally-20161025-gs9ymc.html.

[73] Wikipedia contributors. 2023. Declension — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Declension&oldid=1149572805. [Online; accessed 29-April-2023].

[74] Wikipedia contributors. 2023. Word order — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Word_order&oldid=1151379926. [Online; accessed 28-April-2023].

[75] Oscar Wilde. 1988. *The Importance of being Earnest: a trivial comedy for serious people*. Leonard Smithers.

[76] William Wordsworth. 1807. She Was a Phantom of Delight. In *Poems, in Two Volumes*. Longman, Hurst, Rees, and Orms. Paternoster Row.

[77] Tatu Ylonen, Aaron Campbell, Bob Beck, and Markus Friedland Niels Provos. 2023. ssh – OpenSSH remote login client manual page.

[78] Rolf A. Zwaan and Gabriel A. Radvansky. 1998. Situation models in language comprehension and memory. *Psych. Bull.* 132, 2 (1998), 162.

Received 2023-04-28; accepted 2023-08-11