Say “Hello World!”.

Sandro Ansari

May 2025

Abstract

DroLang is a programming language that looks like natural language. Leveraging React with Vue and Microsoft's ts-parsec library, the goal of this project was to show that a programming language can be made to look like English. Not only that, but on a fundamental level, the interpreter for DroLang relies on semantic composition in order to derive the meaning of its sentences. While this type of language is largely inefficient, there are benefits in accessibility and interesting tie-ins with the field of semantics.

Background

The simplest way to describe my project is a programming language which appears naturalistic. The original programming languages were punch-card based and tied in to the machinery they worked on. Since then, new languages have evolved from low level machine code into higher level abstracted systems. The evolution from Fortran to C to Python (to name a few) showcases how languages have moved in a direction allowing for programmers to more abstractly tell the computer what to do. DroLang is the natural next step in that evolution.

The original vision behind my project was being able to give the computer something that looked like pseudo-code and the machine would do the rest. Obviously, that is a herculean task. Semanticists do not even have an idea what a "chair" is. How on earth would I be able to teach a computer that. Programming languages also need to be deterministic for them to be useful, so leveraging large language models was out of the question. Thus I decided to steer away from these real language complexities while maintaining the benefits of readable, accessible code.

Instead of a natural language programming language, DroLang was going to be a programming language wolf in natural language sheep's clothing. The grammar and lexicon would not match the scale of English by any means, but would offer a decent amount of variability in terms of grammatical framing. Programmers would be able to outline their programs in the way they like while keeping the program execution deterministic.

While a fully implemented natural programming language has the potential to solve many problems, the development of DroLang pimarily focused on the problem of accessibility. Software developers are the first thought when it comes to programming, but many others need to use computers to accomplish complicated tasks in research and other areas. Many researchers use various programs to collect and analyze data, run simulations, and perform other tasks. However, for many, programming is not familiar or easy to grasp. In my own experience teaching programming to children at a summer camp, things as elementary as variables

can be incredibly baffling when you've never been exposed to it.

A programming language that looks like a language you already speak can make working in that language easier for many reasons. You do not need to learn new syntax or keywords or modules or libraries. You can also understand others' code a lot easier if you can just read it like natural English.

This project has provided me an opportunity to explore syntax and semantics on the linguistics side of things while comparing those structures to analogous ones in programming syntax and structures. Figuring out how to allow for different frames of verbs or the multiple ways we talk about loops is the core intriguing component of this problem for me.

I have delved deep into parsers and semantic theory and learned how to make them work together. Having taken classes in syntax, semantics, and compilers, this is not an entirely new area for me, but my understanding of these topics have expanded beyond the scope of those classes. This has allowed me to assemble an artificial recreation of natural language with a lot of expressive power and flexibility.

Naturalistic programming languages have been attempted before with varying degrees of success and depth. My contribution to this field lies not in new goals but in the different strategies employed in my execution. Most projects I have found in this vein attempt to mimic and/or understand natural language in its entirety. This allows for a broader use of other natural languages or more idiomatic speech. DroLang differs from these since it is first a programming language with the appearance of natural language secondary. The main benefit of being accessible is preserved while the pitfalls of ambiguity and the incredible scope of natural language are avoided. DroLang does not understand what you say but is rather a machine that does what it is told to do.

This approach does not offer as much expressive freedom as a fully implemented natural programming language but is still a useful tool. When you progress away from the machine level you lose efficiency. This project does not lose as much efficiency as a comprehensive naturalistic language since it is still a programming language (albeit with some very English-looking syntax). This is not to say it rivals C, but rather that it does not suffer as much from the translation of English or some other language into code.

Being able to read, write, and share code efficiently and easily without much extra knowledge of syntax etc. makes for very effective and portable applications. As stated earlier, I am by no means the first to attempt this, so here is a sample of some previous efforts.

Many tools have looked to leverage natural language to various purposes. The language COBOL attempted to simulate English with it is syntax [Liu and Lieberman, 2005]. This meant defining control structures in a way that matched English framing as shown in Listing 1.

Listing 1: Cobol Loop Frame

```
PERFORM [n] TIMES
        [Statements]
END-PERFORM
```

Listing 1 somewhat mimics the English framing used to show repeated actions. Other languages, including Python, also have syntax that can be somewhat read like English. Readability is important for people to quickly and easily understand code. However, some programming languages gone the extra mile.

Metafor is a project designed for outlining programming projects without over-committing to any given implementation [Liu and Lieberman, 2005]. This project takes in natural language "stories" and creates outlines of classes and methods. It figures out which components ought to be properties and which ought to be classes among other things. This allows individuals to brainstorm without tying themselves to any particular structure. This means programmers can fail faster, and quickly prototype ideas.

Another approach I discovered was Pegasus [Knöll and Mezini, 2006]. Pegasus uses many advanced components including short-term memory to store context, long-term memory to store semantic information, and a way of matching all the components of the supplied sentence to those memories. It is a very sophisticated system that gets closer than many to the natural language ideal at the cost of some performance issues.

Pegasus and Metafor are closer to natural language than other programming languages in terms of their understanding of the input text. However other attempts simply merely try to mimic the appearance of natural language without the computer truly understanding the underlying linguistic phenomena the way Pegasus or Metaphor might. There have been many other examples of natural language approaching languages (multiple of which are listed in Knöll & Mezini). Some of these experiments had interesting ideas applicable to my project. A handful of them allow for fluffy constructions like "I would like ..." or "Please ..." These increase the flexibility. Also, the ability to store the inflections of various words and assign specific meanings to them is very interesting. Not all of these features were feasible to add, but some more fundamental pieces like alternate grammatical frames and a semblance of real semantic composition make DroLang more like English than many standard programming languages.

Goals

The goals of DroLang could be grouped into a few key groups: computational expressivity, natural language freedom, and linguistic fidelity. Balancing these three factors became a constant tug of war. Compromises had to be made in many places, but this push and pull of these three domains kept DroLang focused on its

true objective—becoming a programming language that looks like a natural language.

Computational Expressivity

The first and most important aspect of a programming language is its ability to give instructions to a machine. An important metric here is whether or not the language is Turing complete. The Church-Turing thesis states that any problem solvable by an algorithm can be solved by a Turing machine (a model of computational device with exceptionally simple parameters). If a programming language can be used to simulate a Turing machine's actions, then it is proven that that language can solve any algorithmic problem.

DroLang being Turing complete is as good as any programming language can get. Instead of proving that a language can simulate a Turing machine, one can also look at its features checking for branching control flow (if statements), looping control flow (while loops), and memory storage (vairables). If all these features are present, then the language is Turing complete.

Natural Language Freedom

This goal is the crux of the project. Users should have as much freedom with their input language as possible. The theoretical maximum of natural language freedom would be a language that can accept any valid sentences of English and work. Due to the scope of natural language, this is entirely unfeasible. However, there are a few features that have been included to support this goal. The most important one is alternative argument frames. In DroLang, verbs take arguments just like in English. In English though, there are many ways of supplying those arguments to the verb. Allowing for multiple of these forms is the key way that DroLang supports natural language freedom.

Linguistic Fidelity

The last major goal was to incorporate as much real linguistic theory as possible to allow for increased language freedom and to offer a strong base for the parsing strategies DroLang employs. This goal is actually really important when trying to work with the first two. To allow for language freedom, I could have implemented a large number of manually created frames for all sorts of operations, but this would not be scalable and would make the computational expressivity limited to whatever those frames did. My actual approach leverages the lambda-calculus nature of natural language semantics to allow for exceptionally expressive sentences that are very English-like.

Planning

Creating DroLang has been a long and storied process with many ups and downs. The first step, as with many projects, is researching what others have done in the field. As mentioned earlier, the history of natural-looking languages is an eventful one. Starting with the advent of programming languages, programmers quickly began to bridge the computer-human communication gap. NLC is an unimplemented language designed by Bruce Ballard and Alan Biermann [Ballard and Biermann, 1979]. Their paper outlines a compelling argument for components a natural looking programming language needs and the types of data that such a language should be able to handle. Their model included a scanner, a parser, a semantic engine, and a matrix computer. DroLang has the first three of these pieces as well. I utilize the ts-parsec library's built-in lexer and parser tools to tokenize the incoming sentences and then generate a pseudo-syntax tree. I can then run the top node of the tree and it will compose the meanings from across the syntax trees.

While NLC was a concept, as mentioned earlier, some other languages have gone above and beyond to reach true natural langauge parity. Pegasus stores concepts and ideas in a large network of terms and uses a complicated analysis of sentences at various levels to generate and execute the meaning of the input [Knöll and Mezini, 2006]. This system takes an incredibly comprehensive approach to natural langauge to the point where the program legitimately does its best to actually 'understand' the input and resolve it according to its memory and background knowledge. The fact that a program can actually understand (as far as anything without consciousness can) what its being told to do is truly incredible.

Pegasus actually interpreting its input is in stark contrast with another popular avenue in this field A.I. Artificial intelligence, large language models (LLMs) in particular, has grown incredibly strong in recent years in terms of its ability to understand natural language prompts. There are many tools now from GitHub Copilot to Claude AI that leverage these models and large amounts of online data to generate code for you based on simple pseudo-code-esque prompts. I knew from the beginning however, that this is not the direction I wanted to go. A programming language by definition, should make the computer do what its told to do. You should be able to tell the computer exactly what you want it to do. This should be true relative to the level of abstraction inherent to your language. In C++, you have to worry about pointers and memory allocation, and whatever you tell it to do, it will do (even if it inevitably causes a seg. fault). In Python, you do not need to worry about memory management, but if you tell it to print something it will do exactly that.

I did not want to use LLMs because they don't do exactly what you tell them to do. One of these models will grab bits and pieces from all over the web to cobble together that only probably will do what you want, and even if it does work, there's no guarantee it will do it in an efficient, or secure manner [Fu et al., 2023].

This is a big factor in my goal of computational expressivity. Any user should be able to both write what they want the machine to do and have it completed accurately, and another should be able to read that person's code and know what it will do. This drove me in the direction of natural language parsers.

Initially I looked into the Spacy library for Python, the NLTK library (for various languages) and the Stanford Dependency Parser [Marneffe et al., 2006]. These parsers unfortunately did not mesh with my use-case for two reasons. First, the dependency parsers only did so much to give me a syntax tree. Dependencies do not directly translate to a semantically composable tree. Second, these parsers allowed for entirely too much freedom and thus would make it difficult to restrict the scope of natural language allowable. This was one situation in which natural language freedom was somewhat sacrificed for computational expressivity.

Product

Discussion

Reflection

References

[Ballard and Biermann, 1979] Ballard, B. and Biermann, A. (1979). Programming in natural language: "nlc" as a prototype. pages 228–237.

[Fu et al., 2023] Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M., Yu, J., and Chen, J. (2023). Security weaknesses of copilot-generated code in github projects: An empirical study. *ACM Transactions on Software Engineering and Methodology*.

[Knöll and Mezini, 2006] Knöll, R. and Mezini, M. (2006). Pegasus: first steps toward a naturalistic programming language. pages 542–559.

[Liu and Lieberman, 2005] Liu, H. and Lieberman, H. (2005). *Metafor: visualizing stories as code*.

[Marneffe et al., 2006] Marneffe, M., MacCartney, B., and Manning, C. D. (2006). Generating typed dependency parses from phrase structure parses. pages 449–454.