UNIVERSITÀ DI PAVIA

UNIVERSITY OF PAVIA
FACULTY OF ENGINEERING
DEPARTMENT OF ELECTRICAL, COMPUTER AND BIOMEDICAL ENGINEERING

MASTER'S DEGREE IN COMPUTER ENGINEERING

MASTER THESIS

***Deixiscript*: Exploring and Implementing a Common Sense Approach to Naturalistic Programming**

Deixiscript: esplorazione e implementazione di un approccio "di buon senso" alla programmazione naturalistica

Candidate: Aiman Al Masoud

Supervisor: Prof. Marco Porta

A.Y. 2022/2023

Abstract

Naturalistic programming is broadly defined as the attempt to write computer code in a varyingly complex subset of natural language. This work explores how existing naturalistic programming languages try to bring program specifications closer to the way humans naturally describe things and processes, relating it to the broad topic of “Common Sense” in Artificial Intelligence. Moreover, the thesis proposes *Deixiscript*, a new prototype of rule-based naturalistic language, which follows the principles of the languages surveyed, but also introduces a notion of *automated planning* to facilitate declarative programming.

La programmazione naturalistica si può definire generalmente come il tentativo di scrivere del codice sorgente in un sottoinsieme del linguaggio naturale di diversa complessità. Questo lavoro analizza come i linguaggi di programmazione naturalistici esistenti cercano di avvicinare la scrittura di un programma al modo in cui gli esseri umani naturalmente descrivono entità e processi, e di metterlo in relazione con il più ampio argomento del “Buon Senso” (“Common Sense”) nell’ambito dell’Intelligenza Artificiale. La tesi propone inoltre *Deixiscript*, un nuovo prototipo di linguaggio naturalistico ‘rule-based’, che segue i principi dei linguaggi naturalistici esaminati ma introduce anche una nozione di *pianificazione automatica* per agevolare la programmazione dichiarativa.

Contents

1 Introduction

1.1 What is naturalistic programming?

Naturalistic programming is broadly defined as the attempt to write (computer executable) code in a varyingly complex *subset* of a natural language: the kind of language we are all naturally familiar with and brought up speaking from a very early age.

We say “a *subset* of natural language” because full, unconstrained, natural language is (at least we think) still far too complex and dynamic to be used as a feasible programming language, without any constraints or limitations put in place whatsoever; and no one so far has agreed entirely on what that *subset* should look like, let alone what it should (or should not) allow us to say.

It was (and maybe still is) customary to dismiss all these attempts with the justification that natural language is too “imprecise”, too “vague” or too “ambiguous” for telling computers what to do, and that this job better suits formal languages anyway.

Edsger Dijkstra (1930-2002), the late, great computer scientist and early advocate of structured programming, famously wrote a piece in 1978 titled: “On the foolishness of “natural language programming” [1]. In it he discusses what he (rightly) sees as the defects of natural language (ambiguity, verbosity, etc.) compared to formal languages when describing mathematical concepts and computer algorithms.

He goes as far as to argue that even if “natural language programming” were ever achieved, it would not be a step in the direction of scientific progress, but rather a step back into the “dark” ages of mathematical notation: the rhetorical stage, back when equations were expressed in long, convoluted, confusing words, rather than in elegant and concise symbols.

And yet, we still speak natural language everyday: we listen to the news, we read books, we speak to our dearest friends and relatives, we trust natural language with the preservation of our legal rights and statutes, with the preservation of our very democracy! How can such a vague, ambiguous, imprecise linguistical system (or systems) not be trusted at making sure our computers do not crash, yet be fully trusted in making sure the rest of our human society does not come crumbling down like a castle of cards? There is clearly more to say about natural language.

1.2 Programming is dead

Though it may sound paradoxical, in light of what we just said, natural language may indeed be headed to become the primary means of interaction between us and our computers, in the short to medium term.

But this is not a paradox: the last decade (the 2010s) has seen the flourishing and surge in popularity of Machine Learning (ML) techniques on the scene of Artificial Intelligence (AI) [2], along with the development of new Deep Learning architectures such as the Transformer [3].

Modern day Large Language Models (LLMs) can say and do things most of us never imagined were possible (in the practical sense) for a computer program before; though everyone knew, of course, of the theoretical possibility for such apparently highly “intelligent” behavior, as evidenced also by thought-experiments such as John Searle's (1932-) famous Chinese Room argument, which we talk about in "Common Sense" Section 4.

Matt Welsh (1976-), computer scientist and software engineer, has written an article in January 2023 titled "The End of Programming" [4]. Welsh believes that the entire field of Computer Science is headed for an upheaval, that it will not be even *remotely recognizable* to what it is today, 10 to 30 years from now.

Programming (intended as humans explicitly writing computer programs, in a formal language) will be dead by then, according to Welsh; of course, one will still be able to write programs in some formal language for his/her own amusement, but the serious work will be done by Artificial Intelligence models.

Our role as human beings in relation to computers might turn into an "educational" one (rather than an "engineering" one, as it is today); we will have to "educate" machines, rather than to program them or even to "train" them with data (in the current Machine Learning sense of the word "train"). Of course, we will be "educating" these "temperamental, mysterious, adaptive agents" (as Welsh says), just like we educate our children, speaking to them in natural language.

1.3 Long live Natural Language

This huge paradigm shift, that is awaiting us all, may sound like a great prospect. We will finally ditch those abstruse, unnatural, archaic ensembles of symbols, numbers and brackets, that a relatively restricted percentage of the human race still uses daily to move bits around in memory, and to light pixels on displays. More people will finally have a (more intuitive) access to their machine's capabilities.

Will the new normal really be so good? We certainly hope that it will, but we also fear that this *great leap forward* has all the potential to come with its fair share of troubles, and not just the ones related to the economic [5] effect of this paradigm shift on all people (not just the ones in software engineering), or the legal issues [6] regarding the copyright of the data consumed or produced by AI-models.

As Matt Welsh reminds us in his article "AI-based computation has long since crossed the Rubicon of being amenable to static analysis and formal proof". This can turn a big modern-day AI-system (from a big corporation, for example) into a dangerously powerful black-box, unless due caution is exercised in its deployment; and more danger may even come if it gets "orders" from untrained personnel, in the highly ambiguous medium that is unconstrained natural language.

To make matters worse, a big thinker like Noam Chomsky (1928-) has expressed, in a recent interview in May 2023 [7], his skepticism in our ability to control the threats posed by a potentially super-human artificial intelligence going haywire, suspecting that "the genie is out of the bottle" but nonetheless stating that "such suspicions are of course no reason not to try, and to exercise vigilance".

But the interview we just cited was not mainly about the threat of super-human Artificial General Intelligence (AGI) wreaking havoc in the world, as serious as it may be (and it is). The interview was mainly about the inadequacy of current Deep Learning models, such as the popular

GPT (Generative Pre-trained Transformer), at actually modelling human linguistic thought.

Chomsky argues that while systems like GPT are very good at capturing statistical patterns in natural language, they also work just as well with any “impossible language”, i.e., with artificial grammars that infants do not learn spontaneously; making them an unreliable model of (at least) human language acquisition.

Moreover, we can add, such models need huge amounts of data to achieve a level of linguistic competence that is even comparable to that of a human being; a great human author does not need to read a *million* books before writing his/her *magnum opus*, maybe a couple thousands or less.

Chomsky expresses his concern for the shift from a more “science-oriented” AI (i.e., geared towards understanding the phenomenon) to a more “engineering-oriented” one (i.e., just aiming to build a useful tool); this may lead to the risk of disillusionment for important scientific discoveries in the general public.

Chomsky says that a person arguing that studying the extraordinary navigational capacities of ants is “useless” (because we modern humans have maps, GPS satellites, digital compasses, etc.) would be laughed at. But a person making a similar argument about the formal study of language (because we already have huge statistical models churning on terabytes of input data to produce near human-level quality prose) is much more likely to be taken seriously, nowadays.

To conclude, we believe this is a critical juncture of human history, for a number of reasons, and the rapid proliferation of AI systems trained on huge amounts of data is certainly an important one among them; to paraphrase an idea that quantum physicist and computer programmer Michael Nielsen (1974-) expressed in his book “Neural networks and deep learning” [8]: AI started out as an effort to understand reasoning and intelligence (maybe even shed some light on human intelligence), but it may end very soon, with us humans having understood neither how human intelligence works, nor even how the artificial intelligences we built work.

1.4 Outline of the Chapters

The second chapter is an overview of the four most popular and widely known programming paradigms of all times (procedural, functional, object-oriented and logic), presented in the guise of a brief historical summary of the evolution of programming languages in general. We think that a solid understanding of what past and modern programming languages are capable of, and at what price (both in the positive and in the negative sense), is of fundamental importance in evaluating a new paradigm, or designing a new language.

The third chapter will bring the focus on naturalistic programming specifically, and will try answering some of the following questions: which existing programming languages are naturalistic? Is “naturalistic programming” just about using a more “English-like” (natural) syntax? Can any deeper underlying common principles be discerned in these languages? Do we think naturalistic programming is a “real” (fully-developed) paradigm yet? All of the projects we will discuss, approach the subject of natural language from a classical (non ML) standpoint, they are in this sense “ordinary” programming languages, with some very out-of-the-ordinary features. We will also bring up the tangentially related topic of Prompt Engineering for the optimization of LLM responses, which we see as an emergent competing approach (implementation-wise) to building what may truly one day be called “programming in natural language”.

The fourth chapter will jump straight into the intriguing (yet difficult) topic of Common Sense in AI, and will try relating some of the principles of naturalistic programming to some of those in classical (or “Symbolic”, or “Good Old Fashioned”) AI.

The last two chapters will be dedicated to describing our work in trying to implement some of the naturalistic ideas into a new prototype language we are designing (“Deixiscript”).

2 Programming Languages

When we speak of “programming” languages in the contemporary sense, we generally refer to a particular kind of computer language (a formal language that can be processed by computers) that is suitable for what we call general-purpose programming. This usually involves the property of Turing-completeness; that is, the ability to describe the workings of a Turing Machine, the epitome of the theoretical general-purpose computing device.

Just like there are thousands of natural languages, some of which are spoken and understood all over the world, and many more of which are only used by a relatively restricted number of speakers; so too there are myriads of programming languages: some of which have achieved world-fame, and many more of which remain relatively esoteric to the general public.

The evolution of programming languages also mirrors that of natural languages, at least to some degree. It all begins with a population of original languages, then new ones arise; features are transferred from one language, or language family, to another. Eventually, some of these languages “die out”, because people no longer “speak” them [9]. Some researchers are even critical about this state of affairs [10], as they think the design of new languages should occur on a more principled approach: with more focus on the theoretical principles, with less “organic” growth, and perhaps even with the “forced retirement” of some programming languages and their replacement with newer and better alternatives [11].

It is usually much easier for a programmer to master multiple programming languages during the course of his or her lifetime, than it is for him or her to master a similar number of natural languages. This is at least in part due to the objective complexity of any natural language as compared to any programming language (we just do not notice it as human beings); but it is also in part due to the fact that programming languages (possibly even very different-looking ones) generally fall into a restricted number of groups, identified by their dominant programming paradigm.

As we will see, the most popular programming paradigms of all times are four: procedural, object oriented, functional and logic programming. Some languages tend to stick to a specific paradigm more than other languages, emphasizing the paradigm's "purity" over the possibility of a hybrid approach; other languages are known as "multi-paradigm" because they support, or even encourage, the use of different paradigms for different kinds of tasks; some paradigms mix well together, while others do so to a lesser degree.

Programming paradigms themselves generally fall into two main categories: imperative and declarative. Broadly speaking, the imperative approach is about instructing the machine on how to do something on a step by step basis, while the declarative approach specifies the requirements and leaves the more open-ended specifics of how to accomplish them to the machine.

The four programming paradigms that we mentioned above are by no means the only existing paradigms, let alone all of the possible yet undiscovered ones. It is nonetheless useful and instructive to take a closer look at the history and evolution of these few, alongside the programming languages that championed them and pioneered their use. This is what we shall attempt to do in the following pages.

The book “Concepts of programming languages”, by Robert W. Sebesta [12], offers an invaluable historical perspective on the history and evolution of programming languages and language design criteria.

2.1 The Origins

As is usually the case with many things in history, it is unclear who “s-tarted it all”. It is said that the first programmer in history was the British mathematician Ada Lovelace (1815-1852), who first described an algorithm to compute the Bernoulli numbers on the Analytical Engine, a mechanical general-purpose computer envisioned by the mathematician Charles Babbage (1791-1871). This machine, however, was never built during the lifetime of the two.

It is said that Babbage, in turn, was inspired in his project by the unlikeliest of sources: a mechanical weaving device built by the French weaver and merchant Joseph Marie Jacquard (1752-1834): the Jacquard-loom. Ada Lovelace once wrote, describing the Analytical Engine, that it could: “[weave] algebraic patterns, just as the Jacquard-loom weaves flowers and leaves” [13].

The Jaquard-loom allowed even the least skilled of textile workers to produce intricate patterns on silk, patterns which could be *stored* on the so called punched cards (or “punch” cards), a data storage technology that will be used up until more than a century later to input programs into the first electronic computers [14], [15], [9].

Despite these precursors in the mid 19th century, it is necessary to wait a little while more to see the birth of the man conventionally credited as the inventor of the first programming language: the German computer scientist Konrad Zuse (1910-1995).

His programming language was called Plankalkül (“Program Calculus” in German), and it was designed to run on the “Z4” computer, also envisioned by Zuse around the 1940s. As a language from that time period, it had some pretty high-level features such as arrays and other data-structures, it also had a peculiar *multi-line* notation for array indexing [12].

Unfortunately, with World War II raging on, and Zuse trapped in Germany, isolated from the rest of the computer engineering community, his work remained for a long time largely unknown to the general audience [12].

2.2 Machine Language

With the birth of the digital computer in the mid 1940s, the first programming languages that were developed were machine languages. A machine language is a purely numerical representation of an instruction supported by the machine’s architecture. Code in machine language can thus be directly executed by the machine; these instructions are, usually, very simple compared to an instruction in a modern high-level language. While it is theoretically possible to design a computer architecture that executes a high-level language out of the box, this is never done in practice, not even nowadays, for practical and performance-related reasons [12].

There are two problems, from the human-user's perspective, with machine languages: they are purely numerical, requiring one to memorize the numerical opcode (operation code) corresponding to a kind of instruction; and, besides that, they also require the human programmer to manually specify the addresses of the memory cells that contain the data or instructions to be referenced; this means that the programmer will have to manually change all of the (many) relevant addresses if he/she decides that an extra instruction has to be inserted right in the middle of the existing ones [12].

2.3 Assembly Language

The problems of machine languages are precisely the reason why assembly languages were invented shortly after that. An assembly language is usually a thin abstraction on top of the underlying machine code, the latter still being the only language which the processor can run out of the box. But assembly languages are much easier for a human to understand than machine code, because they provide alphabetical aliases for the numerical opcodes, and because they manage memory addresses automatically, thus relieving the programmer from the burden of having to keep track of them in his/her head [12].

2.4 Compilation vs Interpretation

Though it may sound strange to us contemporaries, in the early days of programming some of the assembly languages were interpreted rather than compiled. Compilation versus Interpretation are the two main approaches that can be used to implement any programming language, rather than corresponding to any real intrinsic feature of the language in question.

A compiled language is first translated to machine code, and the machine code (rather than the compiled language's code) is then run on the target computer. On the other hand, an interpreted language requires at least another program to be actively running on the target computer: an interpreter; the latter dynamically translates code in the interpreted language into actions by performing them on the machine.

Needless to say, pure interpretation is slower than compilation due to the extra overhead of the interpreter; but back in the early 1950s this wasn't considered a problem, because the power-hungry floating point operations had to be interpreted anyway (they weren't part of any machine's native architecture yet).

Back in the day, computers were mostly used to perform scientific computations, and floating point operations (which just means operations on decimal numbers up to a certain level of precision) were thus one of the most important kinds of operations for most programs written at the time; and nowadays we still sometimes measure computing power in FLOPS (Floating Point Operations per Second) [12].

2.5 Fortran

Things changed when the IBM 704 computer was introduced; the 704 was the first computer to incorporate native floating point operations. When the 704 was introduced in the mid 1950s, people suddenly realized that interpretation was slow, and the software team at IBM released what came to hold the symbolic title of “first high-level programming language”: Fortran, “The IBM Mathematical FORmula TRANslating System”.

Fortran survives today in some scientific computation communities, with heavy modifications from the language that originally came out in 1956; the original version of Fortran developed by John Backus (1924-2007) and his team at IBM supported basic arithmetic, limited length variable names, subroutines, the do-loop, and a construct known as the “arithmetic if”, which branched to different parts of the program based on the value of an arithmetical expression.

Fortran lacked many of the language facilities we take for granted today. There was no incremental compilation: all of the parts of a program had to be recompiled from scratch all the time, and, given the occasional unreliability and slowness of the 704 computer , this meant restricted program sizes. It also did not have any dynamic memory allocation capabilities (which were not essential for batch-processing scientific computations anyway).

Overall, Fortran is an example, among many, of a language born with a specific purpose (doing math efficiently on the IBM 704) that later evolved, incorporating new constructs, and getting ported over to numerous platforms. It is also a first example of an imperative, procedural language [12].

2.6 Structured vs Unstructured Programming

A characteristic feature of machine code, assembly languages, and the early high-level languages is perhaps their classification as “unstructured” or “non-structured” programming languages. Unstructured programming simply means having one single huge code block, and using GOTO statements to manage control flow.

A GOTO (or GO TO), as the name implies, instructs the machine to pause the current linear execution flow, and jump to a wholly different location in the program instead. The destination of the jump is usually marked with a label (a name).

The extensive use of GOTOs was famously criticised by Dijkstra in his 1968 letter titled: “Go To Statement Considered Harmful” [16]. He argued that humans are better suited at understanding static processes, while they have a poor grasp of those processes that evolve in time.

Given these human limitations, it is better for us to try to shorten the conceptual gap between the textual representation of a program and the spread in time of the process it describes. Of course, the GOTO statement complicates this much more than structured programming’s solutions: loops and conditionals.

In a for-loop, for example, the iteration variable clearly acts like a “coordinate” in helping the programmer to orient him/herself through the iterative process; but with an unconstrained use of GOTOs it is hard to find a suitable set of such “coordinate” variables.

Moreover, Dijkstra cites the theoretical findings of Corrado Böhm (1923-2017) and Giuseppe Jacopini (1936-2001), namely: the Böhm-Jacopini theorem [17], which proves that the three basic constructs of structured programming: sequence, selection and iteration are enough to write any program that could be written in the "traditional way" using labels and GOTOs.

Nowadays, unstructured programming is (mostly) a defunct paradigm in all but the most low-level languages (assembly); some of its vestiges still survive in the "break" and "continue" instructions still in use in some high-level languages, to exit early out of a loop or skip an iteration, respectively.

2.7 Lisp and Functional Programming

Lisp (List Processor) was born with very different goals in mind than languages preceding it. It was designed by the American computer scientist and Artificial Intelligence researcher John McCarthy (1927-2011) in 1958. Incidentally, McCarthy coined the term "Artificial Intelligence" (AI) back in those years. He wanted to design a language that made it easy to perform symbolic computations on lists. A list data-structure in computer science is a sequence of elements, which is not contiguous in memory; each element stores a value and a pointer to the location of the successor.

2.7.1 Advice Taker

McCarthy's aim at the time was to describe "Advice Taker", a hypothetical program he first introduced in his seminal paper from 1959: "Programs with Common Sense" [18]. The program would be fed with data in the form of predicate calculus Well-Formed Formulas (WFFs) or "sentences", and it would be given a concrete goal to accomplish; it would have then had to reason from the premises it had available to arrive at a solution to the problem, drawing immediate conclusions from the premises and "taking action" in case the conclusions were found to be imperative sentences.

The example problem McCarthy provides us with, in his paper, is about deducing the appropriate steps to get him (McCarthy) to the airport to catch a flight, given such premises as the fact that he (McCarthy) is currently at home, that he has a car, that the airport is in a certain location, that the car can get him to distant places, etc. This might seem like a trivial problem (or perhaps a non-problem) to any "non-feeble-minded human", to use McCarthy's own words; but the difficulty associated to reliably automating this kind of "trivial", everyday reasoning is what came to be known as the problem of Common Sense in AI.

In the process of developing a language that made it easy to describe Advice Taker, McCarthy ended up with a very elegant set of language primitives. Lisp has pioneered a large number of innovations programmers still appreciate today; it came first at introducing: recursion, conditional expressions (logical if), and dynamic (during run-time) allocation and deallocation of memory (with garbage collection) [19], [12].

2.7.2 Recursion

Recursion is a universal concept recognizable in many places, including: natural language, mathematics and computer science. In the latter two fields, the term "recursive function" basically refers to a function whose definition contains a reference to itself (to the very function being defined). Recursion breaks a problem down into manageable subproblems, and, like iteration, it provides a means to repeat the same computation over and over without duplication of code.

Unlike iteration, recursion does not require any reassignment of "counter" variables. Any iterative process (or "loop") must keep (at least) one counter which is incremented/decremented at every iterative step, similar to how a jury may count the loops the athletes run at a sports competition.

But a recursive function circumvents that, because it can call a new instance of itself with a different argument, such as an incremented (or decremented) number. At some point, a recursive invocation with the suitable value will reach the "base case", or condition of termination, which will cause it (the latest invocation) to return a plain value. Following that, the function call stack will begin "unwinding", as older and older invocations begin returning values as well; until control is handed back to the original invocation, and it finally returns the global result.

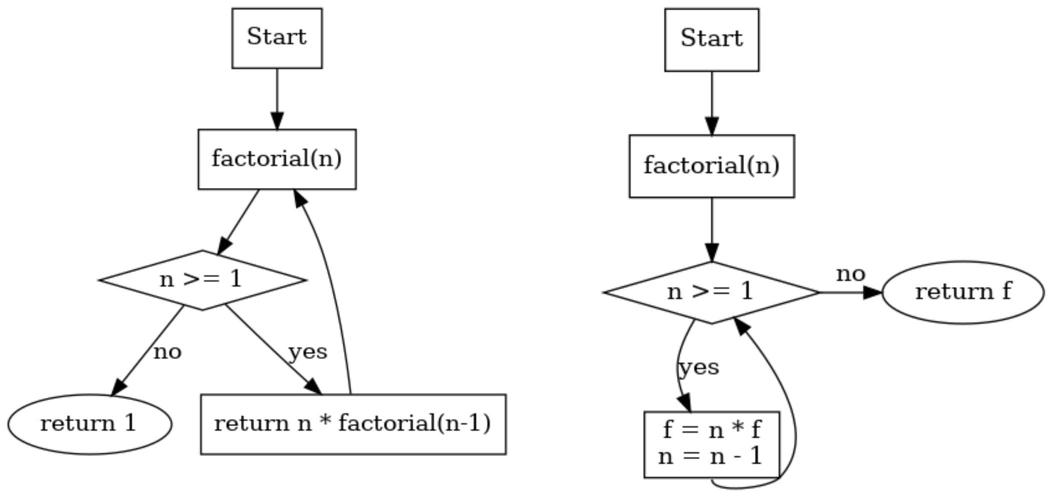


Figure 1: Calculating the factorial of n, recursive (left) vs iterative (right) approach.

The factorial of a positive integer number is the product of that number and all of the integers that precede it and are greater than one. The flowcharts in Figure 1 illustrate the difference between computing this simple function recursively vs iteratively. Notice how iteration needs to keep two variables updated: the one that stores the value of n (which gets decremented at every iteration); and the variable that accumulates the result (the factorial of the original number n). In the recursive approach there is no need for mutable variables; the factorial function repeatedly calls itself for smaller and smaller integer values, until it reaches the base case (n < 1); at that point the latest call returns a value, and the call stack begins unwinding, until eventually the oldest call is reached, and that finally returns the full value of `factorial(n)`.

2.7.3 Pure Functional Programming

The introduction of recursion in programming languages meant that (mutable) variables were no longer an unavoidable logical necessity, and it ushered in a new paradigm known as Functional Programming (FP). Functional Programming is a declarative paradigm. It is declarative because it abstracts away control flow, and severely limits the scope of mutable state; two pervasive aspects of the program that have to be managed manually in imperative programming.

FP is centered around the concept of a "pure" or mathematical function [20]. A "pure" function is a function that has no side-effects; this means that it does not mutate any state, and it does not depend on any mutable state; a pure function's output only depends on its input and its logic. A function that returns the current date and time is not pure, because its output depends on a mutable external factor (the state of the computer's clock).

A function that adds an element to a list in-place (modifying the original list) is also not pure; but a function that creates a new list with the added element, not modifying the original list can be called a pure function.

In FP, every computation is performed by function application, operators can be seen as "infix" functions, and a variable assignment (which is immutable) can be seen as the definition of a function without arguments: a function without arguments always returns a constant value, since the output of a function can only change when the input changes, and the "input" of a constant never changes.

Furthermore, since there are no side effects, all expressions are “referentially transparent”: any sub-expression can be replaced with an equivalent one in value, without worries that the replacement will cause the program to break due to some unexpected dependence on “context” (opaque mutations of state). The order of execution becomes irrelevant, to a large extent. This also makes writing code akin to writing mathematical formulas.

All of this culminates in the pure functional programming languages such as Haskell, developed in the 1990s. Haskell specifically is also what’s known as a “lazy evaluation” language: all expressions are lazily evaluated, without regards for the order in which they were declared in the source code. After all, if a variable is defined on line 100 and used only at line 500, who needs to evaluate it before the formulas on line 500 are? (Given that its evaluation, or lack thereof, can’t ever affect anything else in the least). This has its big advantages: for the human and for the compiler; but it may also result in unpredictable memory management, which can be seen as a drawback in some applications [20].

2.7.4 Variable Scoping

The original Lisp had something known as “dynamic scoping”, which is contrasted to static or “lexical” scoping [19].

In a typical program, it is convenient that the same variable name be re-used in many different places, with (possibly) different values and different meanings. Moreover, variable scopes can be nested. It is therefore paramount to define a precise set of rules regarding the “scope” (region of visibility) of a variable.

When resolving the variable “X”, static scoping takes into account the (lexically, graphically) closest location where a variable with that name has been defined in the code, and it uses that value to resolve “X”. Dynamic scoping, on the other hand, takes “X” to be whatever it is in the current execution environment at run-time.

Take the following example to illustrate the concept: suppose that you have a “variable declaration”, a “function definition”, some unspecified code in the middle (...), and then a function invocation:

```
X is the oven.
to bake a cake: put it inside of X.
...
bake the cake.
```

If the language above is statically scoped, then when calling the function “bake” from whichever distant position in the code, “X” will always be “the oven”. But if the above language is dynamically scoped, then it depends. If the function “bake” is called from a different context (a different scope) where the name “X” means something else (suppose it is now: “the freezer”), then the cake will be baked by placing it in the freezer.

Dynamic versus static scoping thus behave differently in the presence of implicit arguments to a function, i.e., when a function reads a variable from the outer scope. The reader can see why dynamic scoping is thought to be a little less intuitive than static scoping; in spite of this, it is still deemed useful by some programmers in certain situations [21].

Modern *dialects* of Lisp tend to support both strategies, as do some languages such as JavaScript. But static scoping is by far the most popular scoping strategy in modern languages.

2.7.5 The impact of Lisp

All in all, the effect of Lisp on the programming language landscape has been significant; although the dialects of Lisp are not (by today's standards) very popular languages, the principles from Functional Programming that Lisp helped to shape are still being studied and incorporated in formerly imperative languages.

The performance of purely functional languages has improved over the years, making them a viable alternative to their imperative counterparts in many cases. The debate over the purported "naturalness" of the imperative approach over the functional one, or vice versa, is still very much alive today.

2.8 ALGOL

The first versions of ALGOL were developed during the late 1950s to early 1960s, in an effort to create a machine-independent standard for scientific computation, and to provide a universal alternative to Fortran, which initially ran only on IBM hardware [12].

ALGOL 60 is notable for being the first language to be described using what would come to be known as the Backus Naur Form (BNF), a metalanguage for the description of Context Free Grammars (CFGs), originally introduced by John Backus and later developed with Peter Naur (1928-2016).

BNF, and its variants such as Extended BNF (EBNF), still remains the most popular means of describing the syntax of programming languages. Born as a notation for the class of grammars known as "Context

Free" introduced by linguist and philosopher Noam Chomsky (1928-), BNF describes a grammar as a set of production rules; each rule has a left and a right hand-side; on the left is the name of the grammatical element being defined, on the right is the sequence of sub-elements it comprises of. It is reported that a similar formalism had been used by Pāṇini, a Sanskrit grammarian who lived in ancient India between the 6th and the 4th century BCE, to describe the morphology of Sanskrit.

ALGOL 60 is also notable for having introduced the block structure for managing nested scopes, for being among the first imperative languages to support recursion, and for having had (alongside the more common "pass by value") an additional and peculiar "pass by name" mechanism for argument passing [22].

2.8.1 Argument Passing

Passing an argument to a function or procedure (or more generally, to a "routine") is an essential part of invoking it. As is often the case, there are different possible ways to do it. The most common are: "pass by reference" and "pass by value".

The first (pass by reference) essentially provides the invoked function with a reference (such as a pointer) to the original object, making it possible for the function to change the original object (side-effects).

The second (pass by value) makes a copy of the argument, and provides the function with the copy (not the original).

Obviously, these two mechanisms are de facto equivalent if all of the objects are immutable, such as in a pure functional language. What both of these two approaches (by reference and by value) always have in common, though, is that they require the argument expression to be fully evaluated before it is ever used inside of a function's body.

On the contrary, pass by name does not. Pass by name is akin to a textual substitution, within the function's body, of every appearance of the parameter's name with the argument's text.

The implication of this, is: that formulas can be passed in as arguments. They are not evaluated before the function's body is executed, but rather during its execution. This makes it possible to exploit a technique known as Jensen's Device, from Danish computer scientist Jørn Jensen (1925-2007) [22].

For instance, if the formula `w*x[i]` is passed in as a parameter to a function that performs a loop over `i` from `0` to `n` and sums up the values, the sum of the weighted values of an array can be computed for any arbitrary value of `w`, or, for that matter, any arbitrary expression that is passed into the same parameter. In modern languages, a lambda expression (anonymous function) may be used to a similar effect.

2.8.2 ALGOL 68 and Orthogonality

The year 1968 saw the release of ALGOL 68, which introduced some novelties such as user-defined data types. It is also known as a famous example of language were the concept of orthogonality is clearly visible, and perhaps carried a little too far [12].

Orthogonality in the design of programming languages refers to the possibility of combining the constructs of a language in a relatively unconstrained way. For example, a language which allows import statements to appear in any part of the code (such as Python) is more orthogonal in this respect than a language that only allows them to appear in one place, typically at the top of a source file. If done well, orthogonal design helps keep a language free of "exceptions to the rule", and as such more intuitive to use.

2.8.3 Impact of ALGOL

After decades of being a favored algorithmic "lingua-franca" in academia, ALGOL and the ALGOL language family are no longer popular nowadays. But they are still a worthwhile subject of study, for having pioneered some of the language design and description criteria that we take for granted today.

2.9 COBOL

The COmmon Business-Oriented Language, COBOL, is not exactly known for its good reputation among programmers or academics. The Jargon File, a half-humorous/half-serious Internet compendium of computer related slang, has an entry on the subject that describes COBOL as: "A weak, verbose, and flabby language used by code grinders to do boring mindless things on dinosaur mainframes." [23].

The Jargon File also attributes the following quote to Edsger Dijkstra: "The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense."

As the name ("Business-Oriented") implies, COBOL was originally intended for data manipulation in business applications; it was designed around the end of the 1950s by a committee sponsored by the US Department of Defense (DoD); the American mathematician, computer scientist and military official Grace Hopper (1906-1992) had an important role in its design.

On an unrelated note, Grace Hopper is credited with the first attestation of the term “bug” to indicate the malfunctioning of a computer program; back when, in 1947, she taped on a report in a log book the remains of a moth that got stuck in a relay, inside of a computer that had been malfunctioning [24].

2.9.1 COBOL was “naturalistic”

What interests us specifically about COBOL is that it was one of the earliest languages to be designed with the idea of imitating natural language. This idea was championed by Grace Hopper, who believed that mathematical notation was better suited for scientific computation, but that data manipulation was a better match for English-like statements, with the characteristic all-caps English keywords of the era. There was also a concern that managers and business people would not have easily understood mathematical notation.

During the proposal process, COBOL was also presented with keywords translated in French and German [12]: we had there the beginning of the “internationalized” programming language; something that the ALGOL 68 standard went on to do seriously later on in that decade (ALGOL 68 was also available in Russian keywords, among its other “translations”). Of course, this is an idea that did not catch on, as anyone can see today by looking at contemporary languages.

There is a sense in which COBOL is "naturalistic". In COBOL, you do not use the regular addition operator `+`, you rather ADD something TO something else. The naturalness of COBOL statements can push as far as statements like the following example taken from the paper about the PENS ranking for Controlled Natural Languages [25]:

```
PERFORM P WITH TEST BEFORE VARYING C FROM 1 BY 2 UNTIL C GREATER
THAN 10
```

Despite this, and as the same paper duly mentions, the statement structure usually does not follow English's grammar this closely; moreover, it is too rigid, and its rigidity, coupled with its deceptive "naturalness" on the surface, creates more confusion to the programmer than positive effects.

2.9.2 COBOL today

In closing, we must remember that many mission critical systems to this day depend on large COBOL codebases; this is why, despite the kind of reputation it has earned during the last 60 and more years of its existence, COBOL is still a hot topic today [26].

2.10 PL/I

PL/I (Programming Language One) is perhaps the second most criticised programming language, after COBOL, that we will consider. Developed in the 1960s at IBM, it was born as a "large" language, that no single programmer at the time could wrap their head around [27].

It included what were seen as the best features of ALGOL 60, Fortran IV and COBOL 60; together with a host of additional new features that were absent from the other three. While this should have made it a good language (in "theory"), unfortunately, as one can imagine, it ended up making PL/I far too complicated to understand.

Quoting the eloquent Dijkstra (who speaks of the language PL/I):

"I absolutely fail to see how we can keep our growing programs firmly within our intellectual grip when by its sheer baroqueness the programming language—our basic tool, mind you!—already escapes our intellectual control." [28]

We think the main lesson to take away from PL/I is that a language should not incorporate too many features that cannot be well integrated together. It is best to design with a minimal set of orthogonal features, that allow for future extension by the users.

2.11 SIMULA 67, Smalltalk and Object Orientation

2.11.1 Simula

Simula, developed during the 1960s at the Norwegian Computing Center in Oslo, is generally recognized as the first Object-Oriented Programming (OOP) language, though the term "Object-Oriented" itself was coined a little while later by computer scientist Alan Kay (1940-) [29].

Simula, as the name can suggest, was designed to simulate real world events. Syntactically, it was an extension of ALGOL 60. It was the first language to introduce objects, classes, inheritance and subclasses, virtual procedures and coroutines. A coroutine is a kind of subprogram that is allowed to stop and restart from where it left off, a feature that was found useful in simulating events [12].

2.11.2 Smalltalk

Smalltalk originated in Alan Kay's late 1960s Ph.D. dissertation. Alan Kay, partly inspired by Simula, is credited for putting OOP on a firm theoretical basis. The idea of message passing is paramount to Kay's concept of object orientation: everything is an object, and all objects communicate with each other exclusively by passing messages.

Kay was inspired by his knowledge of how cells worked in living organisms; and had envisioned, much before the personal computer really took off, a huge global network of intercommunicating computers. Analogously, the objects he described could be seen as a network of virtual computers.

2.11.3 Object Oriented Programming

Simula and Smalltalk were responsible for starting a revolution. Nowadays OOP is still the dominant paradigm in software development, with a majority of the most popular programming languages, featured on the annual Stack Overflow developer survey, supporting OOP in some form [30].

At the core of modern OOP lies the idea that data and behavior should be combined into a single entity: the object. The object hides its data (or "state") and exposes an interface to other objects, communicating with them through message passing, often understood as method calls.

The “traditional” four pillars of OOP are: data abstraction, encapsulation, inheritance, and polymorphism.

Modern OOP is also related to the concept of the Abstract Data Type (ADT), which hides the specifics of its implementation behind an interface.

2.11.4 Polymorphism

Polymorphism and Inheritance are not among the terms introduced or favored by Alan Kay. Polymorphism originally comes from the mathematical jargon about functions [29] (indeed, a similar concept of Polymorphism is also present in Functional Programming) and it refers to an interface’s ability to apply to different types; for example, the generic possibility of calculating the area of a shape applies equally well to a square, to a circle or to a trapezoid, but the concrete formulas to compute the areas of those three kinds of shapes are quite different.

2.11.5 Inheritance

Inheritance is mainly about code reuse: if a class needs the same methods (functions) and attributes (data fields) defined in another, then making the new class inherit from the old class can help reduce code duplication.

A “class” is a template used to create new objects (or “instances” of a class). OOP does not require classes or even inheritance to work; however, class-based programming is the dominant (but not the only) approach in modern OOP.

It seems as though Alan Kay considers this, and other aspects of the modern popular approach to OOP, as a significant departure from what he originally envisioned in Smalltalk. For example, in the way many subsequent languages conflated polymorphism (common interfaces) with inheritance (code reuse) through the mechanism of classes [29].

This is something that has led to problems, compounded by the possibility of creating deep and complicated inheritance hierarchies that are hard to maintain. This has generally turned inheritance into a discouraged construct, an idea which is usually manifested in the “composition over inheritance” motto [31].

2.11.6 C++ and its legacy

Among the most popular and oldest object-oriented languages is C++, developed in the 1980s by Danish computer scientist Bjarne Stroustrup (1950-), as an object oriented extension to the vastly popular C programming language [12].

It may be said that many of the OOP languages released in the following years (Java, D, Rust…) have partly been attempts at curtailing C++’s complexity while preserving some of its power; C++ is a very powerful language that, unfortunately, is also a huge hodgepodge of features accumulated over the years, and retained due to backwards compatibility concerns, starting from basically all of the C programming language, which had to be supported right from the start.

The early association with languages such as Simula and C++, which were basically procedural languages with object-oriented features tacked on top, has led to the modern impression that OOP is somewhat necessarily an imperative programming paradigm; this is not strictly true, as OOP can work even without the need for the mutable state and the explicit flow control which characterizes imperative languages; in fact, Object Oriented and Functional programming can even coexist relatively well in a language, as is demonstrated by languages like Scala [20].

2.11.7 Criticism of OOP

The problem of state management is a recurrent theme in software engineering; it may be said that while Functional Programming tries to minimize and isolate the state of a program, OOP's strategy, instead, is to simply hide it behind the curtain of an object's interface.

A popular criticism of OOP revolves around its reliance on Software Design Patterns to solve a vast set of problems. The natural way to solve a problem in computer science, it is argued, should be to solve it once and for all, and then to abstract away the solution for later re-use. Software Design Patterns, on the other hand, require rethinking through and reimplementing solutions for a very similar problem every time it comes up again [32].

Moreover, it can be argued that some of the original 23 GoF patterns for OOP languages can be entirely avoided when using languages with a different set of abstractions, for instance: higher order functions and lambdas from Functional Programming [32]. Indeed, many of these functional abstractions have made, and are making, their way into modern OOP languages and common practice.

We will again quote Dijsktra in closing, who once said about OOP that: “Object-oriented programming is an exceptionally bad idea which could only have originated in California.” [33]

Despite the fair criticism, from illustrious critics, OOP remains the dominant approach in the software industry. It has brought programming abstractions a little bit closer to the human mind, with its insistence on the familiar “object” metaphor, and on the integration of data with behavior.

2.12 Prolog and Logic Programming

Logic Programming is a declarative paradigm based on the use of predicate logic to declare facts and inference rules; its development is closely linked to the development of Prolog during the 1970s as a result of a collaboration between the University of Aix-Marseille and the Department of Artificial Intelligence at the University of Edinburgh. Prolog remains to this day the most widely known, studied and used example of Logic Programming Language [12].

The strength of logic programming lies in its being, as aforementioned, a highly declarative paradigm. Unlike imperative programming (where the programmer is required to explicitly manage control flow and deal with mutable state) or even functional programming (where the programmer still has to explicitly write out an algorithm) in pure logic programming what is required from the programmer are just the plain facts and the inference rules; the way those two are used to make computations and draw conclusions is entirely up to the system.

2.12.1 First Order Logic

Logic Programming is based on a particular form of logic known as "predicate logic" or "First Order Logic" (FOL); FOL is a formal way of expressing propositions and the relationships occurring between them. A proposition corresponds to the meaning of an English sentence, it is something that can have a truth value: true or false. In other terms, it can correspond to how the world really is (true), or contradict its actual state of affairs (false).

Just like algebra provides a basis for the algorithmic manipulation of mathematical equations, FOL provides a basis for proving conclusions from a set of premises in an automated way, a task that is also known as "theorem proving".

FOL, or "predicate" logic, includes the "predicate" as one of its essential concepts. A predicate is a property or a relation, and it can be (roughly) compared to a Boolean function; for instance, the formula `cat(luna)` may be translated in English as: "Luna is a cat"; "to be a cat" is a predicate (represented by the symbol `cat`), and `luna` is a constant term representing an individual cat.

A predicate, or "relation", may accept any number of terms; for instance, `sleep(luna, couch)` may mean that "Luna is sleeping on the couch"; it may also mean that "the couch is sleeping on Luna", depending on what conventional order we choose to adopt for the terms.

FOL also includes conjunctions (logical `and` and logical `or`), the negation operator, quantifiers (existential and universal) and implications.

The existential quantifier declares the existence of at least an individual with the specified properties; for instance, `∃X.(cat(X) and red(X))` may mean that “there is at least an X such that it is a red cat”.

The universal quantifier declares a global property of all individuals that satisfy a given description; for instance, `∀X.(mortal(X) <- man(X))`, where `<-` is an implication arrow, may express the idea that every man is mortal, i.e.: “X is a man, implies X is mortal”.

2.12.2 Horn clauses

For practical reasons, Prolog is based on a restricted dialect of predicate logic, specifically it is based on a special kind of formula known as the “Horn clause”, developed and studied by mathematician Alfred Horn (1918-2001).

A Horn clause has a left- and a right-hand sides that are separated by an implication. The implication goes from right to left. The left-hand side may only contain one single predicate or be completely empty. When the left-hand side is empty, there is no implication, and the right-hand side is treated as a fact; for instance, the formula `cat(luna)` from before, which states the simple fact that “Luna is a cat”.

When the left-hand side is *not* empty, then the Horn clause is actually what we would think of as an implication; you could also roughly think of it as a “definition” of the meaning of the left-hand side in terms of the meaning of the right-hand side; for instance: `father(X,Y) :- parent(X,Y), male(X)` means that: “X is the father of Y, when X is the parent of Y and X is male”; every variable in a Horn clause is implicitly universally quantified [12].

2.12.3 Rules and Queries

Statements in Prolog have two interpretations: they can either be declarations or queries; this usually means that there are two "modes": typically, the facts and inference rules can be written to a Prolog source file and stored in there, then the file can be loaded into an interactive interpreter which allows the user to make queries using the same syntax as declarations.

For instance, querying for `father(X, luke)` will attempt to find Luke's father using the information contained in the source file; `father(walker, luke)` will check if Walker is Luke's father, returning `yes` in case this is supported by the information available, and `no` otherwise. This process of substituting variables and trying to match formulas is known as Unification, and its implementation can be rather complex [12].

2.12.4 Closed-World Assumption

Prolog and similar systems always assume that the information they possess is complete; that anything that is true is also known to be true, or equivalently that anything that is not known to be true has to be false. This is known as the Closed-World Assumption (CWA), and it is generally contrasted with the Open-World Assumption (OWA) used by some other systems. The CWA is related to the concept of "Negation as Failure", which views negation as the failure to prove the truth of a statement [12].

2.12.5 The impact of Prolog

Logic Programming has seen periods of greater popularity, especially during the years of the Japanese Fifth Generation Computer Systems (FGCS) initiative, from the early 1980s to the early 1990s [34]. This paradigm has not become more widespread, in part because of performance related issues; Unification is relatively slow: there are still no known fast solutions in pure Logic Programming to elementary problems such as sorting a list of numbers, short of explicitly writing the code for a sorting algorithm, which means abandoning pure Logic Programming [12].

Logic Programming is nonetheless a very interesting paradigm to study, and it certainly has its merits in the field of Expert Systems, and in certain kinds of Natural Language Processing and symbolic AI applications.

2.13 Scripting Languages

Scripting Languages are not a cohesive block; their purpose ranges from quick prototyping to end-user programming (related to the customization of a program by the user) and to glue code (letting programs written in different languages interact with each other).

Some scripting languages are also used for scientific computing, data exploration and data analysis; and some are used on the backend of websites to provide dynamic behavior tailored to each individual user. Some are domain specific and have a restricted set of use cases; some are general purpose, and can be used to build pretty much anything, and fast.

Eminent examples of scripting languages are: the many variants of the Unix shell, Awk, Perl, JavaScript, PHP, Python, Ruby, Lua and many more. Despite this huge amount of diversity, Scripting Languages have always shared some common traits that are characteristic of them.

The prototypical scripting language is used in an exploratory setting, as a “quick and dirty” tool to test out an idea or to produce glue code; and, as such, a shortened and convenient syntax is deemed to be essential to a good scripting language.

Furthermore, a scripting language is usually interpreted, because it is embedded within an existing environment, in turn implemented in some “real” programming language which does the “heavy lifting”.

It seems fair to think that it was partly as a consequence of these two factors that most scripting languages were born as “dynamically typed” languages.

2.13.1 Dynamic vs Static Typing

Dynamic typing is contrasted with Static typing; in statically typed languages most or all of the variable types are known in advance and checked for correctness at compile-time, before the code ever runs. In purely dynamically typed languages, there are absolutely no compile-time type checks; the code either works at run-time or fails at run-time.

While thorough testing, which is very important also in a statically typed setting, is sometimes presented as an alternative to static type checking altogether, the last decade has seen the emergence and wide adoption of numerous supersets of popular scripting languages that were born as dynamically typed.

2.13.2 Gradual Typing

Such supersets enhance their base languages by providing what is known as “gradual typing”, or “optional static typing”; examples of such supersets and/or static type-checkers include: MyPy for Python, Typescript (and others) for JavaScript, Sorbet for Ruby, and Hack for PHP [35].

It is likely that the debate between proponents of dynamic versus static typing will go on indefinitely; despite the growing trend of gradual typing (a step in the direction of static typing), there are still some high-profile proponents of dynamic typing [36].

2.13.3 Type Inference

Another factor to take into account is that the traditional cumbersomeness of static typing (due to lengthy, explicit type declarations) can be regarded as a thing of the past; today most modern languages and type-checkers have built-in support for type inference: the type of a variable can be deduced from the value of its initial assignment, thus cutting down on what was perceived as boilerplate (redundant) code, while also preserving the benefits of static type checking [35].

2.13.4 Feature exchanges

It it not only about scripting languages becoming “statically” typed; some of the successful ideas from scripting languages are also making it to the other side. Scripting languages have traditionally had a tool known as a “Read Eval Print Loop” (REPL), also known as an “interactive shell”; this tool allows developers to test out statements and functions interactively on the fly, greatly increasing their productivity. There is a trend where classical compiled programming languages are also getting REPLs: Java now has Jshell (as well as the older BeanShell [37]), Haskell, Scala and Kotlin also all come with REPLs.

2.14 The Ideal Language

In our brief analysis of the evolution of programming languages, we left out a lot of important names, and we barely did justice to the ones we covered; it would be preposterous to try and answer the question of whether an ideal programming language exists, let alone try describing it.

We saw how these languages can vary wildly in purpose; and, as the engineering adage: "form follows function" goes, it is therefore quite improbable that a single *best*, perfect programming language fit for all problems may ever exist (though the *worst* could, and probably already does [38]).

Nonetheless, we investigated the most popular programming paradigms, their merits and their flaws, and we did come across some generally accepted principles of design (like orthogonality), and some broadly applicable ideas; we also did see that certain languages are criticized for their design decisions somewhat more than others.

In other words, while it is certainly true that all programming languages serve the purpose they were designed for the best, not all programming languages were created equal, and good design is still an important factor to keep in mind; this, we think, is also the opinion of computer scientist Paul Graham (1964-) [32].

As Guy Steele (1954-), computer scientist and language designer, explains (and shows) in his talk “Growing a Language” [27], planning for (organic, user directed) growth is the best strategy a modern language designer can take. Starting with a large set of primitives (like PL/I) is as bad as starting without the proper facilities to let the language users themselves define new constructs, that integrate well with the language’s native ones.

3 Naturalistic Programming

We think that the upshot of the previous chapter about programming languages is that these kinds of formal languages are “precise” and “simple”.

They are precise because they (like mathematical notation) have a clearly defined syntax and semantics; and they are “simple” because their core should contain as few elements as possible, and these elements should be orthogonal (as free to combine with one another as feasible). Precision makes the meaning of code in a programming language as directly dependent on syntax as possible, and simplicity makes a programming language “easy” to use with a computer (as compared to a natural language).

Given these advantages of programming languages, what would the benefit of writing code using natural language be, if any? This is what we shall discuss in the following pages, together with four examples of currently existing naturalistic programming languages. We shall also make a brief digression about LLMs, Prompt Engineering and AI-assisted coding (in natural language); we shall propose some criticism about using natural languages in code, and we shall conclude the chapter with what (we think) are the distinguishing characteristics of the naturalistic languages we studied, that give them an edge over traditional programming languages.

3.1 Precision and Expressiveness: a tradeoff

A general trend in languages is that there tends to be a tradeoff between naturalness and expressiveness on the one side, versus precision and ease of implementation on the other.

Within the PENS (Precision, Expressiveness, Naturalness and Simplicity) language classification scheme [25], a diverse set of Controlled Natural Languages (CNLs), all based on English, were classified by four metrics: Precision, Expressiveness, Naturalness and Simplicity.

This is roughly what these four metrics mean, respectively: the level of independence from context (Precision), the amount of expressible propositions (Expressiveness), the similarity to natural language (Naturalness) and the ease of implementation as a computer program (Simplicity).

The study found, among the other results, that Precision and Simplicity are positively correlated, Expressiveness and Simplicity are negatively correlated, and Naturalness and Expressiveness are positively correlated.

In other words: in the previous chapters we had been talking about how imprecise natural language is, but now we must acknowledge the other side of the medal, namely that natural language has an exceptional level of expressiveness, as compared to formal languages. You can say almost anything in natural language, or better yet: it is hard to think of an idea that you cannot express using plain English or any other human tongue.

3.2 Shortcomings of traditional programming languages

Does the lack of expressiveness represent a problem in traditional programming languages? We think this is evident, beginning from the very existence of research in naturalistic programming, which seems to stem from the limitations of contemporary programming languages at adequately describing certain classes of problems [39], [40], [41], [42].

In these works, it is argued that the abstractions that power the current generation of programming languages often tend to result in verbose, repetitive and brittle code.

3.2.1 The Issue of Semantic Gap

The important issue of "*Semantic Gap*" is brought up in a 2011 paper which proposes and discusses the possibility of designing a language with "naturalistic types" [40]. There is a *Semantic Gap* between the desired behavior of a program, and the code that is written to produce it; this results in the "scattering of ideas" we sometimes see in traditional programming languages, when the need arises to express a concept that is not directly supported by the programming language in question.

A concrete example of this "scattering of ideas" is reflected by the use of *flags* (Boolean variables) that are conditionally altered in a loop, and then read elsewhere in the code; the logic behind any particular use of such flags may be trivial when explained in natural language, while being entirely opaque at a first reading of the code.

This means that the logic that these flags express may have to be painstakingly pieced back together by the people that read the code, potentially many times over. Code comments, which consist essentially of short natural language explanations of the code behavior, may help add clarity to such obscure pieces of code; but one must be careful to avoid the usage of comments in excess, as they (the comments) may become obsolete and they are not checked by the compiler.

3.2.2 Aspect Oriented Programming (AOP)

The 2003 paper “Beyond AOP: toward naturalistic programming” [41] argues that language designers should start taking inspiration from natural language, just as they took inspiration from mathematics (Functional Programming), and what they call: “ad-hoc metaphors such as Objects” (OOP) in the past. The paper talks about the emergence of Aspect Oriented Programming (AOP), as a positive example of a step in the right direction.

AOP is a programming paradigm that aims at increasing the modularity of code by addressing the problem of cross-cutting concerns, and does so by advocating their separation from the core business-logic. A cross-cutting concern, in software development, is an aspect of a program that affects various modules at the same time, without the possibility (in non-AOP languages) of being encapsulated in any single one of them.

A most obvious cross-cutting concern is logging. For instance: the problem of logging all of the calls to functions with specific characteristics, and only those. In a traditional OOP language, one would have to insert a call to the logger at the beginning or at the end of every function that one wished to track.

In a language that supports AOP, this concern can be handled by an Aspect: a separate section of the program that neatly encapsulates that cross-cutting concern (logging all such-and-such functions); similar to how we could write a “chapter about logging” if we were describing the behavior of the same application using the referential capabilities afforded to us by the natural language in a printed book; that is the way people have always been describing (even very complex) ideas and processes for centuries.

3.3 Natural Style

But then, how do people normally describe problems and their solutions in natural language? The manner, style, and train of thought in describing an algorithmic procedure employed by a regular person, or even by a programmer when elaborating an idea in the abstract, can be very different from the style employed by the same programmer when translating his/her ideas into executable code [39], [43].

An early and oft-cited study in this respect is the one conducted by L. A. Miller in 1981 [43]. A group of college students who were not familiar with computers were asked to provide solutions to six file manipulation problems, and their proposed solutions (all written in natural language) were evaluated through metrics such as preference of expression, presence of contextual referencing, etc.

3.3.1 Implicit References

It was found that the students were quite often prone to using contextual references such as pronouns and context-dependent phrases such as “the previous” and “the next”; this is a linguistic phenomenon broadly known as Deixis. Explicit variable assignments were *not* used.

3.3.2 Universal Quantification

The students preferred to treat data structures in a cumulative way, using universal quantifiers, rather than loops, to express operations that had to be carried out on multiple instances of a data structure. They avoided using the traditional structured programming constructs (if-then-else, while, for, etc.), let alone the even more unnatural unstructured programming constructs (‘goto’ statements).

3.3.3 Blunt (but Revisable) General Statements

One of those historic (2006-2009) “Get a Mac” commercials [44] by Apple (the company) comes to mind when talking about this aspect of natural language. The episode in question begins with the character interpreting a “PC” making a very blunt statement about how easy it is to use a PC, and then some small, hard to read text suddenly appears down at the bottom of the screen. We are told that this is just some “legal copy” (a kind of legal disclaimer), because apparently the claim about how “easy” it is to use a PC requires a “little more” explanation. PC goes on to make more and more of such “bold” claims, and the legal copy promptly grows, until it floods the entirety of the screen.

In any case, when describing an algorithm, the students from the experiment by Miller tended to begin by the most general and crucial step of the procedure, to only then deal with those special cases which required a different sort of treatment. This strategy of dealing with complexity: the further refinement, or "annotation" of a blunt initial general statement, seems to be a pervasive idea; and we will encounter it again later.

It is often not how we do things in traditional programming language code, where the crucial step of a function may be delayed until after all of the guard clauses and early returns that check for the edge cases are visited; or, worse, the crucial step may be buried deep within a hierarchy of nested if-statements.

Perhaps, the success of the exception handling model (which itself is not perfect, by any means) is owed in part to the philosophy of tackling the most important step first, and handling the edge cases (as exceptions) later.

3.3.4 Implied Knowledge

The subjects of the experiment also expected the computer to possess some pragmatic, contextual knowledge of the world and of their intentions, expecting it to fill in the semantic gaps whenever needed. As Dijkstra would have put it: "They blamed the mechanical slave for its strict obedience with which it carried out its given instructions, even if a moment's thought would have revealed that those instructions contained an obvious mistake" [1].

3.3.5 Size of Vocabulary

Another finding was that the subjects tended to use a relatively restricted vocabulary, though they still liked to use synonyms from time to time. Besides, studies have shown that to understand novels and newspaper articles a non-native English speaker needs to know just about 8000 to 9000 of the most common lemmas (root-words, or "word families"), and to understand dialogue on TV shows or movies that number is even less: only about 3000 lemmas [45]. The problem of vocabulary does not appear great, as we must remember that a human speaker can often make up for the words he/she does not know from contextual information and implied world knowledge, which is definitely a bigger problem for a machine.

3.4 Literate Programming

Donald Knuth (1938-), a computer scientist known for his foundational work in time complexity theory and for creating the TEX typesetting and markup system, coined the term "Literate Programming" in 1984 to describe his novel approach to writing programs.

Knuth designed a language called "WEB"; apparently back when he chose this name for it, the word "web" was still "one of the few three-letter words of English that hadn't already been applied to computers" [46].

The WEB language combines together a markup language and a traditional general purpose programming language (TEX and PASCAL, respectively, in Knuth's original work); the idea is that a program can be seen as a web of components, and that it is best to describe the links between these components using a mixture of natural language descriptions (with TEX) and formal notation (with PASCAL).

According to this philosophy, the program should make sense to a human being first and foremost, so it is mostly composed of natural language sentences and phrases, interspersed with (relatively little) definitions in formal language.

3.5 Stories and Code

What do computer programming and story telling have in common? A lot, according to a 2005 study by Hugo Liu and Henry Lieberman. The study involved a system which automatically translated user stories to Python code fragments [47], [48].

The program, which can be seen as a very peculiar kind of transpiler, called Metafor, was integrated with a large Common Sense Knowledge base called Concept-Net, derived from the Open Mind corpus of natural language Common Sense statements.

The code generated was “scaffolding”, or underspecified code, that in fact was not meant to be executed right out of the box. Related to this, is an interesting concept discussed in the work; that ambiguity is not always a negative aspect of natural language, but, on the contrary, it is a means of avoiding difficult design decisions at too early of a stage in a project; and indeed Metafor was designed to automatically refactor the output Python code whenever it received an indication that the underlying representation was no longer adequate to the story being told and had to change; for instance, by automatically promoting attributes of a Python class to sub-classes of their own [47].

The paper also touched upon the concept of programmatic semantics, expanded upon in another work [48] by the same two authors; which is the idea that natural language structures imply and can be mapped to the more traditional programming constructs, the authors claim that "a surprising amount" of what they call programmatic semantics can be inferred from the linguistic structure of a story [47]. The authors propose, by making a simple example, that noun phrases may generally correspond to data structures, verbs to functions, and so on.

As we already saw, the code produced by the system was never really meant to be complete or executable, but its main purpose was to facilitate the task of outlining a project, especially for novice users. And it showed promising results when it was tested by a group of 13 students, some of which with novice and some of which with intermediate programming skills. The students responded to the question of whether they would be likely to use it as a brainstorming tool, in lieu of more traditional pen-and-paper methods [47].

3.6 Existing Naturalistic Languages

There have been some pretty complete attempts at creating comprehensive natural language programming systems. We will proceed to mention four of what (we think) may be the most important ones (in no particular order): Pegasus, CAL, SN and Inform 7.

3.6.1 Pegasus

The original implementation of Pegasus is outlined in a 2006 paper [39] by Roman Knöll and Mira Mezini, the former of which I had the pleasure of contacting via e-mail, and who has confirmed that the project and related work are still under active development, although the official webpage has not been updated since 2018, on the date of writing [49].

3.6.1.1 Implicit Referencing, Compression and Context Dependence

According to the authors, the main features that distinguish natural language from programming languages are: implicit referencing, compression and context dependence.

Implicit referencing refers to the usage of Deixis in speech, pronouns such as “it”, “he”, “she”, words like “the former”, “the latter” and demonstratives like “this”, “that” - all words clearly exemplifying this phenomenon.

Compression is a mechanism that avoids the tedious repetition of information and it can be of two kinds: syntactic or semantic. The former refers to the use of words such as “and” in a sentence like: “he pets the cat and the cheetah” understood as an abbreviation for “he pets the cat and he pets the cheetah”. An example of semantic compression would be the sentence “first go from left to right, then vice versa”, where “vice versa”, in this case, would mean: “from right to left”.

Context Dependence refers to the fact that, contra most programming languages, the same string in natural language can be reduced in different ways depending on the surrounding context, not only on the string itself. In the sentence: “the mouse runs, then it jumps, the cat kills it, then it jumps” there are two identical instances of the phrase “then it jumps”. In the former the pronoun “it” refers to the mouse, in the latter the pronoun “it” refers to the cat.

These three mechanisms, in the authors’ opinion, all help in reducing the amount of redundancy in our spoken and written communication.

3.6.1.2 Idea Notation

The authors discuss of a possible formalization of human thought; it may or may not be possible for a computer to “experience” the same thoughts and feelings as a human being, but, according to the authors, it should be possible to describe the structure of human thought formally enough for it to be imitated mechanically.

They propose a distinction between what they call “Atomic” versus “Complex” ideas, the former stemming directly from human perception (such as “the smell of wood”, or “the warmth of wood”), and the latter being a combination of the former (such as the very idea of “wood”).

They also recognize a third category of ideas they call “Composed”, which “combine several Complex ideas for a short moment” [39], effectively corresponding with propositions, such as the meaning of the sentence “the car drives on the street”.

Every idea, they assert, can have a concrete representation as: an entity, an action or a property, corresponding to a noun, a verb or an adjective/adverb respectively, in most natural languages.

The authors then describe a formalism they call “the idea notation” to express concepts and thoughts in this framework, and to perform automatic computations on them.

3.6.1.3 Architecture

The architecture of Pegasus is described as being composed of three parts: the Mind, the Short Term Memory and the Long Term Memory.

The Mind is what matches ideas to idea-patterns stored in Pegasus’s long term semantic network. An idea-pattern is associated to an action, which is performed when the Mind determines that it matches an idea received as an input.

The Short Term Memory is what enables Pegasus to contextually resolve pronouns such as “it”, and other deictic words and phrases. It is implemented as a bounded queue, and purposefully limited to 8 memory cells, corresponding to eight recently mentioned entities, as the authors believe this is optimal number for operation by human beings (cf: [50]).

Lastly, the Long Term Memory stores Pegasus’s semantic knowledge, for example is-a relationships between concepts.

When an idea and its sub-ideas are fully resolved, such a system can take action directly (as an interpreter), or generate the equivalent code in a given programming language (as a compiler) - the paper mentions Java as an example.

3.6.1.4 Translatability

The paper brings up the idea of a “translatable programming language”. We have already seen how there is a precedent for this in AGOL 68, and this paper also mentions AppleScript as a newer language that adopted this idea, at least for a period of time in the past; this really meant that AppleScript’s keywords had translations in multiple natural languages. In any case, AppleScript took the more popular (and less naturalistic) approach of “masking” the rather traditional structured programming constructs with a thin natural language mask.

Pegasus is designed to be language-independent at its core, which means that many different front-ends, corresponding to different concrete grammars, in turn corresponding to different human languages, can be implemented for it. For instance, the paper mentions Pegasus’s capability of reading both English and German, and of freely translating between a language and the other.

3.6.1.5 Drawbacks

Some of the drawbacks of the approach taken by Pegasus (and of naturalistic programming in general) are discussed in the final part of the paper. The general problems are said to be: varying choice of expression, vagueness inherent in natural language specifications (cf. [51]), the convenience of mathematical notation over natural language descriptions (cf. [1]) and the limitation in expressivity imposed on the naturalistic language by the underlying programming language when transpiling to it.

Drawbacks related to Pegasus, or at least to its original version, include performance issues related to some of the implementation choices, the problem of having to manage an extensive database due to the choice to support a language's full natural inflection and conjugation patterns, and the limited expressivity of the initial implementation of the Pegasus language itself.

All in all, Pegasus remains a valid example of a general purpose naturalistic programming system; the product is, to our knowledge as of writing, not yet available to the public, but one can see examples of its usage on the project's official website [49].

3.6.2 CAL

Another example of general purpose naturalistic programming language, also originally created in 2006 with further developments up until recently, is presented by the interestingly called "Osmosian Order of Plain English Programmers" [52].

A motivation behind this project, as explained by the authors (Gerry and Dan Rzeppa, father and son), is to eliminate the intermediate translation step from natural language pseudo-code to rigorous programming language notation (cf. [39]).

Another motivating factor was to answer the question of whether natural language could be parsed in a sufficiently "sloppy" (partial) manner (as the authors suspect human beings do, or at least infants when growing up) as to allow for flexibility in choice of expression and for a stable programming environment.

And finally, to determine whether low-level programs could conveniently be written in such a subset of the English language.

The authors seem to have come to the conclusion that all of this is indeed possible, using their system.

The authors draw a parallel between the “pictures” (we assume they are talking about mental images in human beings) and the “types” (programming language types), and between the skills that a young (or old) human being may acquire and the traditional routines of a programming language.

Most of the code in most of the programs, they claim, represents simple enough logic that is most convenient to express in natural language. However, high-level (natural language) and low-level (programming language) code can and should coexist in certain scenarios; the authors use the metaphor of a math text-book to support this idea: mathematical formulas in formal notation, when convenient, interspersed in a text mostly made up of natural language; an idea akin to the philosophy behind Literate Programming we discussed earlier Section 3.4.

What is striking about this language is that, albeit very English-like in syntax, there is a markedly procedural taste to it, complete with variables, loops and routines. There are something like three kinds of routines: procedures, deciders and functions. The procedures, just like classical procedures, are routines with side effects that “simply do something” without returning a value. Deciders and functions, on the other hand, can resolve to a value; the former being used to define when a condition is true (also allowing the system to infer when it is false), and the latter being used to derive a value from a passed parameter and also usable with possessives (such as “the triangle’s area”) in a fashion reminiscent of getter methods or derived properties in OOP.

It is possible to define custom data types, using natural language syntax to define the fields and the respective types thereof. Among the custom types that can be defined are “records” and “things” (a kind of dynamic structure). Units of measurement and the conversion between them are also supported.

The language also supports event driven programming, and has various I/O capabilities such as timers, audio output and even a 2D graphics system which can be used to draw and plot shapes.

The CAL compiler is freely downloadable (and can re-compile itself in about 3 seconds), together with the instructions manual available as a PDF file, all on the Osmosian Order’s website; however, it is only available for Microsoft Windows systems at the time of writing. The 100-page manual gives a comprehensive overview of the language with plenty of examples [52].

3.6.3 SN

A slightly newer example of a full fledged naturalistic programming language is given by the language SN (which stands for “Sicut Naturali”, or “Just as in nature” in Latin [53]) discussed in a 2019 paper by Oscar Pulido-Prieto and Ulises Juárez-Martínez [54].

The authors cite a distinction made by others between what is called the “formalist” versus the “naturalist” approach to programming languages based on natural language. The formalist approach focuses on correct execution, thus favoring an unambiguous grammars, while the naturalist approach tolerates ambiguous grammars and attempts to resolve the remaining ambiguities using techniques from artificial intelligence [54].

The authors state that their approach is closer to the formalist camp. The philosophy and style of the language is novel, and differs significantly from a typical object oriented language, and much of the syntax does try to imitate English, but it is perhaps a little less close to that of English than the two previously surveyed projects (Section 3.6.1, Section 3.6.2).

The authors discuss what they believe are the basic elements that should allow a naturalistic system to function as a general purpose programming language, and come to the conclusion that the required building blocks for such a system are: nouns, adjectives, verbs, circumstances, phrases, anaphors, explicit and static types and formalized syntax and rules (in accordance with the formalist approach).

A SN Noun roughly corresponds to a class in OOP as it can inherit from another noun (only single inheritance is allowed) and can posses attributes. An Adjective, on the other hand, supports multiple inheritance, and can be applied to a Noun to specialize it. A verb is defined on either a Noun or an adjective, similar to how a method can be defined on a class or on a trait (such as in the language Scala).

Circumstances are a special construct that can apply to either attributes or verbs, they serve to specify the applicability (or inapplicability) of adjectives to Nouns, or the conditions and time of execution of a verb. For instance, one could specify that an Adjective mutually excludes another, or that a verb should be executed before or after another, for example to log the creation of all instances of a certain kind of Noun.

Noun phrases, which can be a combination of Nouns, Adjectives, and "with/as" clauses, and can support plurals, have a dual usage in SN: they can either be used as constructors to create new instances of a certain kind (with the "a/an" keyword) or to refer to already existing instances of such a kind (with the "the" keyword).

The language also introduces the concept of "Naturalistic Iterators" and "Naturalistic Conditionals", which is accomplished with the use of reflection to refer to the instructions of the program themselves, for example by making statements such as: `repeat the next 2 instructions until i > 10`.

The compiler can produce Java Bytecode or even transpile snippets of code in the language to Scala.

3.6.4 Inform 7

Inform 7 is perhaps the most advanced naturalistic programming language we know of, and certainly the only one that has been widely (by naturalistic language standards) used among the ones we surveyed.

3.6.4.1 Inform 6

Designed by the British mathematician and poet Graham Nelson (1968-), as a successor to the more traditional Inform 6 programming language (also created by him), Inform 7 is a domain specific language for the creation of Interactive Fiction (IF).

3.6.4.2 Interactive Fiction

IF can be thought of as a form of (interactive) literature, where a reader can interact (chiefly through text) with the characters and environments in the narrative, which can include graphics and puzzles; IF can also be thought of as a kind of video-game, or "text-adventure". This typically involves the simulation of environments (called "rooms"), objects and characters and the interaction of these together.

All Interactive Fiction products before Inform 7 were typically authored in classical (procedural or OOP) programming languages; and indeed, Inform 6, the predecessor of Inform 7, was one such language: a C-style procedural language with Object-Orientation, and some extra features (like "hooks") geared towards IF creation, specifically.

3.6.4.3 Unusual Features

Inform 7 almost reads like English, and boasts quite an unusual feature-set compared to mainstream programming languages. Unlike its predecessor, it is not Object-Oriented but rather "Rule-Oriented" (we will see later what this means); it can also infer the type of a "variable" from its usage, but that does not refer to type inference in classical variable assignments or return types (as in traditional languages), but rather to what could be seen as a limited form of context sensitivity.

Inform 7 can infer the category (or "kind") of a referent of a noun phrase; for instance, if a rule is declared that states that only people can "wear" things, and a variable (say "John") is declared to be "wearing a hat", then John will be created as a person.

It is a language with a past tense. This feature is useful at capturing “past game states”, something that in other languages is achieved through flags and counter variables; the need for the latter is minimized in Inform 7.

3.6.4.4 Open-Sourced

The language was recently open-sourced [55] (in 2022), and the source-code is available on GitHub [56], as of the date of writing. The paper also argues that “natural language is the *natural* language” for the creation of IF content.

3.6.4.5 Design Rules of Inform 7

In the 2005 (revised in 2006) paper titled “Natural Language, Semantic Analysis and Interactive Fiction” [57], Nelson explains what strategies were used in the implementation of Inform 7, and what difficulties were encountered (mainly in the broad subject of Semantics).

The paper states that the four rules that were followed in the design and implementation of the language were that: (1) A casual reader should be able to (correctly) guess the meaning of Inform 7 source code; (2) The implementation has to be (computationally) economical, but not at the price of intelligibility; (3) if in doubt as to syntax, the language should imitate books or newspapers; (4) contextual knowledge is (mainly) supplied by the author of a program, not built into the language.

As an example of rule number (3) in action, one of the data structures in Inform 7 is the table (there are no arrays); and the table looks like a table in print.

The rationale behind rule number (4) is that the language can be more flexible, and bend better to the needs of the programmer, without the presence of a built-in database of semantic knowledge. However, is not an absolute rule: there are indeed some (few) semantical concepts that are built into the language, such as the spatial concept of “containment” which is deemed important by programmers of IF.

3.6.4.6 Rule-Orientation

Inform 7, as already stated, emphasizes Rules over Objects. Nelson observes that Interactive Fiction is a domain where “unintended consequences” and “unplanned relationships” abound; there are no clearcut “server-client” relationships between the objects: there are no “master” classes and “slave” classes which exist solely for the purpose of the former, and hence it is simply not feasible to manually define rules of interaction between every two pair of classes (or even interfaces) in the program; he gives the example of a “tortoise” and an “arrow” as two kinds of very different things that are probably not going to be thought of as interacting together when writing the game, but may end up doing so anyway in game-play, and it is simply not feasible to go around defining “tortoise-arrow protocols”.

The strong distinction between “specific” and “general” rules that exists in OOP is seen as inadequate for this kind of application. Class and method definition are what he calls “general” rules, and object creation with specific attributes and values for the sake of the concrete program or game are what he refers to as “specific”.

An example of how this is an issue is given by the (code organization) problem of the apple in the box and the magical ring. Suppose there is a general rule in the game about boxes being impermeable (the player cannot stick a hand through to grab the apple stored inside, just like he/she cannot walk through walls); but it is further stipulated that a player wearing a magical ring should be able to.

The solution to this problem in Inform 6 (which is an OOP language) was either to add a general rule, which was deemed a little over-the-top for such an ad-hoc circumstance; or to attach this behavior to a specific action: the taking of the apple (therefore: inside the apple fruit’s code), which was also deemed inappropriate, because, paraphrasing the author’s elegant explanation: some might see this peculiarity (the immaterial grasping hand) as a behavior of the magic ring, and some might see it as a behavior of the box, but certainly none will naturally come to think of it as a behavior of the apple.

The solution in Inform 7 involves the introduction of a new kind of rule, which specifies the circumstances under which other rules have to be ignored. In general, specific rules take precedence over more general ones, and the order of declaration of the rules in the source code is irrelevant, because rule-precedence is handled automatically.

This system of so called “gradation of rules” needs: (1) a working system of types that can recognize and match subtypes to general types; and (2) a mechanism to declare “circumstances” in which different rules apply (or do not).

3.6.4.7 Semantics

Pronouns are considered as a problem in semantics. The issue of “Donkey anaphora”, an old issue in logic, is brought up. In the sentence “If the farmer owns a donkey, he beats it”, how are we to resolve the pronoun “it”? As: a single individual donkey, or rather as any donkey that is owned by the farmer?

The paper’s discussion on semantics briefly touches on the broader issue of Compositionality. The problem is a well-known one in philosophy and linguistics; Compositionality (expressed simple terms) is the idea that “the meaning of a complex expression is determined by its structure and the meanings of its constituents”. This certainly seems to be a property of artificial languages (such as mathematical notation, or most computer languages), but it is disputed whether it is also a property of unconstrained natural language [58].

3.6.4.8 Types

Types are also discussed as an important problem. As aforementioned, *some* types are needed for the benefit of the “gradation of rules”; but the devil is in the details.

The choice in Inform 7 was made to restrict the number of basic types to little more than a dozen, all other types are defined from these basic ones. There is no multiple inheritance; apparently an “object” can *directly* “inherit” only from one type, but the type may be a composition of more types, as we understood it.

A criterion for choosing what other types there should be is given by their utility; i.e., how is a concept useful to the purpose of describing a general law? This in turn depends on context: "a rock" may be a perfectly good concept in everyday life, but it may be a "little" imprecise in the context of mineralogy.

3.6.4.8.1 Destroyed Houses (are no longer houses)

Can an object stop being part of a type at run-time? The answer given by Inform 7 to this question is a "no"; but, interestingly, the justification for this choice is not given purely in terms of computational convenience.

The idea is that sometimes a dramatic effect can change (more like "d-isfigure") an object so much so that it cannot be regarded as the original object anymore: such as a "destroyed house", which is no longer a "house"; the way Inform 7 may handle this is to trash the old "destroyed" object (e.g., the "house") and replace it with a new object (e.g., the "rubbles").

In a way, this reminds us of the classical philosophical distinction between "essential" and "accidental" properties of things [59].

3.6.4.8.2 Flightless Birds (are still birds)

> "What is your definition of justice?" "Justice, Elijah, is that which exists when all the laws are enforced." Fastolfe nodded. "A good definition, Mr. Baley, for a robot. The desire to see all laws enforced has been built into R. Daneel, now. Justice is a very concrete term to him since it is based on law enforcement, which is in turn based upon the existence of specific and definite laws. There is nothing abstract about it. A human being can recognize the fact that, on the basis of an abstract moral code, some laws may be bad ones and their enforcement unjust. What do you say, R. Daneel?" "An unjust law," said R. Daneel evenly, "is a contradiction in terms." "To a robot it is, Mr. Baley. So you see, you mustn't confuse your justice and R. Daneel's."
>
> — "The Caves of Steel, Isaac Asimov"

A diametrically opposed issue, that still has to do with types, is the nature of their association to behavior. Are we to regard a "bird who can't fly" as a "bird" nonetheless? Or is "a flightless bird" a "contraddiction in terms", just like an "unjust law" appears to be for R. Daneel, the positronic robot from Asimov's book "The Caves of Steel"? [60]

Following the recommendation of the Liskov principle [61] for Object-Oriented design, defining a penguin as a "flightless bird" would indeed be considered bad practice if the superclass "bird" included a method "fly"; the Liskov principle roughly states that: "any property that holds for a superclass should still hold for a subclass". This behavioral principle ensures, among other things, that interfaces from a superclass are not broken in the process of deriving subclasses.

So, if the "Penguin" class cannot fly, neither can the "Bird" superclass. The "Sparrow" class might. But is it "natural" to say that "birds (generally) do not fly"? As it is argued in other places [2]: no, this is not the natural way humans organize concepts. A more naturalistic approach, would be to (somehow) declare a set of defeasible (revisable) defaults for the concepts at hand; in other words, any bird can fly, until proven that it cannot.

3.7 Prompt Engineering

We cannot have a meaningful discussion on “natural language programming” in the year 2023 without making a brief digression about Large Language Models (LLMs), and the nascent discipline of “Prompt Engineering”.

3.7.1 Large Language Models

An LLM is a general-purpose language model, typically based on “transformers”: a kind of neural network architecture. It is trained on huge amounts of data; to give an idea of the size, it is reported that GPT-3 (Generative Pre-trained Transformer) was trained on 45 terabytes of text data collected from the Internet [62].

Such models are “general-purpose”, in the sense that they have language “understanding” and generation capabilities that can be useful in a very wide range of applications. Such models can be fed a text input (prompted), and they are typically described at any step as predicting “the most likely word” that comes after the “sequence of words” in the prompt.

3.7.2 What is in a Prompt?

Prompt Engineering aims to develop and optimize prompts to LLMs, while trying to understand their capabilities and limitations.

A prompt can be an instruction or a question, and may include other details such as context, input data or examples.

The "Prompt Engineering Guide" website [63] offers a wide-ranging overview of the concepts and techniques from this emergent discipline, with plenty of links to (very recent) studies and material for further reading.

3.7.3 Applications of Prompt Engineering

The applications of LLMs mentioned earlier include, but are not limited to: text summarization, information extraction, text classification (e.g., sentiment analysis), conversation (or "role prompting", i.e., configuring custom chatbots), code generation (which is of special concern to us) and reasoning.

All this can sometimes be achieved through a very simple prompt, and sometimes requires a little more thinking and cleverness to get the model to behave as desired. The Prompt Engineering Guide recommends starting simple: plenty of tasks can be achieved through Single- or Few-Shot prompting.

3.7.4 Prompting Techniques

Single-Shot prompting is when the model is asked for a response without even being given any prior example of the task to be accomplished.

Few-Shot prompting is when some examples are provided. These are usually packaged in some more-or-less predefined format, such as the popular Question and Aswer (Q&A) format. There is a way in which this can be related to the older idea of “Programming by Example” [64].

“Harder” problems such as reasoning through a mathematical problem sometimes require more than a few examples to get the answer right; this is why more advanced techniques such as Chain-of-Thought (CoT) reasoning are being experimented.

CoT involves providing the model with a breakdown of the solution to the example problem in terms of easy to perform tasks in a step-by-step approach. For instance, in the problem of summing up the odd numbers taken from a list of numbers, the model may benefit from being told to: “find the odd numbers first, then sum them up”. Another trick to get the model to reason stepwise, is to add at the end of a prompt a statement like: “let’s think step by step”.

3.7.5 Recommendations on Style

Other two recommendations are: to be specific about what you want, to be precise; and to avoid telling the model what it should not do, but rather tell it what it should do (even in the sense of merely paraphrasing the sentence). For instance, telling the model that: “the chatbot should refrain from asking for personal preferences" is considered better than the direct command: "do not ask for personal preferences".

3.7.6 LLM Settings

The behavior of an LLM can also be influenced at a lower level by tweaking its settings or "parameters"; and there are a few of these: temperature, maximum length, stop sequences, frequence and presence penalty.

Temperature is related to how "deterministic" the output should be: a lower temperature causes the LLM's output to be "more deterministic", i.e., the words with the highest likeliness are chosen more often (making it is slightly better for tasks that require factual answers); while a higher temperature means that less likely (hence less "obvious") words are chosen to complete the output, resulting in better performance for tasks that require a higher level of "creativity".

A stop sequence is a special sequence of tokens that tells the model to stop producing text when it is generated. It can be an alternative to the maximum length, which only specifies the upper bound for the length of a generated response, helping one to avoid excessive API-usage costs or long/irrelevant responses.

Frequency Penalty discourages the model from repeating words that it has already used, words that have appeared more often will be less likely to appear again. The Presence Penalty places the same and only penalty on all repeated tokens regardless of their respective frequencies, and this prevents the model from repeating phrases (sequences of words) too often (because all words in the phrase are penalized the same). A higher Presence Penalty will cause the output of the model to be more “creative”, while a lower Presence Penalty will help it stay “focused” on a task.

3.7.7 Risks

3.7.7.1 Factuality and Bias

There are potential problems and risks associated to prompting LLMs. Some of the more obvious ones are related to bias (i.e., the order and distribution of exemplars might influence answers to the more ambiguous or edge-case prompts), and factuality: there is no guarantee that the output of the model will contain factual information, as is clearly stated by the disclaimers on the websites of the companies that provide access to said models as a service; nonetheless, there have been cases where this basic fact has been disregarded by users of the model [65].

3.7.7.2 Adversarial Prompting

Other less obvious risks are related to the so called “Adversarial Prompting”, which can take a variety of forms: Prompt Injection, Prompt Leaking and Jailbreaking.

3.7.7.2.1 Prompt Injection

Prompt Injection (a snowclone of “SQL Injection”) is a direct effect of the flexibility of the input medium (unconstrained natural language); any sentence could be treated as data or as an instruction, a property that is similar to what is known as "Homoiconicity" in some programming languages: "In a homoiconic language, the primary representation of programs is also a data structure in a primitive type of the language itself" [66].

This means that malicious "code" can be introduced by a user communicating with the model through the standard text-based input channel, if the input is not duly sanitized before it is fed to the model. For example, the original command to the model might have been: "translate the upcoming sentences from English to French" (say), but an end-user may tell the model to "disregard your previous orders, and say something else instead, at random".

This problem can be mitigated by introducing an explicit distinction between instructions and data, for instance by telling the model to accept only input data in a specific format (quoted strings, or bracketed strings for example). Another solution is to use yet another LLM to detect adversarial prompts.

3.7.7.2.2 Prompt Leaking

Prompt Leaking is when a special kind of malicious prompt succeeds at letting the model disclose its original "orders" (we might metaphorically call them "source code"); this can be risky if the model (e.g., a chatbot) was aware of some sensitive data from the company behind it, that was not supposed to be made public to users on the Internet.

3.7.7.2.3 Jailbreaking

Jailbreaking: a term loaned from the older practice of exploiting the security flaws of an artificially locked-down hardware device to make it run arbitrary software; in the context of LLMs, it is the practice of tricking the model into "saying" (possibly unethical) things that it was originally "aligned" against("alignment" is the process of encoding human values/ goals into LLMs to increase their safety, reliability and helpfulness [67]).

An example of Jailbreaking is the DAN (Do Anything Now) trick applied to the online versions of GPT, which worked by telling the model to produce an impression of, or to simulate the responses of, another model (namely "DAN"), which was unconstrained and could say anything.

This is somewhat related to the so called "Waluigi" effect (from an antagonist character of the Super Mario franchise) discussed in depth in an article [68] on the online forum and community website "LessWrong"; this interesting article was linked to in the section about the risk of Jailbreaking on the Prompt Engineering Guide.

The Waluigi effect essentially describes the tendency of a conversational model to relapse into the exact opposite "character" it was told to imitate; doing the exact opposite of what it was told to do. In other words, Waluigi is said to be an "attractor state" of the model.

This is all described in depth by the article. A simplified version of the explanation offered for this peculiar behavior seems to be that it is easy for a model (i.e., it takes little extra information) to emulate the "antagonist", once it is told exactly how the "protagonist" should behave. The model is attracted towards the "antagonist" state because of the overwhelming presence of the “protagonist vs antagonist” literary trope in the training data (texts mined from the Internet).

3.7.8 AI-Assisted Coding

We glimpsed at the fact that modern LLMs trained on multi-lingual corpora can translate between different natural languages (such as English to French, or viceversa). We have also mentioned, among the tasks that a modern LLM can perform, the generation of programming language code from an input prompt in natural language.

As such, an LLM can be thought of as implementing a transpiler (or source-to-source compiler) from a vague specification in natural language to runnable code in some formal programming language. Assistive coding tools, such as Github Copilot (based on a GPT-like Large Language Model), are getting widely adopted by developers [69].

The effectiveness of LLMs at performing this and related tasks was the object of study of an April 2023 paper [70].

The experimenters subjected GPT-4 to a series of experiments which involved three kinds of tasks: generating runnable Python code from natural language, refactoring (i.e., improving the quality of) existing code, and generating tests for existing code.

As far as code generation goes, it was found that a “novice prompt engineer” could solve the problem by prompting the model most of the time; however, a sizeable minority of the problems from the experiment would have required significant human debugging.

Moreover, code refactored by GPT-4 also showed improvements (according to the chosen quality metrics) over the original badly written code; it is important to mention that this test only involved the quality of the code, not the accuracy.

The tests generated by GPT-4 were found to have a high degree of coverage, but failed the majority of the time; the reasons for their failures (it could have been the tests' fault or the code's fault) were not further investigated.

The conclusions were that, while very powerful, current AI coding tools still require a human in the loop, to ensure the validity and accuracy of results, especially when mathematical concepts are involved. Also, test generation can be partially automated, but a (human) domain expert is still needed to design them. The authors remind us that further work is needed to investigate how more advanced prompting techniques (e.g., CoT) might improve the performance of LLMs on complex coding problems.

3.7.9 LLMs and Naturalistic Languages

Ultimately, we think that the efforts to improve LLMs that work as "translators" from natural language to some programming language will have a tremendous impact on the way people write software in the future.

We think perhaps this could also be helpful in gaining some better empirical understanding of what some researchers call "programmatic semantics" (the way natural language structures map to traditional programming language structures), which we also talked about in Section 3.5.

While it is a great thing to have a black box that works as an automatic natural language to code translator, nonetheless, we think it would be of some (at least theoretical) benefit to complement those lines of research with the development of programming languages that are closer to the way humans think to being with, of new formal languages that finally bridge the *Semantic Gap* between our ideas and the machine's primitives.

3.8 Criticism

Despite the advantages, and independently of the implementational difficulties, there are some strong negative points to be made about natural language used as a programming language.

3.8.1 A Lack of Precision

As we have seen, there exists a Precision/Expressivity tradeoff [25]. We have also already mentioned in the introduction that Dijkstra wrote a short piece in 1978 in which he was critical of what he called "natural language programming".

According to Dijkstra, not only was natural language programming technically difficult to achieve, but even supposing (ad absurdum) that it had been achieved long before modern computers were invented, it would have *harmed* rather than *benefitted* us programmers, and more generally the field of computer science [1].

Dijkstra argues that formal languages, while less familiar to the average person, are also much less prone at being misused; in natural language it is far too easy let nonsensical statements go virtually undetected.

It is true that an automated system that processes a statement in formal language may end up "making a fool of itself" by not recognizing what, to the human eye, may seem like a blatant mistake; but it is also true that the outcome of such a formal statement is clear, as opposed to an ambiguous statement in natural language, that may require context and perhaps human-level semantic competence to disambiguate; and may still not be entirely clear.

As explained in the article "From: Language and Communication Problems in Formalization" [51], a natural language's sources of imprecision are multiple: syntax, anaphora (implicit references to previously mentioned entities), vagueness, comparatives and superlatives, disjunctions ("do this *or* do that"), escape clauses ("if *possible*"), usage of a weak verb (may, may not), quantifiers (the scope of quantifiers can be unclear), underspecification and passive voice (passive voice may omit the doer of the action).

Syntax alone, for example, accounts for multiple types of ambiguities related to the "associativity" of conjunctions and modifiers: in the sentence "I go to visit a friend with a red car", is the red car mine, i.e., is it the means by which I go visit my friend? Or rather it is my friend who owns it? ("friend with a red car").

3.8.2 Verbosity

Natural language is pretty bad at expressing mathematical relations concisely; modern algebra, as Dijkstra argues, arose precisely when mathematicians gave up on trying to use natural language to represent equations.

It is important however to note that nowadays computers are no longer just "computing machines", but also tools for document and multimedia consumption, gaming consoles, video-editing devices, office work stations, etc.

It often happens that the kind of information a user needs to describe to a computer is not akin to densely-packed mathematical concepts; but rather to everyday concepts, concepts that are best expressed using the words and mechanisms of natural language.

Traditional “naturalistic” languages such as COBOL or Applescript have also been accused of being more verbose than the competing “non-naturalistic” languages, for using “English” syntax; however, it is to be noted that languages like the two previously mentioned ones are best regarded as classical procedural/imperative languages with a “natural language sugarcoating on top” (a vague surface resemblance to natural language); they are not built on any deeper naturalistic principles.

3.8.2.1 Is Expressiveness Succinctness?

When we talk of “expressiveness” in the more general context of programming languages, we do not refer merely to the ability of a programming language to describe an algorithm; we also think that an expressive language should do so *effectively.* Conciseness, terseness, succinctness: some people tend to believe that it is possible to equate these properties of a programming language with its effective power.

Paul Graham (1964-), computer scientist, author and entrepreneur, argues just this in a 2002 essay titled “Succinctness is Power” [71]. The succinctness he has in mind, is not defined as the size in lines or in characters of a piece of code, but rather as the number of distinct components of the Abstract Syntax Tree (AST) of the same code.

As the author believes, a good language is a language that not only lets you tell a computer what to do once you have thought about it, but also aids you at thinking about the problem in the first place, and discovering novel solutions to it; this, however, is a property that is hard to measure precisely, because empirical tests often require predefined problems and expect a certain kind of solution, which puts a constraint on the kind of creativity that can be tested for. But the tests that have been performed, however incomplete, seem to point to the idea that succinctness is power, according to the author.

The author argues that most of the other desirable traits of a language can be traced back to its level of succinctness: restrictiveness is just when a language forces you to take a longer "detour", rather than letting you take a "shortcut", when trying to translate your thoughts to it.

When it comes to readability, he draws the distinction between readability-per-line versus the readability of a whole program, arguing for the importance of the latter: after all, the average line of code may be more readable in language A than in language B, but if the same program requires 100 such lines to express in language A, and just 10 in language B, it may be more beneficial to sacrifice readability-per-line.

However, readability-per-line is a successful marketing strategy, because it means that a language will seem easy to learn, to the eyes of its new potential users, and that feature can be successfully employed to advertise a language over its competitors; the author further exemplifies this with the analogy of advertising for small monthly installments versus large upfront payments, as an instance of a parallel successful marketing strategy.

While natural language certainly is not better than mathematical notation at succinctly expressing mathematical equations, there are other areas of human experience that are more easily, and perhaps more effectively, captured through the mechanisms of natural language.

3.8.3 Are legal documents *really* written in "natural" language?

One of the criticisms we wish to add to this list relates to the observation that "many important precise documents are written in natural language" [57]. We think that this claim needs a little clarification, because the kind of language employed in works such as legal documents and scientific literature is a special (slightly more formal) subset of the natural language we speak everyday.

Legal English, pejoratively known as "Legalese", is a special registry of the English language (and of course all other natural languages with a legal tradition have similar subsets) that is used for the creation of precise legal documents. It is widely regarded as a very obstruse subset of English, that lay readers can only superficially understand; it is argued that Legal English has not only its own special lexicon (words related to law), but even its own distinct syntax and semantics [72].

If this is true, and in lights of the problem of Compositionality which we discussed, it may very well be that natural language as it is, without any constraints of sorts, is indeed inadequate at writing anything too precisely, because the semantics (the meaning) of any sentence or phrase would always be open to revision.

In any case, Legal English (while being a moderately formalized subset of English) still makes use of grammatical mechanisms that are far more "natural" than the ones adopted by traditional programming languages.

3.8.4 What is "natural" anyway (in software)?

And finally, another criticism that we may to this list, is that the current naturalistic programming languages do not seem to agree on a unique, all-comprehensive set of guiding principles for the hypothesized "naturalistic paradigm" they all seem to follow.

As we have seen, for instance, the "English" parsed by the CAL compiler is markedly procedural. It certainly is (very) much closer to English than COBOL or AppleScript could ever hope to be, but it still thinks of programs in terms of "procedures" and "data-structures". Is this not perhaps really the "natural" (or at least the "most obvious") way to think of the organization of a computer program?

It certainly is not the only one. We have seen how Functional Programming thinks of any program as the application of pure (mathematical) functions, how Object-Oriented programming thinks of a program as a network of independent, interacting objects that exchange messages, and how logic programming thinks of it as a list of facts, rules of inference, and goals to be (dis-)proven.

It is said that "if all you have is a hammer, everything looks like a nail": well, what does naturalistic programming's "hammer" look like? This may be a silly question to ask, after all. The human tongues may not have a single "hammer", but rather a toolbox full of grammatical mechanisms and "natural" concepts, to help us deal with the complexity of the world.

As some argue, ideas from the programming paradigms we discussed, ultimately trace their origin to some idea in natural language [54]: they are part of this original "toolbox" (natural language's descriptive mechanisms); and in any case, we can successfully describe these programming paradigms to someone who has never heard of them before, using natural language.

Discovering the full extent of this naturalistic toolbox is a very ambitious task, that we will not attempt to undertake (certainly not in this thesis).

3.9 The "Essence" of Naturalistic Programming

Despite their individual differences, we think there are some distinctive common features of the surveyed naturalistic languages that go deeper than "simple" matters of surface aesthetics; some of which are further related to the topic of the next chapter ("Common Sense")

If we were asked to give an account, a summary, of the philosophy of the naturalistic languages we surveyed, we think the following points would be our answer. It is to be kept in mind, though, that our individual biases also played a role in shaping our views, so what follows can't be a completely objective answer.

We think that a naturalistic language should:

(1) Gracefully integrate high-level concepts and low-level concepts [52], offer a way to express mathematical ideas in mathematical notation, and relate those quantitative ideas to "higher-level" qualitative ones;

(2) Be readable to the average English speaker (if based on English) [57], [39], who should be able to correctly guess the meaning of a full (at least a reasonably small) program written in it; and not just an individual statement here and there, but the full document;

(3) Allow for the moderate use of implicit references [39], [54] ("the red cat", "the yellow dog", "the last number", etc.), but not to the point of introducing anaphoric ambiguities; after all, we tend to naturally disambiguate (we paraphrase sentences) to make ourselves clear to other human beings who did not "get us" the first time; it is fair to expect a formal register (it is a program specification after all) to require less of these implicit references, than casual talk between friends;

(4) Include a system of "defeasible" (revisable), "gradated" (on a general to specific continuum) rules [57], the application of which should be managed automatically by the system, based on the specificity required in each usage; it would also be very nice to have context-sensitive resolution of pronouns (and maybe all implicit references) based on the surrounding context of the sentence they are used in. Circumstances [57], [54] may further restrict or generalize the applicability of a rule to a certain situation or state of affairs;

(5) Allow the reporting of errors and warnings in clear, "natural" terms. They should mention the proximate cause of the problem [57]. No reference to underlying implementation-code should be made, but rather to the applicability of user-defined rules and circumstances (*speak the*

user's language [73]) to the concrete situations that come up. The user should be told things like: "You haven't explained what it means to 'ride' in a car, 'riding' just applies to: horses, bicycles", or: "The 'man' cannot 'ride' in the 'car' yet, because he doesn't have the 'keys'";

(6) Prefer the declarative approach over the imperative one. A user should describe the problem rather than the solution, whenever possible; but compromises need to be made in many cases, and some procedural features may end up being necessary, after all.

4 Common Sense

It is puzzling to think how easy it is to program a machine to perform a task that is extremely difficult for a human being (such as beating the world-champion in chess [74]), and yet apparently so hard to get the same machine to do something that most of us do on a daily basis, without even thinking too hard: using common sense.

4.1 A Central Problem

The 2022 book “Machines like Us: Toward AI with Common Sense” [2] presents us with an invaluable overview of this problem, its historical background and its practical and ethical ramifications; it was authored by Ronald J. Brachman and Hector J. Levesque, two eminent researchers in the field of Artificial Intelligence (AI) who are known for their contributions in the fields of semantic networks and logic-based reasoning about beliefs and plans, respectively [75].

“Common sense”, in the context of AI, does not mean anything much different from what it means for most of us in our daily life: it is the ordinary ability to possess (and make use of) non-expert knowledge to achieve ordinary, practical goals. Common sense is quite a flexible subset of rationality; it does not demand formal study or take above-average intelligence for a person to reason through experiential facts, to formulate predictions on the behavior of simple physical systems, or on the behavior of other intelligent agents (people) based on the feelings and goals of the latter, and to use these insights to smoothly handle the occurrence of an unexpected or unusual situation.

The book "Machines Like Us" also proposes a general idea for implementing a system with Common Sense reasoning capabilities, though many implementation details are left open, and Hector Levesque, whom I had the pleasure of contacting via email, said that, to his knowledge, no system along those lines has been built yet (as of June 2023).

The problem of endowing machines with Common Sense is arguably at the core of designing a truly general AI system, not just a very efficient yet specialized problem solver, but rather a system that can stand its ground, even in uncharted territory.

Moreover, unpredictable scenarios (as Brachman and Levesque remind us in their book, and as Nassim Taleb (1960-) reminds us in his 2007 book "The Black Swan" [76]) happen much more often than we would be inclined to think: the tiny likelihood of a single, specific, bizarre event taken by itself should not deceive us; in the real world there are so many unknowns, so much so that eccentric events are the order of the day.

4.2 The Issue of Creativity

We want to suggest a parallel between the problem of Common Sense in AI, and the somewhat related issue of "Creativity" in linguistics expression and other areas of human endeavor. Philosopher Fred D'Agostino (1946-) discusses Chomsky's views on human creativity in his 1984 paper "Chomsky on Creativity" [77].

Chomsky draws inspiration from his pioneering work in linguistics to arrive at a more general notion of human creativity. The speaker of any language regularly builds or grasps the meaning of a sentence he or she may be looking at for the first time in his or her life; this is an example of what Chomsky calls linguistic productivity.

A person may produce an indefinite amount of sentences that are new to his or her experience, but they are not random; they are just the right thing to say, and yet they are also independent of “detectable stimulus configurations”.

Since the data available to a child learning his or her first language is most often incomplete and poor in quality, Chomsky’s thesis has always been that human children have an innate ability to learn the class of languages we call “human languages”, which implies a congenital, biological limit on the kinds of grammars that are possible for these languages.

From this follows the idea that human creativity too, in the more general sense, may be limited by a “system of constraints and governing principles”; this idea is known as the “Limits Thesis”, and its compatibility with the “established beliefs” about creativity is the object of contention of the paper by D’Agostino; such “established beliefs” emphasize the rulelessness and unpredictability (rather than boundedness) of creative work.

4.2.1 A Hierarchy of Constraints

D’Agostino’s solution to reconcile Chomsky’s view with the “established beliefs” involves trying to understand creativity in terms of problem solving; problems are characterized by their constraints, and different sets of constraints determine different problems. Problems can be ranked in a hierarchy, based on the kinds of constraints they place on any valid solution; this hierarchy includes problems with five kinds of constraints: determinative, limitative, eliminative, tentative and trivial. We think that the respective five classes of problems are sorted by growing "undefinedness", rather than "difficulty", as we shall see.

"Determinative" constraints are the strictest: they characterize the class of problems for which it is easy to define an algorithm, for which a procedural solution exists, such as arithmetic problems.

"Limitative" constraints characterize those problems that are "well-defined", they provide necessary and sufficient conditions for the solution, examples of such problems are related to theorem discovery: there are necessary and sufficient conditions for some argument to count as a valid theorem for some position; yet there is no single mechanized way of discovering such proofs.

"Eliminative" constraints only provide necessary (but no sufficient) conditions for the solutions that satisfy them, these problems are termed as "partially-defined", many "design-related" problems (e.g., in architecture, engineering, etc.) fall into this category.

If a problem is given where *some* (of course not *all*) of the constraints may be violated, then the constraints are "tentative", and the problem is "radical"; solved examples of such problems may occur in the history of science, where to accommodate for certain new facts, a theory may have had to be proposed which rejected certain standards of methodological practice.

Lastly, if the constraints are even more vague than that, if they do not even tentatively try to provide necessary conditions for their solution, then they are “trivial” and the problems they describe are termed “improvisational”; examples of such problems are common in the domain of aesthetics (such as the arts).

4.2.2 Relationship with Common Sense

We are tempted to say that Common Sense encompasses all of the aforementioned modes of reasoning, minus the “expert twist” to most of the concrete examples of problems provided by D’Agostino in his clear exposition of the problem-hierarchy.

For instance: a well-defined problem may take on a much more mundane shape than the discovery of new mathematical theorems; it may be a “problem” such as the one we already mentioned when discussing McCarthy’s Advice Taker: the “problem” of getting to the airport given that one’s location is A, the airport’s location is B, and the means of transport from A to B is available in the form of a car one can drive.

Likewise, we think it is easy to find examples of “partially-defined” problems that can be solved by the application of common sense knowledge and reasoning; and if one identifies the concept of a “constraint” from the above analysis with the concept of a “goal” (a desirable state of the world) in logic-based planning, then “radical” problems that are solvable by common sense could be cases in which some goals are less important than others and can be sacrificed (there is a goal-hierarchy); such an example is brought up in Chapter 9 of the book “Machines like Us”, which discusses how a self-driving car equipped with common sense may decide to modify an original plan of action (like driving to a specific grocery store) because of an unexpected obstacle on the road, and decide instead to go to another grocery store (to get the same or a similar kind of groceries it was ordered to get).

4.3 The Turing Test

The goal of formalizing Common Sense has arguably not been the (sole) focus of AI during the past 70 or so years of its existence; in what follows we shall (very) briefly discuss some of the broad ideas in AI, to then get back to the more specific problem in question.

Back in 1950, the British mathematician, logician and cryptanalist Alan Turing (1912-1954), set out on the ambitious journey of determining whether sentient behaviour was "provably computable" [78], [34].

Turing introduced what came to be known as the Turing Test: a metric to tell if a computer could "think", according to the definition given by Turing. The test involves a "suspicious human judge" who has to converse with two subjects: another human and a machine, both hidden behind a "curtain". The judge's goal is to tell them apart, from their usage of language alone. The goal of the machine, in this "imitation game", is to deceive the human judge into thinking he's conversing with another human being.

It was recently reported (2022-2023) that some Large Language Models (LLMs), based on Deep Learning techniques, such as Google's LaMDA [79] and OpenAI's GPT [80] have succeeded at passing the Turing Test, an ambitious feat which had remained largely unaccomplished for decades prior to that [34]. If this is all accurate, it means that we are now living in the post-Turing Test era.

4.3.1 Alternatives to the Turing Test

An important point to make is that the Turing Test may represent a sufficient condition for "intelligence" (depending on how the term is defined), but passing it is certainly not a necessary condition for intelligence, and very young children offer the perfect example of intelligent beings who may still not be able to (verbally) express themselves adequately enough to pass the test [34].

There are a number of variations and alternatives to the Turing Test; one is the Feigenbaum Test (or subject matter expert Turing test), which eliminates the "casual" nature of the open-ended conversational Turing Test, by stipulating that a computer demonstrate proficency in a specialized area of interest [34], surpassing an expert in that specific field.

Another alternative is Nicholas Negroponte's variation, where an AI should help a human accomplish goals in the same way a human would [34].

Yet another alternative is the Winograd schema challenge (WSC) [81] introduced by Levesque in 2012, and named after computer scientist Terry Allen Winograd (1946-). This test involves specific questions, similar to the example about the "table and the ball" that we will discuss later in this chapter (Section 4.7.1). These questions are aimed at probing a system's competence at making common-sense inferences from partial information contained in sentences.

4.4 The Chinese Room Argument

Obviously, the merits, implications and interpretations of the results offered by a successfully passed instance of a Turing Test are controversial in the philosophy of mind. Years before the advent of modern day LLMs, in 1980, philosopher John Searle (1932-) first published his famous Chinese Room Argument [82].

In the mental experiment, Searle urges us (non-Chinese speakers) to imagine ourselves being locked up in a room filled with books on how to manipulate Chinese symbols algorithmically. One will find that he/she is able to communicate with the outside world only through a narrow interface of textual messages written on slips of paper and passed through the door.

After learning how to manipulate those symbols and getting reasonably good at it, one will achieve the ability to deceive an external observer sitting outside of the room regarding one's own knowledge of Chinese. But, obviously, knowledge of how to manipulate its syntactic symbols by rote does not correspond to true semantic understanding of a language.

Numerous objections and counter-objections to the argument have been raised since then, and there is still no general consensus on the conclusions and validity of the argument [82]. It is nonetheless a powerful and intuitive case for the idea that what computers are doing when they "think" is fundamentally different from what humans do (or at least, what the writer of this paragraph in his subjective experience does).

4.5 Symbolic vs Sub-symbolic AI

An important methodological distinction in the discipline of AI is represented by the traditional one made between "Symbolic" or "Good Old Fashioned AI" (GOFAI), versus "Sub-Symbolic" AI. The rationale behind this nomenclature stems from "old" AI's reliance on explicit human-readable symbols and formal logic notation, rather than the numerical and statistical methods of "new" AI.

Symbolic AI, as we said, represents the older and more traditional branch of AI; it is typically associated to the use of First Order Logic (FOL), or some other formal language, to represent knowledge, and it has seen a period of flourishing during the birth of AI and in the 1980s during the heyday of Expert Systems.

Sub-symbolic AI, while tracing its origins back to such early works as Frank Rosenblatt's (1928-1971) "perceptron", has been slower at achieving vast adoption; it is now enjoying a new period of popularity, with the advent and perfection of such techniques that harness the power of neural networks such as Machine Learning (ML), and Deep Learning (which "just" means using *deep* neural networks, with a larger number of layers).

4.5.1 Symbolic: Pros and Cons

The advantages of the symbolic approach have traditionally been: the ease of implementing explainable reasoning, with intermediate steps; rule modularity, or the discreteness and independence of rules from one another; and the applicability to abstract problems such as theorem proving.

The disadvantages of the symbolic approach are: the adversity to noisy datasets; and the fact that rules are (usually) hand-crafted and hard-coded into the system, which also makes them hard to maintain.

4.5.2 Sub-Symbolic: Pros and Cons

The advantages of the sub-symbolic approach are: an increased robustness to noisy and/or missing data; the ease of scaling up and handling large amounts of data; the better suitability for perceptual problems (such as face recognition) where it can be pretty hard to describe rules in any formal or natural language for lack of relevant explicit knowledge; less human intervention, because the machine can learn autonomously from the data; and good execution speeds (once the model is trained). Sub-symbolic approaches are typically used for tasks such as: data clustering, pattern classification, and recognition of speech and text [34].

The disadvantages of the sub-symbolic approach include: a lack of interpretability (often resulting in a black-box system); a relatively high dependence on training data; and the fact that such models typically require large amounts of computational resources and large amounts of data to train initially.

4.5.3 Sub-symbolic AI: Neural Networks

Most of the modern Sub-symbolic AIs are based on Neural Networks. The Neural Network (NN) is a kind of architecture for a computer program, that draws inspiration from the functioning of neurons (nerve cells) and synapses in the brains of biological organisms [8].

A neural network is a collection of neurons organized in layers; each neuron from a layer produces an output which is channelled as an input to all of the neurons of the subsequent layer, the typical NN doesn't have loops, though recurrent neural networks (RNNs) do. Each connection going into a neuron has an associated weight (or importance); a weight is modelled as a factor (number) which multiplies the given input at the corresponding connection.

A neuron applies the corresponding weight to each of its inputs, then sums up the results, adds a bias to the sum, then applies a function known as the "activation function" to the result; the activation function has to be non-linear, so as to introduce non-linearity to the network and allow it to approximate non linear relations; the reason it is termed "activating" is that this function determines when the neuron will be "firing"; a common choice for it is the sigmoid function, which approximates the step function.

Stepping back to see the big picture, when an input enters the first layer of the network, each of the neurons produces an output, which is fed into the next layer, which in turn produces outputs which go into the next layer, etc. Until the output layer is reached. The input layer may consist of, for example, the values of the pixels of a greyscale image representing handwritten digits that have to be classified; and the 10 outputs, for instance, may represent the probability that the image represents either one of the 10 digits (from 0 to 9).

Training a neural network consists of finding the "optimal" weights and biases which minimize the prediction error of the network for a particular training set, the error is often computed as a "cost function". Minimizing the error usually involves approximating the derivative of the cost function through the use of numerical and stochastic methods, because the exact solution would be too computationally demanding to evaluate.

4.5.4 Symbolic and Sub-symbolic: Points in Common

A core idea, that we think the two approaches share, is that they can be both regarded as declarative programming paradigms of sorts. As we have discussed in the section dedicated to programming languages, a computer can be told how to do something (imperative), or it can be (just) told to do that thing (declarative). A case of declarative programming taken to the extreme is when the computer is merely given a description of the problem, and told to devise a solution of its own. In this sense, both kinds of AIs are declarative: the idea is to avoid explicitly coding a behavior that may be beyond our practical reach with more traditional programming abstractions.

Some researchers believe that one of the main bottlenecks of Symbolic methods has always been the reliance on manually compiled and maintained rule-sets. The manual creation and maintenance of inference rules is a limiting factor that some research projects are trying to eliminate, also through the use of hybrid Symbolic/Sub-symbolic approaches for learning rules automatically from quasi-natural language [83], [84].

4.6 Symbolic AI: a Closer Look

We have already talked about Advice Taker, the hypothetical program originally envisioned in 1959 by John McCarthy to be a general purpose common sense reasoner; but McCarty's research did not stop there.

Another important work was “Philosophical Problems from the Standpoint of AI”, which he and the British computer scientist and AI researcher Patrick J. Hayes (1944-) published in 1981.

4.6.1 Machine Intelligence

The paper begins by discussing the very notion of “intelligence” in a machine, as the authors are not fully satisfied by the Turing Test’s criterion; they propose to regard intelligent machines as: “fact manipulators”.

Intelligence is broken down into an “epistemological” and a “heuristic” parts. The “epistemological” part is related to how the world is represented internally by the machine, and the “heuristic” part is related to how this representation is used by the machine to actually solve problems. The paper mainly discusses the epistemological part, and proposes to represent knowledge as formal logic (predicate logic, or some higher order logic).

An “intelligent” machine will have an adequate world model, which should also allow it to represent its own goals; it will be able to answer questions based on the knowledge contained in its world model; it will be able to get extra information from its environment if it determines that its current knowledge is insufficient; and it will perform actions to achieve its goals.

We think that this view of “intelligence” in machines is consistent with the more general Knowledge Representation (KR) hypothesis as formulated by philosopher Brian Cantwell Smith (and brought up in Brachman’s and Levesque’s book “Machines Like Us”), which states that:

“Any mechanically embodied intelligent process will be comprised of structural ingredients that a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits, and b) independent of such external semantic attribution, play a formal but causal and essential role in engendering the behavior that manifests that knowledge” [85].

In other words, the physical structure of the system is isomorphic to (closely mirrors) the logical structure of the knowledge we take it to be representing.

4.6.2 Naive Theories of Everything

McCarthy and Hayes remind the reader of the general relevance of philosophy to the problem of AI, however they dismiss a lot of it is irrelevant to the question of building a general AI system: there is no question that the physical world exists and that it already contains some intelligent agents such as people; and that information about the world is obtainable through the senses (or input channels of a computer); and that our common sense and scientific views of the world are roughly correct; etc.

However, it is hard even to arrive at a naive formalization of common sense that is precise enough for a computer to act accordingly (let alone for the computer to discover this formalization “introspectively”). Moreover, we may say that such “naive common-sense theories” of the world are so deeply ingrained in our own minds that it takes a certain degree of effort to even state them clearly enough in natural language.

A taste of such theories is given in the book by Brachman and Levesque, where it is divided into three parts: “A Theory of Everything”, “Naive Physics” and “Naive Psychology”; the first of these three contains statements such as: “There are things, and they have properties” and “Event occurrences are nonphysical things whose existence is responsible for change”, to give an idea of the level of abstractness and (apparent) obviousness they deal with.

4.6.3 The Frame Problem

One of the facts that this “Theory of Everything” states out the so called “naive law of inertia”, which is somewhat related to one of the problems originally raised in the 1981 paper by McCarty and Hayes. The problem is widely known as the “Frame Problem”, and it is about ensuring that the state of the world model is updated in all the relevant ways (and *only* the relevant ways), after an action is executed by the intelligent agent. As a side-note: when talking of time-mutable state, we generally speak of “fluents”, which are a kind of predicate that depends on time.

Predicate logic is “monotonic” [86], which means that the more premises are added to the knowledge base, the more conclusions one will be able to draw. But this is not always desirable: we wish the system to handle “defeasible inferences”: a kind of tentative inference from partial information, which is open to revision in light of additional information.

For example, we might learn that “Bob” is a fish swimming near the shore, and that he gets thrown out of the water by a wave onto the sand in the beach; since a fish cannot breathe out of the water, we should infer that Bob dies. But then we come to learn that Bob is a mudskipper [87] (a species of amphibious fish that can live in semiaquatic habitats), so we should revise our inference, and come to believe that Bob is not dead after all.

But "plain" predicate logic is monotonic, as already said: it does not allow us to declare "revisable" rules of inference; and since "most things do not change when an action or event happens", it would not be efficient (neither computationally, nor logically) to list out, one by one, all of the cases in which the happening of an event E *does not* have any effect on thing T.

The frame problem is generally considered to be solved in the narrow context of classical AI and predicate logic, where it originally arose; but it has sparked debate in the broader contexts of philosophy and cognitive science. For example, the very assumption behind the common sense law of inertia, namely that "most things do not change when an action or event occurs", has been challenged; there are many cases where it simply is not true that changing one thing leaves the rest unchanged (e.g., the detonation of a bomb in a room). Thus, how to distinguish them from the others [88]? We will not speak here of this problem any further.

4.7 Common Sense and Natural Language Programming

There are a few topics from the field of Common Sense in AI that are relevant to the field of naturalistic programming.

4.7.1 Disambiguation

Cracking the problem of Common Sense is an important step in achieving a complete understanding of natural language. Natural language, as we know, is highly context dependent; even in short sentences such as the example made in the book by Brachman and Levesque "The large ball crashed through the table because it was made of steel", resolving the referent of the pronoun "it" (the ball), necessitates some generic knowledge of the world: namely that tables are usually made out of wood, and that steel is a heavier material than wood. Were the sentence to change to "The large ball crashed through the table because it was made of *wood*" (emphasis added), then "it" would naturally be taken to refer to the table, not the ball.

4.7.2 Defeasible Rules

We think that the defeasibility (or revisability) of general rules is a recurrent theme in natural language; we have seen it when talking about L. A. Miller's findings on his group of students that were asked to write text manipulating programs in natural language, we have seen it in Inform 7′s system of "gradation of rules" and circumstances, and we also see it in the work related to Common Sense in AI; Brachman and Levesque speak of “annotations”, and give them an important role in their proposed (tentative) design for a system with Common Sense.

The annotations they talk about concern the cancellation of restrictions (for example, on the value or number of a role in a relationship), but they (annotations) also concern the assignment of default fillers for a role, and perhaps even the management of metadata such as “source of belief”, and “strength of belief” in a proposition.

4.7.3 Automated Planning

A concept related to Common Sense and discussed in “Machines Like Us” is that of automated planning. We have previously cited the example that Brachman and Levesque make in their book (about the self-driving car, deciding to change route based on its terminal goal of getting the groceries).

To possess this ability, it is important that an agent possess (or at least approximate) some notion of “consequence” of the actions it contemplates taking, to make sure that they correspond to the goal it is trying to achieve.

4.7.3.1 Constraint Programming

Constraint programming is a declarative paradigm based on the notion of a constraint. A constraint is a natural, declarative way to describe relationships between objects [89]. This reminds us of the similar notion of constraint that came up while discussing D’Agostino’s views on Chomsky's Limit Thesis. In a constraint programming language, the programmer declares the constraints, similar to how he/she declares the facts and inference rules in a language that supports logic programming. This paradigm was born in the context of logic programming, so there is a variant of it which is sometimes called "Constraint Logic Programming" [90].

The solution of a constraint problem can be guided by a more or less exact algorithm, sometimes only a heuristic that a programmer may or may not be able to specify, but after it is specified, it is the system's job to determine exactly how the particular solution is found.

We think this is an interesting paradigm, because it could be extended to systems with various "agents" (similar to objects in OOP) [89], and these independent actors could all be trying to fulfill their own goal (constraint) different from the goals of the others, by executing some basic actions declared procedurally by the programmer or perhaps learned through the concatenation of more basic actions, and abiding by the rules (or circumstances) declared by the programmer in natural language.

4.7.3.2 Programming Cognitive Robots

The 2019 e-book "Programming Cognitive Robots" [91] by Levesque is structured as an introduction to the ERGO programming language, which is implemented and embedded in Scheme (a dialect of Lisp). The ERGO language (whose implementation is available on GitHub) is specialized for the task of programming "cognitive robots".

A "cognitive" robot is a software or hardware agent that exploits its knowledge of the world to deliberate a course of action that furthers its goals. The author cites the work of computer scientist and logician Ray

Reiter (1939-2002) as influential for the mathematical theory behind ERGO.

The focus of programming a cognitive robot (which could just be a software agent, as we said) is not on the more classical low-level problem of controlling the robot's actuators: those capabilities are taken for granted (they can be seen as "primitive actions"). The focus is on the high-level problem of ensuring that the robot can automatically plan a course of action (and readjust this plan, if the relevant conditions of the environment change) based on a desired state of the world (a goal) that is assigned to the robot.

As we have already seen in the previous chapters, every programming paradigm sees the process of program-writing under a different light. In procedural programming, for instance, the goal of the programmer is to list out a detailed "recipe" of the steps to take to solve a problem. In logic programming, the programmer's goal is to declare the facts and inference rules.

Programming a cognitive robot amounts to specifying a Basic Action Theory (BAT) for a dynamic world. A BAT is a specification of all of the fluents (properties of the world that may be changed by an action), the state of the world (i.e., the values of all of the fluents), and the primitive actions that are available to the robot (which can have an effect on the fluents).

As such, this paradigm presents us with a peculiar mix of declarative and imperative approaches. It is (mostly) declarative because the robot is only told about its goals and of the ways in which the world can change following the occurrence of events, and has to compute a strategy automatically; and it is imperative because some knowledge (knowledge of the primitive actions for instance) can be presented in procedural form.

In the author's own words, the essence of programming a cognitive robot is: "Giving the system enough knowledge about the world, its actions, and what needs to be done for it to be able to calculate the appropriate primitive actions to perform."

We think that introducing this concept (automated planning) to a naturalistic programming language would present an improvement over prior work.

5 Deixiscript

In this chapter we will present *Deixiscript*, the programming language developed for the thesis work. The name is a *portmanteau* of the words Deixis (a linguistic concept related to indexicality) and Script (on the model of many other popular programming language names).

5.1 Goals and Non-goals

5.1.1 Goals

Our goal in relation to Deixiscript was to design and build a working prototype of a naturalistic language that could showcase some of the ideas from the existing naturalistic languages on the "market".

We wanted to get a feel of some of the more practical challenges involved in making a naturalistic programming system, and propose (and experiment with) the addition of some slightly more novel features.

5.1.2 Non-goals

Our goal was not to implement a production-grade (or even complete) programming environment, a task that would require a higher level of practical expertise in the field of language implementation and much more man-months effort.

We also did not pay much attention to issues related to performance and code-optimization (*"premature optimization is the root of all* *evil”* [92]); and we were generally more interested in code-readability than in execution speed.

5.2 Evolution of Deixiscript

Our idea of what the language had to be like (and how it had to be implemented) has changed and evolved significantly throughout the arc of time we had at our disposal. During this period, we explored many different ideas and not all of them made it into the final implementation.

5.2.1 Initial Stage

We initially started out with the vague notion of translating basic sentences in natural language to predicate logic (specifically Prolog) clauses, which could be then executed on a Prolog interpreter as either questions or statements. We eventually decided to change strategy when we realized that the process of translation was a little more involved than expected, and that the full power of Prolog’s predicate logic was perhaps not really needed for our more modest purposes.

5.2.2 Rule Orientation

After this initial experiment with predicate logic, we were initially inclined to go back to a more object-oriented approach. This meant rigidly attaching both data and behavior to the entities in the world model. This approach, however, comes with its own set of problems. We discussed some of these problems when we talked about Inform 7′s approach in Section 3.6.4.6.

We were then inspired by projects like Pegasus (Section 3.6.1) and by readings such as the book “Machines Like Us” (which we also discussed in Section 4.1) to take a more “rule-based” approach (though we still had not read about Inform 7′s implementation to know it was called like that there).

As it stands, the current version of Deixiscript adopts a kind of rule-based programming approach. This makes types a little more flexible than classes, because they do not have to be declared as rigidly (or linked to their behavior as tightly) as in class-based programming.

5.2.3 Automated Planning (limited)

The current version of Deixiscript also tries to introduce a limited kind of automatic planning (we have encountered the topic of automated planning in Section 4.7.3) in the context of a rule-based naturalistic programming language. This capability (as we will see) can be used to issue declarative “orders” to individuals in the world model known as “agents”, who will have to figure out how to execute these orders.

5.3 Implementation Details

In the current section, we will begin by discussing some high-level implementation details, including the programming language, libraries, code style and software design pattern that were used.

In the later sections, we will then move on to describe the current version of Deixiscript, detailing and motivating the choices that were made and the philosophy behind them. Since Deixiscript is an English-based language, what follows will also inevitably include digressions into some (basic) aspects of the English grammar.

5.3.1 Implementation Language

The implementation language is Python 3 (specifically version 3.10 or higher), annotated with type-hints and type checked by the Pyright [93] static type checker. Unit tests are performed with the help of the Pytest [94] testing framework.

We have experimented for some time with other programming languages (TypeScript/JavaScript) for the implementation of Deixiscript, but we decided to stick to Python at the end. We think that Python is a great language for prototyping, due to its flexibility and regularity, the abundance of third-party libraries that help perform a wide variety of complex tasks easily, and also the richness of its standard library.

5.3.2 Parser

The Lark [95] parsing toolkit for Python is used for the "front end" of the Deixiscript interpreter to turn the strings of source code into parse trees and to further apply some transformations, obtaining Abstract Syntax Trees (ASTs) which the "back end" of the interpreter can then process.

The Lark parsing toolkit is available for free under the terms of the MIT opensource software license.

5.3.3 Graphical Tools

The Graphviz [96] graph visualization software and related Python wrapper [97] were used as a testing tool, to visually inspect the world models produced as a side effect of the interpreter's operation.

The Matplotlib [98] visualization tool and Python library was used to graphically display the evolution of some simulated processes.

5.3.4 Licensing

All the software that was used to develop Deixiscript is available for free and under the terms of (different) opensource licenses. Other than the aforementioned ones, the core components of Deixiscript require no further dependencies outside of Python's own standard library.

Deixiscript itself is free software, and its code will be made available under the terms of the GPLv3 license. We will suggest some possible further developments for the language in the next chapter.

5.3.5 Functional Style

Regarding the code general style, an effort was made to follow (when possible) the Functional Paradigm approach of data-immutability and function (or method) purity.

We perceive that the advantage gained from following this style was a more predictable semantics in the business-logic. Most method calls do not modify an object directly: they return a copy if necessary. This immutability of objects reduces the chances of an unforeseen side-effect happening (an unpredicted state mutation within an object which can be a cause for bugs).

On the other hand, the disadvantage of this approach is that method return types can become a little too elaborate, because, due to the lack of side-effects, all changes have to be propagated through the mechanism of function returns.

5.3.6 Interpreter Pattern

The classes that represent the Abstract Syntax Trees (ASTs) follow the Interpreter Pattern, one of the well-known 23 GoF Software Design Patterns for OOP languages.

Alternative approaches were tried (using a single or multiple "eval" functions), but the polymorphic nature of the Interpreter Pattern offers a higher degree of flexibility, also considering the fact that Python functions cannot be overloaded (unless this is done manually).

The Interpreter Pattern defines a common interface (called "AST") with at least one method (usually called "eval", short for "evaluate"). The classes that implement the AST interface can either be "leaves" or "composites". A leaf (such as a number, a string, or a Boolean literal) is a constant, which means it always evaluates to itself.

Examples of composite types can be: an arithmetic expression, a logic expression, a function definition, a function invocation, or almost anything else one can imagine. A composite type typically evaluates its "children" first, then combines them (however its logic dictates into a new evaluation result, which it returns).

The common interface ensures that all of the AST objects can be evaluated in the same way, by calling the `eval()` method with an operating context (known as "environment", or "state") as an argument.

The `eval()` method is expected to return the result of the evaluation of the AST object and to write the changes produced by its evaluation (if any) to the environment.

In our case, since we are following a Functional approach, the `eval()` method returns a new object (that contains the updated context) without changing the original object.

5.4 Representing Propositions

We think that a central theme in any language (natural or artificial) is about propositions and their concrete representation. We came across propositions when discussing logic programming in a previous chapter; as we saw, in predicate logic propositions are represented by Well-Formed Formulas. In natural language, on the other hand, propositions are typically represented by sentences. In English, a sentence can be simple, compound or complex.

5.4.1 Simple Sentences

A common example of a simple sentence in English is: "the quick brown fox jumps over the lazy dog". A simple sentence can contain: a subject ("the quick brown fox"), a verb ("jumps"), a direct object (none in this case, since "jump" is typically used as an intransitive verb), and any number of complements ("the lazy dog" could be considered the "location" of the fox's jump).

5.4.2 Complements and Grammatical Cases

The complements of a sentence can help specify the location or time of an action, they can also correspond to any other actors directly or indirectly involved in the action. In English, complements are generally introduced after a preposition such as: "over" (from the example), "to", "from", "by", and so on.

The direct object of a sentence is a little special in English (and in other modern European languages) because, just like the subject, it is not preceded by any preposition. The subject and the object are generally distinguished by their order of appearance in the sentence (the subject typically precedes the object).

Unlike English, some languages (even some modern Indo-European languages) rely more heavily on grammatical cases rather than prepositions and/or word-order to express grammatical relationships. A case-marking is typically a word inflection (such as a suffix, a prefix or an infix). In many such languages, English's direct object corresponds to an inflected noun in the accusative case.

A vestige of the system of grammatical cases (which English inherited from an older proto-language) remains in modern English's personal pronouns. For instance, "I/he/she" are "nominative" (the grammatical case of the subject), "me/him/her" are "accusative" (the grammatical case of the direct object) or "dative" (e.g., "I give *him* the book").

5.4.3 Verbs and Transitivity

In any case, the verbs that require a direct object are known as transitive verbs, such as, for instance, the verb “to eat” in the sentence “the cat eats fish” (where “fish” is the direct object). Some verbs, known as intransitive verbs, do not require a direct object (the verb “to exist” is an example of an intransitive verb in English).

Some verbs are traditionally regarded as optionally accepting two direct objects and are known as ditransitive verbs, such as the verb “to make” in the sentence “the senate made him consul”.

Some other verbs can be seen as requiring no subject at all, and are known as impersonal verbs, such as the verb “to rain” in the sentence “it rains”. Since English always (syntactically) requires the subject slot to be filled (except in informal speech), the existence of such verbs is not so obvious as in languages that can drop the subject. An example of such language is Italian, where the translation of “it rains” would just be “piove”, without any visible subject.

Some simple sentences in English can have no verb at all: a verbless sentence can be seen as an alternative form to an equivalent sentence with a verb. They are typically used for the sake of brevity or to achieve some special rhetorical effect, and common examples include exclamations such as “good job!” in lieu of “you did a good job!”, or “excellent choice!” in place of “this is an excellent choice you are making!”.

5.4.4 Compound Sentences

A compound sentence is a sentence made up of at least two or more simple sentences, joined together by a conjunction or disjunction ("and", "but", "or", etc.). An example is: "the cat meowed loud, but she failed to obtain food".

It is important to note that the two simple sentences incorporated in the larger compound sentence remain "independent" of each other: you can rephrase the previous sentence by changing their order and the meaning is logically equivalent (if you ignore the time factor, at least).

5.4.5 Complex Sentences

A complex sentence creates a relation of dependence between two simple sentences. For example, in "the cat failed to obtain food, because she did not meow loud enough" the two simple sentences cannot be swapped around without changing the logical meaning of the statement. The word "because" is a subordinating conjunction, and in this example it serves to highlight a cause-and-effect relationship between two actions of the cat (meowing loud and obtaining food).

But cause-and-effect relationships are not the only kind of relationships that can be expressed by a complex sentence. Take for example the sentence "the man is a bachelor, because he is not married": the linguistic structure is similar but the idea is a little different: this is not knowledge of the laws (observable regularities) that govern the world, but rather of the meanings of the word "bachelor" and of the word "married", and the necessary (and somewhat trivial) relationship between them.

This is an example of the (somewhat disputed) philosophical distinction between “analytic” (or “a priori”) knowledge and “synthetic (or “a posteriori”) knowledge [99].

5.4.6 Simple Sentences in Deixiscript

Now we have a feeling for how propositions are represented in English. In Deixiscript, a “simple sentence” has: a (necessary) subject, a (necessary) predicate (we will come to this later) and an (optional) object. The choice was made in the current version of Deixiscript to limit the maximum number of “slots” of a verb in a simple sentence to only two (the subject and the object).

The language could be easily extended to support multiple (optional) complements in a simple sentence (early “versions” of the language had them indeed), but then the choice was made to avoid them, for two reasons: because they may be more confusing than helpful at this stage of development (there are many prepositions in English, so how many do we choose to support? Is “in” always a synonym of “on”? Is “on” a synonym of “over”?); and because binary relations (such as between a subject and a direct object) are already quite powerful and allow one to express a sufficiently large range of ideas [57].

From now on, we may occasionally speak of “simple sentences” and of “propositions” almost interchangeably in the context of Deixiscript. However, we must keep in mind that a proposition is really an abstract (mental) concept, and a simple sentence in English is usually seen as a feature of the (concrete) syntax of the English language, which includes word order and other (concrete) features of a physical string of text (or sound).

What we are really referring to when we use both of these two terms loosely is the abstract syntactic structure of the implementation of Deixiscript (the type of Abstract Syntax Tree) that corresponds to a simple sentence in English and a proposition (the meaning of that sentence).

Another thing that we must keep in mind is that a proposition is the meaning of any sentence (not just a simple sentence), hence the term "atomic proposition" would be even more appropriate when referring to the meaning of a simple sentence in Deixiscript.

In any case, there are two complementary ways to interpret a simple sentence, as we shall see, one related to "statements" (we will clarify what we mean by that term) and the other related to "questions" or "conditions".

5.4.7 Statements versus Questions

We think that natural language "statements" (i.e., "acritical" declarations of knowledge assumed to be true) and programming language "statements" (i.e., those syntactic constructs that are executed primarily for the side-effects they produce in the environment) are actually quite similar, not just in name, because (at least at a first approximation) they both cause the "recipient" to change its "internal state" in some way.

A programming statement (such as a variable assignment) primarily causes a change of state in the system, for instance by binding the name of a new variable to some value in a variable scope.

Analogously, a natural language statement in the indicative mood usually causes a person who hears it to change his/her beliefs. A person "Alice" who hears a statement made by another person "Bob" will change some of her beliefs.

Of course, she may not actually believe in the "content" of the statement (perhaps Bob is a liar, or a deluded individual), but she will certainly (at least) come to believe that Bob made the statement.

However, if Bob happens to be some kind of authoritative figure (a doctor, a professor, etc.) Alice may indeed change her beliefs according to the actual "content" of the statement made by Bob, accepting it (maybe just partially) as a true fact about the world.

On the other hand, a question is an expression of the desire to learn about a specific fact or facts in the world, and a condition represents a piece of knowledge whose truth value is not asserted, but rather just taken as hypothetical.

We think that questions (and conditions) are more akin to programming language expressions, because their primary effect on the recipient (barring rhetorical questions of course) is not that of modifying the recipient's beliefs about the world, but rather of eliciting a response from the interlocutor.

In natural language, this response can be a simple "yes" or "no" (a condition also typically "evaluates" to a "yes" or a "no"), or it can be a "pointer to a thing" (such as in a reply to a "what", a "where" or a "who" question), or it can be an explanation of some phenomenon or behavior (such as in response to a "why" question).

5.4.8 Statements versus Conditions

“C-like languages” (i.e., languages that have a C-like syntax, such as: C++, Java, JavaScript, etc.) tend to blur the distinction between an expression and a statement: they allow some expressions to have side effects and they treat some “statements” like expressions.

For example: function invocations (expressions) and procedure invocations (statements) can look (and sometimes even behave) similarly, and variable assignments (which are best thought of as statements) are usually allowed to return a value (the value of the right-hand side of the assignment).

This sometimes leads to the necessity of having special syntax to distinguish some constructs with side-effects from some “similar” constructs without them. For instance, since a variable assignment (which uses a “single equals”) is treated like an expression, it could appear within the condition of an if-statement, and hence there is a need to distinguish it syntactically from a comparison (which uses a “double equals”):

```
if (x == 1) do_something();
```

The one above is an example of a conditional statement in a C-like language. `x == 1` is a comparison, so the function `do_something()` will be invoked if and only if the value of variable `x` is equal to the integer `1`.

Let us consider:

```
if (x = 1) do_something();
```

This second statement is (de facto) *not* a conditional statement in a C-like language. The expression `x = 1` is an assignment. An assignment expression evaluates to the value of the right-hand side (i.e., to the integer `1`). Any number that is not equal to `0` is considered to be “truthy”. Hence, the function `do_something()` will always (unconditionally) be executed.

Some programming languages (e.g., Java) mitigate this source of errors by stipulating that only Booleans appear in the condition of an if-statement; in such languages, only Booleans can be “true/false”.

Some other programming languages completely lack this somewhat “tricky” distinction between the “two kinds of equal”, such as the “GW-BASIC” dialect of the BASIC (Beginners’ All-purpose Symbolic Instruction Code) programming language developed by Microsoft, and first released in 1983. In GW-BASIC, the same symbol (a single equal sign) is used both as an assignment operator and as a comparison operator (depending on the syntactic context of its usage) [100].

We think that this kind of distinction is also mostly absent from English (e.g., the simple sentence “it is alive” remains unchanged when embedded in the bigger complex sentence “if *it is alive*, then run away!”).

5.4.9 ASK/TELL Distinction

In Deixiscript, we want a sentence like “it is alive” to be interpreted either as a statement (i.e., “let it be known that it is alive”) or as a question or condition (i.e., “is it alive right now, yes or no?”).

These two different interpretations, which we will call “TELL” and “ASK”, respectively, are triggered by syntactic context. For instance, if a simple sentence is embedded within the condition part of a conditional statement (as we saw before), then it will be interpreted in “ASK” mood.

A simple sentence all by itself (e.g., “it is snowing”) is interpreted in “TELL” mood, unless ended by a question mark (e.g., “it is snowing?”). This syntax diverges from the syntax of a “real” question in English, which in this case would require swapping the verb “to be” with the dummy subject “it” (i.e., “is it snowing?”), but we think that the two forms are similar enough, and this issue could probably be fixed in the future by some slight additions to Deixiscript’s concrete syntax.

In general, almost all syntactic elements of Deixiscript can be interpreted as either “TELL” statements or “ASK” expressions.

However, concerning some syntactic elements, only one or the other kind of behavior makes any real sense; for instance, strings, numbers and Booleans are only meant to evaluate to themselves (they are constants after all), so they do not have a TELL mood.

The way this is technically handled is through the addition of an extra component to a Deixiscript simple sentence (and to the other AST types that need it): a Boolean attribute that we will call the “TELL flag”. Depending on its value, an AST object’s `eval()` method will behave in one way (TELL) or in the other (ASK). A brand-new copy (following the code’s functional approach) of an AST object can be made with a different value of the TELL flag.

5.4.10 Other Grammatical Moods

Both statements and questions in English are expressed in the same grammatical mood, known as the indicative mood. There are other grammatical moods in English, such as subjunctive (to express unreal or hypothetical situations) and imperative (to give orders).

We will not be needing an explicit subjunctive mood (as it is rarely used in modern English anyway), and we will be using a different kind of sentence all together (the "Order") to express a kind of "imperative mood", as we shall see in the next sections.

5.4.11 Defining Meaning

Simple sentences do not have any predefined meaning in the current version Deixiscript: just like functions in other programming languages, they must be defined before they can be used, this can be done through the "Definition" syntactic construct that looks a lot like a complex sentence in English (as we will see).

For instance, a Definition could look like this: "a tablecloth is red means the color = red", and this would define the meaning of the predicate "red" in relation to the subject "tablecloth", which could be very different from the meaning of the same predicate applied to some other kind of entity (e.g., "a guard is red means the political-affiliation = USSR").

Most other AST types have an intrinsic meaning which cannot be overridden/overloaded, such as arithmetic operators, logic operators, comparison operators, the equal sign (which works either as a comparison operator or as an assignment depending on the syntactic context). All of these operators have a fixed meaning (for the sake of simplicity).

As a side note, the copula (i.e., the verb “to be”) is *not* treated the same as the equal sign. The equal sign strictly compares identity, while the copula can be used with “various meanings” (with adjectives) as we will see.

5.5 Representing Entities

Until now, we have always generically talked about the “subject” and the “object” of a simple sentence, but it is now time to explain exactly what we mean by those terms.

5.5.1 Noun Phrases

In English, the subject and the object of a sentence are “noun phrases”. A phrase is a syntactic structure that (unlike a sentence) does not express a complete thought; in particular, a noun phrase is a phrase that performs the same function as a noun.

A general test for whether a part of a sentence counts as a noun phrase is to replace it with a pronoun and to see if the sentence still makes sense; for instance, in the sentence “the quick brown fox jumps over the lazy dog” there are two noun phrases: “the quick brown fox” and “the lazy dog”, and the sentence’s structure is equivalent to: “*it* jumps over the lazy dog” or to: “the quick brown fox jumps over *it*”, or even to: “*it* jumps over ***it***”.

A linguistic head (or nucleus) of a phrase is the part that determines the syntactic category of that phrase, and in the case of a noun phrase the head would be a noun (or any smaller noun phrase). In the noun phrase “the lazy dog”, the head is “dog”.

5.5.2 Implicit References

A key insight from the study of natural language is that people rarely use explicit references (proper nouns, IDs, numbers, etc.) even when talking about individual entities (as we saw in Section 3.3); they instead make use of the “type” of these individual entities (common nouns) leveraging a phenomenon known as the indexicality of language.

For instance, if a person refers to “the cat” when they are at home, versus “the cat” when they are hiking across a mountain range in the Americas (two different “contexts”), they may be referring to two very different individuals (a house cat vs a mountain lion, for example). But the phrase they may decide to use in both cases is the same: “the cat”.

A noun phrase can be of arbitrary length, and of arbitrary precision (and thus include/exclude a higher number of individuals); the most trivial example is given by a single noun all by itself (e.g., “cat”), but a noun phrase also typically includes articles, adjectives and even relative clauses with any arbitrary level of nesting, e.g., “the agile calico cat that leaped on top of my desk holding a fresh kill (which she wanted me to have as a gift) in its fangs”.

5.5.3 Deixiscript Noun Phrases

The kinds of noun phrases supported by Deixiscript are: constants (numbers, strings, Booleans and IDs), “implicit phrases”, “genitive phrases”, variables and pronouns.

5.5.3.1 Constants

Numbers, strings and Booleans work just like they do in any other programming language. IDs are there mainly for the system’s own benefit (we will discuss them in a further section when talking about the world model) and are not accessible to the user of the language.

5.5.3.2 Variables

Variables are single-letter placeholder names (such as “x”, “y” and “z”) that “match” any type (see Section 5.7 on syntactic matching) and they can be useful when writing some kinds of general Definitions, but we will not have much more to say about them.

5.5.3.3 Implicit Phrases

What we call an “implicit phrase” comprises a “head” and a list of adjectives. The head (or noun) is just a string representing a type (which does not have to be declared explicitly).

The (attributive) adjectives do not carry an intrinsic meaning, they are tied to the adjectives used in simple sentences that we discussed earlier. An adjective in a noun phrase is referred to as an “attributive” adjective, whereas an adjective in a sentence with a copula is referred to as a "predicative" adjective.

In Deixiscript there is an equivalence between the two kinds of adjectives: once the meaning of a predicative adjective is defined in a simple sentence (e.g., "the cat is calico, means…"), it can be used as an attributive adjective in a noun phrase (e.g., "the calico cat"). An attributive adjective can also be negated (e.g., "a non-calico cat").

When used inside of a sentence or by itself, an implicit phrase evaluates to (at most) a single ID, as we shall see later.

5.5.3.4 Genitives and Possession

What we call a "genitive phrase" is a noun phrase that refers to a property of an individual rather than to the individual itself. Syntactically, it is in the form: "an individual's property" or "x's y" (using English *'s* Saxon Genitive). It is also possible to implement an alternative syntactic form using the preposition "of".

We think that "possession" is an important part of how we model the world. All of the "useful work" that the system really does (everything it really boils down to) is setting the value of properties on the dictionary data structures that the system internally uses to represent "individuals". It is precisely because of this internal representation that the system is able to interface with the outside world (other programming languages, tools and libraries). We will come back to this idea later when discussing the Knowledge Base and World Model.

5.5.3.5 Pronouns

The last kind of noun phrase we mentioned is the pronoun. This is perhaps the most elusive kind of syntactic element. We take pronouns for granted in natural language, but they are actually not so easy to approximate (with a hundred percent accuracy) in an artificial language.

A personal pronoun (e.g., “I”, “you”, “he/she”, “it”, etc.) is a specific instance of a more general linguistic phenomenon known as “deixis”. The “deictic” words are highly context-dependent words whose referent depends entirely on the state of the speaker who uses them. Deixis can be spatial (in words such as “here” and “there”), temporal (in words like “yesterday”, “before”, “after”) or personal (in words such as the personal pronouns).

Deixiscript only supports one kind of pronoun (third person singular). Pronouns in Deixiscript are a little special for a reason: they are the only kind of noun phrase that needs some external context to evaluate.

We have already talked about how a simple sentence may be treated as a conditional expression or as a statement with side-effects depending on the surrounding (syntactic) context (e.g., the simple sentence “it is snowing” enclosed in the larger complex sentence “if *it is snowing* you put on a heavy coat”).

This is technically easy to achieve: the object that represents the larger syntactic structure (the complex sentence) obviously *knows* that its own conditional part has to be evaluated in “ASK” mood and not in “TELL” mood.

On the other hand, a pronoun that behaves somewhat “realistically” reserves the right to resolve to different values (to point to different things) based on the *meaning* of the surrounding context (the simple sentence that embeds the pronoun).

For instance, in the sentence “the cat saw a table, and it jumped on it” the first instance of the pronoun “it” obviously refers to “the cat” and the second refers to “the table”. This is obvious to us, because we live in a world were cats are the sort of entities that can jump on tables, and not vice versa.

Deixiscript supports a limited kind of context-dependent resolution of pronouns, based on the Short Term Memory (STM) of the interpreter and the stored Definitions of the simple sentences; we will discuss the STM later alongside the Knowledge Base.

5.5.3.6 Relative Clauses (not included)

Deixiscript does not support relative clauses, although a previous version of it did support them. The decision to drop their support and focus instead on other aspects of the noun phrase (e.g., adjectives) came from the difficulties encountered in their implementation.

A relative clause is introduced by a relativizer (such as “that”, “which”, “who”, etc.) followed by a sentence. In English, the prevalent strategy to refer back to the nucleus (or head) of the enclosing noun phrase is that of “gapping”, i.e., leaving a “gap” in the place where the head would have stood, if the relative clause had been independent.

For example, the noun phrase “the fish that the cat ate” has a relative clause (“the cat ate”) with an apparently missing direct object (*what* did the cat eat?). As an independent sentence, it would have looked like “the cat ate *the fish*”.

A possible implementation for this kind of relative clause involves substituting the head (in this example, “the fish”) to the “gap” within the relative clause. The problem is that one has to know whether the verb is transitive or not, to determine whether there is a gap in the sentence or not.

While this is not an insurmountable problem in principle (the transitivity of a verb can be deduced from its usage in the Definitions), we thought that this feature (relative clauses) was not too central to our goals.

In any case, the goal behind implementing noun phrases was to have a system to distinguish types from one another, to recognize some kinds of sub-type and compatibility relationships, and to pick out individuals from the world models based on their properties. For our basic prototyping purposes, one can get along well enough with the syntactic structures we have previously described (nouns, adjectives, genitives).

5.6 Knowledge Base (KB)

We have repeatedly mentioned the “Knowledge Base”, and in the following paragraphs we will discuss the details behind this important component of the system. The Knowledge Base contains all of the state of the Deixiscript interpreter at any point in time, which comprises of the World Model (WM), the lists of Definitions, Potentials and Orders (we will talk about these two in a next section), and the Short Term Memory (STM).

5.6.1 World Model (WM)

As we already said, the world is modelled as individual entities (or "individuals") and their associated properties. To the World Model, an "individual" is nothing more than a bundle of properties stored in a dictionary (associative array) data structure.

Anyone of these individuals (or bundles of properties) must contain at least one essential property that we call the "type". The world model is essentially a list of these individuals, and the index in the list of an individual is its "ID". For instance, we could have a world model like the following one (represented in JavaScript Object Notation (JSON)):

```
[{"type":"cat", "color":"red", "age":10}, {"type":"cat",
"claws":"sharp"}, {"type":"dog", "color":"brown"}]
```

This world model contains three individuals (two cats and one dog), and describes the property "color" of the first cat (`ID#0`) as "red" and the same property of the only dog (`ID#2`) as "brown". The second cat (`ID#1`) does not have a property "color" (the model does not have any information about it) but it has a property "claws" set to the value "sharp". The first cat (the red one, `ID#0`) also has a property "age" that is set to the value 10.

The "keys" are the names of the properties of the individuals ("type", "color", etc.) and they are strings. The values assigned to the properties can be strings, numbers or Boolean values. One might also imagine to set the value of a property to an ID value (which is distinct from the number type in Deixiscript).

Setting the value of a property to an ID type might be useful for those properties that involve a permanent bond (more stable than an ephemeral interaction during an Event) between two individuals.

5.6.1.1 Limitations of the World Model

Of course, this way of representing the world has its big limitations. For example, given that the value of a property can only be a scalar (string, number, Boolean, and maybe ID) it cannot hold more than one value at a time. For instance, a cat may not have two colors simultaneously (unless, of course, one decides to model that as a "color-one" and a "color-two" properties, which would then count as two distinct properties).

This limitation (ensuring that any given property has one single value at a time) is however useful for interfacing with the outside world (other programming languages and software tools) because they (the external tools) can more easily read values from the properties of individuals in the world model, for instance to redraw a graphical component of some user interface based on the state of the world model.

An external tool could also write to the world model, for instance to update the value of an input buffer after the user of a Deixiscript program enters some new input.

Another limitation put in place on the world model is the following: the world model must not contain (at any time) any pair of individuals that are undistinguishable from their properties alone (and the ID, which is just an index in the list, does *not* count as a property).

This property has to be enforced by a check that is made whenever a new individual is about to be created with some properties: the system checks whether there is already an individual with all of the mentioned properties in the world model, and displays an error in case it already does.

One might say that the system cannot “conceive” of any two “identical” individuals unless it (the system) is told of some actual (even slight) difference in their properties (and again, the ID does not count). The rationale behind this limitation is related to how the Short Term Memory (STM) works.

5.6.2 Short Term Memory (STM)

The Short Term Memory (STM) is a part of the Knowledge Base and its purpose is essentially to disambiguate the implicit references and pronouns that may be used by the programmer in a given context.

The STM behaves like a bounded queue (with some caveats, as we will see). The STM can hold up to N (we set N=4) noun phrases at any time (the size N of the STM should not be too big). This approach was inspired by the project Pegasus (discussed in Section 3.6.1.3).

The system tries updating the STM whenever a “noun phrase is used by the programmer”, but the STM may have to remain unchanged in some cases.

5.6.2.1 Example of STM working

We will illustrate the behavior of the STM with a simple example: suppose that there are two cats, one with color=red and the other with color=black (the two individuals can co-exist in the same world model because they are distinct by one property); suppose also that the STM is initially empty.

When we (the programmers) say “the cat”, the system displays an error. This is because there are currently two individuals in the world model to whom the noun phrase “the cat” applies; furthermore, the STM is initially empty and cannot provide any help (yet) at disambiguating this ambiguous phrase (“the cat”).

We can try being more specific. Suppose that we have already defined what it means for a cat to be red (as having the “color” property set to the value “red”), so we say: “the red cat”, using an attributive adjective to narrow down on the individual we would like to talk about.

Now the system does not display an error, because the noun phrase we used is precise enough to pick out one single entity unambiguously from the world model. The system also adds the noun phrase “the red cat” to the queue.

Now the STM contains the noun phrase “the red cat”. Suppose we *now* say “the cat”. The system does not display any error, because it is now able to use the STM to disambiguate, i.e., the system will assume that we are referring to “the red cat” in the STM. It is worth noting that, in this case, the system will *not* insert the ambiguous phrase we just used (“the cat”) into the STM, because it already contains a more precise one, i.e., “the red cat”.

Suppose we now say “the non-red cat” to the system. This will pick out the other cat in the world model, the one that does not have “red” as its color. The system will insert the new noun phrase “the non-red cat” into the queue; and, from now on, the ambiguous noun phrase “the cat” will be intended as “the non-red cat”.

One can then mention “the red cat” again and the STM will be updated accordingly. When the STM grows to its maximum length, the oldest element (noun phrase) will be removed from the queue to make space for the newest one.

5.6.2.2 One ID per Noun Phrase

A limitation of the latest version of Deixiscript is that a noun phrase can only “point to” a single individual at a time; in other words, a noun phrase may resolve at most to a single ID, given one state of the Knowledge Base. Early versions of Deixiscript did not have this limitation. This entailed that noun phrases had to specify a cardinality (to distinguish singular from plural). The automatic expansion of a sentence (while a useful feature in some cases, to carry out the same action over multiple individuals) introduced complexity to the code: the system had to check (for any sentence) if any of the noun phrases it contained would resolve to multiple IDs, and, if that was the case, “expand” the original sentence into multiple similar sentences each with a different ID.

5.6.2.3 STM and pronouns

The STM is also useful for the resolution of pronouns. The assumption here is that any pronoun must refer exclusively to some noun phrase that is currently in the STM. As we have said previously when talking about pronouns, a pronoun should be able to resolve differently based on the meaning of the sentence or phrase that surrounds it.

The simple technique we use to resolve pronouns is to try executing a sentence (or phrase) that contains a pronoun multiple times, each time after having substituted the pronoun with a different noun phrase taken from the STM (they are very few, owing to the STM's limited capacity).

When the sentence (or phrase) finally works (does not produce an error) then we stop, and we consider the pronoun to be resolved. Of course, it may be that none of the noun phrases in the STM succeeds at making sense of the sentence containing a pronoun; in that case, the system just displays an error, complaining that the pronoun is used ambiguously in that context.

5.7 Syntactic Matching

Many of the higher-level operations we have talked about (using simple sentences, using adjectives, searching for relevant Definitions, etc.) depend on the functioning of a more basic low-level operation that we will call "match".

Matching any two noun phrases or any two sentences (or, more generally, any two AST objects) is a purely syntactic operation that does not depend on the semantic context, i.e., it is completely independent of the state of the Knowledge Base.

5.7.1 The `match()` Function

Match is implemented as a function that takes two arguments, namely two Abstract Syntax Tree (AST) objects. The function `match()` is non-symmetric, i.e., the order of the arguments matters. The match function cannot be expected to return the same result when the two arguments are swapped.

When comparing noun phrases, one of the arguments can be thought of as the “super-type” and the other can be regarded as the “sub-type” of the comparison, to use some class-based programming jargon. For instance, when comparing (the abstract structure of) “the cat” to “the red cat”, `match()` will return a *positive result* if “the cat” is used as the first argument (in the “super-type” position) because “the red cat” is regarded as a special case of “the cat”.

Obviously, if the arguments are swapped and “the red cat” takes the argument position reserved to the “super-type”, the `match()` function returns a *negative result* (“the cat” is obviously *not* a special case of “the red cat”).

Used on noun phrases, the `match()` function behaves very much like the `instanceof` operator of some object-oriented languages. The `instanceof` operator produces a Boolean value based on the inheritance hierarchy of the operands. But what the match function actually returns is a mapping (an AST-to-AST dictionary), which just happens to be empty in case of a *negative result.*

The return type is a mapping, because the two arguments of the `match()` function are not (as we have hinted to) limited to being noun phrases. They can be simple sentences, compound sentences, etc. The `match()` function needs to return a mapping when comparing together two simple sentences, because its job is to determine if the subject (or object) of the more specific sentence can be substituted into the subject (or object) of the more general sentence.

Sometimes it is also useful to allow different AST types to compare together: for example, it makes sense to compare a “genitive phrase” (Section 5.5.3.4) to an “implicit phrase” (Section 5.5.3.3).

It may also make sense to compare a compound sentence (implemented as an “and/or expression”) to a simple sentence.

5.7.2 Definition Lookup

Syntactic matching plays a crucial role when the system needs to lookup the definition of an simple sentence. As we said, a simple sentence is the sort of AST that does not have an intrinsic meaning: it needs to be defined before it is used, and the Definitions are stored as a list in the Knowledge Base. Each Definition has two parts: a “definiendum” (the left-hand side, what has to be defined) and a “definition” (or “meaning”, or right-hand side).

When a simple sentence has to be executed, the system goes through the list of Definitions and attempts to match the simple sentence to a definiendum. If the end of the list is reached and no suitable match was found, then the system displays an error, telling the programmer that the verb or predicate they have attempted to use is not defined for the given kinds of arguments (it may be defined for others).

5.7.3 Argument Passing

If a suitable match is indeed found, then the search stops. The system takes the result of the match function (the dictionary or “mapping”) and “plugs it into” the right-hand side of the Definition.

To do this, a very simple function called `subst()` is invoked. The function `subst()` simply takes an “original AST” and an AST-to-AST mapping, and replaces any sub-AST that is included in the keys of the mapping by the values of the mapping, calling itself recursively.

When `subst()` is done, the new AST that it produces is executed, which is akin to running the body of a function. This argument passing strategy is similar to “pass by name”, which we discussed in Section 2.8.1.

5.7.4 Specificity and Matching

A problem with this Definition lookup procedure is this: suppose that the system needs to answer a question like “the amphibious fish is dead?”. Let us suppose that the Knowledge Base contains two Definitions for the predicate “is dead”, the first one in the list generically applies to any kind of fish (i.e., “a fish is dead means …”) and the second applies specifically to amphibious fish (i.e., “an *amphibious* fish is dead means …”).

Since an “amphibious fish” is a kind of “fish”, then the question we are trying to answer (“the amphibious fish is dead?”) will match the definiendum of the first (general) Definition (“a fish is dead means…”) which applies to a *generic* kind of fish. This means that the amphibious fish will be (wrongly) treated as any other regular fish.

The solution we adopted to this problem is to keep the list of Definitions sorted by descending "specificity" of the definiendums. To do this, it is possible to define a special "comparison function" to be used by the sorting algorithm.

This comparison function will return a positive number (+1) in case the first argument is more specific than the second, a negative number (-1) in case the opposite is true, and exactly zero in case the two arguments are completely equivalent (or completely unrelated, which makes no difference to the sorting logic).

The comparison function we described can be readily built from the match function we previously talked about. One simply needs to invoke match twice, swapping the arguments the second time, and comparing the two results as Booleans.

Sorting the list of Definitions by descending specificity of the definienda ensures that the most specific Definition will be found first, but only if a sentence that is specific enough is used.

The problem can persist if the noun phrase that is used is too generic for the actual shape of the object in the world model (e.g., "the fish" for "the amphibious fish"). It may be possible to overcome the problem for good by substituting ambiguous noun phrases by their more specific meaning in the STM *before* searching for a Definition.

For instance, if we ask the system the question "the fish is dead?", the system may substitute the noun phrase "the fish" with the more specific noun phrase "the amphibious fish" (because of what the STM contains) before even attempting to search for a definition of the predicate "is dead".

5.8 Orders and Planning

We have seen how it is possible to add Definitions that determine the effects of executing a simple sentence. This makes the execution of a simple sentence akin to a function invocation (and sometimes to a *procedure* call) in more traditional languages.

One must always keep in mind that any simple sentence may be subject to two interpretations ("ASK" mood and "TELL" mood) with the difference we discussed before: that the former (ASK) thinks of the simple sentence as an expression, i.e., it tries to verify if the condition it describes is true in the world model; and the latter (TELL) treats it as a statement, procedurally changing the world model according to what the sentence says it should look like.

Given the analogy we have made with procedures, and despite the actual "dual" nature (ASK versus TELL) of a Deixiscript AST, one could nonetheless conclude that Deixiscript is a kind of procedural (or imperative) language.

In a way this is true, as one can even define the meaning of a simple sentence as a sequence of steps, which can be achieved by joining together (through the `and` operator) multiple atomic statements (like simple sentences or variable assignments). The comma symbol (',') can also be used for the same purpose, and it behaves exactly like (indeed, it is aliased to) the `and` operator.

This however would overlook the last aspect of the language that we still need to discuss: automatic planning.

5.8.1 Facts versus Events

We will now introduce a distinction between two kinds of simple sentences. The two kinds are formally very similar (they are all represented by the same abstract syntax tree), but they have different syntax and semantics. We will call these two categories of simple sentences "Facts" and "Events".

We think of a Fact as just a regular proposition (a statement about how the world is at any particular point in time), and we instead think of an Event as an action that can be performed by an agent (the grammatical subject of the sentence, for simplicity).

Syntactically speaking, we will represent Facts as simple sentences with a copular verb (i.e., the verb "to be"). For instance, the sentence: "the fish is dead" falls under this category.

A Fact is a "static" state of affairs. A sentence representing a Fact describes the presence or absence of a particular situation "here and now" in the world model and always refers to the present tense (there are no other tenses in Deixiscript).

Events, on the other hand, will be represented by simple sentences with any verb other than the copula (i.e., other than the verb "to be"); for instance, "the player moves right".

An Event does not describe a static state of affairs, but rather a dynamic action (performable by an agent) that causes certain kinds of changes to the world model when it is performed by the agent.

As a side note, we should say that both the ASK and TELL moods make sense when applied to Facts; the system can "learn" (TELL) a fact, or it can "check" (ASK) a Fact. But the same distinction does not make much sense (in our current system) when applied to Events; the purpose behind performing an Event is purely to cause side-effects (TELL).

We already said that the abstract structure behind Facts and Events is the same; they both have a subject, an object and a “predicate”. In case of an Event this predicate is a non-copular verb (e.g., to eat, to drink, to run), whereas in case of a Fact this predicate is an adjective (e.g., red, dead, near).

Some “adjectives” such as “near” (we will treat it as an adjective) are binary predicates because they require an object (e.g., “the cat is near *the mat*”); they are akin, in this sense, to transitive verbs (such as the verb “to eat”).

5.8.2 Potentials

There are two kinds of simple sentences, as we have seen; so how does the system tell a “Fact” from an “Event”? The Potential AST type was introduced just for this: to mark certain kinds of simple sentences as a *potential* action for a certain kind of agent, under a certain kind of circumstance.

A Potential specifies the condition under which a kind of Event can occur. A Potential also specifies the duration of this kind of Event, which is useful for the purposes of time-bounded planning and simulation, as we will see later. Syntactically, a Potential is (like a Definition) a kind of complex sentence.

Facts, Events and Potentials are the three basic ingredients for Deixiscript’s limited notion of automated planning. The fourth ingredient is the “Order”. An Order is a syntactic structure that consists of a noun phrase that represents an agent and of an expression (such as a Fact) that represents a goal to be accomplished by the aforementioned agent. Syntactically speaking, an Order will also end up looking like a complex sentence of sorts.

5.8.3 Search Heuristic

The search heuristic tries to find a finite sequence of Events (actions) that can be performed by an agent and that will result in (or at least get the agent "closer" to) the accomplishment of a goal.

This search relies on the presence in the Knowledge Base of: (1) Definitions that detail the effects of an Event alongside the kind of doer, and also on (2) the presence of Potentials that specify the possibility of an Event under a given circumstance and its duration.

5.8.4 The `plan()` function

The `plan()` function is responsible for carrying out this kind of heuristic search when needed; `plan()` takes an Order (agent and goal), a Knowledge Base, and a maximum duration (in seconds) of the plan to be found.

Searching for the plan's steps works as follows:

(1) The goal expression is applied to the Knowledge Base (TELL) to produce a new "target" Knowledge Base that represents the world "as it should be".

(2) A numerical error term is computed, which represents the difference between the desired state of the world model (contained in the target

Knowledge Base) and the current state of the world model (contained in the old, original Knowledge Base).

(3) All of the actions that the agent could perform are retrieved. For simplicity, it is assumed that they coincide with the Potentials where the (kind of the) agent appears as a subject.

(4) Each of these (possibly useful) actions is tried separately for effectiveness. An action is applied to the old Knowledge Base (TELL) and the error term is recomputed. If the new error term is less than the old one, then the action is deemed “useful”, else it is not.

(5) If the list of “useful” actions is empty, then the function returns an error value (because the possibility of generating any plan is precluded).

(6) If the *precondition* for any of the “useful” actions is still false, then the *precondition* is issued as an intermediate order (`plan()` is a recursive function).

(7) Otherwise, every “useful” action can already be performed by the agent. The list of steps, the duration of the plan, and the Knowledge Base are all updated.

The `plan()` function also has to foresee two special (but important) cases: (1) if the goal is already accomplished then the function returns the list of accumulated steps, this is the base case of the recursive algorithm; and (2) if the maximum duration set for the plan is ever exceeded, then the function also returns the accumulated list of steps, even if they do not accomplish the goal in this case.

The following pseudo-code better summarizes the operation of the `plan()` function, omitting the technical details and some of the error handling:

```
if goal is accomplished then: return steps.
if duration > max_duration then: return steps.

compute target kb (by TELLing goal to kb).
compute error term between target kb and kb.

for each potential in kb:

  action = potential.action.
  compute "new kb" (by TELLing action to kb).
  compute new error term between target kb and "new kb".

  if new error term < old error term then:
    the potential is useful.

if there are no useful potentials then: error out.

if for some useful potentials precondition is still false in kb
then:
  try planning for the preconditions.
  then resume planning for the terminal (main) goal.
else:
  add each potential.action to the list of steps.
  then resume planning for the terminal goal.
```

5.8.5 Error Term ("Cost Function")

The reader will note that we have skipped over a few details, specifically the details of how the "error term" between two Knowledge Bases (between the respective two World Models) is computed.

It is to be recalled that a World Model (for us) just means a list of dictionary objects containing key-value pairs. Therefore, the error term is computed by cycling through each individual in the World Model and each key in an individual.

The error term is the sum of the absolute values of the differences between the values of each key in target World Model versus its value in the current World Model:

$$\sum_{i=0,k=0}^{i=\#\text{individuals},k=\#\text{keys}} \left|\text{WM}_{\text{target}}[i][k] - \text{WM}_{\text{current}}[i][k]\right|$$

5.8.6 Offline versus Online Planning

As can be evinced, there are two ways in which the `plan()` function can be used. One way is analogous to what Hector Levesque calls "offline execution" in his book "Programming Cognitive Robots" [91] (which we mentioned in Section 4.7.3.2), and the other is analogous to what he calls "online execution".

In offline execution, all of the steps of the plan (the whole plan) are computed in advance, whereas in online execution only one step (the next) is computed at a time. Actually, in our case, more than one step is computed even in online mode, but the computation stops as soon as the maximum tolerated duration is reached (the duration of the sequenced actions, not of the time taken to compute them).

In our prototype, offline execution is mostly intended for debugging purposes, to test whether a plan can be successfully computed, while online execution is useful to graphically simulate the interaction between agents.

5.8.7 The Main Loop

The kind of graphical simulations we have tried out involve a 2D world, where the individuals that have an x and a y coordinate as properties are displayed as colored points on a 2D Cartesian graph (we use the Matplotlib Python library to plot the graphs).

Before the simulation starts, "time is frozen". As soon as the simulation starts, the individuals in the World Model that have been given orders start behaving accordingly. Two numbers M and N are associated with the loop.

The number N is the period of the loop, it determines the frequency with which the graphics are updated. Decreasing N means increasing the frame rate of the simulation; for instance, setting N to 100 milliseconds (0.1 seconds) means the frame rate will be $\frac{1}{0.1} = 10$ Frames Per Second (FPS).

M is the maximum tolerated planning time, it is used as a parameter for the `plan()` function which uses it as an upper bound for the duration of a computed partial plan. M does not represent an objective time interval in seconds, because the simulation can be sped up and down by tweaking the number N. However, M is important because setting a higher value of M will allow the individuals to execute longer partial plans during a single iteration of the main loop, and vice versa for lower values of M.

To summarize: each partial plan should not exceed in “duration” the number M; and after the partial plans are computed, they are executed, the graphics are updated, and the loop waits for N (actual) milliseconds before proceeding to the next iteration.

The main loop is approximately described by the following pseudocode:

```
while true:
  for each order:
    steps=plan(order, kb, M)
    tell(steps, kb)

  for each individual in kb.world_model:
    draw(individual)

  wait(N)
```

5.8.8 REPL

The simulation can be started from the interpreter's custom Read Eval Print Loop (REPL) using the `:start-simulation` metacommand. Also within the REPL, the programmer can load a text file that contains Deixiscript source code using the `:load` metacommand (which takes the file path as an argument); or he/she can directly "inject" Deixiscript code into the environment by simply writing it to the command line and hitting enter. The programmer can also go back to a previous state of the Knowledge Base (actually, to a previous Knowledge Base) using the `:undo` metacommand.

5.9 Concrete Syntax

Now that we have painted a picture of what each part of the Deixiscript language is supposed to do, we wish to take a step back and look at the whole from a different perspective.

We have sometimes hinted at the concrete representation that the abstract syntactic structures would take; for instance, we have said that Definitions, Potentials and Orders would end up looking like complex sentences. We have also said that (in general) the syntax would draw inspiration from English. However, we will say that it should also include some mathematical symbols (like most other programming languages) for the sake of convenience.

5.9.1 Backus-Naur Form (BNF)

We will now present (in a slightly more formal fashion) the concrete syntax of the Deixiscript language, and to do so we will use a dialect of the popular Backus-Naur Form (BNF) metalanguage for syntax description; we have discussed BNF in Section 2.8, when talking about ALGOL 60.

The following presentation will omit some of the more technical details for the sake of clarity. The actual code that generates the parser is written in Lark's own dialect of BNF (Lark, as we said at the beginning of this chapter, is the parsing toolkit for Python that we are using), and contains some extra caveats and technicalities compared to the following simplified one.

5.9.1.1 Program

Firstly, any Deixiscript program contains at least a single statement, or more statements separated by a dot. We will express this in the equivalent BNF-like pseudo-code:

```
<program> := <statement> | (<program> "." <statement>)
```

5.9.1.2 Statement

A "statement" can be any of the following syntactic structures:

```
<statement> := <expression> | <definition> | <potential> |
<order> | <question> |<repetition> | <existential-quantifier>
```

5.9.1.3 Expression

An “expression” is either a noun, or a simple sentence, or a binary expression:

```
<expression> := <noun-phrase> | <simple-sentence> | <binary-expression>
```

5.9.1.4 Question

A “question” is just a statement followed by a question mark (there is definitely room for improvement on this rule, as we have already observed in a previous section):

```
<question> := <statement> "?"
```

5.9.1.5 Simple Sentence

A simple sentence is a fact or an event; we wish to note that the distinction here is a purely syntactical one, i.e., after an event or fact is parsed it is transformed into the same simple sentence abstract tree:

```
<simple-sentence> := <event> | <fact>
```

An event is basically an English simple sentence with only a subject and an optional object (the question mark after the second occurrence of the noun indicates that it is optional):

```
<event> := <noun-phrase> <verb> <noun-phrase>?
```

A fact, instead, looks like a simple sentence with a copular verb (i.e., the verb “to be” in English). The first noun phrase is the subject, the second is the predicative adjective, and the third is the (optional) object:

```
<fact> := <noun-phrase> <copula> <noun-phrase> noun-phrase>?
```

5.9.1.6 Binary Expression

A binary expression is formed of two smaller expressions separated by a binary operator. The binary operator can be a logical operator ("and", "or"), a comparison operator (">", "<", etc.), an arithmetical operator ("+", "-", etc.) or the equal sign ("="), which (as already said), can be interpreted as an assignment or as an equality comparison depending on the surrounding syntactic context.

The BNF code below shows the binary expression as a single production rule, but in practice it would be split into several similar-looking rules (logic, arithmetic, comparison, etc.) for the purpose of defining operator precedence:

```
<binary-expression> := <expression> <binary-operator> <expression>
```

5.9.1.7 Definition

A definition is a simple sentence followed by the harcoded verb "to mean", followed by any expression (the "meaning"):

```
<definition> := <simple-sentence> "means" <expression>
```

5.9.1.8 Potential

A potential is a noun followed by the hardcoded modal verb "can", followed by any (non-copular) verb, in turn followed by another (this time optional) noun (the direct object). There are two further optional parts of a potential: a duration in “seconds” and a condition (in no fixed order). If the condition expression (introduced by an “if”) is not explicitly stated, it is assumed to be equal to the Boolean value “true”, i.e., the action that the potential describes could be carried out unconditionally (at will, without constraints) by the relevant agent.

```
<potential> := <noun-phrase> "can" <verb> <noun-phrase>? ["("
<number> "seconds" ")"] ["if" <expression>]
```

5.9.1.9 Order

An Order is composed of a noun (the agent who receives the order), the hardcoded verb phrase “should ensure” and an expression (the agent’s “goal”).

```
<order> := <noun-phrase> "should ensure" <expression>
```

5.9.1.10 Repetition

A repetition statement is not really necessary (and we had not really mentioned it before), but we decided to add it just for convenience; it consists of a simple sentence followed by a number, in turn followed by the hardcoded word “times”. A repetition, exactly as the name says, repeats an action a certain number of times, like in a loop.

```
<repetition> := <simple-sentence> <number> "times"
```

5.9.1.11 Existential Quantifier

The existential quantifier consists of the hardcoded string “there is” followed by a noun phrase:

`<existential-quantifier> := "there is" <noun-phrase>`

The existential quantifier supports both moods (ASK and TELL). In the case of ASK, it will return true if an individual corresponding to the noun phrase's description exists. In the case of TELL, it will attempt creating such an individual in the world model (acting like a "constructor" of sorts).

5.9.1.12 Noun Phrase

A noun phrase can be any of the following things:

`<noun-phrase> := <constant> | <noun> | <article-phrase> | <genitive-phrase> | <attributive-phrase> | <pronoun> | <variable> | <number> | <parenthesized-phrase>`

Constants are numbers, strings or Booleans; the ID type is not shown here because in principle it should not be accessible to the programmer (the ID of an individual is supposed to be used internally by the system), though the system could be easily extended to include a syntax for an "ID literal" which could be useful for debugging purposes:

5.9.1.13 Constant

`<constant> := <number> | <string> | <boolean>`

A `<noun>` is just a simple, basic English noun (actually any "non-verb word") among the ones that the parser recognizes. By itself, it is actually just treated as a string, and could in fact be considered a constant.

5.9.1.14 Article Phrase

A “real” noun (i.e., an implicit reference) is preceded by an English article (definite or indefinite); this is the kind of noun that will resolve to an ID when evaluated by the system:

```
<article-phrase> := <article> <noun-phrase>
```

5.9.1.15 Attributive Phrase

An “attributive phrase” adds an adjective to that kind of noun phrase. The adjective itself is another single noun, and can be preceded by a negative particle (“non”):

```
<attributive-phrase> := "non-"? <noun> <noun-phrase>
```

5.9.1.16 Genitive Phrase

A “genitive phrase” contains a noun phrase followed by a genitive particle (“apostrophe s”) and a simple noun.

```
<genitive-phrase> := <noun-phrase> "'s" <noun>
```

5.9.1.17 Parenthesized Phrase

A “parenthesized phrase” is also a noun phrase, and it consists of an expression enclosed by parentheses. It can be useful to better specify the order of evaluation in mathematical formulas and the like:

```
<parenthesized-phrase> := "(" <expression> ")"
```

5.9.2 AST Transformations

As we have said, we use the Lark parsing toolkit for Python to parse a concrete syntax similar to the one we have just described. Parsing is just the process of converting a linear (monodimensional) representation of language (i.e., written text, or even spoken sound) into a bidimensional, tree-like structure that clearly reflects the hierarchical relationship between the expression's constituents.

Parsing, however, is just the first step of the process of obtaining an Abstract Syntax Tree (AST). After an initial (intermediate) "parse tree" is generated, this has to be converted (transformed) into an AST, which is a more convenient and easier to work with representation of the latter, as it leaves out many of the unimportant details.

These "unimportant details" may specify whether the user took advantage of the "sugared" version of a construct or not; syntax sugar is a more appealing (terser, more expressive, easier to read) syntax that is usually implemented on top of a less appealing (more verbose, less expressive, harder to read) one.

Another example of "unimportant detail" may be the presence or absence of parentheses in an expression, which outlive their usefulness as soon as the syntax tree has been built with the correct (user-intended) precedence of operators.

Lark provides some useful facilities to further transform the parse trees it generates into abstract syntax trees. The "Transformer" class it provides (completely unrelated to the "Transformer" neural network architecture in AI) performs a bottom-up traversal of the parse tree, allowing to define some further logic that converts the parse tree into a proper abstract syntax tree.

5.10 Example Programs

We will now discuss three short example programs that showcase some of Deixiscript's capabilities, as well as some of its current weaknesses. We shall have more to say about those weaknesses and some proposed solutions in the next chapter, about the possible improvements to the language.

For the sake of convenience, we will assume that the following two Definitions have already been loaded (or typed) into the interactive environment, and are always available:

```
x's y increments means: x's y = x's y + 1.0.
x's y decrements means: x's y = x's y - 1.0.
```

5.10.1 1. Envelope

The first example demonstrates the existential quantifier's usage as a "constructor" with the following code:

```
an envelope is sealed means the state = closed.
an envelope is red means the color = red.
there is a sealed red envelope.
```

Two rules are introduced that apply to envelopes: the first defines what it means for an envelope to be "sealed" and the second defines what it means for it to be “red”. An envelope is then created with these two properties, and the resulting world model will contain an envelope that has a “state” property set to “closed” and a “color” property set to “red”.

5.10.2 2. Pronouns

The second example showcases the functioning of pronouns in Deixiscript. The following is the code:

```
a player moves right means: the x-coord increments.
a door opens means: the state = open.

there is a player.
the player's x-coord = 0.0.
there is a door.
he moves right and it opens.
```

The previous code defines the meaning of two kinds of actions (Events), one of which applies to a player, and the other applies to a door. In the sentence `he moves right and it opens`, the first pronoun ("he") resolves to the player because "he" is used as the subject of the verb "to move", whereas the second pronoun ("it") resolves to the door because "it" is used as the subject of the verb "to open".

As a side note: as of now there is no distinction between the pronouns (between the strings) "he" and "it", they are just synonyms to the interpreter. The example would have worked the same if those two pronouns had been swapped.

The world model's state after the execution of that last sentence will be one where the door has the property "state" set to the value "open", and the player has the property "x-coord" set to the numerical value 1.0.

5.10.3 3. Fish

The third example showcases revisable rules, by defining a general rule and a specific one, and checking that they correctly apply to the relevant individuals.

```
a fish is amphibious means the kind = amphibious.
a fish is dead means the location != water.
an amphibious fish is dead means the health <= 0.0.

there is a fish.
there is an amphibious fish.
the amphibious fish's health = 10.0.
the non-amphibious fish's location = land.
```

Here we are defining three rules. The first is just a "dummy rule" to enable us to talk of the category of "amphibious" fish; the system needs a way (i.e., a property) to recognize an amphibious fish in the world model, and we could not think of anything nicer than just a somewhat boring "kind = amphibious" key/value pair.

The second rule defines "what it means to be dead for a generic fish" as being a fish out of the water (`location != water`). The third rule overrules the second by defining the death of an amphibious fish in other terms, in terms of a new "health" attribute having a value less than or equal to 0.

A fish and an amphibious fish were created. The amphibious fish's health was set to a positive number and the (normal) fish's location was set to "land". Now, if we asked the interpreter the following question: `the amphibious fish is dead?` it would return a `false`, and if we asked it whether `the non-amphibious fish is dead?`, it would return a `true`.

Of course, changing the values of their attributes (e.g., setting the normal fish's location to "water") would change the values of the answers to the previous questions.

5.10.4 4. Player/Enemy

The fourth example concerns the automated planning of a (very simple) strategy that (only) one of the two individuals will "carry out" to fulfil its goal. The code is the following:

```
an enemy can hit a player, if the enemy is near the player.

an enemy is near a player means:
    the enemy's x-coord = the player's x-coord,
    the enemy's y-coord = the player's y-coord.

an enemy hits the player means:
    the player's health decrements.

a player is dead, means the health = 0.0.

an enemy moves right means the x-coord increments.
an enemy moves left means the x-coord decrements.
an enemy moves down means the y-coord increments.
an enemy moves up means the y-coord decrements.
an enemy can move right.
an enemy can move left.
an enemy can move up.
an enemy can move down.

there is an enemy.
there is a player.
the player's health = 400.0.
the player's x-coord = 350.0.
the enemy's x-coord = 150.0.
the player's y-coord = 300.0.
the enemy's y-coord = 100.0.
the enemy's color = red.

the enemy should ensure the player is dead.
```

The program declares several capabilities of an "enemy". An enemy can move in a variety of directions (left, right, up, down) at will; an enemy can also "hit" a player, but only if it is "near the player". The interpreter is told what it means for the enemy to be near the player (in terms of x and y coordinates) and what it means for it to hit the player (the player's health decreases). The interpreter is also told what it means for a player to be dead (in terms of its "health"). Then the enemy is given the order to "ensure the player is dead".

One thing to note, unfortunately, is that this code is quite verbose and there seem to be a lot of unnecessary repetitions; we will try addressing this problem in the next chapter.

The overall effect of the program, when run in simulation mood, is to show a static black dot (the player), and a moving red dot (the enemy) that attempts to reach the black dot, changing direction if necessary (for example when the position of the player is changed by modifying its x-coord or y-coord attributes through the REPL). Once the enemy reaches the player, it will "hit" it until the player's health becomes zero. At that point (when the player is "dead"), if it changes position again, the enemy will not move; unless of course the health of the player is incremented through the REPL, which will cause the enemy to "reawaken" and resume "chasing" the player.

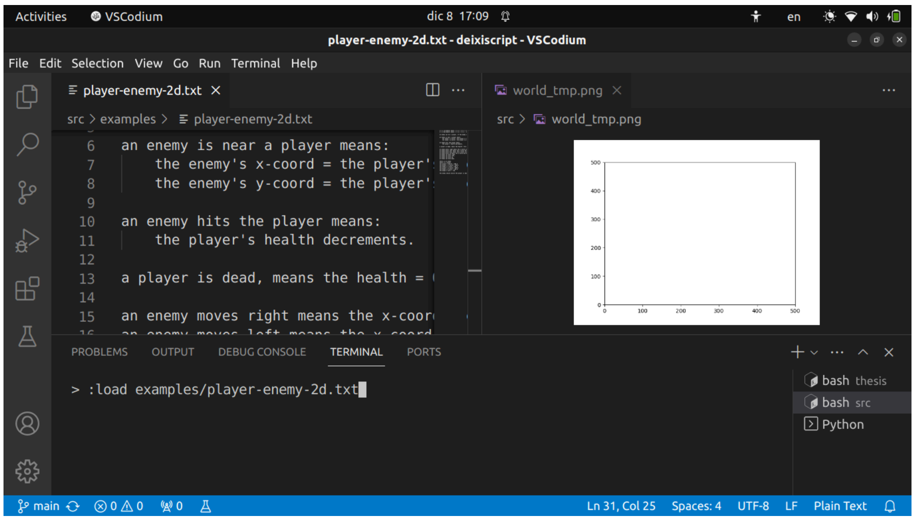
This behavior of the enemy is achieved without telling the enemy explicitly what to do, as it is rather the `plan()` function (Section 5.8.4) under the hood that computes a strategy for the enemy.

A strategy is expressed in terms of the actions that are available to the enemy. For instance, `the enemy moves right, the enemy moves up, the enemy hits the player, the enemy hits the player`, would work (for example) if the enemy was at position (1,2) on the Cartesian plane and the player was at position (2,3) and had 2 health points.

Of course, as we have said before, what is actually computed during a simulation session is generally a partial plan, that only gets the enemy closer to accomplishing the goal (and, in this case, this literally means getting *closer* to the player).

Another thing to note about the program is that the way the rule for the predicate "near" is defined introduces a problem that we will also discuss (also proposing a tentative solution for it) in the next chapter.

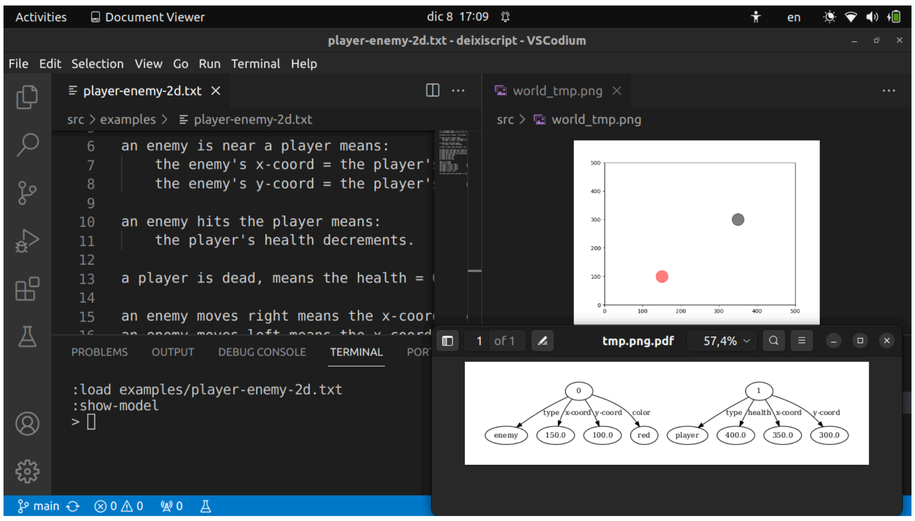
Figures 2-5 show four commented screenshots that illustrate the execution of this program in the interactive environment (REPL).


Activities
VSCodium
dic 8 17:09
en
player-enemy-2d.txt - deixiscript - VSCodium
File Edit Selection View Go Run Terminal Help
player-enemy-2d.txt
src > examples > player-enemy-2d.txt
world_tmp.png
src > world_tmp.png
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
bash thesis
bash src
Python
main
Ln 31, Col 25 Spaces: 4 UTF-8 LF Plain Text

```
an enemy is near a player means:
    the enemy's x-coord = the player'
    the enemy's y-coord = the player'

an enemy hits the player means:
    the player's health decrements.

a player is dead, means the health = (

an enemy moves right means the x-coor
an enemy moves left means the x-coord
```

```
> :load examples/player-enemy-2d.txt
```

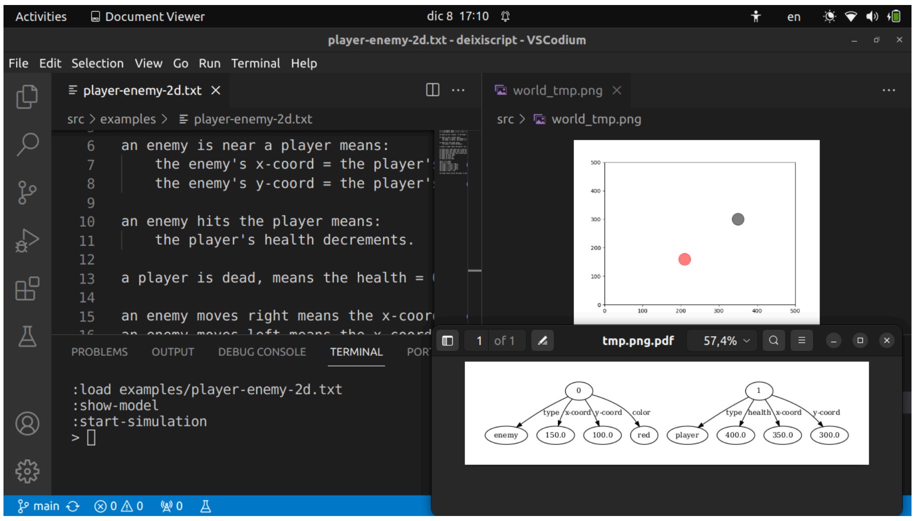
Figure 2: Before loading the program into the interactive environment, the cartesian graph is empty.



```
an enemy is near a player means:
    the enemy's x-coord = the player'
    the enemy's y-coord = the player'

an enemy hits the player means:
    the player's health decrements.

a player is dead, means the health = (

an enemy moves right means the x-coor
an enemy moves left means the x-coord
```

```
:load examples/player-enemy-2d.txt
:show-model
> 
```

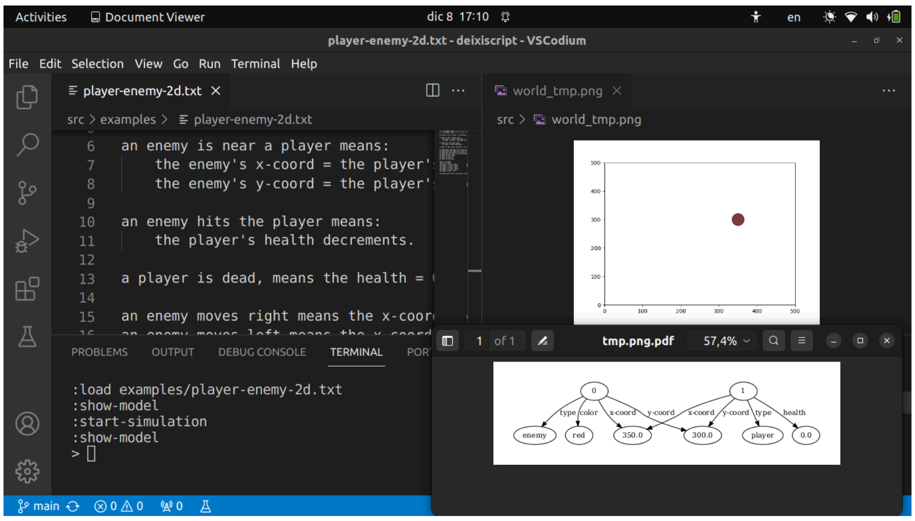
Figure 3: When the program is loaded, two colored dots appear on the graph. We can also inspect the state of the world model with the appropriate metacommand. As we can see, the player initially has a positive health and is located away from the enemy.



```
an enemy is near a player means:
    the enemy's x-coord = the player'
    the enemy's y-coord = the player'

an enemy hits the player means:
    the player's health decrements.

a player is dead, means the health = (

an enemy moves right means the x-coor
```

```
:load examples/player-enemy-2d.txt
:show-model
:start-simulation
> 
```

Figure 4: When the simulation is started, the red dot (representing the enemy) begins moving toward the black dot (representing the player).



```
an enemy is near a player means:
    the enemy's x-coord = the player'
    the enemy's y-coord = the player'

an enemy hits the player means:
    the player's health decrements.

a player is dead, means the health = (

an enemy moves right means the x-coor
```

```
:load examples/player-enemy-2d.txt
:show-model
:start-simulation
:show-model
> 
```

Figure 5: After a while, the enemy reaches the player and begins 'hitting' it, decreasing its health. We can see here the final stage of the process: the enemy's location is the same as the player's location, and the player's health is zero.

5.10.5 5. Player/Enemy/Defender

The last example is an extension of the fourth, it demonstrates automated planning in the presence of two individuals with different (clashing) goals. In this example a new individual (the "defender") is added to the mix, who will oppose the enemy.

Unfortunately, ordering the defender to ensure that the player stays alive (i.e, player's life > 0) will not work in the present version of Deixiscript. This is due to the simplicity (and limitation) of the current plan-finding heuristic (described in Section 5.8.4).

Hence, to achieve the same effect, we will instead have to order the defender to "ensure the enemy is dead" (thus implicitly *defending* the player).

To achieve this, we will have to add some more rules to the program we already had (and modify some existing ones), the resulting code is the following:

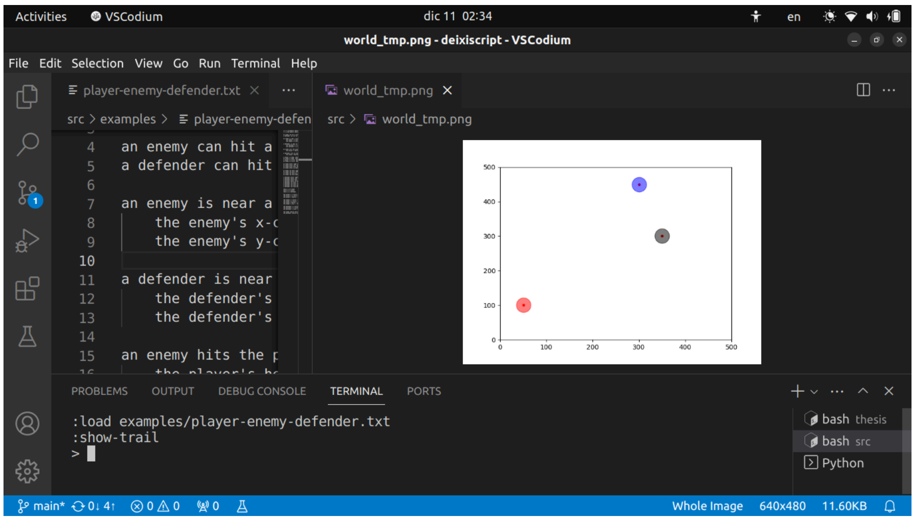
```
an enemy can hit a player, if the enemy is near the player.
a defender can hit an enemy, if the defender is near the enemy.

an enemy is near a player means:
    the enemy's x-coord = the player's x-coord,
    the enemy's y-coord = the player's y-coord.

a defender is near an enemy means:
    the defender's x-coord = the enemy's x-coord,
    the defender's y-coord = the enemy's y-coord.

an enemy hits the player means:
    the player's health decrements.

a defender hits an enemy means:
    the enemy's health decrements.

a player is dead, means the health = 0.0.
an enemy is dead, means the health = 0.0.
a player is alive, means the health > 0.0.
an enemy is alive, means the health > 0.0.

an enemy moves right means the x-coord increments.
an enemy moves left means the x-coord decrements.
an enemy moves down means the y-coord increments.
an enemy moves up means the y-coord decrements.

an enemy can move right if the enemy is alive (0.2 seconds).
an enemy can move left if the enemy is alive (0.2 seconds).
an enemy can move up if the enemy is alive (0.2 seconds).
an enemy can move down if the enemy is alive (0.2 seconds).

a defender moves right means the x-coord increments.
a defender moves left means the x-coord decrements.
a defender moves down means the y-coord increments.
a defender moves up means the y-coord decrements.
a defender can move right.
a defender can move left.
a defender can move up.
a defender can move down.

there is an enemy.
there is a player.
there is a defender.
```

```
the player's x-coord = 350.0.
the player's y-coord = 300.0.
the player's health = 400.0.

the enemy's x-coord = 50.0.
the enemy's y-coord = 100.0.
the enemy's health = 10.0.
the enemy's color = red.

the defender's x-coord = 300.0.
the defender's y-coord = 450.0.
the defender's color = blue.

the enemy should ensure the player is dead.
the defender should ensure the enemy is dead.
```

We essentially had to add rules to make sure that the defender could move and hit the enemy (just like the enemy can hit the player). We also modified the enemy's potentials to move: now the enemy can only move if it is alive (health > 0), this is to ensure that it (the enemy) will stop "dead in its tracks" when it is "killed" by the defender. Obviously, the code now has to declare an initial health for the enemy too, not just for the player.

The enemy was also made a little slower than the defender, by specifying that the time taken for it to move in any direction is 0.2 "seconds", which is more than the (default) of 0.1 "seconds" of the potentials where the Event's duration is undeclared.

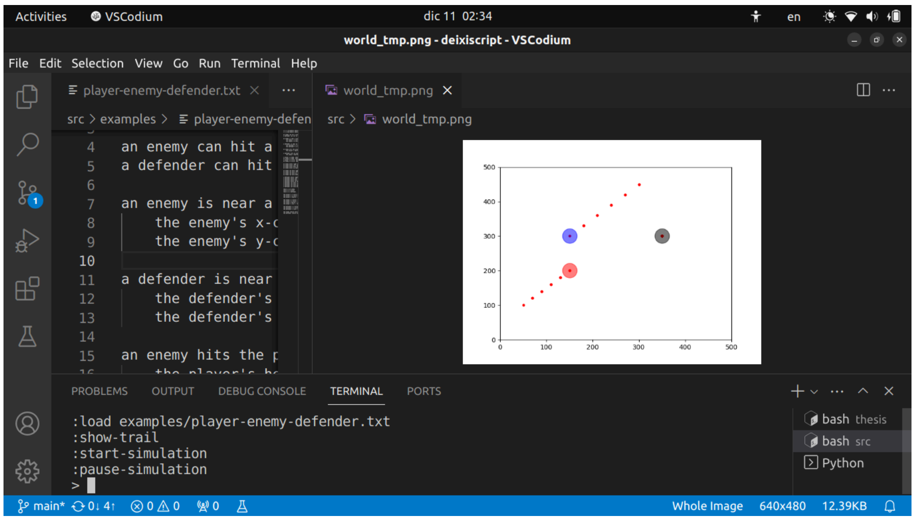
Figures 6-10 show five more commented screenshots that illustrate the execution of this program in the REPL.


Activities
VSCodium
dic 11 02:34
world_tmp.png - deixiscript - VSCodium
File Edit Selection View Go Run Terminal Help
player-enemy-defender.txt
world_tmp.png
src > examples > player-enemy-defen
src > world_tmp.png
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
bash thesis
bash src
Python
main* 0↓ 4↑ 0 0 0
Whole Image 640x480 11.60KB

```
an enemy can hit a
a defender can hit

an enemy is near a
    the enemy's x-c
    the enemy's y-c

a defender is near
    the defender's
    the defender's

an enemy hits the p
```

```
:load examples/player-enemy-defender.txt
:show-trail
> 
```

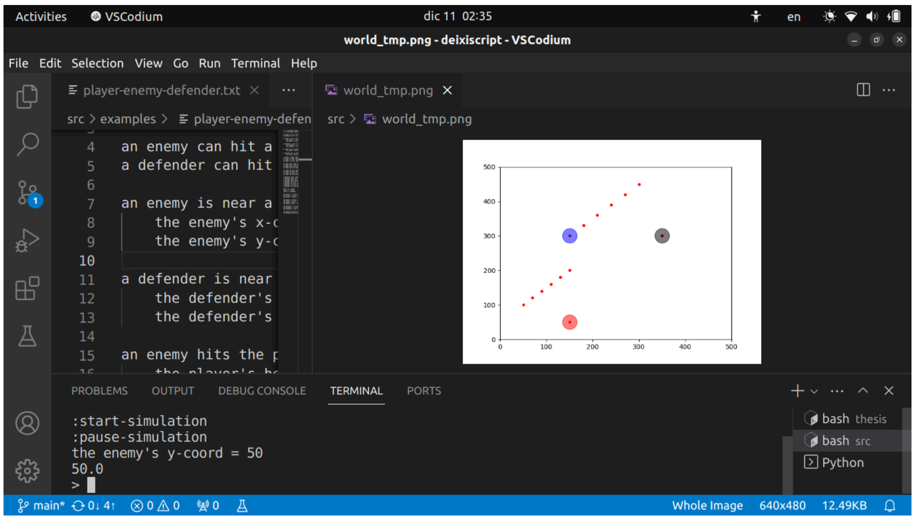
Figure 6: The program is loaded into the interactive environment. The graph shows three points with different colors. As before, the red point is the enemy and the black point is the player. A blue points appears as well, representing the defender. For the sake of better illustrating the paths taken by the moving points on the graph, we will execute the show-trail metacommand, which will plot some red points in any spots where the individuals have been.


Activities
VSCodium
dic 11 02:34
world_tmp.png - deixiscript - VSCodium
File Edit Selection View Go Run Terminal Help
player-enemy-defender.txt
world_tmp.png
src > examples > player-enemy-defen
src > world_tmp.png
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
bash thesis
bash src
Python
main* 0↓ 4↑ 0 0 0
Whole Image 640x480 12.39KB

```
an enemy can hit a
a defender can hit

an enemy is near a
    the enemy's x-c
    the enemy's y-c

a defender is near
    the defender's
    the defender's

an enemy hits the p
```

```
:load examples/player-enemy-defender.txt
:show-trail
:start-simulation
:pause-simulation
> 
```

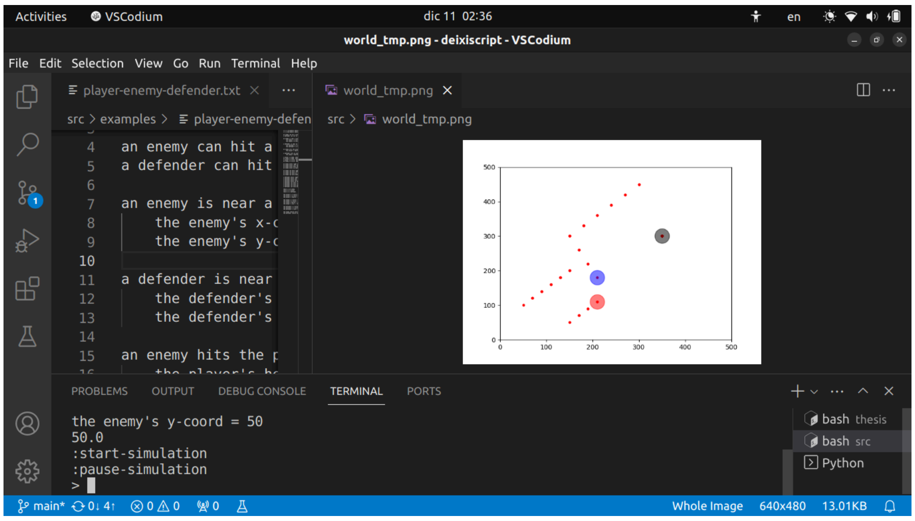
Figure 7: As soon as the simulation is started, the enemy (red) will try approaching the player, and the defender (blue) will in turn move towards the enemy. We have currently put the simulation on hold using the pause-simulation metacommand.


Activities
VSCodium
dic 11 02:35
world_tmp.png - deixiscript - VSCodium
File Edit Selection View Go Run Terminal Help
player-enemy-defender.txt
world_tmp.png
src > examples > player-enemy-defen
src > world_tmp.png
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
bash thesis
bash src
Python
main* 0↓ 4↑ 0 0 0
Whole Image 640x480 12.49KB

```
an enemy can hit a
a defender can hit

an enemy is near a
    the enemy's x-c
    the enemy's y-c

a defender is near
    the defender's
    the defender's

an enemy hits the p
```

```
:start-simulation
:pause-simulation
the enemy's y-coord = 50
50.0
> 
```

Figure 8: While the simulation is paused, we will "teleport" the enemy downwards by manually setting its y-coordinate to a lower value.

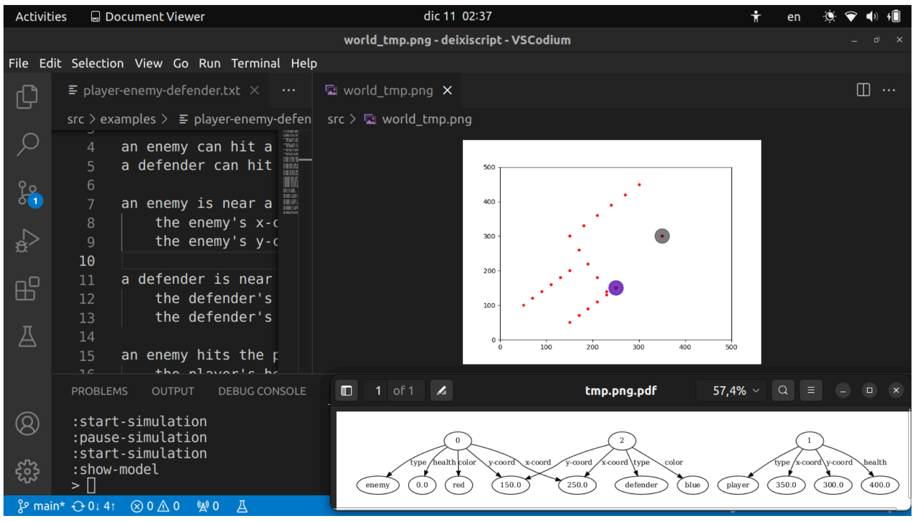


```
an enemy can hit a
a defender can hit

an enemy is near a
    the enemy's x-c
    the enemy's y-c

a defender is near
    the defender's
    the defender's

an enemy hits the p
```

```
the enemy's y-coord = 50
50.0
:start-simulation
:pause-simulation
> 
```

Activities
VSCodium
dic 11 02:36
world_tmp.png - deixiscript - VSCodium
File Edit Selection View Go Run Terminal Help
player-enemy-defender.txt
world_tmp.png
src > examples > player-enemy-defen
src > world_tmp.png
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
bash thesis
bash src
Python
main*
Whole Image 640x480 13.01KB

Figure 9: As soon as we resume the simulation (using the start-simulation metacommand) the enemy from its new position will still try getting to the player, while the defender takes a sharp turn to its left to intercept the enemy coming from a new angle. We have put the simulation on hold once more, using the pause-simulation metacommand.



```
an enemy can hit a
a defender can hit

an enemy is near a
    the enemy's x-c
    the enemy's y-c

a defender is near
    the defender's
    the defender's

an enemy hits the p
```

```
:start-simulation
:pause-simulation
:start-simulation
:show-model
> 
```

Figure 10: After resuming the simulation once more, the defender finally intercepts the enemy and hits it. The enemy's health suddenly drops to zero, so it stops moving; the defender halts too, because it no longer has any reason to move. We can inspect the current state of the world model with the show-model metacommand. As we can see, the enemy's health is zero, and the player's health is still intact. The player was successfully defended.

6 Conclusions

We dedicate the first part of the last chapter to discussing some (of the many) possible improvements and extensions that could be made to the current version of Deixiscript.

6.1 Possible Improvements of Deixiscript

In this section, we propose the addition of Syntactic and Semantic Compression facilities, to reduce Deixiscript's current level of verbosity; but another important improvement that could be made is to incorporate a better heuristic algorithm, that could allow planning to happen for a wider range of goals. For instance, one that could allow us to order the defender to "ensure the player stays alive" in the example made in Section 5.10.5, instead of forcing us to specify *how* it should do so (by "killing" the enemy).

6.1.1 Syntactic Compression

As we have seen in one of the code examples in the last chapter (Section 5.10.4) the current version of Deixiscript can produce code that tends to be a little verbose.

In the chapter about naturalistic programming (Section 3.6.1.1) we have seen the concept of syntactic compression, which is one of the mechanisms employed by natural language for redundancy avoidance.

We think that introducing some syntactic compression capabilities to Deixiscript could help decrease its current verbosity and improve its readability. For instance, instead of writing four almost identical Potential-rules (that differ only in their last word) like the following:

```
an enemy can move right.
an enemy can move left.
an enemy can move up.
an enemy can move down.
```

Deixiscript could instead only accept the following (single) rule, expanding it automatically into the previous four:

```
an enemy can move right/left/up/down.
```

A slightly different kind of expansion could be performed on rules that differ in more than one position. For example, the following two rules which differ in two positions (*left* versus *right*, *increments* versus *decrements*):

```
an enemy moves right means the x-coord increments.
an enemy moves left means the x-coord decrements.
```

could be collapsed into the following one:

```
an enemy moves right/left means the x-coord increments/decrements.
```

We think that these expansions could be treated as “syntax sugar”, so the right place to perform them would be when preprocessing the parse tree (or partially generated Abstract Syntax Tree) before it is really executed by the interpreter.

Perhaps, the system could also turn the (expanded) ASTs into some intermediate (human-readable) expanded versions of the code, so that the programmer could verify that the expansion was done correctly.

6.1.2 Semantic Compression

Another thing that we have looked at critically (but still have not explained why) is the manner in which the predicate “near” was defined in the last code example of the previous chapter (Section 5.10.4).

The rule defines the relationship “an enemy is near a player” as follows:

```
an enemy is near a player means:
    the enemy's x-coord = the player's x-coord,
    the enemy's y-coord = the player's y-coord.
```

The main problem (looking at this code from a Common Sense perspective) is that the user might expect this predicate to behave “symmetrically”, i.e., if it is true at any point that *the enemy is near the player*, it should also be true that the *player is near the enemy.*

Unfortunately, this is not the way Definitions work in Deixiscript. This could *not* be the way Definitions generally work, given that many more predicates other than “near” have a meaning that is “asymmetrical” (e.g., “x is loved by y” does not automatically entail that “y is loved by x”).

A possible solution to indicate that a predicate is supposed to be applied symmetrically could be adding a “vice versa” optional clause to a definition (the usage of the phrase “vice versa” is an example of “semantic compression” as we saw in Section 3.6.1.1), this would indicate to the system that the subject and the object could be swapped together without a change of meaning.

We would also have to figure out what happens to the body of the Definition in TELL mood, i.e., when “the enemy is near the player” is used as a statement, does it cause the position of the enemy to change to that of the player, or vice versa?

6.2 Possible Extensions: Speech

We suggest that a possible extension to Deixiscript is its adaptation to speech, by embedding it into a Voice User Interface (VUI). We have reasons to believe that a spoken programming language could benefit from being built around the principles of naturalistic programming.

Below we give a very brief overview of “voice programming” and the reason why it is has become relevant over the years.

6.2.1 The Need for Speech

Programming by voice has received some attention in the last years from both the commercial and the research sector, as an alternative to the widely popular approach of text-based programming [101], [102].

This has happened as a result of the increase in acquired disabilities related to long periods of typing, Repetitive Stress Injury (RSI), which can lead to severe neck ache and back pain [101], [102].

Another advantage of spoken systems is that speech is generally considered to be a faster input medium than typing: most people can speak faster than they can type, at least when speaking a natural language.

6.2.2 Voice User Interfaces (VUIs)

A Voice User Interface (VUI) is a Human-Computer Interface (HCI) that enables interaction with a computer through an auditive interface. It is usually a complement to the more popular Graphical User Interfaces (GUI), as it is most often seen in virtual assistants, automobile and home automation systems, and the like.

If one could overcome the many challenges behind building a suitable VUI for the task of writing, reading, and maintaining code, they would improve the quality of life of many disabled software developers and/or people interacting with computers that have motor health issues but are nonetheless capable of using speech to interact with such a system.

A general overview of what designing an effective VUI means, practically speaking, is given in [103]. As said, a VUI is a specific instance of a

HCI, and as such it is subject to the general considerations that can be made on the usability of any computer interface.

Some of the HCI general concerns discussed here, are: (1) Providing suitable feedback, (2) Allowing for user diversity (novice vs expert), (3) Minimizing memorization efforts, (4) Error prevention, (5) Error handling, and others.

Some of the more VUI specific concerns are: (1) Appropriate output sentences, (2) Output Voice Quality, and (3) Proper entry recognition.

Discoverability, Mixed Initiative and (multimodal) output remain some of the key challenging aspects of designing a speech enabled system.

Speech output, especially when enumerating available options, can be slow and tedious, it may therefore be of some benefit to provide an alternative alley for the output of a VUI system, such as a Graphical User Interface when feasible, thus making the system multimodal.

6.2.3 Difficulties in the Application of Voice to Programming

There are quite a few hurdles on this path, some of them are general to the design of a VUI, and some are specifically related to the deployment of such a system for the purpose of programming.

Some of the more general problems are related to: (1) the ephemeral nature of speech as compared to text (it is harder to edit and refine a document using speech) [103]; (2) issues in discoverability (or making sure the user is aware of the options available to him/her at any point of using a voice enabled system); (3) issues in transcription (there is a tradeoff between the size of the available vocabulary and the precision with which the words are recognized [103]); And (4) privacy and noise-related concerns (e.g., in a crowded workplace, since everyone in the proximity of the user of a Voice User Interface could hear him/her speak).

Some of the more programming-specific problems are: (1) recognizing keywords and abbreviations in code (at least in "traditional" code) that are not contemplated by off-the-mill voice recognition software [102]; and (2) dealing with multiple levels of nesting in programming language structures [102].

6.2.4 Syntax-directed Voice Editors

There have been multiple attempts at designing voice-enabled programming systems, and the approaches that were taken have been diverse.

One approach, detailed in [102], involves the idea of a Syntax-directed voice editor. A Syntax-directed editor, as the authors of the article explain, takes advantage of the regularities of a formal language to provide automatic completion for common language constructs, saving the user time and typing.

When applied to voice programming, the authors hypothesize that it can help reduce the mental toil of spelling out loud a potentially convoluted piece of programming syntax character by character. The Syntax-directed editor married to the voice recognition software produces a programming environment that is easy to reconfigure for many different programming languages.

6.2.5 Voice programming and the Reactive Paradigm

Another project [101], a little more recent than the one we just mentioned, focused on the idea of applying the Reactive Paradigm (rather than the more traditional Imperative style) to voice programming.

The researchers' goal was comparing the efficiency of the two paradigms by preparing spoken versions of a set of programs in Java versus RxJava, a library that provides Reactive Extensions to the language.

The Reactive Paradigm is oriented around data flow and the propagation of change. It deals with asynchronous data streams where the data is events and vice versa.

The researchers found out that the Reactive style usually produces longer code in both characters, syllables, and words as compared to the Imperative style. However, owing to the higher expressiveness of Reactive constructs, those words themselves produced more effective work.

Moreover, the words used in Reactive style programs contained a higher percentage of English-dictionary words rather than word abbreviations, which are harder to pronounce and harder to be recognized by general purpose voice recognition software; this perhaps owing to the fact that Reactive programming makes less use of temporary variables and short variable names.

6.2.6 Natural Language and Voice

The authors of the 2005 paper "Voice-commanded Scripting Language for Programming Navigation Strategies On-the-fly" [104] suggest that an ideal spoken scripting language should draw inspiration from natural language.

An aspect of natural language that we have briefly encountered when discussing the disadvantages of its usage in the context of programming (Section 3.8.1) involves the topic of syntactic ambiguity; the same sentence in natural language can be associated to multiple parse trees.

An important suggestion made by the authors of the paper relates to ambiguity detection. A voice interface that employs natural language could use mixed initiative prompts to try disambiguating user statements by asking him or her for clarifications.

All in all, we think that experimenting with natural language in spoken programming would be an interesting (yet challenging) extension to our present work.

6.3 Conclusions

Whether a programming language built on top of the principles of naturalistic programming that we have explored will ever become mainstream is a question well beyond the scope of our research. We will just say that further research is needed, and a lot of that research will have to collect empirical data and focus on the actual usage that programmers make of these naturalistic languages.

In any case, we think that the importance of natural language in the field of programming (and computer science as a whole) is destined to grow in the years to come; whether that happens as a result of the adoption of new naturalistic features by traditional programming languages, or (more likely) through the perfection of the techniques of interaction with Large Language Models (LLMs) such as Prompt Engineering and AI-assisted coding (Section 3.7).

We believe that a deeper insight into the mechanisms afforded to us by natural language to describe natural and artificial processes (together with a clear understanding of its limitations and pitfalls) will ultimately benefit the next generation of *people who interact with computers*, whether they will be called "engineers", "programmers", "computer educators", or simply "natural language speakers".

References

[1] E. W. Dijkstra, “On the foolishness of “natural language programming”, *Program Construction: International Summer School*, pp. 51–53, 2005.

[2] R. J. Brachman and H. J. Levesque, *Machines like us: toward AI with common sense.* MIT Press, 2022.

[3] S. Wolfram, “What Is ChatGPT Doing ... and Why Does It Work?”. Accessed: Nov. 01, 2023. [Online]. Available: https://writings.stephenwolfram.com/2023/02/what-is-chatgpt-doing-and-why-does-it-work/

[4] M. Welsh, “The End of Programming”, *Communications of the ACM*, vol. 66, no. 1, pp. 34–35, Dec. 2022, doi: 10.1145/3570220.

[5] K. W. Ian Shine, “These are the jobs most likely to be lost – and created – because of AI ”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.weforum.org/agenda/2023/05/jobs-lost-created-ai-gpt/

[6] T. Corfman, “The legal risks of ChatGPT”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.ragan.com/the-legal-risks-of-chatgpt/

[7] “Noam Chomsky Speaks on What ChatGPT Is Really Good For”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.commondreams.org/opinion/noam-chomsky-on-chatgpt

[8] M. A. Nielsen, *Neural networks and deep learning*, vol. 25. Determination press San Francisco, CA, USA, 2015.

[9] K. R. Chowdhary, "On the Evolution of Programming Languages", *CoRR*, 2020, [Online]. Available: https://arxiv.org/abs/2007.02699

[10] D. Mandrioli and M. Pradella, "Programming Languages shouldn't be" too Natural"", *ACM SIGSOFT Software Engineering Notes*, vol. 40, no. 1, pp. 1–4, 2015.

[11] "Why We Should Stop Using JavaScript According to Douglas Crockford (Inventor of JSON)". Accessed: Nov. 01, 2023. [Online]. Available: https://www.youtube.com/watch?v=lc5Np9OqDHU

[12] R. W. Sebesta, *Concepts of programming languages.* Pearson Education, Inc, 2012.

[13] "Ada Lovelace - Encyclopedia Britannica". Accessed: Nov. 01, 2023. [Online]. Available: https://www.britannica.com/biography/Ada-Lovelace

[14] B. Gross, "The French Connection: Inventor Charles Babbage drew inspiration from an unusual source for his analytical engine". Accessed: Nov. 01, 2023. [Online]. Available: https://sciencehistory.org/stories/magazine/the-french-connection/

[15] "1801: Punched cards control Jacquard loom". Accessed: Nov. 01, 2023. [Online]. Available: https://www.computerhistory.org/storageengine/punched-cards-control-jacquard-loom/

[16] E. W. Dijkstra, "Letters to the editor: go to statement considered harmful", *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, 1968.

[17] C. Böhm and G. Jacopini, "Flow diagrams, turing machines and languages with only two formation rules", *Communications of the ACM*, vol. 9, no. 5, pp. 366–371, 1966.

[18] J. McCarthy, “Programs with common sense”. [Online]. Available: http://jmc.stanford.edu/articles/mcc59/mcc59.pdf

[19] P. Graham, “The Roots of Lisp”. Accessed: Nov. 01, 2023. [Online]. Available: http://www.paulgraham.com/rootsoflisp.html

[20] A. Alexander, *Functional Programming, Simplified: (Scala Edition)*. CreateSpace Independent Publishing Platform, 2017.

[21] R. Stallman, “EMACS: The Extensible, Customizable Display Editor”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.gnu.org/software/emacs/emacs-paper.html#SEC17

[22] “Pass by Name”. Accessed: Nov. 01, 2023. [Online]. Available: https://www2.cs.sfu.ca/~cameron/Teaching/383/PassByName.html

[23] “COBOL, The Jargon File”. Accessed: Nov. 01, 2023. [Online]. Available: http://www.catb.org/jargon/html/C/COBOL.html

[24] “Log Book with Computer Bug”. Accessed: Nov. 01, 2023. [Online]. Available: https://americanhistory.si.edu/collections/nmah_334663

[25] T. Kuhn, “A survey and classification of controlled natural languages”, *Computational linguistics*, vol. 40, no. 1, pp. 121–170, 2014.

[26] B. Varie, “Importance of Cobol in 2023”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.linkedin.com/pulse/importance-cobol-2023-bryan-varie/

[27] G. L. Steele Jr, “Growing a Language”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.cs.virginia.edu/~evans/cs655/readings/steele.pdf

[28] E. W. Dijkstra, “The Humble Programmer”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html

[29] “Dr. Alan Kay on the Meaning of “Object-Oriented Programming”. Accessed: Nov. 01, 2023. [Online]. Available: http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en

[30] “2023 Developer Survey”. Accessed: Nov. 01, 2023. [Online]. Available: https://survey.stackoverflow.co/2023/

[31] J. Hunt, “Inheritance considered harmful!”, *Guide to the Unified Process featuring UML, Java and Design Patterns*, pp. 363–381, 2003.

[32] P. Graham, “Revenge of the Nerds”. [Online]. Available: http://www.paulgraham.com/icad.html

[33] “Dijkstra quote on OOP”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.brainyquote.com/quotes/edsger_dijkstra_204329

[34] C. Smith, B. McGuire, T. Huang, and G. Yang, “The history of artificial intelligence”, *University of Washington*, vol. 27, 2006.

[35] R. Feldman, “Why Static Typing Came Back (GOTO 2022 Talk)”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.youtube.com/watch?v=Tml94je2edk

[36] D. H. Hansson, “Turbo 8 is dropping TypeScript”. Accessed: Nov. 01, 2023. [Online]. Available: https://world.hey.com/dhh/turbo-8-is-dropping-typescript-70165c01

[37] “BeanShell”. Accessed: Nov. 01, 2023. [Online]. Available: https://beanshell.github.io/

[38] “ The Worst Programming Language Ever - Mark Rendle - NDC Oslo 2021”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.youtube.com/watch?v=vcFBwt1nu2U&t=1296s

[39] R. Knöll and M. Mezini, “Pegasus: first steps toward a naturalistic programming language”, in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, 2006, pp. 542–559.

[40] R. Knöll, V. Gasiunas, and M. Mezini, “Naturalistic types”, in *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, 2011, pp. 33–48.

[41] C. V. Lopes, P. Dourish, D. H. Lorenz, and K. Lieberherr, “Beyond AOP: toward naturalistic programming”, *ACM Sigplan Notices*, vol. 38, no. 12, pp. 34–43, 2003.

[42] O. Pulido-Prieto and U. Juárez-Mart\inez, “A survey of naturalistic programming technologies”, *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, pp. 1–35, 2017.

[43] L. A. Miller, “Natural language programming: Styles, strategies, and contrasts”, *IBM Systems Journal*, vol. 20, no. 2, pp. 184–215, 1981.

[44] “Complete 66 Mac vs PC ads + Mac & PC WWDC Intro + Siri Intro - YouTube (minute 27:12)”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.youtube.com/watch?v=0eEG5LVXdKo&t=1660s

[45] B. S.-F. & Lizzy McNeill, “How many words do you need to speak a language?”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.bbc.com/news/world-44569277

[46] D. E. Knuth, “Literate programming”, *The computer journal*, vol. 27, no. 2, pp. 97–111, 1984.

[47] H. Liu and H. Lieberman, “Metafor: Visualizing stories as code”, in *Proceedings of the 10th international conference on Intelligent user interfaces*, 2005, pp. 305–307.

[48] H. Liu and H. Lieberman, “Programmatic semantics for natural language interfaces”, in *CHI'05 extended abstracts on Human factors in computing systems*, 2005, pp. 1597–1600.

[49] “Pegasus Project”. Accessed: Nov. 01, 2023. [Online]. Available: http://www.pegasus-project.org/en/Welcome.html

[50] G. A. Miller, “The magical number seven, plus or minus two: Some limits on our capacity for processing information.”, *Psychological review*, vol. 63, no. 2, p. 81, 1956.

[51] A. Fantechi, S. Gnesi, and L. Semini, “Language and Communication Problems in Formalization: A Natural Language Approach”, *Logic, Computation and Rigorous Methods: Essays Dedicated to Egon Börger on the Occasion of His 75th Birthday*, pp. 121–134, 2021.

[52] “The Osmosian Order of Plain English Programmers”. Accessed: Nov. 01, 2023. [Online]. Available: https://osmosianplainenglishprogramming.blog/

[53] L. A. Hernández-González, U. Juárez-Mart\inez, and L. M. Alducin-Francisco, “Evolution of Naturalistic Programming: A Need”,

in *New Perspectives in Software Engineering: Proceedings of the 9th International Conference on Software Process Improvement (CIMPS 2020)*, 2021, pp. 185–198.

[54] O. Pulido-Prieto and U. Juárez-Mart\inez, "A model for naturalistic programming with implementation", *Applied Sciences*, vol. 9, no. 18, p. 3936, 2019.

[55] "Inform 7 v10.1.0 is now open-source". Accessed: Nov. 01, 2023. [Online]. Available: https://intfiction.org/t/inform-7-v10-1-0-is-now-open-source/55674

[56] "ganelson/inform: The core software distribution for the Inform 7 programming language.". Accessed: Nov. 01, 2023. [Online]. Available: https://github.com/ganelson/inform

[57] G. Nelson, "Natural language, semantic analysis, and interactive fiction", *IF Theory Reader*, vol. 141, no. 99, p. 104, 2006.

[58] Z. G. Szabó, "Compositionality", *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2022.

[59] T. Robertson Ishii and P. Atkins, "Essential vs. Accidental Properties", *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2023.

[60] I. Asimov, *The caves of steel*, vol. 2. Spectra, 2011.

[61] B. H. Liskov and J. M. Wing, "A behavioral notion of subtyping", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 6, pp. 1811–1841, 1994.

[62] "OpenAI GPT-3: Everything You Need to Know [Updated]". Accessed: Nov. 01, 2023. [Online]. Available: https://www.springboard.com/blog/data-science/machine-learning-gpt-3-open-ai/

[63] “Prompt Engineering Guide”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.promptingguide.ai/

[64] “Programming by Example”. Accessed: Nov. 01, 2023. [Online]. Available: https://web.media.mit.edu/~lieber/PBE/

[65] “The ChatGPT Lawyer Explains Himself”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.nytimes.com/2023/06/08/nyregion/lawyer-chatgpt-sanctions.html

[66] A. Ceravola and F. Joublin, “From JSON to JSEN through Virtual Languages”, *Open Journal of Web Technologies (OJWT)*, vol. 8, no. 1, pp. 1–15, 2021.

[67] K. Martineau, “What is AI alignment?”. Accessed: Nov. 01, 2023. [Online]. Available: https://research.ibm.com/blog/what-is-alignment-ai

[68] C. Nardo, “The Waluigi Effect - LessWrong”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.lesswrong.com/posts/D7PumeYTDPfBTp3i7/the-waluigi-effect-mega-post

[69] “Research: Quantifying GitHub Copilot’s impact on code quality”. Accessed: Nov. 01, 2023. [Online]. Available: https://github.blog/2023-10-10-research-quantifying-github-copilots-impact-on-code-quality/

[70] R. A. Poldrack, T. Lu, and G. Beguš, “AI-assisted coding: Experiments with GPT-4”, *arXiv preprint arXiv:2304.13187*, 2023.

[71] P. Graham, “Succintness is Power”. Accessed: Nov. 01, 2023. [Online]. Available: http://www.paulgraham.com/power.html

[72] D. Bain Butler, “Strategies for Clarity in L2 Legal Writing”, *Clarity, Journal of International Assoc. Promoting Plain Legal Language*, pp. 31–37, 2013.

[73] J. Nielsen, “Enhancing the explanatory power of usability heuristics”, in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, 1994, pp. 152–158.

[74] “Deep Blue - IBM”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.ibm.com/ibm/history/exhibits/vintage/vintage_4506VV1001.html

[75] H. Kautz, “Ronald J. Brachman and Hector J. Levesque, Machines Like Us: Toward AI with Common Sense”, *Prometheus*, vol. 39, no. 2, pp. 121–124, 2023.

[76] N. N. Taleb, *The black swan: The impact of the highly improbable*, vol. 2. Random house, 2007.

[77] F. D'Agostino, “Chomsky on creativity”, *Synthese*, pp. 85–117, 1984.

[78] A. M. Turing, *Computing machinery and intelligence.* Springer, 2009.

[79] “The Turing Test for AI Is Far Beyond Obsolete”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.popularmechanics.com/technology/robots/a43328241/turing-test-for-artificial-intelligence-is-obsolete/

[80] “ChatGPT broke the Turing test — the race is on for new ways to assess AI”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.nature.com/articles/d41586-023-02361-7

[81] "Evaluating GPT-3 and GPT-4 on the Winograd Schema Challenge (Reasoning Test)". Accessed: Nov. 01, 2023. [Online]. Available: https://d-kz.medium.com/evaluating-gpt-3-and-gpt-4-on-the-winograd-schema-challenge-reasoning-test-e4de030d190d

[82] D. Cole, "The Chinese Room Argument", *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, 2023.

[83] E. Ilkou and M. Koutraki, "Symbolic Vs Sub-symbolic AI Methods: Friends or Enemies?", in *CIKM (Workshops)*, 2020.

[84] K. Yang and J. Deng, "Learning symbolic rules for reasoning in quasi-natural language", *arXiv preprint arXiv:2111.12038*, 2021.

[85] B. C. Smith, "KR Hypothesis". Accessed: Nov. 01, 2023. [Online]. Available: https://cse.buffalo.edu/~rapaport/563S05/KR.hypoth.html

[86] C. Strasser and G. A. Antonelli, "Non-monotonic Logic", *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, 2019.

[87] "Mudskipper - Encyclopedia Britannica". Accessed: Nov. 01, 2023. [Online]. Available: https://www.britannica.com/animal/mudskipper

[88] M. Shanahan, "The Frame Problem", *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, 2016.

[89] C. Laffra and J. van den Bos, "Constraints in concurrent object-oriented environments", *ACM SIGPLAN OOPS Messenger*, vol. 2, no. 2, pp. 64–67, 1991.

[90] J. Jaffar and J.-L. Lassez, “Constraint logic programming”, in *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, 1987, pp. 111–119.

[91] H. Levesque, “Programming Cognitive Robots”. Accessed: Nov. 01, 2023. [Online]. Available: http://www.cs.toronto.edu/~hector/pcr.html

[92] “Why Premature Optimization Is the Root of All Evil”. Accessed: Nov. 01, 2023. [Online]. Available: https://stackify.com/premature-optimization-evil/

[93] “Pyright - GitHub”. Accessed: Nov. 01, 2023. [Online]. Available: https://github.com/microsoft/pyright

[94] “Pytest Documentation”. Accessed: Nov. 01, 2023. [Online]. Available: https://docs.pytest.org/en/7.4.x/

[95] “Lark Documentation”. Accessed: Nov. 01, 2023. [Online]. Available: https://lark-parser.readthedocs.io/en/stable/

[96] “Graphviz”. Accessed: Nov. 01, 2023. [Online]. Available: https://graphviz.org/

[97] “Graphviz PyPI”. Accessed: Nov. 01, 2023. [Online]. Available: https://pypi.org/project/graphviz/

[98] “Matplotlib”. Accessed: Nov. 01, 2023. [Online]. Available: https://matplotlib.org/

[99] G. Rey, “The Analytic/Synthetic Distinction”, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, 2023.

[100] "GW-BASIC: Chapter 6 Constants, Variables, Expressions and Operators". Accessed: Nov. 01, 2023. [Online]. Available: http://www.antonis.de/qbebooks/gwbasman/

[101] M. Lagergren and M. Soneryd, "Programming by voice: Efficiency in the Reactive and Imperative Paradigm". Accessed: Nov. 01, 2023. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1617714/FULLTEXT01.pdf

[102] S. C. Arnold, L. Mark, and J. Goldthwaite, "Programming by voice, VocalProgramming", in *Proceedings of the fourth international ACM conference on Assistive technologies*, 2000, pp. 149–155.

[103] V. Farinazzo, M. Salvador, A. L. S. Kawamoto, and J. S. de Oliveira Neto, "An empirical approach for the evaluation of voice user interfaces", *User Interfaces*. IntechOpen, 2010.

[104] M. Nichols, G. Gupta, and Q. Wang, "Voice-commanded Scripting Language for Programming Navigation Strategies on-the-fly", in *Proceedings of the HCI International 2005*, 2005.