



Article

A Model for Naturalistic Programming with Implementation

Oscar Pulido-Prieto ^{*,†}  and Ulises Juárez-Martínez 

Division of Research and Postgraduate Studies, Tecnológico Nacional de México/I.T. Orizaba, Oriente 9, Emiliano Zapata Sur, Orizaba C.P. 94320, Veracruz, Mexico; ujuarez@ito-depi.edu.mx

* Correspondence: opulidop@ito-depi.edu.mx; Tel.: +52-272-72-57056

† Current address: Oriente 9 No. 852 Col. Emiliano Zapata, Orizaba 94320, Veracruz, Mexico.

Received: 8 May 2019; Accepted: 16 September 2019; Published: 19 September 2019



Featured Application: A general-purpose naturalistic language can be used to develop software systems using a syntax closer to natural languages, but formal enough to avoid ambiguity. This is useful for tasks such as requirement transformation, maintenance and software evolution.

Abstract: While the use of natural language for software development has been proposed since the 1960s, it was limited by the inherent ambiguity of natural languages, which people resolve using reasoning in a text or conversation. Programming languages are formal general-purpose or domain-specific alternatives based on mathematical formalism and which are at a remove from natural language. Over the years, various authors have presented studies in which they attempted to use a subset of the English language for solving particular problems. Each author approached the problem by covering particular domains, rather than focusing on describing general elements that would help other authors develop general-purpose languages, instead focusing even more on domain-specific languages. The identification of common elements in these studies reveals characteristics that enable the design and implementation of general-purpose naturalistic languages, which requires the establishment of a programming model. This article presents a conceptual model which describes the elements required for designing general-purpose programming languages and which integrates abstraction, temporal elements and indirect references into its grammar. Moreover, as its grammar resembles natural language, thus reducing the gap between problem and solution domains, a naturalistic language prototype is presented, as are three test scenarios which demonstrate its characteristics.

Keywords: naturalistic programming; Controlled Natural English; expressiveness; formal languages; general-purpose programming; very high-level languages

1. Introduction

For a number of decades, programmers have sought the means by which to employ natural languages in software development [1]; however, the use of these languages is limited by the inability of computers to detect and resolve ambiguities, the resolution of which requires the human cognitive process. In response to this limitation, various authors have proposed mechanisms for defining controlled and formalized versions of a natural language that can be processed by a computer without ambiguity.

Various paradigms, including object-oriented programming (OOP) and model-driven architecture (MDA), have, in recent years, enabled a form of software development in which abstractions are modularly defined. The client uses natural language to describe its requirements, which are adapted into formal language and then dispersed or grouped by force into a programming language, which requires correct documentation explaining the context in detail. Often, the programmer's lack of

discipline or the software's delivery times result in deficient or non-existent code documentation, complicating software maintenance and evolution.

A programming paradigm must both be generic enough to express abstractions regardless of domain and, at the same time, describe enough expressive elements to support general purpose languages. While OOP defines encapsulated abstractions without considering the domain, it lacks the mechanisms necessary to encapsulate non-functional requirements scattered among abstractions, which, in turn, implies a transformation of client requirements [2]. Aspect-oriented programming (AOP) allows encapsulating dispersed code; however, since requirements are defined in natural language, transformations are still required to adapt them to the paradigm. Therefore, although AOP contributes to solving modularity problems, it does not afford expressiveness in terms of client requirements [3].

Biermann and Ballard [4] described limiting factors for the use of natural languages for programming, stating that natural languages have a wide range of syntax and semantics, which forces programmers to learn both a large number of grammatical constructions and their respective semantics. Another limitation is that natural languages, although more expressive, are more verbose and require more characters than a traditional programming language. Finally, as natural languages are ambiguous, clear and precise instructions are required to solve a problem. Given that complex syntax can be solved using subsets of a natural language and that these subsets facilitate verbose but self-documented instructions which obviate further documentation, ambiguity is a limiting factor that could persist in an English subset if an adequate syntax is not established or, on the contrary, efforts are made to make it the least verbose possible.

Ambiguity occurs because computers lack the efficient mechanisms to resolve it completely. This requires a correct subset of a natural language to be defined which, thus, must be "naturalistic". Naturalistic programming is based on this principle and, as in natural languages, a naturalistic programming language must be capable of making reference not only to objects, functions or aspects, but also to other instructions, using instructions such as "the first word in the previous paragraph" [5]. Naturalistic programming provides mechanisms for complementing already existing abstractions, thus requiring programmers to develop software that is closer to a natural language, but with the restrictions of a formal language, thanks to expressive grammar.

This article presents a conceptual model based on that proposed by Lopes, who theorized on the integration of structural, temporal and reflexive indirect references into programming languages [5]. The conceptual model includes the use of noun phrases, the distinction between plural and singular nouns for data collection, and a flexible definition of sentences that enables the simulation of a form of ellipsis in instructions, as well as providing a context for sentences in the form of circumstances. In addition, Knöll et al. [6] stated that, in natural languages, noun phrases are used as indirect references and show how these references can be used in the definition of naturalistic programming languages.

This conceptual model led to the design of a general-purpose programming language called SN, which enables the use of indirect and temporal references as in natural languages, while at the same time being formal enough to eliminate ambiguities and describe abstractions using a subset of the English language. A basic Application Programming Interface (API) was designed from this language, as were several test scenarios in order to verify program correctness in terms of syntax, and, finally, a questionnaire was implemented to evaluate its code expressiveness in terms of the English language.

This article comprises the following sections: Section 2 presents the background and related studies; Section 3 presents the conceptual model proposed; Section 4 describes the naturalistic language SN; Section 5 presents a description of the compiler and the challenges that emerged at the design stage; Section 6 presents three test scenarios; Section 7 discusses the statistical results of students' assessment of the expressiveness of the SN prototype language; and finally, Section 8 presents the conclusions and possible research lines for the future.

2. Background

In [7], the authors describes programming models as a set of properties that enable the design of a programming language, stating that a functional model is that which possesses the minimum elements required to define a programming language as an implementation of a universal Turing machine. They described a functional model as having the following elements: a lack of side-effects, which implies referential transparency and, therefore, immutable control structures; recursion as a functional mechanism to solve repetitive operations; and the ability to define a system of types for the use of lambda calculus.

Booch et al. [8] defined the elements they identified for the object model as: abstraction; encapsulation; modularity; and hierarchy. For these authors, abstraction is the capacity to take the most relevant elements for the system, while encapsulation consists of hiding the internal details of an object and offering a controlled mechanism to access them, and modularity is the ability to divide an abstraction into smaller units. Finally, hierarchy is the classification and ordering of abstractions where inheritance, a term which describes a relationship where a class “is a” type of another class, and aggregation, where a class “is part” of another class, are observed.

In [2], the authors defined AOP as a complementary paradigm focused on encapsulating non-functional requirements. As a complementary paradigm, AOP serves to support other paradigms, notably AspectJ, which is a language complementary to Java. The authors described the following elements in the join point model: *join point*, which is an identifiable execution point in the code; *pointcut*, which is a mechanism for specifying and quantifying join points; and *advice*, which is the additional functionality that is added to the join point. It should be noted that advice encapsulates a non-functional requirement that is applied to a join point through the pointcut.

Videira Lopes et al. [5] opened a discussion on the elements that are observed in the natural languages and can be added to AOP to describe abstractions with a level of expressiveness closer to English, while maintaining the formality of a traditional programming language. The authors stated that three types of references are observed in anaphoric relations: structural references; temporal references; and reflexive references. *Anaphora* is a natural language property that enables an entity (real or abstract) to be related to a pronoun or noun phrase without losing context. Traditional programming languages provide support only for explicit structural references such as “out.println(str)”, but not for indirect references such as “execute all methods”, and do not support reflexive references such as “execute the previous three instructions” and temporal references such as “execute the method before printing the string”. On the other hand, the authors stated that, while *AspectJ* uses a rudimentary form of temporal reference, it lacks the capacity to describe information taken from the context, in which a reflexive and temporal reference such as *the last operation*, which makes reference to any previous operation regardless of its context, could help to define expressiveness mechanisms closer to natural language.

Unlike anaphoras, cataphoras refer to elements described later in the same text, with both comprising endophoras, which are indirect references defined in the same text. On the other hand, exophoras refer to elements that are defined in another text. Endophoras and exophoras enable deixis, which consists of the use of indirect references whose meaning depends on the context in which they are used. As deixis is mainly observed in the form of pronouns or noun phrases, the same sentence can exist in different texts with their own context [9].

In this article, the concept of “entity” is used as a means of describing an identifiable “thing” using a signifier as a type, which, in turn, is a noun or adjective. (In semiotics, a signifier is defined as a component of the linguistic sign, which is formed by three elements: a referent, which is a real or abstract ‘thing’; a signifier, which is the symbolic representation of the referent; and a meaning, which is the representation of a mental idea [10]. On the other hand, Ferdinand de Saussure mentioned that, as there are signs without a referent, he only considered the signifier and the meaning [11].) Types are directly related to indirect references, because the use of noun phrases implies typing and, therefore, knowledge of the entity to which reference is made.

A noun phrase is group of words whose nucleus is a noun and which optionally contains one or more adjectives and a determiner, such as *the first element* [12]. A verb phrase is a group of words whose nucleus is a verb with direct and indirect compliments and a preposition, while complements to a verb phrase can also be noun phrases. Noun and verb phrases form the grammatical subject and predicate of a sentence [13].

Naturalistic programming is a paradigm based on the use of elements from the natural languages for the design of more expressive programming languages that are closer to said natural language. Videira Lopes et al. [5] described the advantages and limiting factors of both the natural languages and the programming languages currently used in software development. Their article mentions that natural languages are of little use given their ambiguity, while programming languages are difficult to maintain if the code is inadequately documented, as, due to their formality, the requirements have been transformed and adapted to the language. As a solution, the authors proposed a middle ground, namely a formal programming language featuring elements from both natural languages and anaphoric references.

Another detail to be taken into account is that mentioned by Clark et al. [14], namely that a programming language based on a formal representation of a natural language could belong to one of two categories, either formalist or naturalist. A formalist language lacks ambiguities because it focuses on computers processing it correctly, while the ambiguities of a naturalist language are resolved by means of mechanisms taken from artificial intelligence, meaning that ambiguities remain in the grammar. Based on both definitions, a naturalist language can be implemented with a broader subset of a natural language than a formalist language. In addition, it should be noted that, although, based on [14], the definition of “formalist” is closer to the approach reported here, the present article adheres to the “naturalistic” definition addressed in [5,12].

Scala is a multi-paradigm programming language that combines functional programming with OOP. As its code is compiled to obtain Java bytecode, the bytecode generated with Java and Scala is interchangeable [15]. For Scala, everything is an object, including primitive types and methods, enabling a method to be assigned to one variable or used as a parameter in another.

Finally, the term programming semantics describes the relationship between natural language structures and basic programming structures, where noun phrases are data structures, verbs are functions and adjectives are properties, as defined by Liu and Lieberman [16], Mihalcea et al. [17]. The present article differs from these definitions, in that nouns are treated as basic abstractions and adjectives as complementary features, while a noun phrase is a refinement that results from combining a noun with one or more adjectives, which introduce the attribute to define properties. In addition, as the SN language reported in this article produces Scala code, the mapping between the source code and a programmatic equivalent is carried out automatically. Programmatic semantics must not be confused with formal semantics, which are the mathematical and precise representation of the meaning of a programming language [18]. The present article only covers programmatic semantics, as the compiler generates Scala and AspectJ code as an intermediate language and their formal semantics are already validated [19,20].

Related Work

The following are the highlights from a comprehensive review reported in [21]: *Pegasus* is presented in [12] as a tool that enables the development of software that functions through a natural language entry based on a mechanism that resembles the human brain. *Attempto Controlled English* (ACE) is presented in [22–25] as a formalized version of the English language that can be transformed into first order logic. *Computer-Processable Language* (CPL) is presented in [14,26] as a natural controlled language developed with a naturalistic focus and featuring an embedded unambiguous language called CPL-Lite. *Metaphor* is presented in [27] as a tool that was envisaged to support novice programmers by enabling them to describe a problem in natural language and then returning Python code sketches. *Natural Language Computer* (NLC) is presented in [4] as a programming language focused on the

management of arrays that appear on the screen by using a subset from the English language to resolve the expressiveness problems of the C language. *Macho* is presented in [28,29] as a tool that enables the generation of source code based on a simple sentence in English. Presented in [30,31], *Natural Language Command Interpreter* (NLCI) is a tool focused on generating executable code from an entry in English and is based on an ontology generated from an API. As already mentioned in the survey, while these languages propose solutions from the perspective of natural language, none provide a programming model that describes the elements required for the design of general-purpose naturalistic programming languages.

3. Conceptual Model

In [21], various research articles dealing with the implementation of languages with naturalistic characteristics are reviewed, noting that the authors focused on resolving particular problems and mainly used nouns and verbs. Apart from nouns and adjectives, humans express themselves using pronouns, which are words that replace a noun or phrase and whose use depends on the context; however, the studies reported using, in general, the neutral pronouns *it* and *this* for singular words, and *these* or *those* for plurals. They observed that few studies made use of noun phrases, complementing them with adjectives when a noun with more specific characteristics was required. Based on the foregoing, it can be inferred that the relevant entities for software development resemble not only nouns but also noun phrases that result from the specialization generated by adjectives and to which, ideally, reference can be made via pronouns or, even, the noun phrases themselves. The use of phrases and pronouns enables indirect reference to be made to previously (or subsequently) defined entities, a common feature of natural languages. In contrast, in programming languages, identifiers are used to store values and references, while working directly with types would allow these languages to work with indirect references aside from pronouns such as “it”, as in the case of languages such as Haskell. Working with types enables the creation of instructions where identifiers are not used, which is common in natural languages, where types are used and identifiers are reserved for elements to be highlighted. For example, *append a string to a file* [6].

Another property of natural languages is their capacity to describe things non-sequentially or as a consequence of something, a concept defined as a circumstance, thus enabling situations to be associated with contexts. To solve circumstances in programming, some languages define events as objects associated with others for executing a process derived from an action. The disadvantage is that events require one object to be associated with another, leading to inexpressive definitions from an English language perspective. The AspectJ join point model is similar to the concept of circumstances, because it defines a context for a *join point* by means of *advice*. The disadvantage is that, as the join points are based on Java syntax, a change in method signature requires a review of the aspect code.

Based on the elements described above, a naturalistic language is composed of nouns, adjectives, verbs, circumstances and phrases, with said elements contributing to ensure that the language uses syntax closer to the English language and with a sufficiently elevated level of formality in order to avoid ambiguities in the code. A naturalistic syntax also requires mechanisms that enable the verification of the lines of code that are found before or after the execution of an instruction. Based on the foregoing, the minimum elements considered for the definition of a naturalistic conceptual model for the design of general purpose programming languages are as follows:

1. A noun as a base abstraction that is either singular or plural.
2. An adjective as a complement to the noun.
3. A verb as an action undertaken by a noun.
4. A circumstance as a determinant that responds to events.
5. A phrase used to define instructions with a complexity beyond that of the subject and the predicate.

6. The element must enable the definition of instructions in terms of those elements previously described (anaphoras).
7. As types are defined explicitly and statically, they are used in the construction of instructions comprising phrases.
8. The conceptual model is implemented via a textual language that offers a level of expressiveness similar to a natural language, but which is formalized in order to avoid ambiguity.

In the English language, nouns are used to describe entities, while adjectives complement nouns by providing particular features. Nouns and adjectives are combined in noun phrases, which, in sentences, function both as the grammatical subject and as a direct and indirect complement to the verb. Verbs define actions that affect entities and are combined with direct and indirect objects to form the predicate of a sentence. If a noun, adjective or verb requires a context, its execution is associated with a sentence by means of a circumstance. The use of phrases results in more complex instructions that enable reference to be made by means of types, be they nouns or adjectives, depending on the context, which function as indirect references (anaphoras).

Some of the desirable, but not required, elements are listed below:

1. Complete support for the deixis, which implies that the instructions are defined according to the elements that are described not only before but also after the same text.
2. Support for identifiers that enable work with the abstractions by means of direct references.
3. Property-based typing, which requires the classification of a noun based on its properties, and, therefore, has the capacity to add new properties to an abstraction when it is created.

The use of full deixis enables reference not only to previously defined elements but also to elements later defined in the same text (cataphoric references). Using indirect references enables work without identifiers, although these do enable elements to be highlighted when there are many entities of the same type. Finally, it has been observed that typing in OOP is only allowed via classes, interfaces or mixins (in some object-oriented languages, a mixin is a class used for defining methods and fields that can be used by other classes without being in the same hierarchy, thus avoiding the *diamond problem* [32]); therefore, it is proposed that an entity be classified not only with nouns and adjectives but also by means of specific properties that do not exist in other entities of the same type, as reported in [6].

This section describes circumstances, sentences, indirect references and explicit typing, with other model elements described in Appendix A.

3.1. Circumstance

A circumstance is defined as the set of things that occur around a fact, expressed as a sentence, that affect the meaning of the sentence in some way. In the conceptual model proposed, a circumstance provides mechanisms for describing the manner in which the noun and the elements that comprise it interact with other nouns, in order that the programmer can correctly express ideas and restrictions for instances without them being dispersed. Given that this study proposes an approach in which a noun is complemented with adjectives in order to create instances, a circumstance enables the establishment of restrictions that are based on which adjectives are permitted, which adjectives are required, and which adjectives are not permitted for the composition. A circumstance is a sentence which enables a noun to be instantiated, wherein a value is assigned to an attribute of the noun, or which enables the execution of a verb to be established in accordance with another.

3.2. Sentence

The combination of a verb and a noun (or phrases that contain them as a nucleus) results in a sentence with a particular meaning that depends on the definitions both of the verb and the noun, further to the manner in which the behavior of the latter is modified by an adjective. The conceptual

model describes grammatically imperative instructions with an elliptic subject (in natural languages, an elliptic subject is a grammatical subject which is omitted from a sentence without losing its context) and which are in the second person with the “orders” directed at the system, and indicative instructions in the third person, which have a grammatical subject. It should be noted that these grammatical modes are only evident for humans, as a computer cannot distinguish them due to its lack of reasoning. One example of an imperative instruction is *add 5 to result*, while an example of an indicative instruction is *the System prints the Number*.

In the present study, interrogative instructions are a special case because, although, in theory, there are no problems working with them in an implementation, they are not considered for the design of the SN language proposed here.

3.3. Indirect Reference

The conceptual model integrates the use of indirect references to describe elements previously defined in the code or defined at another site, as is the case with exophoras. These references function as a replacement for identifiers, which are considered optional elements in the implementation of the conceptual model. Nouns act as types that serve to create instances, while, at the same time, the type is used as a mechanism for referring to said instances thanks to the use of pronouns, numerals or ordinals. Similarly, the properties of a noun serve as a mechanism for selecting particular abstractions, with the instruction *the Number with 50 as value* thus functioning as an indirect reference for a number with a value of 50.

To obtain a set of *Numbers*, reference can be made to each number based on various criteria: *the first Number*; *the 3rd Number*; *the last Number*; or, even *it*, which refers to the last number used or derived from a particular operation.

3.4. Explicit Typing

The model presented in this article requires explicit typing due to the fact that, in natural languages, people express their ideas in accordance with types, referring to things such as *the House*, *my House*, *the first House*, or *all Houses*, instead of *the House H*. This means that every entity defined is, in turn, a noun whose type was clearly defined and whose properties either serve to identify it or are combined with adjectives that represent particular properties that provide it with some characteristic that distinguishes said entity from other similar ones. For example, *the House with “green” as color* can be defined as *the Green House*, where the property *color* directly becomes the adjective *Green*. Finally, a type is expected to form a hierarchy that requires specialization, meaning that a base noun serves as a general element from which types will be generated and which will thus provide more concrete characteristics. For example, *the House* can be defined as a specialization of *the Building*. As can be seen, the concept is similar to the hierarchy of the object model, due to the fact that said model draws on the idea of semantics and is, therefore, a fundamental part of natural languages.

4. Language

The conceptual model proposed in this article serves as a basis for defining naturalistic general-purpose programming languages. This section presents the most important relevant elements of the SN language, an English-based programming language derived from the elements established in the proposed conceptual model. The remaining elements can be found in Appendix B.

Using indirect references, SN returns Java *bytecode* as a result and enables a higher level of description by defining a process similar to a verb phrase, which, conceptually, ensures that it is similar to a function or method. It also defines nouns that, optionally, have a plural form, as well as adjectives that combine with nouns during either definition or instantiation. Lastly, SN presents a limited capacity for describing circumstances for a noun or an adjective. Given that the main purpose here is to implement the model, the optimization of the code was not taken into account. It should

be noted that, given the extent of SN grammar, only its most relevant elements are described in the present study.

In the SN language, reserved words are written in lower case, while it is recommended that the first letter of nouns and adjectives is in upper case. For attributes and verbs, said letter is kept in lower case, although there is no problem with the use of upper and lower case letters, as long as reserved words such as nouns, adjectives, verbs or identifiers are not used. This mode of use is recommended for creating a correct distinction between the main abstractions and the rest of the words. A *Hello World* example is shown below:

```
1 main Example:
2 System prints ‘Hello World’.
```

As can be observed, an abstraction *main* is defined on Line 1, which functions as a point of entry to the program, while, on Line 2, the sentence *System prints “Hello World”* contains the noun *System*, which possess a verb *prints* which, in turn, accepts a direct object, which, in this case, is the message to be printed. The Java equivalent is shown below:

```
1 class Example {
2 public static void main(String...arg) {
3 System.out.print(‘Hello World’);}}
```

4.1. Abstractions

SN has three types of abstractions: nouns; adjectives; and a main abstraction. These abstractions are described in this section.

4.1.1. Noun

Nouns and adjectives are defined in order to create base and complementary abstractions. Nouns are base abstractions that can refine other nouns (inheritance) or be combined with adjectives in order to obtain a complementary behavior. An example of a noun definition is shown below:

```
noun House.
```

The signature is simple: the reserved word *noun* is used, followed by the name of the abstraction. In natural languages, nouns are narrowed subtypes of other nouns. SN enables the refining of a noun to achieve a more specialized behavior. Based on this, the noun *House* is defined as a specialized form of *Building*, as shown below:

```
noun Building.
noun House is a Building.
```

where *Building* describes a base noun, with another noun *House*, presenting more concrete behavior than that of the functionality of *Building* in order that the code is reused and a hierarchy of types is created. As in other programming languages, the multiple inheritance of nouns is not allowed in SN.

By default, the noun *Thing* is the root of the types used in the SN language. All the nouns defined will implicitly extend said type or some of the subtypes. The type *Things* is used as a root for all the plurals defined by SN and can be used to handle every noun without the explicit definition of a plural. To define a plural, SN uses the following instruction:

```
noun House is a Building with plural as Houses.
```

The above makes it possible to create the plural noun *Houses* for the noun *House*.

4.1.2. Adjective

With the exception of plurals, adjectives are described in a similar way, as shown below:

adjective *Luxurious*.

where the adjective *Luxurious* is defined in order to be combined with a noun. Refining an adjective is also possible, with an example shown below:

adjective *Big*.

adjective *Luxurious* is *Big*.

The adjective *Luxurious* is a refined form of the adjective *Big*; however, unlike nouns, an adjective can be refined from two or more adjectives, as shown below:

adjective *Costly*.

adjective *Luxurious* is *Big* and *Costly*.

The adjective *Luxurious* is now defined as the union of the adjectives *Costly* and *Big* and can be combined with the noun *House* to define a particular type of house. The instantiation of an adjective requires the noun it complements:

noun *Mansion* is a *Luxurious House*.

By default, the root of the adjectives from the SN language is *Adjective*, with all the adjectives that are defined implicitly extending said adjective or one of its subtypes.

4.2. Instance Creation

For SN, the term instance is used in the same context as OOP: a set of data in memory and accessed through an identifier or, in the case of SN, an indirect reference. To create an instance of a singular, the reserved words *a/an* are used, followed by the name of the noun, as shown below:

a *House*.

For a plural instance, the reserved word *some* is used, as shown below:

some *Houses*.

As proposed in [6], SN enables instance creation without requiring constructors, wherein the attributes can be associated with a value via the preposition *with* regardless of order or whether not all of them are assigned, while any subsequent process can be associated with another by means of circumstances. An example is given below:

a *House* with 5000.0 as *price*.

where the attribute (Attributes are described in Appendix B.3) *price* is a *Real Number*, which comprises two basic SN types in the API: the noun *Number*; and the adjective *Real*. (Although “Real” can, grammatically, also be a noun, for the numerical context, it was chosen to work as an adjective, leading to the adjective “Integer” also being used. Both are combined with the noun “Number” for handling real and integer numbers, while a circumstance is used to ensure mutual exclusion.) The value of *price* is assigned when the instance is created. With these changes, the noun *House* is defined as follows:

noun *House*:

attribute *price* as a *Real Number*.

Moreover, SN enables the creation of instances using an “instantiation-time” combination for merging a noun with one or more adjectives, which enables the definition of decoupled abstractions (both nouns and adjectives) that are combined as required. An example is shown below:

a *Luxurious House*.

4.3. Noun Phrases in Sentences

It should be noted that the examples given in the previous section are stored in a pile instance, in order that each instance can be accessed based on its type and position, for which reason, the SN language uses noun phrases and their position by means of ordinals. An example of their implementation is presented below:

```
the first House
the last Building
the second House
```

On occasions, a noun with an attribute of particular value is required, irrespective of its position. For these cases, the clause marker *where* is used, which works in a similar way to SQL. Given the following instances:

```
a House with 20,000.00 as price.
a House with 60,000.00 as price.
a House with 40,000.00 as price.
```

An example of their implementation is presented below:

```
the first House where price is equal to 40,000.00
the last House where price is lesser than 60,000.00
```

where the instructions indicate that the instances referenced are not only identified by their position but also by the value of the attribute *price*. In this particular case, *the first House where price is equal to 40,000.00* refers to the last instance, while *the last House where price is lesser than 60,000.00* refers to the second because the precedence of the ordinals is lower than that of the attributes, while priority is lower than that of the type. For example, given the following abstractions:

```
noun Building with plural as Buildings:
  attribute price as Real Number.
adjective Costly.
adjective Big.
noun House is a Building with plural as Houses.
noun Mansion is a Big and Costly House.
```

and the subsequent instances:

```
1 a House with 20,000.00 as price.
2 a Costly House with 60,000.00 as price.
3 a Mansion with 80,000.00 as price.
4 a Building with 10,000.00 as price.
```

while, given the following instructions:

```
1 System prints the first Building.
2 System prints the first Building where price is equal to 60,000.00.
3 System prints the second Building.
4 System prints the last House.
5 System prints the Big House.
6 System prints the House where price is equal to 30,000.00.
```

Line 1 prints the value of the first instance, while Line 2 prints the value of the second instance, and Line 3 prints the value of the second instance. Line 4 prints the value of the third instance, while Line 5 prints the value of the third instance, and Line 6 prints an error, as no house has a value of 30,000.00.

Finally, it is sometimes necessary to work, not with one instance, but with a group of them, with SN enabling indirect references to collections of instances that are later grouped into plurals. An example is presented below:

```
all Houses
the last 3 Houses
all Houses where price is greater than 25,000
the last 3 Houses where price is equal to 500,000
```

Based on the previously defined *Building* abstractions:

```
1 System prints all Buildings.
2 System prints the first 3 Buildings.
3 System prints the last 2 Buildings.
4 System prints the first 2 Costly House.
```

Line 1 prints all the instances, while Line 2 prints the first three instances, and Line 3 prints the last two instances, with Line 4 printing Lines 2 and 3, as they contain the adjective *Costly*.

4.4. Local Identifiers

As the use of indirect references not only enables instructions to be written in a form closer to natural language but also causes a large number of instances to become difficult to handle, SN also enables the use of local identifiers. For example, the instance defined *a Luxurious House* can be associated with an identifier, as follows:

```
house is a Luxurious House.
```

where the reserved word *is* corresponds to the operator = of other programming languages. The same applies for literals (numerals and strings) and other identifiers, as shown below:

```
h is house.
num is 5.
str is 'A string'.
```

where the identifier *house* from the previous example is associated with the identifier *h*, while, in the next line, the identifier *num* is associated with the literal 5 and, finally, the identifier *str* receives a string.

4.5. Circumstance

The SN language uses circumstances to define the context of a sentence execution. In this way, a circumstance is used to establish conditions to which nouns, adjectives and verbs react.

4.5.1. Circumstances Applied to Instances

A feature of the circumstance is that it enables the establishment as to which adjectives are required, which are not permitted and which are mutually exclusive at the moment of instantiation. For example, the abstraction *Building* is defined in the context of a realtor, as are the restrictions for the moment at which work with the instances begins, as shown below:

```
1 noun Building:
2   circumstance: this cannot be Integer or Real.
3   circumstance: Office and Habitable are mutually excluded.
4   circumstance: this requires Habitable or Office.
```

As can be seen, three circumstances are described which indicate that an instance of *Building* cannot be combined with *Integer* and *Real* adjectives and which, by necessity, must be combined with *Office* or *Habitable*; however, these, in turn, cannot be combined with each other. The foregoing enables the description of mutual exclusion mechanisms for non-invasive adjectives:

```
1 adjective Office:
2   circumstance: this cannot be Habitable.
3 adjective Habitable:
4   circumstance: this cannot be Office.
```

It should be noted that the example provides the definition of circumstances for both adjectives, although only one is needed to grant mutual exclusion.

With this in mind, the following example describes how circumstances work:

```
1 a Building.
2 an Office Building.
3 an Integer Building.
4 a Habitable Building.
5 an Office and Habitable Building.
```

In the above example, Lines 1, 3 and 5 will raise an error because those instances are forbidden. Line 1 breaks the circumstance defined on Line 4 of *Building*, while Line 3 breaks the circumstance defined on Line 2 of the same noun, and Line 5 breaks the circumstances defined on Line 3 of *Building* and those defined on Lines 2 and 4 in the adjectives *Office* and *Habitable*.

A circumstance enables the conditioning of the execution of a verb (verbs are described in detail in Appendix B) both before and after an instance is created. However, to date, this has been restricted solely to the use of verbs that are found in the same noun as the circumstance and the attribute, although the provision of an extended functionality for said mechanism is also an objective:

```
1 adjective Printable:
2   verb print itself:
3     System prints ‘‘A new house was created’’.
4   circumstance: print this after this is created.
```

As can be seen, the verb *print* that provides the adjective *Printable* for the instance will be executed when it is combined after the instantiation. For example:

```
a Printable House.
```

The following will be printed:

```
A new house was created
```

4.5.2. Circumstances Applied to Attributes

In the case of the attributes, a circumstance enables the execution of a verb when attempts are made to assign the value of the attribute:

```
1 noun House:
2   attribute description as String.
3   verb itself prints desc as String:
4     System prints desc.
5   circumstance: it prints description
   when the description is assigned.
```

As can be seen, the verb *prints* is executed when the value of the attribute *value* is assigned. For example:

```
a House with "A beautiful house" as description.
the description of the House is "Another description".
System prints the description of the House.
```

The following will be printed:

```
A beautiful house
Another description
```

This is due to the fact that *when* is considered the equivalent of the advice *before* in AspectJ.

Another case to be considered is that which occurs when it is necessary to ascertain whether an attribute has a particular value, an example of which is presented below:

```
1 noun House:
2   attribute description is "".
3   verb itself prints desc as String:
4     System prints desc.
5   circumstance: it prints description
   when the length of description is 0.
```

Taking the above example, the output with this circumstance would uniquely be *Another description* due to the fact that the execution of *prints* is conditioned for the length of *description* to be 0:

```
a House with "A beautiful house" as description.
the description of the House is "Another description".
System prints the description of the House.
```

The following will be printed:

```
Another description
```

This output occurs due to the fact that, during instantiation, value was assigned to *description*, meaning that the circumstance verifies whether or not the length is 0, while, on the other hand, the subsequent instantiation gives the following:

```
a House.
the description of the House is "Another description".
System prints the description of the House.
```

where the following will be printed:

```
Another description
Another description
```

This is due to the fact that value is not assigned in *description*, for which reason its length is 0.

4.5.3. Circumstances Applied to Verbs

Finally, given an assignation of the variable of a noun *Mansion*, the following entails:

```

1 noun Mansion:
2   attribute price as an Integer Number.
3   verb assign p as Integer Number to itself:
4     the price of this is p.
5   derived attribute string as a String:
6     a String with ‘‘Mansion valued as ’’.
7     add price to the String.
8     return it.
```

It is observed that there is no mechanism for validating whether a null or empty value is given to the verb *assign* on Line 3, which is resolved by means of a circumstance and a verb tasked with the following validation:

```

1 adjective Validated:
2   verb itself validates val as Integer Number:
3     execute the next instruction when val is null.
4     the price of this is 0.
5     execute the next instruction when val is not null.
6     the price of this is val.
7   circumstance: this validates val instead assign val to something.
```

As observed, the execution of *assign* on *Mansion* on Line 3 is replaced by that of *validates* between Lines 2 and 6, which is presented in the adjective *Validated*. Furthermore, the description provides the conditions in order that the circumstance occurs when assigning a *Integer Number* to ‘‘something’’. The compiler is tasked with validating that both the instruction *validates* and *assign* cohere with the type required for the particular context of *Validated*. For example, if *something* does not correspond to *Validated*, the substitution will not be carried out. The same occurs if the adjective is not combined with a noun that possesses the verb *assign*. Thus, two examples are obtained:

```

a Mansion.
a Validated Mansion.
the price of it is null.
System prints the first Mansion and newline.
System prints the last Mansion and newline.
```

Each will print two distinct things:

```

Mansion valued as null
Mansion valued as 0
```

The Scala and AspectJ equivalent is shown below:

```

class Mansion {
  var price : Number with Integer
  def assign(p : Number with Integer) { this.price = p }
  def string = ‘‘Mansion valued as ’’ + price}

trait Validated {
  def validates(val : Number with Integer) {
    if(val == null)
```



```

        this.price = 0
    else
        this.price = val}}

1 aspect Validated_ValidatorAspect {
2   Object around(Validated v, naturalistic.lang.Integer $val) :
3     execution(* Validated+.assign(naturalistic.lang.Integer+)) &&
4     target(v) &&
5     args($val) {
6     Object rn = null;
7     try {
8       rn = v.validates($val);
9     } catch (Exception e) {
10    rn = proceed(restrictedMansion, $val);}
11    return rn;}}

```

In this case, the AspectJ advice indicates that, when the execution point of the method *assign* defined in *Validated* or its subtypes is reached (Lines 2 and 3), the method *validates*, defined within *Validates* (Line 8), will be executed instead. Lines 4 and 5 define the objects and arguments required to work with *validates*, while Line 10 places the original execution, which is carried out if the *validates* method gives an exception. It should be noted that this example is an optimized version of the result that occurs during SN compilation, as it is resolved using Java Reflection.

4.6. Naturalistic Control Structures

While traditional programming languages are accustomed to using instructions associated with blocks for delimiting the scope of an iterator or conditional, natural languages describe repetitive instructions, which indicate up to which part of the text the actions are scoped. This section presents naturalistic control structures, while their grammar is presented in Appendix B.

4.6.1. Naturalistic Iterator

The SN language enables work with reflexive instructions in order to carry out iterations, with the iterator functioning as an anaphoric or cataphoric reference.

Naturalistic iterators enable the generation of instructions that solely specify how many instructions will be repeated and under which conditions said iteration will take place. The instruction does not have access to the context of the other instructions which it affects, nor is it clear whether an instruction really does appear before or after. This situation is caused by the nature of a computer, which does not know the context of an instruction. An example is presented below:

```

i is 0.
repeat the next 2 instructions until i > 10.
System prints i.
add 1 to i.

```

As can be seen, blocks or marks are not defined, while the compiler is tasked with working with the instructions indicated by means of *the next 2*. If an iterator is required to repeat a block of instructions a specific number of times, the *times* reserved word is used:

```

numbers are some Numbers.
repeat the next instruction 50 times.
an Integer Number; and add 2 to numbers.

```

In the example, the iterator will be repeated 50 times, with a new number added to the collection of numbers. Finally, if it is necessary to work with each element of a plural, such as the plural *numbers* from the previous example, the following construction is used:

```
i is 1.
repeat the next 2 instructions for each numbers as number.
multiply i by 2.
add it to number.
```

The example takes the collection of numbers that was created in the previous example and performs a transaction on each of its values, represented by the variable *number*.

4.6.2. Naturalistic Conditional

A naturalistic conditional is defined as resembling the manner in which naturalistic iterators express themselves, as they possess the same characteristics and limitations, with the difference that, instead of indicating that a set of instructions is repeated N times, it indicates that the set of instructions will be executed only if a particular condition is fulfilled. An example is presented below:

```
flag is false.
execute the next 3 instructions when flag is equal to false.
System prints ‘‘Optional output’’.
System prints ‘‘Another output’’.
flag is true.
System prints ‘‘From the outside’’.
```

where the three instructions subsequent to the definition of the condition are executed if the flag has value *false*, otherwise, only *From the outside* will be printed.

5. Compiler

This section details the most important characteristics of the SN language compiler. It should be noted that, being the first prototype of SN, special emphasis was placed on the functionality and correctness of the result, leaving aside the efficiency and speed of the bytecode generated. The decision was also taken to use two existing programming languages due to the fact that they possess the features required to implement the elements considered in the design of the SN language.

The grammar was implemented using ANTLR v4 (ANother Tool For Language Recognition) [33]. At the time of writing, there were 68 rules and 137 tokens, due to the complexity of the instructions, which contain sentences with noun phrases comprising nouns and adjectives, further to sentences with a deictic subject where the complements of the verb are also noun phrases. The number of tokens is high because prepositions and auxiliaries are used for the correct generation of the verbs. Output at the first step of compiling produces Scala and AspectJ code, with both codes later compiled by their own compilers to generate bytecode.

Scala is a multi-paradigm programming language that combines object-oriented programming with functional programming [34]. Version 2.12.3 (not the SBT (the *Simple Build Tool* is a build tool that works similarly to Maven or Ant [35]) release) was chosen as the main language for the present study, using intermediary code generation to obtain bytecode.

Chosen to carry out the work with circumstances, AspectJ (1.8.10) is an aspect-oriented language based on Java, which uses the signature of the methods to identify execution points into which code can be inserted [3].

To optimize the SN grammar tests, it was decided to use the Scala and AspectJ languages as intermediate languages, meaning that, for each SN noun, a Scala-class source code is generated, while, for each adjective, a Scala *trait* (a trait is a mixin implementation for Scala) source code is also generated.

Nouns and adjectives have attributes and verbs, which are translated into Scala fields and functions, respectively, with the exception of derived attributes, which are translated into Scala functions. All circumstances defined in a noun or adjective are grouped into an AspectJ complementary aspect. Finally, bytecode generation is conducted by AspectJ and Scala compilers.

The decision was taken to use *Reflection* to simplify the process of code transformation, as the use of indirect references requires the intensive use of explicit *casting* (the term *casting* is used in OOP to indicate the conversion of the type of object into any other type within its hierarchy. [36]) at Scala level, which complicated the debugging of the first programs written in SN. This was because several examples consist of type conversions that are not part of the noun hierarchy and a correct type analysis is required to achieve proper functionality.

Although avoiding explicit conversion leads to an integrity problem, this decision was made in order to restrict the use of complex grammatical constructs. As a result, *Reflection* contributed negatively to bytecode efficiency, although it did simplify code generation, while its use in AspectJ resulted in fragile signatures in which the implementation of advice is decoupled from the pointcuts, thus reducing the effect of fragility.

5.1. Scala Generation

Once the Scala and AspectJ source code is generated, their compilers are used to generate the executable bytecode. First, the Scala compiler is invoked, generating at least one *class* file per noun (now translated into a Scala class) and adjective (now translated into a Scala trait). It should be noted that, if a composition is undertaken at instance-time, an additional *class* will be generated for each different composition. This occurs because, internally, the Scala compiler translates instance-time compositions into inner classes (within the corresponding function), while the successive instances of that composition are actually types of that inner class. Given the following example:

```
noun House.
adjective Big.
adjective Costly.
```

whose Scala result is as follows:

```
class House
trait Big
trait Costly
```

if there is an SN implementation such as the one shown below:

```
main Example:
  h1 is a Big House.
  h2 is a Costly House.
  h3 is a Big and Costly House.
  h4 is a Costly and Big House.
```

the resulting Scala code is as follows:

```
object Example extends scala.App {
  var h1 = new House with Big
  var h2 = new House with Costly
  var h3 = new House with Big with Costly
  var h4 = new House with Costly with Big}
```

Therefore, the Scala compiler will generate the following *class* files:

```
Big.class
Costly.class
House.class
```

```
Example.class
Example$.class
Example$$anon$delayedInit$body.class
```

```
Example$$anon$1.class
Example$$anon$2.class
Example$$anon$3.class
Example$$anon$4.class
```

where the first three *class* files are the two adjectives and the defined noun. The following three *class* files refer to elements that Scala generates for executing the program, which, as they are specific to Scala, are beyond the scope of this article. Finally, the remaining four *class* files correspond to each of the inner classes that the Scala compiler generates for the time instance composition made when an object is instantiated.

Another notable detail is that working with such high level grammatical elements ensures that the signatures for the methods in the bytecode are ambiguous. For example, the signature of the verb *plus num as Number to itself* in bytecode was simply *plus*, which caused compiler problems when working with said verb, because it could not be verified whether or not the sentence was translated as *num plus 5* or *plus num 5*. Thus, for the translation into Scala, the decision was taken to generate methods with a name that would include not only the verb but also the position of the noun invoking it, as well as the number of arguments and the preposition (if any). Thus, if the noun is as follows:

```
noun Number:
  verb itself plus arg as Number.
```

the signature is generated as follows:

```
class Number
  public Number plus(arg : Number)
```

With the changes, the signature is now generated as follows:

```
class Number
  public Number itself_plus_arg(arg : Number)
```

Thus, the ambiguity caused by signatures is controlled.

In the SN language, as indirect references are an alternative for local variables, every verb translated into a Scala method has a *ItMethodCompanion* object that handles said references. When an instance is created, it is stored on a list contained by *ItMethodCompanion* and, if an indirect reference is used, *ItMethodCompanion* searches for it based on position (using ordinals), type (nouns and adjectives) and specific values for the attributes.

To work with iterators and conditionals, SN undertakes a process of semantic analysis in order to ascertain whether or not there are sufficient instructions (one or more) after the definition of the iterator or conditional. In the event of these not being sufficient, a semantic error is presented, while, in the event of there being sufficient instructions, they are arranged in order that *while* or *for* blocks are generated for the iterators, or *if* blocks are created for the conditionals. Given the fact that iterators and conditionals do not describe blocks, the SN language solely provides scope for the local variables. This means that, if a variable is defined in an instruction related to a conditional or an iterator, the definition will be moved to just before the blocks, while the point at which it is employed will be used to provide it with an assignation.

5.2. AspectJ Generation

Once the Scala code is generated, the SN compiler verifies whether there is a circumstance and, if not, the compiler ends its execution, otherwise it invokes the AspectJ compiler that generates a *class* file for each noun or adjective that has at least one circumstance. Given that Load-Time Weaving (LTW) is used, code weaving is not performed, as this process occurs when the classes are loaded during the execution of the program.

LTW enables the generation of bytecode, from AspectJ, without directly altering it, as occurs in the generation of AspectJ compile-time weaving. The decision was made to keep the bytecode unweaved, thus making it possible to analyze it without depending on its aspectual counterpart. As both AspectJ compile-time weaving and LTW are processes pertaining to AspectJ, they are outside the scope of this article.

As a result of the execution, a *class* file is generated with the name of the noun or adjective it complements, plus the suffix *_ValidatorAspect*. For example, if the following noun is given:

noun House:

circumstance: Big and Costly are mutually excluded.

the AspectJ compiler will generate the following *class* file:

```
House_ValidatorAspect.class
```

6. Test Scenarios

Three test scenarios are presented below, with the first using indirect references to work with the basic operations for integers, while the second presents the implementation of a database connection, and the third describes an approximate equivalent of the example of files presented in [5].

6.1. Case 1: Operations with Numbers

This scenario presents the noun *Number* that describes the basic arithmetic operations of addition, subtraction, multiplication and division. Its equivalent in the SN language is presented below. While it should be noted that this noun is part of the “low level” API for the SN language, a simplified version is used to demonstrate the properties of the language.

```
1 abstract noun Number is a Thing with plural as Numbers:
2   verb add number as Number to itself.
3   verb itself plus number as Number.
4   verb subtract number as Number from itself.
5   verb itself minus number as Number.
6   verb multiply number as Number by itself.
7   verb itself times number as Number.
8   verb divide number as Number by itself.
9   verb itself by number as Number.
10  verb leftover number as Number by itself.
11  verb itself mod number as Number.
12 adjective Integer.
```

The noun is described on Line 1 with a plural definition, while, on Lines 2 to 11, verbs representing basic mathematical operations are presented, with only Line 10 emphasized because the term “leftover” could be confusing as it is a word chosen for representing a remainder operation. Lines 2, 4, 6, 8 and 10 represent operations that alter the *Number* instance. Lines 3, 5, 7, 9 and 11 are operations that return another instance of *Number* without altering it. The complementary adjective is presented on Line 12, which has specific implementations for handling integer numbers, and is conceived to combine with *Number*, as this noun is abstract and incomplete. Its Scala equivalent is the following:

```

1 abstract class Number extends Thing {
2   def add(number : Number) : Number
3   def plus(number : Number) : Number
4   def subtract(number : Number) : Number
5   def minus(number : Number) : Number
6   def multiply(number : Number) : Number
7   def times(number : Number) : Number
8   def divide(number : Number) : Number
9   def by(number : Number) : Number
10  def leftover(number : Number) : Number
11  def mod(number : Number) : Number
12 trait Integer

```

As can be seen, it uses an abstract class and a *trait* in order to undertake the composition. In Scala, the part for the plurals is easily managed using a list, as shown below.

```
var numbers = List(new Number with Integer)
```

The following code presents the manner in which the verbs are invoked:

```

1 main Operations:
2   an Integer Number with 5 as value.
3   add 2 to the Number.
4   the Number plus 3.
5   the second Number minus the first Number.
6   an Integer Number with the value of the third Number as value.
7   subtract 1 from the fourth Number.
8   divide the fourth Number by 2.
9   170 mod 9; and an Integer Number with the 6th as value.
10  leftover it by 3.

```

Line 1 comprises the point of entry to the program, while Line 2 creates an instance for *Number* and assigns the value of 5. It should be noted that the instances are intended to be created and their attributes assigned according to the programmer's needs, be this in terms of the definition of the noun, in order to give it predetermined values, or during instantiation as part of the preposition *with*. On Line 3, 2 is added to the instance, where, in this case, the expression *the Number* refers to the first (and, up to this point, only) instance. Another instance derived from the verb *plus* results from adding 3 to the first instance, in which case, the instance will have a value of 10. (The conceptual difference between the verbs *add* and *plus* is created on Line 4. This is the first addition to the instance, while the second returns a new instance of said addition, without altering the previous instance.) On Line 5, a third instance derived from the verb *minus* is created and results from subtracting the value of the first instance (7) from the second (10), which means that the third instance will have a value of 3. On Line 6, a fourth instance is created, whose value is that of the third instance (3). On Line 7, 1 is subtracted from the fourth instance (3), which finishes with a value of 2. On Line 8, the fourth instance (2) is divided by 2, giving a value of 1. Line 9 creates an instance whose value will be 8, the remainder of the division of 170 by 9. Finally, Line 10 creates an instance whose value is equal to the remainder from the previous instance (8) on being divided by 3, the result of which is 2.

As can be seen, in this case, it was possible to manage without the use of identifiers and, in their place, indirect references to instances were used according to the moment at which they were instantiated. Furthermore, this study presents a mechanism for instantiation and assignation that simplifies the definition of constructors, which enables the description of nouns with a level of detail that will depend on the needs of the programmer. The equivalent of said code in Scala is the following:


```

1 class Operations extends App {
2   val num1 = new Number with Integer
3   num1.value = 5
4   num1.add(2)
5   val num2 = num1.plus(3)
6   val num3 = num2 minus.(num1)
7   val num4 = new Number with Integer
8   num4.value = num3.value
9   num4.subtract(1)
10  val num5 = num4.divide(2)
11  val num6 = 170.mod(9)
12  val num6 = new Number with Integer
13  num7.value = num6
14  num7.leftover(3)}

```

As can be seen, identifiers were used for correctly working with the instances. In this case, constructors were not used with arguments, while the assignation of the attribute *value* was undertaken in a separate instruction.

6.2. Case 2: Database Connection

In this scenario, an example of connection to a database by means of naturalistic queries is presented. SQL is a query language with a high level of expressiveness, a feature which facilitates implementation in the SN language without major problems. Two very simple abstractions are presented below, *Car* and *Person*:

noun Car:

```

attribute model as a String.
attribute maxSpeed as a Integer Number.

```

noun Person:

```

attribute name as a String.
attribute age as a Integer Number.

```

To enable a separation between the database connection and the nouns, the adjective *Storable* is presented, thus refining the adjective *DBEntity*.

```

1 adjective Storable is DBEntity:
2   circumstance: execute assign this when this is created.
3   verb assign itself:
4     the driver is ‘‘org.postgresql.Driver’’.
5     the jar is ‘‘C:\\pgsql.jar’’.
6     the user is ‘‘admin’’.
7     the password is ‘‘admin1’’.
8     the URL is ‘‘jdbc:postgresql:
//localhost:5432/agenda’’.

```

Based on these abstractions, the following example presents a mechanism that enables three to be converted into database entities with the ability to query, insert, delete, and modify information:

```

1 main Select:
2   a DBPersistent String with ‘‘Person1’’ as value.
3   an Storable Person with the String as name.

```

```

4  some Strings.
5  add ‘‘name’’ to these.
6  add ‘‘age’’ to the Strings.
7  select the Strings from the Person.
8  System prints these and newline.

```

Line 1 describes an abstraction *main*, which is the starting point of any program in SN. Line 2 describes the instance of a noun *String* that is complemented with an adjective *DBPersistent*, which turns the string into an abstraction storable in an attribute of a database tuple. Line 3 indicates the creation of an instance of the noun *Person* that is complemented with *Storable*, in order that this instance functions as an element equivalent to a tuple of a database. Line 4 indicates the creation of a plural type *Strings*, which will store the attributes of the table to be accessed. Lines 5 and 6 add the strings *name* and *age*, which are the attributes of the table *Person* that you want to extract from the database. On Line 7, the database query is made by means of the verb *select*, which returns a plural of the type *Person*, if a plural is not defined, and the type *Things*, the root of plurals, is then returned. Finally, Line 8 prints the values. Other database-related operations are shown in Appendix C.

6.3. Case 3: Encoding Process

This scenario is based on the example proposed in [5], as it is a representation of code written in Java. Given that it only describes an approximate representation, there is no guarantee that it can work without a profound analysis of the API being carried out in order to translate it into the SN language and, thus, present a functional example. Firstly, description is given of three nouns which possess verbs tasked with reading, writing or cleaning data. The main example is presented below.

```

1 noun Encoder:
2  verb encodeDuration bytes as Byte Numbers in itself.

1 noun InputStream:
2  verb read bytes as Byte Numbers from itself.
3  verb empty itself.

1 noun OutputStream:
2  verb write bytes as Byte Numbers into itself.
3  verb empty itself.

1 noun Example:
2  verb itself encodeStream input as InputStream and output as OutputStream:
3    an Encoder.
4    some Byte Numbers.
5    repeat the next 3 instructions until empty input.
6    some Byte Numbers; read the Byte Numbers from input;
   and add it to the Byte Numbers.
7    encodeDuration these in the Encoder.
8    write it into output.
9    execute the next instruction when read the Byte Numbers from
   input are lesser than the first Byte Numbers.
10   this patch the last Byte Numbers and 0.
11  verb itself patch bytes as Bytes and number as Byte Number.

```

The following description focuses solely on the noun *Example*. As can be seen, an instance of *Encoder* is created on Line 3, while an instance of *Numbers*, which is also delimited by means of the adjective *Byte*, is created on Line 4. Line 5 describes an iterator which repeats the following three

instructions until *input* is empty. Line 6 shows the creation of another instance of *Byte Numbers*, which receives the value of the result of the verb *read* for *input*, which receives the first instance of *Byte Numbers* as a parameter. Line 7 executes *encodeDuration*, which receives the instance of *Byte Numbers*, which was created in the previous line and is invoked by means of the indirect reference *these*. The result of Line 7 is written in *output* on Line 8. Line 9 describes a conditional, which indicates that the subsequent instruction will be executed if the bytes which were read from *input* are lower than the value of the first instance of *Byte Numbers*. Line 10 calls to the verb *patch*, which receives the last instance of *Byte Numbers* that had been read, while Line 11 provides the definition without implementing the verb.

7. SN Evaluation

As seen in the scenarios presented in the previous section, SN's grammar enables work with indirect references, as well as reducing the use of identifiers, as mentioned by Lopes et al. [5], while maintaining a grammar similar to the English language.

It should be noted that evaluating expressiveness from the perspective of natural language is practically impossible. This is because, as mentioned in [28], there is no standardized mechanism for evaluating translation between *man pages* (the software documentation for systems based on UNIX), i.e., there is no mechanism reported that objectively evaluates a program and what it is expected to do and then describes these intentions in natural language. In addition, the mechanisms reported in other studies reviewed focus on the code that resulted from the translation (where applicable) or, in its absence, experimental tests conducted on subjects with different levels of programming knowledge (ranging from children to software architects and programmers). For this reason, instructions and code fragments were selected to cover the largest number of valid SN constructions, in order that students with different levels of experience in both programming and English are subsequently able to evaluate their expressiveness from the perspective of natural language.

7.1. Programming Example

As part of the debugging process, an undergraduate student with knowledge of objects and, therefore, presenting object-oriented reasoning, performed the first tests on the SN language. As a result, the student designed exercises, in which the problems posed were solved but which were not naturally expressive. Presented below is one of these exercises, which consists of a mechanism for describing mathematical expressions naturalistic, with, as can be seen, an object-oriented cognitive process used for the programming, therefore differing from what is expected in a naturalistic language:

```
noun SpeedFormula:
  attribute speed is 0.
  verb itself calculateSpeed distance as Number and time as Number:
    speed is distance divided by time.
  System prints speed.
```

As can be seen, while the example works correctly, the name of the abstraction is a composition using an adjective with a noun. While this is not significantly problematic, it is based on the human cognitive process and, thus, is a non-naturalistic description. The same happens with the verb, which is the union of a verb and an adjective. An example of its use is given below:

```
main CalculateV:
  System prints "Write the distance: " and newline.
  System reads.
  dst is integer of it.
  System prints "Write the time: " and newline.
  System reads.
  time is integer of it.
```

```

System prints dst and ‘/’.
System prints time and newline.
fv is a SpeedFormula.
fv calculateSpeed dst and time.

```

A naturalistic approach is presented below:

noun Formula:

```

circumstance: this must be Percentage or Speed.
circumstance: Percentage and Speed are mutually excluded.

```

adjective Speed:

```

attribute distance as a Real Number.
attribute time as an Real Number.
derived attribute result as a Real Number:
distance / time; and return it.

```

As can be seen above, the noun was decoupled from the adjective, thus maintaining the context of the sentences while providing an extension mechanism where another adjective that only shares the attribute *result* is added, with this adjective returning the result of the operation. The following example is presented:

adjective Percentage:

```

attribute quantity as a Real Number.
attribute percent as an Integer Number.
derived attribute result as a Real Number:
aux is the real of percent; aux / 100; quantity * it; and return it.

```

There are now two adjectives that enable work with two types of formulas. The use of both formulas is presented below:

main Main:

```

a Speed Formula with 10 as distance and 5 as time.
System prints the result of it and newline.
speed is a Speed Formula with 10 as distance and 5 as time.
System prints the result of speed and newline.
System prints the result of a Percentage Formula with 10 as percent
and 500 as quantity and newline.
a Percentage Formula with 10 as percent and 500 as quantity and newline.
System prints the result of it.

```

Although the naturalistic version contains more code, it is, in turn, more readable according to the context of the program while also enabling the extension of the functionality of *Formula*. Based on both the foregoing and that reported in [37,38], users were not permitted to perform their own exercises. These studies mention that people will try to construct sentences in natural language without considering the language restrictions (in this case, those of SN). These two studies report problems in both the translation process and the interpretation of the text, because the participants used ambiguous sentences or ones with elements not permitted by the tools reported by the authors.

7.2. Questionnaire

An analytical evaluation was chosen instead of a practical one, in order that prior experience with the paradigm did not bias the answers, as, during the initial tests of SN, it was observed that those who work with SN without experience in a naturalistic context seek to shape ideas according to the

dominant paradigms. For example, SN enables work with syntax that is closer to OOP, but this is achieved at the cost of sacrificing the naturalistic expressiveness expected from SN. Based on the above, it is observed either that a process of “unlearning” the dominated paradigm is required or that subjects have no prior programming knowledge in order to reduce bias in the evaluation of SN expressiveness. On the other hand, were participants to be solely taught to program in SN, this would create another bias where already knowing the paradigm and syntax would mean that subjects would consider the code expressive, not because of its naturalness, but because of their own previous experience.

The tests were performed using a Likert scale-based questionnaire [39] because no standardized mechanism is reported for comparing the correctness of code to its description in natural language [28]. For example, Chomsky [13] presented the sentence “colourless green ideas sleep furiously” which, while grammatically correct, lacks meaning. In fact, this sentence can be implemented in SN with only minor modifications, giving “the Colourless and Green Ideas sleep furiously”.

Undergraduate students from the computer systems engineering degree program of the *Tecnológico Nacional de México (National Technological Institute of Mexico), campus Orizaba* participated in the tests, in which 49 students reviewed examples of programs written in SN. Students ranged in age from 18 to 32, with, when asked to estimate their level of programming knowledge, 46.9% considering themselves as having a medium level, 2% considering themselves as having a high level, and 51.1% considering themselves as having a low or zero level. In terms of their mastery of English, 8.1% of the students considered themselves as having a high level, 53.1% considered themselves as having a medium level, and 38.7% considered themselves as having a low or zero level.

A scale was established with values between 1 and 4, where 1 is *not expressive* and 4 is *totally expressive*. The tests ranged from analyzing simple instructions to understand their meaning and context to analyzing complete programs that perform tasks such as arithmetic operations via English sentences, defining nouns, adjectives and their combinations, or database management. Questions 4 and 5 consisted of two true or false questions.

The Likert scale was adapted to four alternatives, as options such as “very expressive” were considered too diffuse to be distinguished from the “totally expressive” or “self-documented” option, which already covers the positive range, while, at the opposite extreme, the difference between “little expressive” and “inexpressive” is more evident. In addition, an explicitly neutral option using “moderately expressive” was retained. The list of questions, with their respective percentages, can be found in Appendix D.

Table 1 presents a summary of the results for the questions regarding expressiveness levels, while Table 2 presents the percentages of answers given to the two true or false questions.

As noted in Table 1, the expressiveness level of the language tends to be moderate when students without language knowledge are asked to evaluate instructions, presenting a tendency to *little expressive* when examples are shown with several indirect references, such as in Question 19. This is an expected result, derived from the complexity of understanding instructions in which identifiers are dispensed with, while questions such as Question 3 revealed that a simple instruction with indirect references is understood at least moderately by more than half of the students, a situation repeated in Questions 6 and 7. It should be noted that Question 8, which, together with Question 16, obtained the most positive results of all questions, shows SN code and its equivalent in Java, while Question 16 refers to the use of *instead* in a circumstance. It is possible that Question 8 had such a high positive percentage because the students were able to compare the code with its counterpart in Java.

Table 2 presents two questions, where Question 4 asked students to correctly choose from one of the two options that print the sentence based on how they are presented on screen, while Question 5 asked students to identify another value printed from the example shown in Question 4. A little more than half of the students answered correctly, which also shows that indirect references, although not entirely obvious, are not an unknown concept either.

The remarks made by the students show that, as Spanish is their native language, it was difficult for them to solve complex elements with indirect references, in addition to the fact that many were

looking for an implementation similar to languages such as Java, PHP, C# or Haskell, which are known to them.

Table 1. Question Results.

Question	Inexpressive/ Not Useful	Little Expressive Little Useful	Moderately Expressive Moderately Useful	Highly Expressive Very Useful
1	10.2%	36.7%	40.8%	12.2%
2	2%	38.8%	51%	8.2%
3	8.2%	34.7%	36.7%	20.4%
6	10.2%	34.7%	49%	6.1%
7	2%	38.8%	46.9%	12.2%
8	4.1%	34.7%	38.8%	22.4%
9	2%	26.5%	57.1%	14.3%
10	4.1%	36.7%	46.9%	12.2%
11	4.1%	32.7%	51%	12.2%
12	10.2%	28.6%	49%	12.2%
13	6.1%	32.7%	38.8%	22.4%
14	14.3%	32.7%	40.8%	12.2%
15	14.3%	20.4%	57.1%	8.2%
16	6.1%	30.6%	40.8%	22.4%
17	8.2%	28.6%	53.1%	10.2%
18	0	8.2%	59.2%	32.7%
19	4.1%	42.9%	40.8%	12.2%
20	10.2%	22.4%	59.2%	8.2%
21	6.1%	22.4%	55.1%	16.3%
22	0	20.4%	59.2%	20.4%

Table 2. True/False Questions.

Question	Yes	No
4	55.1%	44.9
5	51%	49

8. Conclusions and Future Work

The conceptual model presented in this article is derived from the analysis undertaken on the elements that provide the most relevant information in a sentence written in English. In addition, a tendency to use one or more of the elements identified, such as indirect references, pronouns or ellipsis, was observed in several related studies. The model elements were selected to enable the design of general-purpose naturalistic programming languages that are expressive from a natural language perspective and do not lose their formality, without requiring data dictionaries or artificial intelligence techniques.

There is one great disadvantage to the excessive use of indirect references, namely the complexity for the programmer of analyzing the code. For this reason, they were also included in the model, although not as crucial elements, because alternative ways of working with unique values, such as unrepeatable attributes for instances, can be defined in a similar way to the *primary keys* of SQL.

Based on the questionnaire answered by the students and the research carried out, it was concluded that the conceptual model does not require more elements and serves as a basis for defining the features that a general-purpose naturalistic programming language should possess. Moreover, the questionnaire obtained a high level of expressiveness for SN in students who had no knowledge of it, while the main criticisms recorded were focused on the verbosity of the language and reflect a marked resistance to a new paradigm, a common occurrence in the history of programming languages.

Based on the scenarios used to evaluate the correctness of SN code, which were added to the questionnaire completed by the students, it was observed that SN contributes to reducing the gap between the problem and solution domains, but only when the source code is written using constructions similar to the English language. This results in a self-documented code, but one that

is more extensive and faces the significant design challenge of maintaining naturalness. As a result, SN requires a learning process that focuses on not only knowing the grammar but also adapting the cognitive process, because, although SN enables the programming of abstractions similar to those found in Java or Scala, it loses the naturalness of the instructions, which is its main objective.

As a concept test, while SN reflects a high level of expressiveness compared to the English language, it lacks sufficient optimization for use in real environments. The combination of adjectives and nouns during instantiation time enables abstractions to be decoupled, which increases modularity. The use of indirect references increases the naturalness of the code, but, in turn, complicates the handling of a large number of instances, although SN allows the use of identifiers to solve this problem. Finally, the use of circumstances enables the definition of a context for nouns, adjectives, attributes and verbs, which increases the modularity of the code by encapsulating elements that, in other paradigms, are scattered among the abstractions, with the exception of AOP. However, offering a syntax closer to English, as the circumstances are based on the syntax in a similar way to AspectJ pointcuts, means that the robustness in the definition of verbs that this entails contributes to maintaining control of the circumstances.

In future research, SN grammar will be refined for the conceptual model to be more completely validated. As part of this refinement, support will be given in order that programmers are able to define embedded grammars in describing elements of a particular domain, as, in scientific documents, formal language is separate to natural language. Naturalistic iterators and conditionals will be analyzed to verify whether they have the adequate level of naturalness. The coherence of the indirect references will be evaluated to avoid incoherent syntax instructions such as *add 2nd to the Numbers*.

Furthermore, as Scala and AspectJ were used as intermediary languages to facilitate the analysis of the compilation process and ensure the correctness of the code (which was achieved), the compilation process will be modified to generate bytecode without intermediary languages. In this way, SN will be an independent language, which will improve its optimization and, in turn, will enable the testing of complex applications. Finally, as the construction, analysis and verification of SN was conducted by generating Scala source code, the formal evaluation of the language was postponed because Scala is an already validated language, while the correctness of the intermediate source code was validated via experimentation. With this in mind, a formal evaluation of SN will be performed.

Author Contributions: Conceptualization, O.P.-P. and U.J.-M.; Formal analysis, O.P.-P.; Investigation, O.P.-P.; Methodology, O.P.-P.; Software, O.P.-P.; Supervision, U.J.-M.; Validation, O.P.-P.; Writing—original draft, O.P.-P.; and Writing—review and editing, O.P.-P. and U.J.-M.

Funding: This research received no external funding.

Acknowledgments: The authors would like to thank the Consejo Nacional de Ciencia y Tecnología (CONACyT) for the support provided for the development of this research.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ACE	Attempto Controlled English
AOP	Aspect-Oriented Programming
API	Application Programming Interface
CPL	Computer-Processable Language
JVM	Java Virtual Machine
LTW	Load-Time Weaving
NLC	Natural Language Computer
NLCI	Natural Language Command Interpreter
OOP	Object-Oriented Programming
SBT	Simple Build Tool
CONACyT	Consejo Nacional de Ciencia y Tecnología

Appendix A. Conceptual Model Description

This section presents a detailed description of the elements of the conceptual model.

Appendix A.1. Noun

In natural languages, the noun is the word used to identify a concrete “thing” or a set of “things”. The noun is the principal part of the grammatical subject (*the Number described before*), the complement of a verb (*the System prints the Number*) or the object of a preposition (*add 5 to the Number*). A noun has characteristics (which, in turn, are nouns) and undertakes actions (defined as verbs); moreover, in the English language, a noun possesses a grammatical number, which enables it to be used in the plural (*Numbers*) or the singular (*Number*) form. A naturalistic abstraction that enables the use of descriptions according to the number and use of types offers a mechanism for defining instructions by means of indirect references. Based on the foregoing, the noun described in the naturalistic conceptual model takes the properties of the noun from the English language. (In the romance languages, the noun also has gender, which serves to indicate which article is to be used to complement it. Gender is not considered for the design of the conceptual model proposed in this study, as, in English, nouns are used with a gender neutral article.) A noun is a word with a particular meaning that designates a common name for a thing or concept, which is either tangible or intangible. It undertakes actions that are reflected in the form of verbs, or functions as an object or complement of the verb.

Based on the definition of a noun, the abstraction that it represents requires the capacity to describe plurals, which, in natural languages, is achieved by means of numerals (either numbers or ordinals) or qualifiers (*all, some* or *none*). One problem consists in the fact that English contains nouns whose written form changes in the plural, as in the case of *child* and *children*. The conceptual model considers whether or not the name of the abstraction represents a plural, as there are words which represent groups of entities, such as *list* (which indicates a list of entities) or *block* (in the context of a group of houses).

Further to the foregoing, a noun requires diverse properties, which, in turn, are other nouns which both form it and give it identity. Said properties are also used to create particular instances for the noun itself or to identify it among the various instances of the noun. It should be noted that, in the context of the noun and, thus, naturalistic programming, the instance is the conceptual equivalent of the linguistic term *significant*, given that an instance is the representation of an idea abstracted from a referent, which is a real concept.

For example, *Number* is a noun which describes an abstraction used to represent magnitudes, measurements or orders. Despite the diverse types of *Numbers*, the noun encompasses all of them without the need to go into particular details.

Appendix A.2. Adjective

The adjective was taken as an abstraction, given that it is used to complement the noun in the natural languages. An adjective provides the noun properties which, rather than common to all entities, are common to only some in particular. According to the grammatical form, three forms of the adjective are considered, which are the attributive (indicating the characteristic of a noun: *the integer*), predicative (linked to the noun with a copulative verb: *the number is an integer*), or nominal (acting as a noun: *show all numbers and print the integer*).

For example, *Natural* is an adjective used to complement the noun *Number*, meaning that, when one talks about a *Natural Number*, one makes reference to a particular concept and leaves others to one side, such as *Integer* or *Complex Number*.

In languages such as English, an adjective also has superlative and comparative forms, with the former describing a characteristic to its maximum degree, such as *the greatest Number*, which implies that the noun has to know which of its instances coincide with the adjective. On the other hand, the comparative adjective enables two nouns of the same type to be compared, for example *number A*

is *greater than number B*. As can be seen in this case, the verb is the word *is* complemented by the comparative adjective *greater*, which implies that the noun is also required to verify which is greater. However, this case would only require the verification of the greater of the two instances that are being compared.

Appendix A.3. Verb

The noun is expected to respond to actions represented by verbs, meaning that the verb is equivalent to the functions of other paradigms, such as functional programming or the methods of OOP. In English, a verb is the nucleus of the verb phrase and is divided into two types: finite, when the nucleus is a finite verb; and non-finite, when the nucleus is a non-finite verb, namely the infinitive, participle and gerund.

The conceptual model proposed in this article only considers the use of verbs in their simple or participle form in order to simplify its implementation in grammar, with the phrase such as *the number added to the list* preferred to more complex structures, such as *the number has been added to the list*. Moreover, the use of ellipsis (while formal English requires a noun or pronoun to be specified before the verb, in an informal context, people talk to each other (or other non-human entities) using sentences with an elliptic subject, such as *take it*, whose correct form is *you take it*) at least requires that the subject of the sentence appears in the same sentence. For example, in the context of a list of numbers from which one seeks to identify the third element, the expression *get the third number* contains an indirect reference to the person executing the instruction, who is classified as the subject *elliptical*.

According to the literature, the authors of various studies have been found to use the elliptical subject to indicate to the system which instructions to execute. For example, *get the third number from the stack* implies that the programmer must wait for the system to obtain the third number from the stack. An equivalent of said instruction that is oriented to objects is *theStack.get(2)*, in which *theStack* is an identifier that indicates a list containing numbers.

For example, an *Integer* provides mechanisms for expressing sentences that exercise an effect on the same or other integers, meaning that the expression *add 25 to the Integer Number* implies the description of a mathematical operation expressed in a natural language.

Appendix A.4. Property-Based Typing

Property-based typing makes it possible to use not only nouns and adjectives to identify a particular entity, but also the properties that the entity possesses, whether the entity holds a particular value for a specific property or the property exists in the entity even though it does not have an assigned value.

Appendix B. SN Grammar

The full grammar of SN is presented in this section.

Appendix B.1. Program Structure

In SN, a compilation unit is composed of a package definition, which is a list of types imported with a nickname used to avoid expressions such as *A is a naturalistic.lang.Integer naturalistic.lang Number with 7 as value* when two types share a name but a different namespace (or context).

```

program ::= package
          imports
          abstractions

package ::= 'package' full_id '.' '\n'

imports ::= import
         | imports import

```

```
import ::= 'import' full_id 'as' ID '.' '\n'
```

An example containing a package definition and an import definition is given below:

```
package shop.printer.
import naturalistic.lang.System as System.
import naturalistic.sql.DBEntity as Entity.
```

The basic abstractions are *adjectives*, *nouns* and execution points for a program represented with the special abstraction *main*.

```
abstractions ::= noun_def '\n'
              | adjective_def '\n'
              | main_def '\n'
              | abstractions noun_def '\n'
              | abstractions adjective_def '\n'
              | abstractions main_def '\n'
```

The rule *abstractions* is composed of the rules *noun_def* and *adjective_def*, which are explained later in this section.

Appendix B.2. Main

The reserved word *main* is used to describe execution points, which are abstractions used for executing an SN program, in a similar way to methods used in Java and C *main*.

```
main_def ::= 'main' ID ':' '\n' abstraction_body

abstraction_body ::= attribute_def
                  | derived_attribute_def
                  | verb_def
                  | circumstance_def
```

The rule *abstraction_body* is composed of the rules *verb_def*, *attribute_def*, *derived_attribute_def* and *circumstance_def*, which are defined later in this section.

Appendix B.3. Noun

Based on the model, the reserved word *noun* is used followed by the name of the noun and, finally, the instruction finishes with a period. Optionally, the construction “with plural” is added to indicate a plural noun. The formal grammar for describing a noun is as follows:

```
noun_def ::= noun_signature '.'
          | noun_signature ':' '\n' abstraction_body

noun_signature ::= 'noun' ID
               | 'noun' ID plural_def

plural_def ::= 'with' 'plural' 'as' ID
```

As can be seen, the grammar enables abstractions to be defined via a variant for working with plurals. In the case of SN, the plural is considered a non-sorted set with repeated values, as it is expected that these details are taken into account during the creation of plural instances.

In the natural languages, a noun can be used as a more specialized form of another noun, which enables humans to express themselves using a level of “hierarchy”, given that all nouns are *things*. Based on the foregoing, a noun is refined in order to provide more concrete behavior, creating a hierarchy of types by means of the words *is a*, as shown in the following grammatical rule:

```
noun_signature ::= 'noun' ID
                | 'noun' ID plural_def
                | 'noun' ID 'is' a_an ID
                | 'noun' ID 'is' a_an ID plural_def
```

Appendix B.4. Adjective

The SN language uses a form of *mixin* composition similar to that observed in Scala and used in both the definition of the noun and its instantiation, which enables the definition of adjectives that complement the behavior of the nouns during instantiation. The grammar of an adjective is shown below:

```
adj_def ::= adj_signature '.'
         | adj_signature ':' '\n' abstraction_body
adj_signature ::= 'adjective' ID
               | 'adjective' ID 'is' adj_ex
adj_ex ::= ID
        | adj_comma 'and' ID
adj_comma ::= ID
           | adj_comma ',' ID
```

The notation indicates that the name of the adjective goes before the name of the noun. Similar to the paradigms oriented to objects, adjectives are also used in a similar manner to a composition mechanism during the design stage. Based on the foregoing, the following grammar defines a noun which combines with an adjective during its definition:

```
noun_signature ::= 'noun' ID
                | 'noun' ID plural_def
                | 'noun' ID 'is' a_an ID
                | 'noun' ID 'is' a_an ID plural_def
                | 'noun' ID 'is' a_an adj_ex ID
                | 'noun' ID 'is' a_an adj_ex ID plural_def
                | 'noun' ID 'is' adj_ex
                | 'noun' ID 'is' adj_ex plural_def
```

Appendix B.5. Attribute

A noun comprises attributes that, in turn, are nouns. To describe the elements that comprise a noun, the use of a colon (:) is required when the noun is defined, while the subsequent line begins with a tabulator. The reserved word *attribute* indicates that an attribute will be defined and that it uses the word *as* to establish the type. The grammar for working with attributes is presented below:

```
attribute_def ::= '\t' attribute_signature '.'
attribute_signature ::= 'attribute' ID
                    | 'attribute' ID 'is' attr_val
                    | 'attribute' ID 'as' noun_phrase
```

A noun that describes the contract for the sale, by a realtor, of a house with two attributes—one, a *Client*-type and another, a *House*-type—is presented below:

```
1 noun Contract:
2   attribute buyer as Client.
3   attribute property as House.
```

The manner by which access to the attributes of a noun is achieved is as follows:

- 1 a Client with ‘John’ as name.
- 2 a Luxurious House with 1,000,000.00 as price.
- 3 a Contract with the House as property and the Client as buyer.
- 4 System prints the buyer of the Contract.
- 5 System prints the property of the Contract.

As can be seen, access to the attributes enables the creation of instructions which function conceptually as direct complements, while the noun that contains them becomes the indirect complement of a sentence. It should be noted that said construction is the equivalent, in Java, to `System.out.print(contract.buyer)`, where `contract` is an instance for the class `Contract` in which the field was defined `buyer`.

Appendix B.6. Verb

Given that a sentence comprises a subject and predicate, a verb that complements the noun is required, indicating which actions it undertakes and whether or not it alters both its own state and that of other nouns. Generally, programmers express themselves using constructions in which it is expected that it is the computer executing the action (from a grammatical perspective). This study proposes a mechanism that indicates where, in the instruction that calls the verb, the noun containing the verb can be found. In this case, said position is established by means of the word *itself*. The rules tasked with describing this class of instruction are presented below:

```
verb_def ::= '\t' verb_signature ':' instructions
          | '\t' verb_signature '.' '\n'

verb_signature ::= 'verb' ID 'itself'
```

Their implementation in the SN language is the following:

```
noun House:
  attribute availability is true.
  verb sell itself:
    available is false.
```

As can be seen, the reserved word *verb* which precedes the name of the verb is used, while, after the name, the position occupied by the noun is defined when it is used in an instruction. In this case, it is indicated that the noun acts as the *direct object* of the verb:

```
a House.
sell the House.
```

where the sentence *sell the House* indicates an instruction in which the verb is *sell*, while *the House* indicates the noun invoking it, and which, in the signature, is defined as *itself*.

Given that the reference to the noun itself is found in the signature, it also varies its position depending on the type of sentence, which is formed by changing the position in order to create signatures with different complexities. In this case, a definition of the verb is presented, in which the noun it defines acts as an *indirect object*, while the argument is a *direct object*. Based on the foregoing, the rule *verb_signature* is observed in the following form:

```
verb_signature ::= 'verb' ID 'itself'
                | 'verb' ID preposition 'itself'
                | 'verb' ID args preposition 'itself'
```

```

args ::= ID 'as' noun_phrase
      | args_comma 'and' ID 'as' noun_phrase

args_comma ::= ID 'as' noun_phrase
            | args_comma ',' ID 'as' noun_phrase

```

It is implemented as follows:

```

1 noun House:
2   attribute owner as Client.
3   verb add buyer as Client to itself:
4     the owner of this is buyer.

```

while its invocation is shown in the following example:

```

a House.
a Client.
add the Client to the House.

```

where *add* is the verb, *the Client* is the direct object, and *the House* is the indirect object, which also contains the instruction and is, therefore, represented in the signature with *itself*.

Finally, support is provided for defining verbs where the noun that contains them acts as the *grammatical subject* in a form similar to the syntax of languages such as Java. The rule integrating this kind of construction, *verb_signature*, is presented below:

```

verb_signature ::= 'verb' ID 'itself'
                | 'verb' ID preposition 'itself'
                | 'verb' ID args preposition 'itself'
                | 'verb' 'itself' ID args
                | 'verb' ID args

```

This last case is notable, given that the construction *'verb' ID args*, by default, is considered as *'verb' 'itself' ID args*. An example of its use is presented below:

```

noun House:
  attribute availability is true.
  verb itself sold:
    availability is false.

```

This is used in the following manner:

```

a House.
the House sold.

```

where the verb *sold* is the predicate of the sentence, while *the House* is the grammatical subject, which, in turn, contains the verb, with the signature represented by *itself*.

Appendix B.7. Circumstance

As the circumstances were described in the main text, this section only presents the grammar pertaining to the implementation of the concept. Circumstances are divided into three: instance circumstances; attribute circumstances; and verb circumstances.

```

circumstance_def ::= 'circumstance' ':' instance_circumstance
                  | 'circumstance' ':' attribute_circumstance
                  | 'circumstance' ':' verb_circumstance

```

Appendix B.7.1. Instance Circumstance

The grammar is as follows for circumstances associated with instances:

```

instance_circumstance ::= cannot_circumstance
                        | mutex_circumstance
                        | requires_circumstance
                        | i_circumstance

requires_circumstance ::= abstraction_name
                        'cannot' 'be' type_list

abstraction_name ::= 'this' | ID

type_list ::= ID
           | adj_comma 'and' ID
           | adj_comma 'or' ID

mutex_circumstance ::= noun_phrase 'are' 'mutually' 'excluded'

requires_circumstance ::= abstraction_name
                       | requires type_list

i_circumstance ::= 'execute' verb_c_call
                c_opt 'this' 'is' 'created'

attribute_circumstance ::= 'execute' verb_c_call
                        c_opt
                        'this' 'is' 'created'

verb_c_call ::= ID c_obj
            | ID preposition c_obj
            | ID c_objs preposition c_obj
            | c_obj ID c_objs

c_objs ::= c_obj
       | c_objs_comma 'and' c_obj

c_objs_comma ::= c_obj
              | c_objs_comma ',' c_obj

c_obj ::= 'it'
       | 'this'
       | literal
       | c_access_attr
       | ID

c_access_attr ::= c_nested_attr c_obj

c_nested_attr ::= 'the' ID 'of'
              | c_nested_attr ID 'of'

c_opt ::= 'when' | 'before' | 'after'

```


Appendix B.7.2. Attribute Circumstance

The grammar is as follows for circumstances associated with attributes:

```

attribute_circumstance ::= execute_c_verb
                        | attr_must_cond

execute_c_verb ::= 'execute' circ_verb_call c_opt
               c_attr_access_s 'is' c_e_v_opt

c_e_v_opt ::= 'assigned' | c_attr_objs
           | relational_opt
           c_attr_access_s

not ::= 'not' |

relational_opt ::= not ID 'than'
               | not ID 'or' 'equal' 'than'
               | not 'equal' 'to'
               | 'distinct'

c_attr_objs ::= c_attr_objs_comma 'and' c_attr_obj
             | c_attr_objs_comma 'or' c_attr_obj

c_attr_objs_comma ::= c_attr_obj
                  | c_attr_objs_comma ',' c_attr_obj

c_attr_obj ::= literal
            | c_attr_access

c_attr_access_s ::= c_attr_access
                | c_nested_attr 'something'

c_attr_access ::= c_nested_attr 'this'
               | c_nested_attr ID

```

Appendix B.7.3. Verb Circumstance

The grammar is as follows for circumstances associated with verbs:

```

verb_circumstance ::= 'execute' verb_c_call c_opt_v
                  verb_call_sign is_ex
                  | verb_c_call c_opt_v
                  verb_call_sign is_ex

c_opt_v ::= c_opt
         | 'instead'

is_ex ::= 'is' 'executed'
        |

verb_call_sign ::= 'something' ID s_a_s
                | 'something' ID preposition s_a_s
                | ID s_a_s preposition 'something'
                | 'something' preposition s_a_s
                | 'something' 'is' relational_v_opt
                ID

```

```

s_a_s ::= i_v_params 'and' ID
        | ID

i_v_params ::= ID
            | i_v_params ',' ID

relational_v_opt ::= relational_opt
                 | 'is' not
                 | 'equal' 'to'
                 | 'is' not
                 | 'distinct' 'to'

```

Appendix B.8. Derived Attribute

On occasions, attributes are needed whose value depends on other attributes or verbs, leading to the proposed definition of derived attributes, whose implementation is similar to that of a verb, but whose syntax and invocation are that of an attribute.

For example, if the formula for the area of a square ($R = S \times S$) is known, the definition is composed of the result (*result*) and the length of the sides (*side*), while the naturalistic expression for defining it is *the result of the Square*, where *result* is an attribute whose value depends on a calculation that, in turn, depends on the value of the sides.

The rules involved in said definition are described below:

```

derived_attribute_def ::= '\t' derived_attribute_signature
                       ':' instructions

derived_attribute_signature ::= 'derived' 'attribute' ID
                              'as' noun_phrase

```

This is thought to be because, as can be seen in the sections on attributes and verbs, the invocation depends on the signature of a verb, while the preposition *of* is used for the noun. Therefore, the attributes derived combine the flexibility of an attribute with the potential of a verb, enabling the definition of instructions where the value of an attribute is derived from others, as is the case with a noun that describes the price of a house on which a 15% tax is levied:

```

noun House.
adjective Priced:
  attribute price as a Real Number.
  attribute tax as a Real Number.
  derived attribute total as a Real Number:
    divide tax by 100.
    add 1 to it.
    multiply it by price.
    return it.

```

This is used in the following manner:

```

a Priced House with 200,000.0 as price and 15 as tax.
System prints the total of it.

```

and is printed as follows:

```

230,000

```

The implementation is more extensive because the noun was decoupled, with the implementation of said noun depending on an adjective while conceptually enabling the definition of new types of prices, as seen below:

```
adjective Discount:
  attribute price as a Real Number.
  attribute tax as a Real Number.
  attribute discount as Real Number.
  derived attribute total as a Real Number:
    divide tax by 100.
    add 1 to it.
    multiply it by price.
    subtract discount from it.
    return it.
```

This is used in the following manner:

```
a Discount House with 20000.0 as price, 5000.0 discount, and 15 as tax.
System prints the total of it.
```

and is printed at follows:

```
18,000
```

As can be seen, decoupling the implementation obtains descriptions that are more extensive but read naturally.

Appendix B.9. Plurals

While it is expected that plurals possess attributes and verbs, given that their definition depends on the noun they represent, their elements are defined in the same noun, with the reserved word *plural* placed first. The rule for attributes, verbs and derived attributes are presented below:

```
attribute_signature ::= 'plural' 'attribute' ID
                    | 'plural' 'attribute' ID 'is' attr_val
                    | 'plural' 'attribute' ID 'as' noun_phrase

verb_signature ::= 'plural' 'verb' ID 'itself'
                 | 'plural' 'verb' ID preposition 'itself'
                 | 'plural' 'verb' ID args preposition 'itself'
                 | 'plural' 'verb' 'itself' ID args
                 | 'plural' 'verb' ID args

derived_attribute_signature ::= 'plural' 'derived' 'attribute' ID
                              'as' noun_phrase
```

An example of their implementation is presented below:

```
noun House with plural as Houses:
  plural attribute sorted is false.
  plural attribute house_list as List
  plural verb add house as House to itself:
    house_list add house.
```

When bytecode is generated, at least two *class* files are created, one for the abstraction *House* and another for the abstraction *Houses*, with the components of each distributed to each file.

Appendix B.10. Noun Phrase

Noun phrases are grammatical constructions whose nucleus is the noun. It is considered that, for its implementation in the SN language, the constituents of a noun phrase are a noun and, optionally, one or more adjectives. The rules for defining noun phrases are as follows:

```
noun_phrase ::= ID
              | adjective_phrase ID

adjective_phrase ::= ID
                  | adj_comma 'and' ID

adj_comma ::= ID
            | adj_comma ',' ID

instance ::= singular_instance
           | plural_instance

singular_instance ::= a_an noun_phrase

a_an ::= 'a'
       | 'an'

plural_instance :: 'some' noun_phrase

np_with ::= 'with' with_attrs

with_attrs ::= with_attr
            | with_attrs_comma 'and' with_attr

with_attrs_comma ::= with_attr
                  | with_attrs_comma ',' with_attr

with_attr ::= obj 'as' ID
```

Based on the foregoing, the rule *singular_instance* takes the following form:

```
singular_instance ::= a_an noun_phrase
                   | a_an noun_phrase np_with

seek_instance ::= 'the' noun_phrase
                 | 'the' ordinal noun_phrase

ordinal ::= 'first'
           .
           .
           .
           | 'tenth'
           | 'last'

np_where ::= 'where' where_attrs

where_attrs ::= where_attr
             | where_attrs_comma 'and' where_attr

where_attrs_comma ::= where_attr
                   | where_attrs_comma ',' where_attr
```

```

where_attr ::= ID 'is' compared obj
compared ::= 'greater' 'than'
           | 'greater' 'or' 'equal' 'than'
           | 'lesser' 'than'
           | 'lesser' 'or' 'equal' 'than'
           | 'equal' 'to'
           | 'distinct' 'from'

```

The foregoing is integrated into the rule *seek_instance* in the following manner:

```

seek_instance ::= 'the' noun_phrase
              | 'the' ordinal noun_phrase
              | 'the' noun_phrase np_where
              | 'the' ordinal noun_phrase np_where

```

Finally, the following grammar is required for working with sets of elements (it should be noted that the clause marker *where* also applies for plurals):

```

seek_instance ::= 'the' noun_phrase
              | 'the' ordinal noun_phrase
              | 'the' noun_phrase np_where
              | 'the' ordinal noun_phrase np_where
              | 'the' 'first' integer noun_phrase
              | 'the' 'last' integer noun_phrase
              | 'all' noun_phrase
              | 'the' 'first' integer noun_phrase np_where
              | 'the' 'last' integer noun_phrase np_where
              | 'all' noun_phrase np_where

```

Appendix B.11. Sentence

The basic instruction is the sentence, formed by a subject and a predicate; however, there are sentences in which the subject is inferred from the context. In this class of sentences, it is said that the subject is *elliptic*. SN language enables work with this class of grammatical construction via a restriction that consists in the fact that the verb invoker will always be placed at end of the instruction, just after a preposition, such as in the sentence *sell the House*, where the system is inferred to be the elliptical subject. The grammar for the construction of sentences is presented below:

```

verb_call ::= ID subject
           | ID preposition subject
           | ID objs preposition subject
           | subject ID objs

objs ::= obj
      | objs_comma 'and' obj

objs_comma ::= obj
            | objs_comma ',' obj

obj ::= 'it'
      | 'these'
      | seek_instance
      | literal
      | access_attr

```

While, grammatically, the non-terminal symbol *subject* is the equivalent to the non-terminal symbol *obj*, this distinction is made in order to exemplify that which, at an implementation level, the compiler translates as the invoker and complements of the verb. Another detail to note is the reserved words *it* and *these*, where *it* refers to the last singular instance that was created or that was returned by the verb, while *these* refers to the last plural that was used or that was returned by a verb.

Appendix B.12. Access to Attributes

The following rules are defined for accessing attributes:

```
access_attr ::= nested_attr obj
nested_attr ::= 'the' ID 'of'
              | nested_attr ID 'of'
```

An example that becomes an integer in a string, and vice versa, is presented below:

```
an Integer Number with 4 as value.
the string of the Number.
a String with '4' as value.
the integer of the String.
```

To assign a value to an attribute, the reserved word *is* is used, followed by that which is sought to be assigned to the attribute. The rules are presented below:

```
nat_assign ::= access_attr 'is' obj
```

An example of the assignment of the value of an attribute is presented below:

```
a House with 40,000.00 as price.
the price of the House is 25,000.00.
```

Appendix B.13. Naturalistic Iterator

As naturalistic iterators were presented in the main text, this section only presents the rules.

```
iterator ::= 'repeat' 'the' iterator_signature iterator_condition
iterator_signature ::= ord_one 'instruction'
                    | ord_many numeric_literal 'instructions'
iterator_condition ::= obj 'times'
                    | 'until' condition
                    | 'for' 'each' list 'as' ID
list ::= these
      | seek_instance
      | access_attr
ord_one ::= 'previous'
        | 'last'
        | 'next'
ord_many ::= 'first'
          | 'previous'
          | 'next'
```

Appendix B.14. Naturalistic Conditional

A naturalistic conditional is defined as resembling how naturalistic iterators express themselves, as they possess the same characteristics and limitations, with the difference that, instead of indicating that a set of instructions is repeated N times, it indicates that the set of instructions will be executed only if a particular condition is fulfilled.

As naturalistic iterators were presented in the main text, in this section, only rules involved in naturalistic conditionals are presented.

```
decision ::= 'execute' 'the' decision_signature 'when' condition
decision_signature ::= ord_one 'instruction'
                    | ord_many numeric_literal 'instructions'
```

Appendix B.15. Composite Sentence

Languages such as Java are used to place instructions within instructions in order to reduce the number of lines of code, by, for example, making the call to method as an argument for another call to method. The SN language does not permit this class of construction due to the fact that the ambiguity of the code increases to untreatable levels without the use of more complex techniques, which is found to be beyond the reach of the model proposed. To resolve this problem, the SN language enables work with various instructions on one line, separated by a semi-colon (;), which the SN compiler will treat as one single instruction. However, this does not discount the fact that the use of these constructions lends a higher degree of naturalness to a naturalistic language. The rules that give life to this class of instructions are presented below:

```
composite_instruction ::= semicolon_instructions 'and' nested_instruction
semicolon_instructions ::= nested_instruction ';'
                       | semicolon_instructions nested_instruction ';'
nested_instruction ::= instance
                   | verb_call
                   | nat_assign
                   | nested_iterator
                   | nested_decision
```

Based on said rules, the equivalent of a Java instruction such as $A.b(C.d(E.f()))$ is expressed in SN, as shown below:

the E f; the C d it; and the A b it.

Omitting the fact that it replaces variables with indirect references, this example functions based on the indirect reference *it*, which saves the result of any previous instruction. In this case, the result of the sentence *the E f* is stored in the reference *it* and subsequently used as a direct object in the sentence *the C d it*. The result of this sentence is substituted for the value that was previously assigned to *it*. Finally, the sentence *the A b it* uses the indirect reference *it* again as a direct object, in this way returning the result of the sentence, which is also stored in the indirect reference *it*.

Appendix B.16. Composite Iterator and Composite Decision

It is common, in the natural languages, that, at the end of a paragraph, mention is made that the content of the text will be repeated based on a condition, or, on the contrary, that the text described in the paragraph is conditioned by something. For this reason, SN has inbuilt support for adding conditionals and iterators to the composite instructions, in order that these constructions indicate that what was written will be repeated or executed if the specified criterion is met. The rule that describes the iteration that occurs within a composite instruction is presented below:

```
nested_iterator ::= 'repeat' iterator_condition
```

The following rule is presented for working with conditionals within a composite instruction:

```
nested_decision ::= 'execute' 'when' condition
```

An example is presented below (the ; indicates that the subsequent instruction continues on the same line, although, due to questions of space, it is divided into two):

```
a Number.
add 1 to the Number; repeat 10 times;
and execute when the Number is equal to 0.
```

The approximate equivalent in Java for the previous instruction is shown below (*add 1 to the Number; repeat 10 times; and execute when the Number is equal to 0.*):

```
int theNumber = 0;
if(theNumber == 0) {
    for(int i = 0; i < 10; i++) {
        theNumber += 1;}}}
```

As can be seen, the conditional is found in the most external position, meaning that it will encompass all of the instructions, which, previously, have run from left to right. On conversion into code, it is analyzed and placed at the beginning of the set of instructions, which also occurs with the iterator. This can be seen, for example, in the following instruction:

```
a Number.
add 1 to the Number; repeat 10 times; subtract 8 from the Number;
and execute when the Number is equal to 0.
```

Its equivalent in Java is the following:

```
int theNumber = 0;
if(theNumber == 0) {
    for(int i = 0; i < 10; i++) {
        theNumber += 1;}
    theNumber -= 8;}
```

As can be seen, while the conditional includes all the instructions, the iterator only includes that which precedes it.

Appendix B.17. Local Identifiers

Object-oriented languages generally have a clear syntax in which the identifier is given first, followed by the method and, lastly, the arguments, in the event there are any. On the other hand, in a natural language, human beings use their reasoning to resolve the ambiguities derived from descriptions in which nouns and not phrases are used. This creates sentences that a naturalistic compiler could resolve incorrectly, meaning that sentences such as *print page* and *page read* cause an ambiguity from the perspective of a compiler lacking the cognitive process for understanding the meaning of each word. Various authors resolve this problem through the use of dictionaries, heuristics or directly by reducing the flexibility of the language. For example, a compiler lacking this capacity could misinterpret the following instruction:

```
divide A by B.
```


Using reasoning, a human being infers the previous sentence as “take *A* and divide it between *B*” using the preposition *by*, which could be used to indicate that the person executing the action is *B*, with said operation, as directed by the preposition, resulting in interpretation problems in cases such as the following:

A divided by B.

Again, while the human cognitive process helps to provide an understanding of the context, a compiler could translate it by taking *divided* and executing its verb *A* with the parameter *B*. For example, the SN compiler cannot understand if both instructions are translated into Scala code, as follows:

```
divide.itself_A_by_arg(B)
A.divided_by_arg(B)
```

or, in contrast:

```
B.divide_arg_by_itself(A)
B.A_arg_by_itself(divide)
```

The two previous examples exemplify the ambiguity of working with identifiers when seeking to integrate a linguistic element such as the deictic subject (*divide A by B*). Said problem was resolved by means of validation undertaken on the table of symbols, which resulted in them not being able to be used as identifiers for words that have been defined as verbs, given that the compiler will indicate a semantic error while also limiting the obligatory use of a word in both contexts.

To resolve said ambiguity problem, the decision was taken to limit the format of the verb, in order that the noun that executes it is a grammatical subject or an indirect object, thus discarding the ditransitive verb.

Based on the foregoing, some of the rules that are defined in this section receive a small modification in order that they support the use of local identifiers.

The assignation is modified in the following manner:

```
nat_assign ::= access_attr 'is' obj
            | ID 'is' obj
```

The arguments for the verb (and, therefore, the subject) are modified in the following manner:

```
obj ::= 'it'
      | 'these'
      | seek_instance
      | literal
      | access_attr
      | ID
```

Finally, the values that an attribute can take during its instantiation are as follows:

```
attr_val ::= instance
           | literal
           | ID
```

Appendix B.18. Complementary rules

```
full_id ::= ID
          | full_id '.' 'ID'
```

```
attr_val ::= instance | literal
```

```

instance ::= a_an noun_phrase

literal ::= ID | numeric | string | character

numeric ::= num | numeric num

num ::= '0' | '1' | '2' | '3' | '4'
      | '5' | '6' | '7' | '8' | '9'

string ::= '\'' char '\'''

character ::= '\' char '\'

char ::= UNICODE

preposition ::= 'a' | 'about' | 'after' | 'against'
              | 'among' | 'an' | 'around' | 'as' | 'at'
              | 'before' | 'between' | 'by' | 'during'
              | 'for' | 'from' | 'is' | 'in' | 'into'
              | 'like' | 'of' | 'on' | 'out' | 'over'
              | 'than' | 'through' | 'when' | 'with'
              | 'without' | 'to' | 'under'

```

Appendix C. Database Access with PostgreSQL

In this section, the scenario shown in Section 6.2 is extended to show examples of insert, update and delete for database operations.

Appendix C.1. Storable (Adjective)

```

adjective Storable is DBEntity:
  circumstance: execute assign this when this is created.
  verb assign itself:
    the driver is 'org.postgresql.Driver'.
    the jar is 'C:\pgsql.jar'.
    the user is 'admin'.
    the password is 'admin1'.
    the URL is 'jdbc:postgresql://localhost:5432/agenda'.

```

Appendix C.2. Person (Noun)

```

noun Person:
  attribute name as a String.
  attribute age as an Integer Number.

```

Appendix C.3. Insert

```

main Insert:
  a Storable Person with the first String as name.
  a DBPersistent String with 'John Doe' as value.
  the name of the Person is it.
  a DBPersistent and Integer Number with 25 as value.
  the age of the Person is it.
  some Strings.
  add 'name' to these.
  add 'age' to the Strings.

```

```

insert the Strings into the Person.
select the Strings from the Person.
System prints these and newline.

```

Appendix C.4. Delete

```

main Delete:
  a Storable Person.
  a DBPersistent String with ‘‘John Doe’’ as value.
  the name of the Person is it.
  some Strings.
  add ‘‘name’’ to these.
  add ‘‘age’’ to the Strings.
  delete the Strings from the Person.

```

Appendix C.5. Update

```

main Update:
  a Storable Person.
  add ‘‘age’’ to these.
  a DBPersistent and Integer Number with 50 as value.
  the age of the Person is it.
  a Persistent String with ‘‘Jane Doe’’ as value.
  the name of the Person is it.
  some Strings.
  add ‘‘name’’ to these.
  add ‘‘age’’ to the Strings.
  update these from the Person.
  select the Strings from the Person.
  System prints these and newline.

```

Appendix D. Questionnaire

This section presents the questionnaire that was applied to the students along with the percentages for each answer.

Appendix D.1. Question 1

For assignment instructions such as *A is 25*, 12.2% stated that the language is *highly expressive*, 40.8% stated that it is *moderately expressive*, 36.7% described it as *little expressive*, and 10.2% described it as *inexpressive*.

Appendix D.2. Question 2

For instantiation instructions such as *a Speed Formula with 10 as distance and 5 as time*, 8.2% stated that the language is *highly expressive*, 51% stated that it is *moderately expressive*, 38.8% described it as *little expressive*, and 2% described it as *inexpressive*.

Appendix D.3. Question 3

For invocation instructions such as *System prints the first Number where its value is 62 and newline*, 20.4% stated that the language is *highly expressive*, 36.7% stated that it is *moderately expressive*, 34.7% described it as *little expressive*, and 8.2% described it as *inexpressive*.

Appendix D.4. Question 4

Based on the following assignment instructions:

```
a is 25.
b is 54.
c is 62.
d is 86.
e is 35.
f is 28.
g is 55.
h is 33.
i is 24.
j is 5.
k is 2.
```

For invocation instructions with indirect references such as *System prints the last 2 Number and newline*, it was observed that, for the question “what variable (or variables) stores the result of the instruction?”, 55.1% of the students answered correctly that the variables *J* and *K* are those stored in the indirect reference.

Appendix D.5. Question 5

For invocation instructions such as *System prints the first Number and newline*, it was observed that, for the question “what value is printed from the instruction?”, 51% of the students answered correctly 25.

Appendix D.6. Question 6

For abstraction definition instructions such as *noun Formula*, *adjective Percentage* and *adjective Speed*, 6.1% stated that the language is *highly expressive*, 49% stated that it is *moderately expressive*, 34.7% described it as *little expressive*, and 10.2% described it as *inexpressive*.

Appendix D.7. Question 7

For instance instructions such as *a Speed Formula* and *a Percentage Formula*, 12.2% stated that the language is *highly expressive*, 46.9% stated that it is *moderately expressive*, 38.8% described it as *little expressive*, and 2% described it as *inexpressive*.

Appendix D.8. Question 8

Given the following instruction:

```
adjective Speed:
  attribute distance as a Real Number.
  attribute time as an Real Number.
  derived attribute result
  as a Real Number:
  distance / time; and return it.
```

whose Java equivalent is:

```
class SpeedFormula {
  float distance; float time;
  float result() { return distance / time; }}
```

Overall, 22.4% stated that the language is *highly expressive*, 38.8% stated that it is *moderately expressive*, 34.7% described it as *little expressive*, and 4.1% described it as *inexpressive*.

Appendix D.9. Question 9

For adjective refinement instructions, 14.3% stated that the language is *highly expressive*, 57.1% stated that it is *moderately expressive*, 26.5% described it as *little expressive*, and 2% described it as *inexpressive*.

Appendix D.10. Question 10

Given the next instruction:

```
an Speed Formula
with 10 as distance
and 5 as time;
and System prints
the result of it
and newline.
spf is an Speed Formula
with 10 as distance
and 5 as time;
and System prints
the result of spf
and newline.
System prints
the result of an Speed Formula
with 10 as distance
and 5 as time and newline
System prints
the result of a Percentage Formula
with 10 as percent
and 500 as quantity and newline.
```

Overall, 12.2% stated that the language is *highly expressive*, 46.9% stated that it is *moderately expressive*, 36.7% described it as *little expressive*, and 4.1% described it as *inexpressive*.

Appendix D.11. Question 11

Given the next example:

```
noun Person:
circumstance:
Male and Female are mutually excluded.
circumstance:
this requires Male or Female.
adjective Male.
adjective Female.
```

Overall, 12.2% stated that the language is *highly expressive*, 51% stated that it is *moderately expressive*, 32.7% described it as *little expressive*, and 4.1% described it as *inexpressive*.

Appendix D.12. Question 12

Given the following example:

```
noun Person:
circumstance:
```

this cannot be Sick or Dead.
 circumstance:
 this cannot be Wounded and Dead.
 adjective Dead.
 adjective Wounded.
 adjective Sick.

Overall, 12.2% stated that the language is *highly expressive*, 49% stated that it is *moderately expressive*, 28.6% described it as *little expressive*, and 10.2% described it as *inexpressive*.

Appendix D.13. Question 13

Given the next example:

noun Test:

verb test:

System prints the first Number
 where its value is 62 and newline.
 numbers are the first
 five Numbers in this verb.
 System prints numbers and newline.
 System prints the first
 Number and newline.
 System prints the last
 2 Number and newline.

The results of which are as follows:

Valor asociado a it	: Variable
62	: c
[25, 54, 62, 86, 35]	: numbers
[25, 54, 62, 86, 35]	: numbers
25	: A
[5, 2]	: ‘‘those’’

Overall, 22.4% stated that the language is *highly expressive*, 38.8% stated that it is *moderately expressive*, 32.7% described it as *little expressive*, and 6.1% described it as *inexpressive*.

Appendix D.14. Question 14

Given the following example:

noun Person:

attribute age is an Integer Number
 with 0 as value.
 attribute i is the real of ‘‘0’’.
 verb add aa as Integer Number to itself:
 add aa to age.
 verb itself print:
 System prints ‘‘Printing’’ and newline.
 verb itself print2:
 System prints ‘‘Printing2’’ and newline.

For handling circumstances such as *circumstance: add the i of this to the age of this after the age of this is assigned*, 12.2% stated that the language is *highly expressive*, 40.8% stated that it is *moderately expressive*, 32.7% described it as *little expressive*, and 14.3% described it as *inexpressive*.

Appendix D.15. Question 15

Based on the previous example, and given the circumstance *circumstance: this print after the age of something is assigned*, 8.2% stated that the language is *highly expressive*, 57.1% stated that it is *moderately expressive*, 20.4% described it as *little expressive*, and 14.3% described it as *inexpressive*.

Appendix D.16. Question 16

Finally, given the circumstance *circumstance: it print2 instead something print*, 22.4% stated that the language is *highly expressive*, 40.8% stated that it is *moderately expressive*, 30.6% described it as *little expressive*, and 6.1% described it as *inexpressive*.

Appendix D.17. Question 17

Given the next example:

```
a Persistent String with ‘BMW’ as value.
System prints it and newline.
an Entity Car with the first String as model.
some Strings.
add ‘model’ to these.
select the Strings from the Car.
System prints these and newline.
```

Overall, 10.2% stated that the language is *highly expressive*, 53.1% stated that it is *moderately expressive*, 28.6% described it as *little expressive*, and 8.2% described it as *inexpressive*.

Appendix D.18. Question 18

For embedded grammars, 32.7% stated that the language is *very useful*, 59.2% stated that it is *moderately useful*, 8.2% rated it as *little useful*, while no one described it as *not useful*.

Appendix D.19. Question 19

Given the following example:

```
an Integer Number with 1 as value.
add 1 to the first Number.
System prints it and newline.
subtract 1 from the first Number.
System prints it and newline.
an String with ‘Hi’ as value.
System prints it and newline.
52 + 4.
System prints it and newline.
```

Overall, 12.2% stated that language is *highly expressive*, 40.8% stated that it is *moderately expressive*, 42.9% described it as *little expressive*, and 4.1% described it as *inexpressive*.

Appendix D.20. Question 20

Given the next example:

```
execute the next 5 instructions when
left is a Numeric.
| execute the next 4 instructions when
right is a Numeric.
```

```

| | execute the next instruction when
  operator is equal to ‘+’.
| | | the value of left plus the value
  of right; and return it.
| | execute the next instruction when
  operator is equal to ‘-’.
| | | the value of left minus the value
  of right; and return it.
execute the next 4 instructions when
  left is a Variable.
| res is the string of the value of left.
| concat operator with it.
| concat the value of right with it.
| return res.
return ‘Error’.

```

Note that the “|” is not part of the code and was placed to represent how the logic of the instructions is structured.

Overall, 8.2% stated that language is *highly expressive*, 59.2% stated that it is *moderately expressive*, 22.4% described it as *little expressive*, and 10.2% described it as *inexpressive*. On the other hand, 71.4% of respondents stated that such structuring is more complex than that presented in other programming paradigms, while 28.6% stated that it is simpler.

Appendix D.21. Question 21

Based on the following code:

```

numbers are some Numbers.
add 5 to numbers.
add 6 to numbers.
add 9 to numbers.
add 1 to numbers.
add 0 to numbers.
add 3 to numbers.
repeat the next instruction for each
  element of numbers as number.
| System prints number and newline.
System prints ‘Write a message: ’.
System reads.
concat it with ‘The message is: ’.
System prints it and newline.

```

Note that the “|” is not part of the code and was placed to represent how the logic of the instructions is structured.

Regarding the use of plurals, 16.3% stated that the language is *highly expressive*, 55.1% stated that it is *moderately expressive*, 22.4% described it as *little expressive*, and 6.1% described it as *inexpressive*.

Appendix D.22. Question 22

Finally, respondents referred to the usefulness of a naturalistic language in the following way: 20.4% stated that the language is *very useful*, 59.2% stated that it is *moderately useful*, 20.4% rated it as *little useful*, and no one described it as *not useful*.

References

- Sammet, J.E. The Use of English As a Programming Language. *Commun. ACM* **1966**, *9*, 228–230. [[CrossRef](#)]
- Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.M.; Irwin, J. Aspect-oriented programming. In *ECOOP'97—Object-Oriented Programming*; Akşit, M., Matsuoka, S., Eds.; Springer: Berlin/Heidelberg, Germany, 1997; pp. 220–242.
- Laddad, R. *AspectJ in Action: Practical Aspect-Oriented Programming*; Manning Publications Co.: Greenwich, CT, USA, 2003.
- Biermann, A.W.; Ballard, B.W. Toward Natural Language Computation. *Comput. Linguist.* **1980**, *6*, 71–86.
- Lopes, C.V.; Dourish, P.; Lorenz, D.H.; Lieberherr, K. Beyond AOP: Toward Naturalistic Programming. *SIGPLAN Not.* **2003**, *38*, 34–43. [[CrossRef](#)]
- Knöll, R.; Gasiunas, V.; Mezini, M. Naturalistic Types. In Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2011), Portland, OR, USA, 22–27 October 2011; ACM: New York, NY, USA, 2011; pp. 33–48. [[CrossRef](#)]
- Van Roy, P.; Haridi, S. *Concepts, Techniques, and Models of Computer Programming*, 1st ed.; MIT Press: Cambridge, MA, USA, 2004.
- Booch, G.; Maksimchuk, R.A.; Engle, M.W.; Young, B.; Conallen, J.; Kelli, A.H. *Object-Oriented Analysis and Design with Applications*, 3rd ed.; Addison Wesley Longman: Boston, MA, USA, 2007.
- Lyons, J. Deixis, space and time. *Semantics* **1977**, *2*, 636–724.
- Ogden, C.K.; Richards, I.A.; Ranulf, S.; Cassirer, E. *The Meaning of Meaning. A Study of the Influence of Language upon Thought and of the Science of Symbolism*; Harcourt, Brace & World, Inc.: New York, NY, USA, 1923.
- de Saussure, F.; Baskin, W.; Meisel, P.; Saussy, H. *Course in General Linguistics*; Columbia University Press: New York, NY, USA, 2011; pp. 65–70.
- Knöll, R.; Mezini, M. Pegasus: First Steps Toward a Naturalistic Programming Language. In Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06), Portland, OR, USA, 22–26 October 2006; ACM: New York, NY, USA, 2006; pp. 542–559. [[CrossRef](#)]
- Chomsky, N. Three models for the description of language. *IRE Trans. Inf. Theory* **1956**, *2*, 113–124. [[CrossRef](#)]
- Clark, P.; Murray, W.R.; Harrison, P.; Thompson, J. Controlled Natural Language: Workshop on Controlled Natural Language, CNL 2009, Marettimo Island, Italy, June 8–10, 2009. Revised Papers. In *Controlled Natural Language: Workshop on Controlled Natural Language, CNL 2009, Marettimo Island, Italy, June 8–10, 2009. Revised Papers*; Chapter Naturalness vs. Predictability: A Key Debate in Controlled Languages; Springer: Berlin/Heidelberg, Germany, 2010; pp. 65–81. [[CrossRef](#)]
- École Polytechnique Fédérale Lausanne (EPFL). Scala Online Resources. 2019. Available online: <https://docs.scala-lang.org/learn.html> (accessed on 20 July 2019).
- Liu, H.; Lieberman, H. Programmatic Semantics for Natural Language Interfaces. In Proceedings of the CHI '05 Extended Abstracts on Human Factors in Computing Systems (CHI EA '05), Portland, OR, USA, 2–7 April 2005; ACM: New York, NY, USA, 2005; pp. 1597–1600. [[CrossRef](#)]
- Mihalcea, R.; Liu, H.; Lieberman, H. NLP (Natural Language Processing) for NLP (Natural Language Programming). In Proceedings of the 7th International Conference on Computational Linguistics and Intelligent Text Processing (CICLing'06), Mexico City, Mexico, 19–25 February 2006; Springer: Berlin/Heidelberg, Germany, 2006; pp. 319–330. [[CrossRef](#)]
- Cann, R. *Formal Semantics: An Introduction*; Cambridge University Press: New York, NY, USA, 1993.
- Cremer, V. *Foundations for Scala: Semantics and Proof of Virtual Types*; Technical Report; École Polytechnique Fédérale de Lausanne: Lausanne, Switzerland, 2006.
- Belblidia, N.; Debbabi, M. Towards a Formal Semantics for AspectJ Weaving. In Proceedings of the 7th Joint Conference on Modular Programming Languages (JMLC'06), Oxford, UK, 13–15 September 2006; Springer: Berlin/Heidelberg, Germany, 2006; pp. 155–171. [[CrossRef](#)]
- Pulido-Prieto, O.; Juárez-Martínez, U. A Survey of Naturalistic Programming Technologies. *ACM Comput. Surv.* **2017**, *17*:1–17:39. [[CrossRef](#)]
- Fuchs, N.E.; Schwitler, R. Attempto Controlled English (ACE). *arXiv* **1996**, arXiv:cmp-lg/9603003.
- Fuchs, N.; Schwitler, U.; Schwitler, R. *Attempto Controlled English Language Manual Version 3.0*; Institut für Informatik der Universität Zürich: Geneva, Switzerland, 1999.

24. Fuchs, N.E.; Kaljurand, K.; Kuhn, T. Reasoning Web. In *Reasoning Web*; Chapter Attempto Controlled English for Knowledge Representation; Baroglio, C., Bonatti, P.A., Maluszyński, J., Marchiori, M., Polleres, A., Schaffert, S., Eds.; Springer: Berlin/Heidelberg, Germany, 2008; pp. 104–124. [[CrossRef](#)]
25. Kuhn, T.; Bergel, A. Verifiable Source Code Documentation in Controlled Natural Language. *Sci. Comput. Program.* **2014**, *96*, 121–140. [[CrossRef](#)]
26. Clark, P.; Harrison, P.; Jenkins, T.; Thompson, J.; Wojcik, R. Acquiring and using world knowledge using a restricted subset of English. In Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference (FLAIRS 2005), Clearwater Beach, FL, USA, 15–17 May 2005; AAAI Press: Menlo Park, CA, USA, 2005; pp. 506–511.
27. Liu, H.; Lieberman, H. Metafor: Visualizing Stories As Code. In Proceedings of the 10th International Conference on Intelligent User Interfaces (IUI '05), San Diego, CA, USA, 9–12 January 2005; ACM: New York, NY, USA, 2005; pp. 305–307. [[CrossRef](#)]
28. Cozzie, A.; Finnicum, M.; King, S.T. Macho: Programming with Man Pages. In Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS'13), Napa, CA, USA, 9–11 May 2011; USENIX Association: Berkeley, CA, USA, 2011; p. 7.
29. Cozzie, A.; King, S.T. *Macho: Writing Programs with Natural Language and Examples*; Technical Report; University of Illinois at Urbana-Champaign: Champaign County, IL, USA, 2012.
30. Landhäußer, M.; Hug, R. Text Understanding for Programming in Natural Language: Control Structures. In Proceedings of the Fourth International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE '15), Florence/Firenze, Italy, 16–24 May 2015; IEEE Press: Piscataway, NJ, USA, 2015; pp. 7–12.
31. Landhäußer, M.; Weigelt, S.; Tichy, W.F. NLCI: A natural language command interpreter. *Autom. Softw. Eng.* **2016**, *27*, 839–861. [[CrossRef](#)]
32. Bracha, G.; Cook, W. Mixin-based Inheritance. In Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications (OOPSLA/ECOOP '90), Ottawa, ON, Canada, 21–25 October 1990; ACM: New York, NY, USA, 1990; pp. 303–311. [[CrossRef](#)]
33. Parr, T. *The Definitive ANTLR 4 Reference*, 2nd ed.; Pragmatic Bookshelf: Raleigh, NC, USA, 2013.
34. Odersky, M.; Spoon, L.; Venners, B. *Programming in Scala: A Comprehensive Step-by-Step Guide*, 1st ed.; Artima Incorporation: Walnut Creek, CA, USA, 2008.
35. Inc., L. SBT Reference Manual. 2019. Available online: <https://www.scala-sbt.org/1.x/docs/Faq.html> (accessed on 20 July 2019).
36. Eckel, B. *Thinking in Java*, 3rd ed.; Prentice Hall: Upper Saddle River, NJ, USA, 2002.
37. Bruckman, A.S. Moose Crossing: Construction, Community, and Learning in a Networked Virtual World for Kids. Ph.D. Thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1997.
38. Landhäußer, M.; Hey, T.; Tichy, W.F. Deriving Time Lines from Texts. In Proceedings of the 3rd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE 2014), Hyderabad, India, 31 May–7 June 2014; ACM: New York, NY, USA, 2014; pp. 45–51. [[CrossRef](#)]
39. Likert, R. A Technique for the Measurement of Attitudes. Ph.D. Thesis, New York University, New York, NY, USA, 1932.



© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).