Grigori Sidorov, Alexander Gelbukh. *A Hierarchy of Linguistic Programming Objects.* Proc. ENC'99, Segundo Encuentro de Computación, Pachuca, Hidalgo, September 12–15, 1999.

# A Hierarchy of Linguistic Programming Objects

Dr. Grigori Sidorov,
Dr. Alexander Gelbukh

Centro de Investigación en Computación, Instituto Politécnico Nacional,
Av. Juan de Dios Bátiz, A.P. 75-476, C.P. 07738, México D.F.,
+52 (5) 729-6000, ext. 56544, 56602, fax 586-2936,
{sidorov, gelbukh}@pollux.cic.ipn.mx, gelbukh@micron.msk.ru

**Abstract**[*]

In the paper, the general structure of some programming objects useful in linguistic analysis is suggested. The structure of the most abstract linguistic object is discussed; such an object[1] can serve as the root of the hierarchy of linguistic objects. We suggest that this abstract linguistic object be a *linguistic sign*. Two possible methods for hierarchy of linguistic programming objects construction are discussed (with parameters and with inheritance).

Also discussed is the structure of *signifier*, *syntactics,* and *semantics* objects, that are properties of a linguistic sign. Syntactics and semantics should be defined through the most common linguistic structures that naturally appear in linguistic algorithms, namely, a string, tree, and network. This is determined by the relations between linguistic signs in language.

As an example, a simple program that uses the discussed object hierarchy to perform sentence and paragraph detection for ASCII text with tables is described.

## 1. Introduction

It is well-known that object-oriented approach to modeling complex objects greatly simplifies the models and software development [3]. Linguistic modeling is not an exception since language is extremely complex and vague system. Object-oriented approach in linguistics has been discussed in many recent works [4, 5, 7, 9]. However, these works do not start from the very beginning of the problem: from the discussion of the most abstract linguistic objects, in spite of the practical and theoretical importance of this issue.

In an object hierarchy always there is the most abstract object[2], the root, which is the parent of all the other objects. The object-oriented point of view at the linguistic modeling gives rise to a few interesting questions.

- Is it true that any linguistic object belongs to one of the traditional levels known in theoretical linguistics: phonology, morphology, syntax, semantics, and pragmatics [1, 6]?
- If so, is it necessary to have a set of separate most abstract linguistic objects for each level, i.e., independent object hierarchies for the levels?
- If not, what should the most abstract linguistic object be, i.e., what features are common to all linguistic entities?
- What properties[3] should such an object have?

Linguistics is highly structured science. Traditionally there are some different levels of research (syntax, etc.), which are usually studied separately. In the modern linguistics the tendency for interlevel interactions is obvious (cf. the lexical component in generative grammars), however, these interactions are external, not an organic part of the corresponding theories, because usually they start at one of the fixed levels. In our opinion, the synthesis of ideas of different levels of linguistic research is very important, provided that it allows for preserving the level structure where it is relevant and necessary.

---

[*] The work done under partial support of CONACYT, Project 26424-A, and SNI, Mexico.

[1] Unless especially mentioned, by an o*bject* we mean what corresponds to a *type* or *class* in programming languages. This usage should not be confusing in the context and better corresponds to the usage common in linguistics: a *morpheme* is not called a *linguistic type* or *class*, but instead a *linguistic object*, while it can be subdivided (subclassed in programming terminology) into such *linguistic objects* as *prefix*, *suffix*, *infix*, etc.

Probably the reason for this terminological difference is also linguistic: the English words *class* and *type* require a complement in genitive plural: “*class of morphemes*”, while “*class* `morpheme`” sounds ungrammatically and confusing outside the programming slang.

[2] Or a set of objects in case of a set of related hierarchies that do not have any common root.

[3] Members and methods, as they are called in programming.

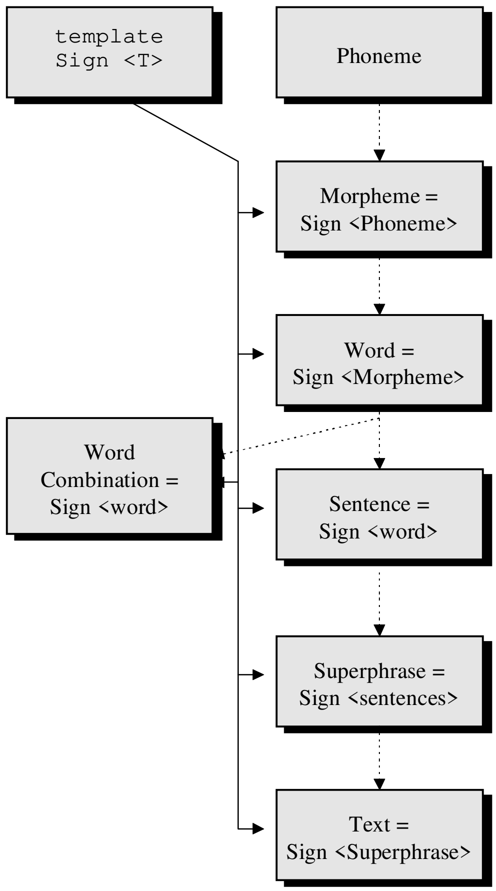


Fig. 1. The linguistic object hierarchy using parameters.

In particular, we think that there should exist some features common for all the linguistic objects, and this set of features should determine the base object linguistic object hierarchy. This abstract object should not belong to any of the traditional linguistic levels but instead should organically unify them.

It is useful to address the history of linguistics in search for a concept that would have always considered important by different linguistic schools and would unify all the traditional linguistic levels.

Thus, what concept was (1) most widely discussed in linguistics in its historical development, (2) is abstract enough and (3) does not belong to any specific linguistic level but instead is common for all such levels? It is a *linguistic sign*. Thus, the corresponding programming object (type, or class) can be used as the most abstract linguistic object in the object hierarchy.

What is a *linguistic sign*? Practically all linguistic theories that proclaim the explanations of the basic language phenomena use this notion. Starting from the ancient Greece, then F. de Saussure, Prague linguistic school, Ch. Morris, L. Bloomfield and other linguists discussed this notion a lot[4]. A very good discussion of the practical use of this notion for language description can be found in [6], from that we adopted some of the discussed ideas.

In simple words, a sign is a *function*: a way to express (associate) some meaning with some observable thing. The thing used to express the meaning is called *signifier*, the meaning expressed with it is called *signified*. However, the signs used in language have yet third, key feature: they affect each other when are used together. All phenomena related to such mutual affection are called their *syntactics*[5] [6]. These three properties of the sign are so important for it that they are called *sides*[6], like sides of some physical objects: an object can't have only one side or not have any.

- Signifier is the "visible" part of the sign, usually the substance. Roughly speaking, for a dollar coin this is the metal disk with the corresponding picture; for the word "bird" these are letters of this word: b-i-r-d (or the corresponding phones).
- Signified is the relation between a sign, the concept, and the world object. Roughly speaking, for a dollar coin this is the value of one dollar; for the word "bird" this is the concept of a bird, a flying creature with feathers.
- Syntactics is the relations between signs in a flow of speech. This is a specific and very important part of a *linguistic* sign. Roughly speaking, it tells us what words can be used together and in what form, for example, *bird* + *flies* is a grammatical English phrase, but *bird* + *fly* or *in* + *flew* are not; the word *well* in the context *well done* has another meaning than in the context *deep well*[7], etc.

Signs that are not linguistic do not have any syntactics: a dollar coin used together with a ten-dollar bill or a hundred pounds check keeps exactly the same "meaning" of one dollar value.

[4] The only exception seems to be the generative linguistics introduced by N. Chomsky, that does not pay much attention to this concept, probably because the sign seems to be rather abstract entity and generative linguistics was always looking for formal and clear models. However, even in this school one of the sign's properties, syntactics, is deeply worked out.

[5] In Spanish: *significante*, *significado*, *sintáctica*.

[6] The term *sides* was introduced first for signs that are just functions signifier → signified and referred to these *two* sides, like two sides of a leaf of paper without which a leaf can't be thought of. In that time, syntactics was not introduced yet.

[7] In Spanish: *bien hecho*, *pozo profundo*.

Signified is also called semantics[8] of the sign. The term "meaning" is generally avoided in the precise definitions, though can be used when it is clear that it refers to semantics.

The majority of linguistic entities are signs[9], because they have signifier, semantics, and syntactics. Some entities don't have semantics (like phonemes).

So far the concept of sign had little practical consequences because it is too abstract. However, in our opinion it proves to be useful in object-oriented programming as the most abstract linguistic entity.

# 2. Linguistic object hierarchy

As we have seen, the objects of the linguistic hierarchy should be derived from the sign.

However, most of linguistic objects are complex, i.e., their parts, the signifier, signified and syntactics are structures[10], or at least sets. We suggest that these parts be considered as sets, or more complicated structures, of other linguistic objects, i.e., signs (the only partial exception is signifier, see Section 3).

There are two basic methods to construct a linguistic objects hierarchy. The first is to consider a sign a parametrized class (template in C++), and the second is to use inheritance.

## *2.1 The method using parameters*

Different types of signs differ in the type of the constituent signs: a phrase is at least a set[11] of words, a word a set of morphemes, etc. How can this be reflected in the hierarchy? Unfortunately, C++ inheritance does not allow for construction of objects in this way: one can't just derive a *set of Xs* from the class *X*. To solve this problem, parametrized classes, or so-called templates, can be used. In simple words, a parametrized type represent objects, having a constituent(s) of some variable type *X*, e.g., *set of Xs*.

Use of parametrized abstract structures such as *set of Xs* allows to interpret traditional levels without separating them completely. Objects (types) of each level are constructed from the names of the objects of the previous level, which is the type of their constituents, as a parameter (e.g., morphemes are constructed from phonemes; they in their turn are the material for words; sentences are constructed from words, etc.).

The hierarchy built using templates and their parameters is shown on Fig. 1. With the dotted lines the template parameters are related with the corresponding objects, and with solid lines the dependencies between a template and its instance are shown. Note that in this type of hierarchy there is no inheritance at all, since we use the Whole-Part relationships instead of more traditional Is-A relationship.

The `Sign` template is a linguistic sign that has `semantics`, `syntactics`, and `signifier` objects as its properties, or parts. All the other linguistic entities but the `phoneme` are based on the `sign` template with different parameters.

From the programming point of view these objects can be regarded as polymorphic collections for data storage. In the method with templates the type of the objects constituting these sets is a parameter: e.g., a the semantics, syntactics, and signifier of a `word` are sets of the corresponding values for `morpheme`.

The `phoneme` has no semantics, i.e., its corresponding polymorphic collection is empty. The other objects are: the `morpheme`, the `word` (lexeme), the `word combination` (collocation), the `sentence`, the `superphrase unit`, and the `text`. All of them are instances of the `sign` template.

In runtime, the objects are constructed by the program level by level. At each level, their `signifiers` are constructed from the objects that are the template parameters, their `syntactics` are their own syntactics, and their `semantics` is constructed from the `semantics` and `syntactics` of the parameter and their own. For example, the meaning of a word is constructed from meanings of morphemes and their position in word; the same for the sentence, etc. There can be exceptions, e.g., phraseologisms, when a unit can't be directly constructed as a plain combination of its parts. Then the corresponding parameter does not participate in `semantics`, but instead only in `syntactics` and `signifier`.

It is useful to give here some comments about the discussed hierarchy.

- The term "word" is used in the sense of lexeme, i.e., an abstract unit that have a wordforms as its substance [6: 99]).
- Sometimes morpheme and word (lexeme) are considered not to be signs [6], instead, only morphs and wordforms are considered signs. We prefer the opposite point of view, introducing the object `representation` for morphs and word forms, where they are treated as `signifiers` (see Section 3).

---

[8] By *semantics* we will mean *semantic and pragmatics*, because the difference between them is not significant for our discussion.

[9] Be will use terms *sign* and *linguistic sign* as synonyms.

[10] *Containers* in C++ terminology.

[11] We say *at least a set* to avoid the discussion of the precise structure at this point. Again, this corresponds to a container: something consisting of some elements.

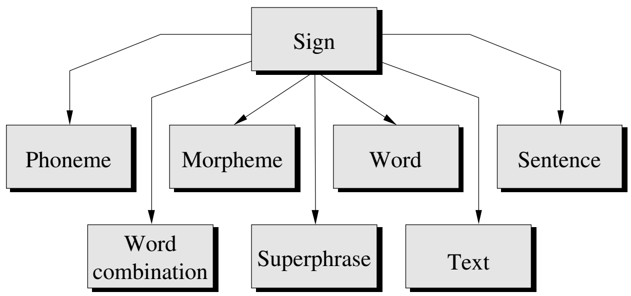


Fig. 2. The linguistic object hierarchy using inheritance.

The presence of parameters allows to organize a work of the program like a kind of conveyer where the corresponding objects are processed, and the result is transduced to the next object (e.g., after processing of phonemes, the morphemes are processed, after this words, etc.). This reflects one of traditional approaches to linguistic program development, introduced in the frame of the Meaning ⇔ Text theory [6].

### *2.2 The method using inheritance*

In our opinion this method of linguistic object hierarchy development is more flexible and productive.

In this method a full-scheme object-oriented approach is used. This method allows to preserve polymorphism, and so seems better than the method of templates, though the structures of hierarchy are very much alike. The hierarchy is shown on Fig. 2.

The difference with the first method is that the objects have no parameters and they are not instances of the `sign` as a template, but instead are derived from it and, thus, inherit from it. The sign here is not a template, but the parent object. These objects should have specific methods for handling other signs (e.g., morphemes should process signs in a different way than sentences).

The program that uses such a hierarchy for text analysis should implement a sort of a loop, that starts with one, the most elementary, sign and then goes up like a spiral, creating the more complex objects in hierarchy, but not loosing the created ones that instead become parts of the signifier, syntactics, and semantics of new constructed objects.

We will not discuss in details the possible relations between signifier, syntactics, and semantics of the two signs in their combination; such a linguistic investigation is beyond the frame of this article. Here will only give a few examples.

- When the English morpheme -*s* ‘plural’ is incorporated in a word, its signifier is added to the signifier of a word; its semantics is added to the word’s semantics. At the same time it is added to the word’s syntactics, because, for example, the relative clause should now have the verb in plural (“… *which are/were* …”). At the same time, the syntactics of a word influences the signifier of the morpheme -*s* ‘plural’, prescribing what variant (-*es* or -*s)* is allowed.
- Another example deals with phraseology. Two words *hot* and *dog* separately mean something absolutely different than together: *hot dog*. This proves that their semantics is determined by their syntactics. By these examples we demonstrate that sign properties can influence any of the properties of another sign.

## 3. *Signifier*, *syntactics*, and *semantics* objects

What are the objects that constitutes sign’s properties? As it was mentioned above, they are polymorphic collections that contain other signs. The only partial exception is a *`signifier`* object. It can contain not only signs but also some material forms, because the signifier is a bridge between the world of signs and the material world, so it must be able to deal with both signs and real world object (such as acoustic features, etc.). The suggested structure of this object is shown on Fig. 3.

The base object is a `material form`. From this object derived are the *`substance`* and *`representation`* objects. The *`substance`* is a physical, material form of linguistic entities, e.g., acoustic features, and the *`representation`* is an abstraction for a set of entities: e.g., morphs -*s* and -*es* represent the morpheme -*s* ‘plural’.

The next level objects for the *`substance`* are: `acoustic`, `written`, `gesticulating`, maybe more objects can be added.

Yet another level of objects for the `representation` contains: *`phone`*, *`suprasegment unit`* (intonation, stress, etc.), *`morph`*, and *`word form`*. We don’t need to introduce any more representations because the higher structures (sentences, texts, etc.) consist of word forms. Representations can have different substances. For example, morphs can have `written` or `acoustic` substance.

The *`sign`* object has `syntactics` and `semantics` object as its properties. We will not discuss here their structure; in fact, this structure is one of the main topics of the computational linguistics in general. There are works that discuss this question in practical aspect [2, 8], though not in terms of objects. Thus, the structure of `syntactics` and `semantics` objects will be discussed only from the point of view of corresponding linguistic structures that can be their properties.

There are three basic kinds of structures that are widely used in nearly all linguistic models: a *string* (linear structure), a *tree*, a *network*.

For example, the linear structures are the only possible surface structures, the tree structures are widely used in

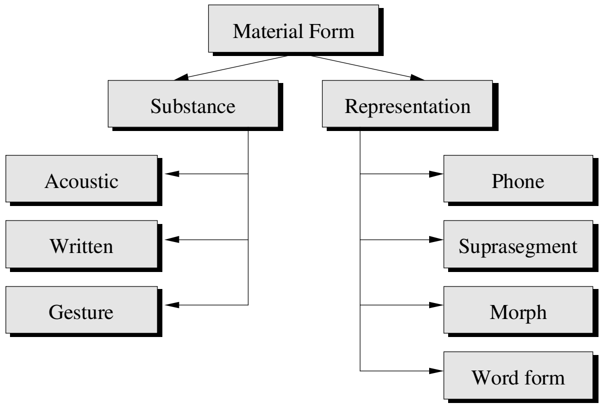


Fig. 3 The material form object hierarchy

syntax research, the network structures often appear as semantic network for lexicon or world knowledge modeling, and also are used for semantic representation of the text.

We will try to show that the presence of these structures corresponds to the relationships of signs.

The sign can have different number of relations with other signs, see Fig. 4. The significance of the particular relations does not matter here, instead, only the number of the relationships is important.

It is obvious that the linguistic structures correlate with possible number of relations between signs. The signs that only allow for *one* arrow going to the sign and *one* going from it ("*one and one*"), form a string. Those that allow for *one* arrow to the sign and *many* ones from it ("*one and many*"), form a tree structure. Finally, those that allow for *many* arrows to the sign and *many* ones from it ("*many and many*"), form a network structure. The *many-and-many* relation can not be implemented in a flow of speech because at a moment of speaking it is possible to produce just one sign. This is the reason to consider it only as a possible structure for `semantics`, not `syntactics`. The other two possible types of relations between signs occur in a flow of speech and constitute sign's `syntactics`.

The `syntactics` object should have the corresponding properties for handling string and tree structures; a `semantics` object should have a property for the network structures. Thus, any object in the hierarchy is able to operate with possible linguistic structures, determined by intersign relations.

For example, a *word* is a linear structure of morphemes plus relations between them. In linguistics these relations usually are not described as a tree structure, but it can be interpreted with such a formalism. The root of the word is the head of the tree, and the grammatical morphemes are the nodes. At the same time, the word is a part of all possible types of networks, i.e., a part of a lexicon, that defines its meaning.

One more example: the sentence is a linear structure of words with relations that form a syntactic tree.

# 4. Example: A program for text structure elements detection

As a simple example of usage of objects similar to those discussed in the paper, we will present a program that uses them for a little bit different task: for the functions performing parsing in a text converter. However, the parsing objects can't be derived from the base linguistic object (the sign), because they have different nature. In contrast to the objects described above, that are declarative, the parsing objects are procedural. So it may be useful to have a *syntactic parser* object that makes sentences from words, or *superphrase parser object* that makes superphrase unit (or paragraphs) from sentences, etc.

We used the developed objects in a simple program that can serve as an example of manipulation with such objects. This program detects sentences and paragraphs in a plain ASCII text containing non-linguistic elements such as tables or multi-column text, headers, abbreviations, etc.

Actually this program is a first stage of a text parser, since a parser can not proceed with a text if it is unaware of what it should analyze. Usually a text parser first of all deals with sentences. Paragraphs may be necessary for some information retrieval system which can analyze the co-occurrences of words in the same paragraph.

The problem of sentence detection is not as trivial as it could seem since we can't treat the period as an unambiguous sentence end marker. There are some situations where a period doesn't denote the sentence end (abbreviations, numbers, etc.); sometimes we can't find the period where it is expected (headers, etc.); sometimes the sentences are organized in an unusual manner (such as in tables or multi-column text, where each column contains its own text data). Also the page numbering with empty lines before and after it may interfere the processing because usually the empty lines denote paragraph boundaries, which are also sentence boundaries.

From the algorithmic point of view, some simple heuristic rules were used, such as:

- If a period appears after a one-letter word, then this most likely is an abbreviation and not a sentence boundary,
- If a paragraph boundary (several spaces at the beginning of the line) is encountered, then if the previous line also is a paragraph and all the letters are capitalized, then this is a header, otherwise it is a separate paragraph (sentence), etc.

Such rules allow to process all situations except tables or multi-column text. In traditional structural approach, a

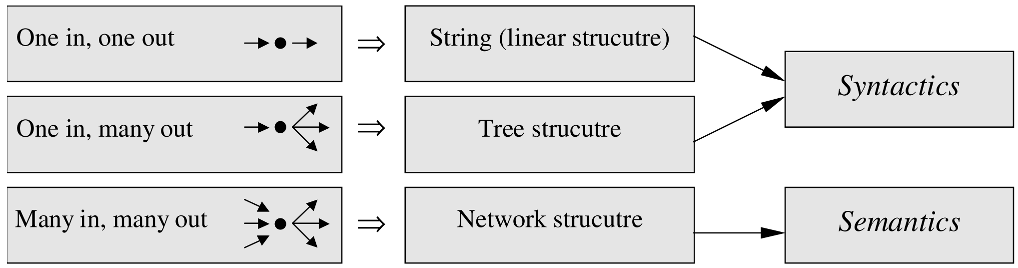


Fig. 4 Relations between signs and corresponding structures.

lot of buffers, flags, etc., would be necessary to deal with such data, and even then correct processing of the data would be not guaranteed. Using object-oriented approach it is possible to create a new parsing objects for each table's column, and they process the columns as if the text in columns were a non-tabular text.

To solve the problem, the following objects were created. First of all a *sign* was created, and *sentences* and *words* like in the suggested hierarchy were derived from it. In such a simple program we ignored some important properties of the sign and dealt only with the graphical forms. Then the line object was used. The input text is broken down into lines basing on its graphical form. The lines are not true language signs with all the sign's properties, instead, they have only `signifier` and `syntactics`. As a simple solution, the corresponding object was still derived from the `sign`, but its semantics was not used, i.e., it was empty. As an addition to declarative objects (sentence, line and word), we designed some procedural object (*text parsing objects*).

Thus, sentences are the parser's output, lines are its input, and words are the entities that it operates upon.

The program works in the following way. It takes lines of the text one by one and analyses their syntactics (according to the rules above) to form the paragraphs. In the same manner it analyses the syntactics of words and punctuation marks to form the sentences.

We tested this program on some Spanish, English, and Russian texts. Only minor changes were necessary to reflect the difference of the punctuation rules for each particular language (we had to rewrite only one virtual method of the *sentence*).

# 5. Conclusions

The usage of linguistic objects hierarchy simplifies and unifies the linguistic software development. In object-oriented approach to development of natural language processing algorithms, it is useful to use an abstract linguistic object that corresponds to the *linguistic sign*, and the hierarchy of linguistic objects derived from it.

The usage of the most widespread structures in linguistic analysis (string, tree, network) is determined by the properties of a sign's syntactics. These structures must be constituents (members, properties) of the corresponding objects.

A simple example of usage of the developed linguistic objects was presented.

**Bibliography**

1. Allen, J. *Natural Language Understanding*. The Benjamin/Cummins Publishing Co., 1995.

2. Berleant, Daniel. *Engineering "word experts" for word sense disambiguation* // Natural Language Engineering 1: 339-362 (1995).

3. Booch, G. *Object-oriented analysis and design, with applications*. 2nd ed., Redwood City, CA: The Benjamin/Cummins Publishing Co., 1994.

4. Daelemans W., Gazdar G., Smedt K. de, *Inheritance in Natural Language Processing*. Computational Linguistics, Vol.18, No.2, 1992. pp. 205–215.

5. Hudson A. *English Word Grammar*. Oxford: Blackwell, 1990.

6. Mel'čuk I. *Cours de Morphologie General*. Vol. 1.; Rus. transl.: Course of general morphology, Moscow–Viena, 1997.

7. Levrat B., Amghar T. *Using Classes of Objects, Polymorphism and Object Oriented Programming Paradigms in the modelizaton of synonymy*. Dialogue'96, Computational Linguistics and its Applications, Proceeding, Moscow, 1996. pp. 128-130.

8. Small, S.L., and C.J. Rieger. *Parsing and comprehending with word experts (a theory and its realization)*. In Lehnert and Ringle (eds.), Strategies for Natural Language Processing, 1982, pp. 89-147.

9. Zajac R. *Inheritance and Constraint-Based Grammar Formalisms*. Computational Linguistics, Vol.18, No.2, 1992. pp. 159–180.