

Acknowledgements

This project is the first part of my final project, being started at the department of Technical Informatics, which is part of the Faculty of Information Technology and Systems at Delft University of Technology, The Netherlands, and it will be continued at the department of Mathematics and Computer Science, of the Transilvania University of Brasov, Romania, under the supervision of Conf. Dr. D. Marinescu. The work was performed at the Knowledge-Based Systems (KBS) chair of the Mediamatica department.

First of all, I would like to thank my project leader Dr. drs. L.J.M. Rothkrantz of the KBS chair, for the support, attention and the great ideas reached during our weekly meetings.

Further more I would also like to thank Prof. dr. H. Koppelaar for providing some great documentations; Mr. M.L. Hodge for providing the software and the ideas in the creation of the icons; Mr. D. Danciu for helping with the optimal form of the theoretical approach, and last, but not least, Mr. B. Tatomir, my brother, for all his support.

Iulia Tatomir
Delft
December 2003

Abstract

Non-verbal communication plays an important role in human communication. The KBS department of TU Delft University is interested in developing multi-modal systems, that will get closer and closer to making the interaction with the computer more human. From facial expressions and speech recognition, to eyes tracking and sign language communication, all these ways create a larger environment in the interaction with the user.

An uni modal system was created, based on the iconic communication – which means that the user will communicate data only by using an international approved language, which are the icons. The situation considered was the one of a crisis, which includes explosions, fire, flood, car crashes, and so on.

Contents

Acknowledgements.....	2
Abstract.....	3
Chapter 1.....	6
Introduction.....	6
Chapter 2.....	8
Literature survey – The theoretical approach.....	8
2.1 What are the icons?.....	8
2.2 Semiotics.....	9
2.3 Encoding/Decoding.....	11
2.4 Icons and context.....	11
2.5 Human Communication.....	12
Chapter 3.....	14
Methodology.....	14
3.1 Advantages vs. disadvantages in using the icons.....	14
3.2 The design of the icons.....	14
3.3 Creating the interface.....	15
Chapter 4.....	21
Introducing Natural Language Processing.....	21
4.1 The natural language.....	21
4.1.1 Components of an NLP system.....	22
4.1.2 Language problems.....	24
4.1.3 Approaches to NLP.....	26
4.1.4 Language Universals.....	26
4.1.5 Creating the vocabulary.....	27
4.2 Grammars and productions.....	28
4.2.1 Bachus Naur Form.....	29
4.2.2 Connection between a context-free grammar (CFG) and a BNF construction.....	30
4.3 Grammar subdivisions.....	30
4.3.1 Simple Noun Phrases.....	31
Verbs.....	32
Particles.....	32
4.3.2 Bounding the vocabulary with the grammar categories.....	32
4.4 Defining the grammar.....	32
4.5 Properties of BNF grammars.....	35
4.5.1 Emptiness.....	35
4.5.2 Starting tokens.....	37
4.5.3 Recursive rules.....	37
4.5.4 Left-recursive.....	38
4.6 An example.....	38
4.7 Phrase structure and lexical structure.....	39
4.7.1 Parsing techniques.....	40
4.7.2 Top-down parsing (LL).....	41
4.8 Competence and performance.....	42
4.8.1 What is a good grammar?.....	42

Chapter 5	44
Models	44
5.1 Interface	44
5.2 Testing the application	48
5.3 Programmatic wrapping of the problem	50
5.3 The meaning of the sentence - the connection between the user interface and the grammar	51
5.3.1 Conclusions	56
Chapter 6	56
Implementation	56
6.1 Design patterns	57
6.1.1 Singleton	57
6.2 Creating the buttons	59
6.3 Configuration of the data	59
6.3.1 Loading the rules	60
6.3.2 The semantic elements	62
6.3.3 Articles between the words	62
6.3.4 Plural nouns	63
6.4 Syntactic analysis	63
6.5 Semantic analysis	64
6.6 Giving the meaning to the sentence	65
6.6.1 Templates	65
6.6.1.1 'There is – There are'	65
6.6.1.2 Question template	66
6.6.1.3 First person template	67
6.7 Client – server side	68
6.8 Hints	70
Chapter 7	72
Testing – the user book	72
Chapter 8	75
Conclusions	75
References	76

Chapter 1

Introduction

The topic of the project is to create an application that gives the possibility to the users to communicate using an international ‘language’ – the icons. We take in consideration the crisis situations: fire, car crash, etc. and also the fact that the communication can take place without the user knowing an international language, which is in our case English, and even the fact that he/she may not be able to speak. Until now this is a uni modal system, this means that the only way to transmit information is using the icons, which is a visual channel of communication.

The central part of this project is the one based on natural languages, our main concern being the way to transform the icons into text, with less grammar errors as possible. That’s why we had to create our own language, constructing in a matter of speaking our own way of communicating, including all the levels: vocabulary, ontology, grammar rules, also taking care of the syntactical and semantic problems. The main idea was to permit the ‘free’ expression of ideas according to the specific context, only by using the icons. It would have been useless to add text between the icons, because our goal was to code as much as possible the communication, making it as atomic as possible.

We have started this approach by concentrating first on the design of the interface, knowing the fact that it should be as simple as it can to be used. Taking in consideration a person placed in a crisis situation, being able to transmit information, for example, from a PDA (Personal Digital Assistant). We are well aware of the fact that one can describe a situation in many ways, the spoken language being very flexible according to the human needs – but the context of this very project constrained us to impose certain rules of sending messages. For example, some of the constraints are the length of a sequence of icons that will be sent, the order of the icons, and so on.

Our wish was to create an icon editor, meaning that the user would have freedom of creating the sentences, modifying them before sending them, browsing through the icons, etc. It is like we try to replace the words with icons, keeping the same basic ideas of grammar rules and order in concepts.

This application is conceived as being a client-server one. In our case the client is the application running on the user’s machine. The input information would be the sequence of icons. On the moment the user will press the ‘SEND’ icon, the correctness of the sequence will be checked, and, in case of success, the output, which will be the translated text, would be transmitted to the server side. From this point the application can be developed both on the server site and on the client one. But this is not the goal of our project.

In chapter two we present the theoretical fundament of the problem, starting with the concept of icon, gathered in the one of semiotic, and passing through the practical use of the icons.

In chapter three, we present our approach to the graphical part of the application: why we choose these icons, why we had defined them in such a manner, the arguments for the way the interface looks like, and all the things that are related with the appearance of the application, within reasoning.

In the next chapter, chapter four, we pass to a short presentation of the most important part of the project, defining the grammar for the vocabulary in a situation of crisis. And here it starts the personal contribution: in section 4.4, not taking into consideration the design of the icons.

Chapter five makes a description of the modeling of the project, the structure of the application, and the very first layer of programming elements that will help us to solve the problem.

In chapter six we develop more the presentation of the programming elements, presenting all the things that have been used in syntactical and semantic analysis, the definition of the fix structures of the sentence – the templates, plus some programming tricks, like the design patterns, that are used to ease our work. In the end of this chapter, section 6.8 we also present our way to help the user in using this application, giving him some hints of the sentences that can be constructed stating with a particular icon.

If chapter seven is meant to explain the way this application should be used, like giving the steps of the running application, steps that must be followed in a specific order; in chapter eight we present the final conclusions, and also making a parallel between our application and the others that are already existing, plus the further work.

Apart from chapter three, that gave me the possibility to create the icons in the way I wanted, part from them, the construction of the vocabulary, the definition of the grammar with all that includes: grammar categories, subcategories and the production rules, and continuing with chapter 5 until the end, all those elements are personal contribution.

Problem definition

We have started the approach of this project by reading some appropriate literature, to get some panoramic idea of the context of the application. The next step was to define the vocabulary and the grammar. The form of the grammar changed continuously from the fact that we could find contra-examples for all the intermediary forms, until we have reached the form that is presented in the appendix A. We have continued after that with the implementation of the grammar checker both from syntactical and semantic point. Because the time permitted, we have transformed the application into one of the client – server communication type, and we have started to implement the part in which we are helping the user to construct more easier the sequence of icons – the hints, presented in section 6.8.

Chapter 2

Literature survey – The theoretical approach

The dream of being able to understand and communicate in any language has not yet been satisfied. However, there do exist signs and symbols (icons), which are understood internationally. This project proposes a computer-based iconic communication application, an iconographic approach towards a crisis situation.

The theory of an iconography is that it is understandable by everyone concerned - that is why iconographies worked well in the ancient world where small groups shared the same experiences - but not always so well in today's global village. Between two people any mark can become an understandable icon. A very simple two-way, open ended and interactive iconic communication between sender and receiver would not need to be comprehensible to anyone else other than those two, but could be extended within a family. It should be possible to re-establish communication by returning to the origins of human development when, in part, facial expression and gesture were used.

2.1 What are the icons?

Icons have been in use for a very long time, as early as the Middle Ages complex iconic systems had been developed such as the heraldic coats of arms and systems of astrological signs. In modern society everyone is familiar with icons, both in and out of work: for example, icons on the toilet door, iconic road signs and complex icons on electronic goods. From the everyday context of living to the packaging for the latest products, one can meet icons as a daily occurrence. In the computer world, the use of icons has been an extension of their traditional uses but computer and related technologies offer the unique possibility of exploiting animation and interaction.

Furthermore, a computer interface language, which is consisted entirely of icons, would have many advantages. It would avoid the need for foreign language translation, it would assist those with language and learning difficulties, even the people who can't speak, and it would help in the teaching of new systems. Also, the use of pictures and gestures to convey our ideas is a basic form of communication that two people frequently resort to when they find they share no common language.

Because you cannot speak does not always mean that you cannot write or maybe produce some kind of mark. This possibility is often ignored. It is a basic human need to make your own mark, and is of psychological importance not only to read or recognize something, but also to have some fast personal means of having an input into your care or daily life. Ideally you need a way expressing your thoughts and feelings, negative as well as positive.

All these ideas are based on a concept called: communication. One should understand the meaning of the icon, remembering it if it is not the first time he/she saw it, and so on. It is a problem of sending a message from the designer to the user, of coding and decoding the meaning – as we will see soon. Communication can be described as the transfer and exchange of messages between people, and in iconic communication the icon is the message transferred between the designer and the user.

A computer interface typically offers the user explicit information and options. Communication between humans however, usually includes nonverbal implicit information, perhaps in the form of intonation, gesture or expression, which serves to interpret the given message. Indeed these value-added features of spoken language can often communicate more than the words themselves, particularly if the words are in a foreign language.

Icons offer a rich potential for communication across natural language barriers. If confined to the European arena, the many-shared conventions make their design much simpler and their correct, true interpretation more certain. The computer provides an ideal device for the implementation of a flexible iconic communication system. The need for such a system is made more urgent by the increasingly international nature of commercial, educational and social communication. Examples such as booking a hotel room abroad, ordering machine parts from a foreign subsidiary all provide occasions for such a system to prove its worth.

An icon can be seen first by its perceivable form (syntax), second by the relation between its form and what it means (semantics), and third by its use (pragmatics). In general, the semantics of icons should be based upon their real-world domain, so the semantics of crisis situation should be based on this specific domain. The meaning of the icon may not simply be its denotation but rather its pragmatic effect.

Starting with a more general point of view, we could say that the iconic communication is a metalanguage. According to Hayes-Roth [7], the representation used is a form of metaknowledge. So we define a language to explain another language.

2.2 Semiotics

Lets start first by giving a definition of this term. The shortest definition is that it is *the study of signs*. We can continue on with the question: What is a sign; and so on, but the most important thing is that semiotics could be anywhere (see Chandler [4]).

One of the broadest definitions is that of Eco [6], who states that 'semiotics is concerned with everything that can be taken as a sign'. Semiotics involves the study not only of what we refer to as 'signs' in everyday speech, but also of anything, which 'stands for' something else. In a semiotic sense, signs take the form of words, images, sounds, gestures and objects. Whilst for the linguist Saussure, 'semiology' was 'a science which studies the role of signs as part of social life'.

For Morris [9], semiotics embraced semantics, along with the other traditional branches of linguistic, looking almost the same as the approaches defined above for the icons:

- *Semantics*: the relationship of signs to what they stand for;
- *Syntactics* (or *syntax*): the formal or structural relations between signs;
- *Pragmatics*: the relation of signs to interpreters.

In our application, one of the steps is to transform the icons into text. Semiotics is often employed in the analysis of texts (although it is far more than just a mode of textual analysis). Here it should perhaps be noted that a 'text' can exist in any medium and may be verbal, non-verbal, or both. The term *text* usually refers to a message, which has been recorded in some way (e.g. writing, audio- and video-recording) so that it is physically independent of its sender or receiver. A text is a composition of signs (such as words,

images, sounds and/or gestures) constructed (and interpreted) with reference to the conventions associated with a genre and in a particular medium of communication (Vaillant [21]).

Lets make now the connection between the semiotic and its subdivision - the icons. In the assertion of the semioticians icon/iconic is a mode in which the signifier is perceived as *resembling* or imitating the signified (recognizably looking, sounding, feeling, tasting or smelling like it) - being similar in possessing some of its qualities: e.g. a portrait, a cartoon, a scale-model, metaphors, imitative gestures.

Turning to *icons*, Peirce [16] declared that an iconic sign represents its object 'mainly by its similarity'. A sign is an icon 'insofar as it is like that thing and used as a sign of it'. Indeed, he originally termed such modes, 'likenesses'. He added that 'every picture (however conventional its method)' is an icon. Icons have qualities, which 'resemble' those of the objects they represent, and they 'excite analogous sensations in the mind'. Unlike the index, 'the icon has no dynamical connection with the object it represents'. Just because a signifier resembles that which it depicts does not necessarily make it purely iconic.

Semioticians generally maintain that there are no 'pure' icons - there is always an element of cultural convention involved. Peirce [16] stated that although 'any material image' (such as a painting) may be perceived as looking like what it represents, it is 'largely conventional in its mode of representation'.

Cook [5] asks whether the iconic sign on the door of a public lavatory for men actually looks more like a man than like a woman. 'For a sign to be truly iconic, it would have to be transparent to someone who had never seen it before - and it seems unlikely that this is as much the case as is sometimes supposed. We see the resemblance when we already know the meaning'. Thus, even a 'realistic' picture is *symbolic* as well as iconic. So he proposed the next icon as the international standard for the concept of 'man'.



Figure 1. The icon of a man

It is easy to slip into referring to Peirce's three forms as 'types of signs', but they are not necessarily mutually exclusive: a sign can be an icon, a symbol and an index, or any combination.

Eco [6] argues that an image (e.g. a human face) is made up of smaller word-like units (i.e. eye, nose, hair, etc.). Semioticians define these smaller meaningful units as iconic signs and argue that their combination within the image results in more complex,

meaningful units, called semes, which correspond to a verbal sentence. Thus the level of first articulation is made up of iconic signs. In Eco's analysis the pre-semantic units of second articulation in film are defined as *figurae* (e.g. geometrical elements, light contrast, figure-ground relations).

Pure icons, therefore, rely initially on recall of a previous visual experience on the part of the user (either first or second hand) with sufficient particularity to make their use in a particular context clear to him.

2.3 Encoding/Decoding

Contemporary semioticians refer to the creation and interpretation of texts as 'encoding' and 'decoding' respectively. Even 'encoding' might be more accurately described as 'recoding'. In the context of semiotics, 'decoding' involves not simply basic recognition and *comprehension* of what a text 'says' but also the *interpretation* and *evaluation* of its meaning with reference to relevant codes. What is 'meant' is invariably more than what is 'said'.

We can consider icons as way of coding and decoding from two aspects. One is the presence of the sender and receiver structure, and the other one is the fact that icons encrypt in a way the meaning of a specific concept.

2.4 Icons and context

The interpretation of gestures that accompany spoken language can be related to the context, but can often significantly improve understanding, notably in a situation where the spoken word is difficult to hear or to comprehend. Many people speak little of any foreign language but are able to understand and communicate a surprising amount with intuitive sign language. As well as these multilingual gestures there are many internationally understood symbols (such as arrows to indicate direction and overlaid diagonals to indicate negation), a study of which can usefully be brought to the design of our icons.

Icons alone are meaningless. An icon in a particular context, however, triggers memories and associations in a user's mind that produce what we refer to as the meaning of the icon. Only in a particular context and interpreted by a human mind does the icon have any meaning.

Context is the situation in which we view the icon. It consists of all the things in the field of view that add to or interfere with the icon. We can think of context as all the hints that nudge us toward one particular interpretation of an icon. Each icon has a matrix of possible meanings depending on the contexts and users.

$$\text{Icon}_i + \text{Context}_j + \text{Viewer}_k \Rightarrow \text{Meaning}_{ijk}$$

The message is what the icon is all about. If the user recognizes the icon as representing the concept you needed, the message was understood. There are two kinds of messages: factual and emotional. Factual messages represent specific information. They are ideas or concepts that we want the user to recognize intellectually. Emotional

with non-linguistic. Examples include facial expressions, eye contact, gestures, tone of voice indicative of emotions and proxemics that accompany speech. People simultaneously use non-verbal communication to compliment and supplement verbal communication.

Chapter 3

Methodology

The creation of user interface is not an easy thing to do, because the designer should first of all have ‘the human touch’ while creating the shapes of the objects, choosing colors, expressing movement and so on. It is known that if a picture is a little bit too complicated, two people can understand different things out of it. Even the colors can be tricky. For example, the color red can express danger, warnings, but in the same time is the color of the human healing, life and health.

3.1 Advantages vs. disadvantages in using icons

Let us see first what are the advantages and disadvantages in creating an application based only on icons. One first advantage it could be that the communication through the icons is a fast one. The meaning of an icon is being extracted immediately, because it is a 100% visual method in sending the information, and usually, this way of communication, the visual one, is the most developed one in the human being. The second advantage is that you type less. As the motto of this paper says: If a picture is worth a thousand words, an icon must be worth at least a sentence! Using one icon spares us in typing a lot... plus, these icons are independent from any language, but dependent on the culture – and this is the first disadvantage. Another advantage is the fact that an icon needs no translation, because it is self – explaining. On the other hand, of the disadvantages in using the icons, is the fact that as much as we could try to build an international system of icons, recognizably all around the world, it will be impossible. People are too different to create one way of interpreting an icon.

3.2 The design of the icons

The icons are normally diagrammatic and may be either combined with, or interspersed with, symbols (arrows, numerals etc.). It is unfortunately apparent that their style is as subject to fashion as any other design product, even typefaces, and that the ideal, timeless range of icons is unlikely to be found. It is, however, possible to avoid the excesses of fashion, and to produce icons, which have a long life span and are amenable to future stylistic updating without compromising continuity of recognition. But one thing is for sure: pictographic sign language is more calligraphic in its visual character.

There are, however, existing signs, symbols and icons which are understood internationally and it is the aim of this research to discover the level of subtlety of communication which icons can achieve, and to develop a computer-based iconic language.

When an iconic interface contains many icons it is easy for the user to get confused. There are various remedies to this situation: the careful design of icons, consistency and respect for conventions are all essential.

There are three levels of articulation defined for icons: the first articulation is the iconsign (atomic icon), which can be composed into compound icons; the second level of articulation is represented by visual elements (shape, color, texture, etc.); the third level

of articulation is represented by the dynamic dimension of motion (static or in movement).

Whilst it is possible to move from a thought to its expression without reference to natural language, it is likely that grammatical elements will often be recognisable. When this arises it might be that kinetic icons and nouns represent verbs by static icons, for example:

- To run:  like a movement icon
Torun.ico
- Medicine:  like a static icon
Medicine.ico

Though it is possible for the tense of a verb or number of a noun to be implicit in its context.

Criteria for the icons are that they should be:

- Graphically clear;
- Semantically unambiguous;
- Without cultural, racial or linguistic bias;
- Adaptable (open to modification to express nuance);
- Simple (created within a 32x32 matrix).

One good question that can be raised is why the icons look like the way they look? Why this graphical approach, and not another one? Why are some icons used with background color, and some others have transparent background? First of all I have to say that I have used some icons that have already been created by others, taking them from the huge databases that one can find on Internet. But still, not all of them were to be found, so I have inspired myself from Horthon's book [8], in which I have applied some of the rules mentioned above.

My icons were created in Easy Icon Editor, software that allows you to design by yourself your own icons. I have used my imagination and intuition in creating them, and usually I liked to use the colors as much as possible.

3.3 Creating the interface

As it is desirable that the system should not rely at any stage on any natural language, an icon whose meaning is not immediately obvious to the user should be able to explain its meaning in terms of more fundamental icons or through diagramming its evolution from its source imagery.

In our application, we have defined fourteen categories of icons, according to their meaning. Each one of these categories will have a representative icon that will show the main characteristics, being like a hint for the 'background' list. The categories are: Cars, Elements of crisis, Human actions, People, First aid, Numbers, Yes/No signs, Directions, House, Information, Intonation, Special signs, Military and Time. It would, for instance, be possible to 'click' on a compound icon (standing for an event or concept having several elements) and have its meaning expressed by several static base icons

(standing for those individual elements) – this is the idea of category. The icons from behind are in some cases nuances of the first (main) icon, like the example: starting from ‘People’ category, which distinct icon is the one of a Man, we will have displayed the icons of Man, Woman, Fireman, Policeman, Doctor, Victim, Dead Person, Person with handicap, Soldier, Bomb squad and the general idea of People. In the other situations they develop together a basic idea, being more constructing the category, than defining it.

‘Clicking’ on those basic icons might initiate a simple, but revealing, transformation from a more comprehensive (possibly photographic) image showing the source of the icon, to the icon itself. The use of relevant photographic images as backgrounds to messages could establish the current domain, and the use of colour will be a helpful adjunct but cannot be used to carry essential information that might become lost on a monochrome system.

In designing our icons we used a corpus – based approach. First we generated messages about crisis situations. Next we defined icons for the user concept, and finally we created these icons in ‘logical’ categories.

The icons will therefore comprise several levels. The top level will present the message and will incorporate compound icons and symbols where needed. Let us explain at least the design of the basic icons. For category ‘Cars’ we have choose to display the icon of an ambulance car, because is the most representative for a crisis situation; we always thing about an ambulance, not a police car, not even at a fire truck. The shape of an ambulance is distinguishable by its shape, colour and the special Red Cross sign. We didn’t put any background colour because there is enough contrast between the white colour of the ambulance car and the default grey from the buttons.



Figure 3. The icon of the ambulance car

For the category ‘Crisis elements’ we have picked up the icon of the flames, because it is the most used one, the most suggestive according to the topic of our vocabulary. When you say crisis situation, one of the first, if not the first one, is the word fire. It is so widely spread and often used, because fire is a consequence of a crash, explosion, bombing, and so on. The colours were used to give the image of expansion and a kind of a movement, red representing the heat but also the danger.



Figure 4. The icon of the flames

In the case of expressing directions, what can be more appropriate than to put some arrows? These are one of the only items that are worldwide spread, but there are still some fashion elements everywhere. Why did I choose this kind of arrow? Mostly because the fact that is divided in two sections gives the impression of space and movement. Why a left oriented arrow and not a right one? For no reason, perhaps just

because it was the first one in the list, alphabetically speaking. In this situation, adding a background, like the standard one in a grey colour, would make the arrow become invisible. In fact, for the fact that some icons have a background colour and some other don't, it was just a matter of intuition and common sense. Personally I think that some of the icons look better without a colour on the background.



Figure 5. The icon of the left direction

Let us go further to the presentation to the category 'First aid'. The icon selected to represent this category is actually the icon that has the same name: First aid, and also the text, of course. When you say first aid, first you imagine yourself a box with everything you need inside. The colour of the box was just a random one.



Figure 6. The icon of the first aid box

For The 'Buildings' category we have chosen a house, because it was the clearest image between them, being also expressive in shape and even in colour.



Figure 7. The icon of the house

Talking about human needs in a crisis situation, you think about helping them, so this is the reason I have chosen the icon with this topic, plus it is a way of showing the fact that it is a human activity – the presence of a hand.



Figure 8. The icon of the helping activity

To be in a difficult situation, perhaps quite a dangerous one, you need ways to communicate what you are seeing or hearing, but also receiving some information back. The image of a lightened bulb is a symbol of knowledge; ideas and reason – that is why we have chose this icon in particular.



Figure 9. The icon of the information symbol

In the case of the category called ‘Intonation’, where there are only two icons: the question sign and the exclamation one, the choice was not difficult at all, because the most used one is the question sign.



Figure 10. The icon of the question sign

Not even for the numbers the choice was a random one. We had opted for number one because it is the first of them, representing the start, even if it might not be the most used one. The colour used for these icons was a choice for the moment, and the fact that we didn’t add some background colour, only proves that fact – that we wanted the icons to look different, various. It is a personal opinion and choice.



Figure 11. The icon of number one

Another numerous in elements category is the one called ‘People’, and the basic icon for it is the one that represents a man. Why didn’t we choose the icon with people? Most of the fact that the noun people is by default on plural form, which is an exception in this category, and another reason is that the man is by definition a relevant element. It might look quite expressive, almost individualised, but in this category we have chosen that all the elements should be particular.



Figure 12. The icon of the man

For the category ‘Special signs’ we had to choose between two icons: the one that represents the exit, and the other one was the sign of negation. We have preferred the first one because it is more expressive, being a true special sign. The colour was putted to give some more colours to the application.



Figure 13. The icon of the exit sign

Not a lot of explanations to make on the basic icon of category ‘Time’. The clock defines the time concept. This icon is a little bit too detailed, so we didn’t add any other colour, not even for the background.



Figure 14. The icon of time, clock

The category ‘Yes/No’ can not be used in this step of the application, because the yes/no signs are implying a communication between the client application and the server – which is not our concern by the moment; but still, they appear in the designing interface. Between the two signs we have opted for the one that express a positive assertion, of course.



Figure 15. The icon that express yes

The last category to be analysed is the one of the military elements. For this one we have chosen a tank. Why? We thought that is was the most suggestive one. It is, of course, represented in grey colour, and quite in detail designed, so other colours, even for background were out of discussion.



Figure 16. The icon of the tank

Querying any icon will initiate a move to the second level, at which base icons will seek to explain the meaning of the top-level icon, acting out the meaning if appropriate. A further level could trace the icon's development from suitable photographic references to its source, and at any level natural language could act in support. The user on entering the system would have chosen the natural language, which is active.

Additional weapons in the design armoury are therefore:

- Color, which might be used either expressively or for coding;
- Movement, which is potentially very powerful (most obviously for dealing with time dependent concepts such as movement) but not to be used lightly;
- Background, which can be used as a visual cue to the domain.

The judicious use of colour can be employed to good effect, most obviously for coding, but the system must be able to communicate fully in monochrome. Background images, however, could help to establish context for a message and multimode systems developments are bringing closer the option of using video in that role – like a future step in continuing this project. In fact, since much multi-media production is visual, and hence a lingual, it would be a very appropriate medium in which to use an iconic language.

According to Horton [8], in creating an icon: some concepts are inherently to understand, whether we represent them with icons or with words. For such issue is not whether icons require study, but whether they require less or more study than equivalent word labels. This was the issue with creating the icons for concepts like ‘I’, ‘there are’. They are concepts defined by humans, can’t be touched, can’t be seen. We could have made the compromise of defining some textual icons, like for ‘me’, or ‘I’, the use of a

sign with the letter 'I' on it. We had made the convention that we won't use any words, and even if English is so widely spread. The way to complete these tasks, of adding such constructions, was to create some templates, but this subject will be developed, as we will see next.

Nevertheless, icons are not pictures. Icons are meant to be viewed entirely in a single glance and, once learned, recognized automatically. Overloading an icon with realistic detail may render it less rather than more recognizable, but sometimes it is useful. Icons should be concrete, showing real-world objects, vivid, clearly depicted, and conceptually distinct one from another.

Chapter 4

Introduction in Natural Language Processing (NLP)

Life would be so much easier if we could communicate with the computers conversationally. After all, everyone knows how to use a ‘natural language’ such as English or French, but not everyone knows how to use obscure command like the ones from the Unix environment. The goal of natural language processing (NLP) is to make conversational computer communication a reality.

To achieve that goal, computers must understand language, an extraordinarily difficult task because language is ambiguous. Words can have more than one meaning, pronouns can refer to many things, and what people say isn’t always what they mean.

Natural language processing, or computational linguistics, is a field combining computer science and linguistics that focuses on the problems of modeling language on computer.

Sophisticated NLP systems try to ‘understand’ language by incorporating knowledge of how sentences are constructed grammatically, and through an ability to draw inferences and explain their reasoning. This is in contrast to systems that take a keyword approach (also called template systems or pattern matchers), in which patterns in the input are matched to stored templates.

4.1 The natural language

As we have mentioned above, the basic field in which we develop this application, is the one of natural languages. Lets see now a definition of this term. According to the British Self-Explaining Dictionary, a natural language is a language spoken or written by humans, as opposed to a language use to program or communicate with computers. Natural language understanding is one of the hardest problems of artificial intelligence due to the complexity, irregularity and diversity of human language and the philosophical problems of meaning.

Our approach in this area was to start from a vocabulary and then defining the grammar, not the other way around. This thing is very delicate due to the complexity of the elements involved in the problem: grammar rules, which would create perfectly, correct sentences, speaking from the semantic and syntactical analysis.

In Allen [1] we can find different approaches to the natural language understanding, according to all the layers of knowledge. The different forms of knowledge have traditionally been defined as follows:

- Phonetic and phonological knowledge concerns how words are realized as sounds. While this type of knowledge is an important concern for automatic speech-understanding systems, there is not the space to examine these issues.
- Morphological knowledge concerns how words are constructed out of more basic meanings units called morphemes.
- Syntactic knowledge concerns how words can be put together to form sentences that look correct in the language. This form of knowledge identifies how one word relates to another (for example, whether one word modifies another, or is unrelated).

- Semantic knowledge concerns what words mean and how these meanings are combined in sentences to form sentences meanings.
- Pragmatic knowledge concerns how sentences are used in different contexts and how context affects the interpretation of the sentence.
- World knowledge includes the general knowledge about the structure of the world that languages users must have in order to, for example, maintain a conversation, and must include what each language user must know about the other user's beliefs and goals.

We are dealing only with the syntactic and semantic knowledge. Starting with this idea, we pass now at another problem: defining the grammar.

4.1.1 Components of an NLP system

Linguists divide language into processing levels, a division reflected by natural language systems that incorporate components for some or all of these levels in their architectures. Morphology is concerned with how words are formed from basic meaningful units called morphemes. 'Cover' is a morpheme; add the morpheme 'un' to the beginning and you have another word, 'uncover'. 'Walk' is a morpheme; adding the morphemes 's' and 'ed' to its end to the form 'walks' and 'walked' form various forms of the verb.

Closely connected to morphology is the lexicon, the dictionary of an NLP system. It holds the vocabulary that the system understands and often contains morphological information. Because its organization can differ radically from one system to another, a major design decision is how much and what kind of information is to be included in the lexicon.

NLP systems don't always perform morphological analysis. The system designers might choose to include all possible forms of a word in the lexicon. In such a lexicon the word 'walk' would have four separate entries: 'walk', 'walks', 'walked' and 'walking', while in a lexicon containing morphological information it would have a single base entry of 'walk', plus the information that 'walk' is a regular verb. The system would then use morphological rules to reduce the other forms of the verb.

Syntax is concerned with the grammatical rules for constructing sentences. The syntax of the sentence is analyzed using a grammar that formally defines the structures permitted in the language, and a parser that analyses a sentence according to the grammar and produces a structural description of a sentence.

The syntactic component of system parse each sentence in the input to determine its syntactic structure and produces output in the form of a parse tree that describes the sentence structure. Often a sentence will have more than one possible structure, in which case the system must determine which analysis is most likely to be correct. This determination frequently involves obtaining information from other components of the system such as semantics or pragmatics.

A phrase-structure grammar is often used to define the grammar. This is a context-free grammar consisting or rewrite rules defining the permissible constructs of the language. The grammar defines a sentence as being made up of a noun phrase

followed by a verb phrase. A noun phrase consists of either a determiner followed by a noun or a noun alone. A verb phrase is made up of a verb.

VP -> V NP

Another frequently used grammar formalism is the transition network. Transition networks are finite-state automata consisting of nodes and labeled arcs. The arcs are labeled with the word category being looked for. When it reaches 'pop' it pops out of the network.

A more powerful version called an augmentation network (ATN) is commonly used in NLP. ATNs increase the descriptive power of the grammar by using recursion to define the networks and by using mechanisms such as registers and tests, which increase the number of operations available. A grammar requires a parser that will take an input sentence and analyze it according to the rules of the grammar. Natural language parsers use parsing strategies from programming language theory and those tailored to natural language. Some types of parsers include shift-reduce parsers, chart parsers, and statistical parsers.

Semantics is about the meaning of words and sentences. It must cope with lexical ambiguity, the fact that a single word can have more than one meaning.

For example, by using the icon that represent a fire

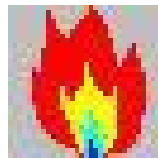


Figure 17. The image of some flames

One would say that it means flames, and some other that it means the verb to burn. Or the icon representing and suggesting a deaf person



Figure 18. The image of a deaf person

Could be interpreted like being a person who simply doesn't want to hear, covering his/hers ears. There are also different ways of expressing the same meaning. Let us take the sentence: The building is on flames. Which would have the next visual representation:



Figure 19. The representation of the sentence: The building is on fire.

One-way of twisting the sentence is the simple reversion of the words, and the sentence would become: There are flames in the building, which has the following appearance:

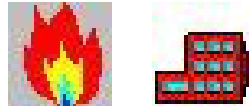


Figure 20. The representation of the sentence: There are flames in the building.

A logical form is the output of the semantic component; it is frequently a first-order predicate logic representation of the meaning of the sentence. In our case, the most general representation looks like this:

<negation> <number> <adjective> <noun> <negation> <verb> <adverb> <number>
<adjective> <noun> adverb>

With some small interference from the fact that after the first noun we could end the sentence with a question sign; plus, keeping in mind that all of these elements are optional according to the syntactical constraints. For example, if we add an adjective to the input data, we must add also a noun, because the adjective could only sit ‘in front’ of a noun, and only a noun.

Most NLP systems perform syntactical processing and then semantic processing, but some go directly to the semantic levels, or perform syntactic and semantic analyses in parallel to try to constrain the number of possible parse trees by using semantic information.

Pragmatics is concerned with language use in context. The pragmatic component of an NLP system is the component that identifies the question ‘Can you tell me what is the time?’ as one which someone would like to receive the exact time, rather than a yes or no answer.

World knowledge is important for understanding and maintaining a conversation. In order to respond properly to a question, an NLP system may need to know what you are referring to. For reasoning purposes it may require an inference mechanism. World knowledge is encoded in a knowledge base that can be accessed by all the components of the systems.

These components of a NLP system must be able to interact and communicate with each other, because knowledge from one component can inform another. There is no single accepted architecture for NLP systems; they can be implemented using many approaches and formalisms. Systems may have some or all of the above components and they can be linked together in various ways.

4.1.2 Language problems

NLP systems implement strategies for handling common linguistic problems caused by ambiguity of language and by the fact that people can use language nonstandard ways and still be understood. A good test of an NLP system is its ability to

cope with, a system can resolve such sentences, determines how successful it is in understanding a reasonably large subset of natural language.

Prepositional phrase attachment is a problem in NLP. A sentence can often be analyzed in more than one-way, producing multiple parse trees for the sentence. Prepositional phrases often produce such ambiguity. Reference to semantics and world knowledge can help to disambiguate the structure. Lexical ambiguity occurs when words have multiple meanings, as was discussed in the example of the ambiguity of the word

Anaphoric reference, or pronoun resolution, is the problem of figuring out what a pronoun refers to. A common technique for resolving this is to create a history list of all previously mentioned noun phrases and assume that a pronoun refers to the most recent noun phrase that meets the constraints (number, gender) of the pronoun and various pragmatic constraints.

Ellipsis refers to sentences that appear to have some parts missing. The missing word or phrase is parallel to one in the previous sentence or phrase:

The doctor helps the man, the fireman the handicapped person.

In this case, 'the fireman the handicapped person' is an elliptical phrase. It lacks a verb but is understandable in the context of the entire sentence because we draw a parallel from the first phrase and assume it's an elliptical form of 'the fireman helps the handicapped person'. But such sentences can't be created still by our application, because there will be created only simple sentences, not at all phrases. The reason why we add such a severe constraint is that a person in a crisis situation doesn't need to send long sentences, and writing novels. The basic sentences are just enough for transmitting the fact data. Plus, everything connected to the grammar would just have been too complicated to deal with.

Quantifier scope is another problem area. Quantifiers such as all, every, some, and no can be ambiguous in their interpretation. Nevertheless, in our application we had defined some quantifiers in order to give more space and wideness to the vocabulary, and also thinking that those words are very useful and often used in explaining something. One of the quantifiers is the negation, which represent a self-standing category in our specific grammar definition. This item can be used next to a noun, where it has the meaning of 'no', or next to a verb, when is negating the activity. Let us give some concrete examples for that. In the first case, we take the sequence:



Figure 21. The problem of <negation> + <noun>

It's meaning being: There is no victim. It is also expressing the fact that specific object is not in a certain place. This construction can be used in a normal sentence, i.e. No doctor run towards the exit, or by activating the first template, which is the one of the expression: There is, There are, and this is our case.

The second situation is placing the negation in front of a verb. We should also mention the fact that a negation could only be positioned in front of a noun or a verb, so that we will accomplish the syntactic production rules. In the verb case, the negation will

generate the adding to the output text or the word ‘can’. So, if we have the sequence of icons like follows:



Figure 22. The problem of <negation> + <verb>

The meaning it would be: ‘The doctor can not help the victim.’

No NLP system has completely solved these problems, and therefore no system can yet cope with unrestricted natural language input.

4.1.3 Approaches to NLP

Different approaches to analyzing natural language exist. A convenient way of distinguishing them is as follows:

- *Linguistic* – Based on encoding formal grammar rules for sentence-level processing. A linguistically oriented system often focuses on the syntactic level, or on syntax and semantics.
- *AI* – Focuses on using world knowledge to understand language.
- *Connectionist* – Uses neural nets for processing language, particularly for lexical disambiguation.
- *Statistical* – Based on extracting statistically significant information from large corpora (millions of words of text).

Most NLP systems use a combination of the linguistic and AI approach. Statistical methods are now receiving much attention, and more systems are likely to incorporate them in the future. The extent to which a system concentrates on certain components of language, attempts to solve a particular linguistic problem area, or chooses one design approach and formalism over another depends upon the type of application, its domain of use, and the philosophy of its designers.

4.1.4 Language Universals

There are more than 6000 languages in the world not counting dialectal differences within languages. All languages in the world share certain features. Here are some **language universals**.

- Wherever humans exist, language exists. Is language innate?
- There are no ‘primitive’ languages. All languages are equally ‘complex’ or equally ‘simple’.
- Any child can learn any language if exposed to it; there, however, is a critical age for learning language.
- All languages change through time. Only the ‘dead’ languages remain unchanged. The structure of language is highly resistant to change. The content of language is most susceptible to change.

- All languages have grammatical categories such as nouns and verbs and semantic categories such as ‘animate’, ‘inanimate’, ‘male’ and ‘female’.
- We can talk about the past, make plans for the future, imagine things that are not in the immediate surroundings and do not even exist, lie and mislead, and pass information to successive generations.
- Language is hierarchical: English, for example, has phonemes or speech sounds (approximately 45), syllables (about 4,000) made up of one or more phonemes, morphemes (about 20,000) consisting of one or more syllables, words (in excess of 500,000) made up of one or more morphemes, and phrases and sentences (unlimited) consisting of one or more words. This permits creation of novel sentences and sentences of unlimited length.
- All natural languages have the **duality of patterning**, i.e. the basic building blocks (phonemes and most syllables) are meaningless, which when combined into larger units (morphemes, words and phrases) become meaningful.

4.1.5 Creating the vocabulary

It is essential to mention that for our application there were multiple ways of defining the problem: first of them was to start from the grammar, and then generate all the words out of it; and the second one was to start the other way around – from the vocabulary, and then build the grammar. This last one is the approach that we have followed.

A good question is: how did I define this specific vocabulary? The answer is very simple: I just took a piece of paper, and started to put down all the key words that I had in mind connected to the topic of a crisis situation. At the beginning there were only thirteen categories, the Military one was not created, because I had considered it to be a little bit too extreme. The reason of adding this last category is the fact that this application will be used by other applications that will require the existence of this specific topic.

Another aspect to be presented, is that the list of the keywords changed on the long of the development of the application, adding more words, sometimes excluding some of them, like ‘dog’, ‘insect’, ‘warning’; words which I have considered at the beginning of being important. Along the developing of the program I have noticed that these words are very isolated ones, and could create just very few sentences, or the fact that they didn’t fit anymore in the generating language.

After having the vocabulary, we had to create some categories of them, grouping them after a specific topic, which would be characteristic for all the icons contained by it. The reason for grouping them was first the fact that we just don’t have place for displaying them all on the screen, especially if we are intending to make this application work on PDA. As we will see in the designing of the interface part, the dimensions of the MainForm – the form that represents the main user interface, has fixed dimensions, due to the fact that the screen of a palm is a very small one, sometimes having problems even with the resolution of the images. The second motive, also a very strong one, is that it is just out of hand, and extremely inefficient to start and search which icons to use. The user should be helped to insert the needed icons, not to block him/her for the first contact to the application. We should create an application that gives as many hints to the user about which icons he should press so that he will transmit a specific fact.

There was the suggestion of grouping the icons after their grammar category, so that in one category there should be numbers, in another one just the verbs, and so on. I have found this approach of being inefficient and too scientific: inefficient, because the distribution of the icons over the grammar categories is not an average one – meaning that there are only ten numbers, three adjectives, and more than seventy nouns. I called this approach too scientific, because a person, especially when he is in a crisis situation, he does not think in verbs, adverbs and nouns, which are abstract elements, but in concrete things, like images. So, defining some special categories, that represent almost the same idea, was the best one.

Let us see now the list of one category, the ‘Crisis situations’ one, and the ‘words’ that are contained in it. The rest of the categories would be presented in Appendix ???

Category *Crisis*

- Flames;
- Electricity;
- Explosion;
- Air;
- Smoke;
- Bomb;
- Gun;
- Knife;
- Toxic;
- Flood;
- Tornado;
- Nuclear threat.

4.2 Grammars and productions

We will have to define a context free grammar (CFG) according to Chomsky’s hierarchy. Thus we move on to consider the notion of a **grammar**. This is essentially a set of rules for describing **sentences** - that is, choosing the subsets of possible combinations that can be constructed, in which one is interested. Formally, a grammar G is a quadruple $\{N, T, S, P\}$ with the four components:

- (a) N - a finite set of **non-terminal** symbols,
- (b) T - a finite set of **terminal** symbols,
- (c) S - a special **goal** or **start** or **distinguished** symbol,
- (d) P - a finite set of **production rules** or, simply, **productions**.

The word ‘set’ is used here in the mathematical sense. A sentence is a string composed entirely of terminal symbols chosen from the set T . On the other hand, the set N denotes the **syntactic classes** of the grammar, that is, general components or concepts used in describing sentence construction.

The union of the sets N and T denotes the **vocabulary** V of the grammar.

$$V = N \cup T$$

The sets N and T are required to be disjoint, so that

$$N \cap T = \emptyset$$

In this situation \emptyset is the empty set.

4.2.1 Backus Naur Form

We go further to introduce the concept of BNF (Backus Naur Form), originally ‘Backus Normal Form’, which is a metasyntax, used to express context-free grammars. BNF is one of the most commonly used metasyntactic notations for specifying the syntax of programming languages, command sets, and the like. It is widely used for language descriptions.

A BNF rule defining a non-terminal has the form:

Non-terminal ::= sequence_of_alternatives consisting of strings of
terminals or non-terminals separated by the meta-symbol |

In classic BNF, a non-terminal is usually given a descriptive name, and is written in angle brackets to distinguish it from a terminal symbol. Non-terminals are used in the construction of sentences, although they do not actually appear in the final sentence. In BNF, productions have the form

$$leftside \rightarrow definition$$

Here ‘ \rightarrow ’ can be interpreted as ‘is defined as’ or ‘produces’ (in some texts the symbol $::=$ is used in preference to \rightarrow). In such productions, both *leftside* and *definition* consist of a string concatenated from one or more terminals and non-terminals. In fact, in terms of our earlier notation

$$leftside \in (N \cup T)^+$$

and

$$definition \in (N \cup T)^*$$

Although we must be more restrictive than that, for *leftside* must contain at least one non-terminal, so that we must also have

$$leftside \cap N \neq \emptyset$$

Frequently we find several productions with the same *leftside*, and these are often abbreviated by listing the *definitions* as a set of one or more alternatives, separated by a vertical bar symbol ‘|’.

From a grammar, one non-terminal is singled out as the so-called **goal** or **start symbol** (the symbol noted with S in the definition of the grammar). If we want to

generate an arbitrary sentence we start with the goal symbol and successively replace each non-terminal on the right of the production defining that non-terminal, until all non-terminals have been removed.

4.2.2 Connection between a context-free grammar (CFG) and a BNF construction

As we have said before, we are using a context-free grammar, based on the definition of some BNF grammar rules. A BNF grammar is similar (and equivalent in power) to a context-free grammar, and most of the times it can mislead us into believing that they are one and the same thing, but in fact there are some basic differences between them. In a context-free grammar, each variable, or non-terminal, has one or more ‘productions’, which are the rules that map the non-terminal to a string of tokens (‘terminals’) and variables. The idea in the CFG is that you repeatedly replace non-terminals with their production strings until you get a string of tokens. In a BNF grammar, each variable has a single production, but the production is more than just a string; it’s a regular expression – or can be one, in case of the BNF structure that you are using.

BNF grammars can be compiled into context-free grammars, and context-free grammars are very easy to convert into BNF grammars. The method is very simple. If all the productions for a non-terminal are written together separated by the symbol ‘|’, all we need to do is reinterpret the symbols.

4.3 Grammar subdivisions

In order to construct a grammar, that should be compact as possible, with robust qualities, we had take the structure of the English grammar, considering that the resulted text will be transmitted to the server side into its English version. We had to split the icons into grammar groups according to their word classes in the sentence. So the categories are: noun, numeral, adjective, verb, negation, sign and adverb; where the Verbs are the elements from the icon category called Human Actions, the Numerals are the ones from the Numbers category, the Adverbs are the Directions, and so on. We are talking now only about the icons that could be selected, and we do not have to choose all of them into one sentence all the time. The user should have the possibility and the freedom to construct the sentence ‘almost’ as he/she wants to. That’s why we have created the templates structures: to permit communication in different ways. It may look more complicated at first, but in fact, these kind of predefined situations will ease the parsing step.

We need to make a small correction: the distribution of the categories is not as simple as it may look, and most of all, it is not enough. It will be a mistake to consider that all the nouns have the same position into the future sentence. For example, if we have the next structure <Noun> <Verb> <Noun>, the next sentence will be a correct one:

The doctor helps the victim.

but the sentence:

The doctor helps the ambulance

is not an accepted entry, even if, initially, ‘the victim’ and ‘the ambulance’ are both nouns. This situation will be taken care in the semantic analysis. We will define a network between categories, and we will even define subcategories of icons to build a very rigorous system. But this subject will be discussed later. Lets see now what are the sentence subdivisions.

4.3.1 Simple Noun Phrases

Noun phrases (NP) are used to refer to things: objects, concepts, places, events, qualities, and so on. The simplest noun phrases consist of a single pronoun, such as he, she, it, and I, or an object. The remaining forms of noun phrases consists of main word, called the head, and other words that qualify the head and identify the nature of the item being referred to. The head of a noun phrase is always a noun. Nouns are divided into two main classes:

- Count-nouns -> nouns used to describe some specific object or set of objects: numbers;
- Mass nouns -> nouns used to describe composites or substances: some, etc...

In addition to the head, noun phrases may contain specifiers, from which we use only cardinals - words indicating the number of objects being described, such as one, two, and so on, but we have made a new category out of it, which is Numbers. In the same manner, we have also qualifiers, which are a noun phrase occur after specifiers and before the head. They consist of adjectives and nouns being used as modifiers. In our case we have used only the adjectives, which are words that attribute qualities to objects yet do not refer to the quality itself.

While noun phrases are used to refer to things, sentences are used to assert, query, or bring about some partial description of the world. The way a sentence is used will be called its mood. So we have:

Declarative (or assertion)
Yes/No question
Wh-question
Imperative (or command)

The wh-question, which is referring to why, where, what, who and so on, will be constructed like templates, and they are defined by adding the question icon, ‘?’, to the end of the sequence of icons. In our case, we took into discussion only the construction of the ‘Where’ questions. In this way, the only combinations available for this topic, is the presence of a noun followed immediately by the question sign. In the same manner, the order will be suggested by adding the imperative sign, ‘!’, and the declarative sentence will be the default one. Nevertheless, there is no difference between the order sentence and the assertion one. We cannot transmit the tone of the sentence, but we can give a hint about the inner state of the user – if he uses the sign ‘!’ it means that he/she is in a serious situation. Also, if there is no question or order sign at the end of the input line, there will be tested a structure of a declarative sentence.

Verbs

We will use only verbs at simple present tense (not even continuous form). And the transitivity or passives verbs are not of our concern. It may look restricted, but by adding the grammar rules for the verbs, the structure of the network will become too complicated. Also, in direct correlation to the verbs are the adverbs. In our situation we have defined only space – adverbs, to give a more precise view of the situation: here, there, and modal adverbs: blocked, unblocked, to represent a delicate crisis situation, in which these words are keywords.

Particles

Some verb forms are constructed from a verb and an additional word called a particle. Particles must be immediately follow the verb or immediately follow the object NP. For example, walk on stairs... Adding these elements to our application, will make the sentence have more meaning, and be closer to the spoken or normal written language.

Prepositional phrases

Another ‘field’ is the prepositional phrases. The most common form of prepositional phrase consists of a preposition (on, from, to, by, through) followed by a noun phrase, which is called the object of the preposition. They can be simple or complex: get out of the car.

4.3.2 Bounding the vocabulary with the grammar categories

For defining our grammar we had ‘spread’ the vocabulary, lexicon in seven categories: negation, number, adjective, subject, verb, adverb and sign. They may not look like the typically grammar categories, and some of them could have been included in the more based ones, but in our case it would be so much easier to work with these ones. As we have already mentioned, the icons are divided into fourteen categories according to their theme, and the way that they would be used, but also in direct connection with our grammar types. Even though some of the words could have had some other meanings, we have defined the sequence of tokens in a certain manner that one icon should have only one meaning (including one grammar category). For example, for the icon ‘flames’, we could have used the noun ‘fire’, or the verb ‘to burn’ etc. In this case it would have been even more difficult to establish which meaning should be choose at a certain moment. Lets see now how they are constructed.

Only nouns construct the category of icons with the name ‘Crisis Events’. The same situation is with the group called ‘Cars’, ‘First Aid’, ‘Buildings’, ‘Information’, ‘Military’ and ‘People’. On the other hand, the categories ‘Directions’, ‘Time’ and ‘Intonation’ are formed only by adverbs, the ‘Numbers’, of course, from numbers, and ‘Yes/No’ icons are included in the adjective section. The other categories are mixed ones: in ‘Human Actions’ we can find three adjectives – blind, scared and deaf, the rest are verbs; in ‘Special Signs’ we can find a noun – exit, a verb – to move, three adverbs – blocked, unblocked and disorder, plus the icon with the negation sign, which is a self standing element by defining its own property: the negation.

4.4 Defining the grammar

In the last chapter, we have set the definition of the grammar in this way: $G := \{N, T, S, P\}$, where N is a finite set of **non-terminal** symbols, T is a finite set of **terminal** symbols, S is a special **goal** or **start** or **distinguished** symbol, and P is a finite set of **production rules** or, simply, **productions**. It is time for us to define the grammar, presenting the production rules:

$S = \text{negation } A \mid \text{number } B \mid \text{adjective } C \mid \text{noun } R \mid \text{verb } E \mid \text{adverb } H$

$A = \text{number } B \mid O$

$O = \text{verb } E \mid P$

$P = \text{verb} \mid B$

$B = \text{adjective } C \mid C$

$C = \text{noun } R \mid \text{noun}$

$R = \text{sign} \mid D$

$D = \text{negation } F \mid F$

$F = \text{verb} \mid G$

$G = \text{verb } E \mid E$

$E = \text{adverb} \mid I$

$I = \text{adverb } H \mid H$

$H = \text{number } J \mid J$

$J = \text{adjective } K \mid K$

$K = \text{noun } L \mid \text{noun}$

$L = \text{adverb}$

Number, adjective, verb, noun, adverb, sign and negation are terminal symbols, and the symbols $A \rightarrow R$ are non-terminals. As it can be noticed by testing the definition of the grammar, the only imposed symbol it may look of being the presence of at least one noun. The way to avoid the generation of the empty string is the fact that we should always have at least two entrance symbols. First should be the one from the production of the starting symbol, and the last one should be the terminal.

Only by looking at the way these rules are combined, we could distinguish two categories of tokens: the starting and the finishing element. Actually this was the starting

point in the construction of the grammar. We took into consideration all the possibilities of how these icons can come along. Our wish is to give as many possibilities of combination as possible taking care of the English grammar rules, because this is the base construction for our grammar. In the end, we have to have a text written in English, which should mean what the user wanted to transmit. To be easier to distinguish the way the categories can be combined, we are presenting a diagram with the network structure between grammar types, lighting also the starting and the ending elements.

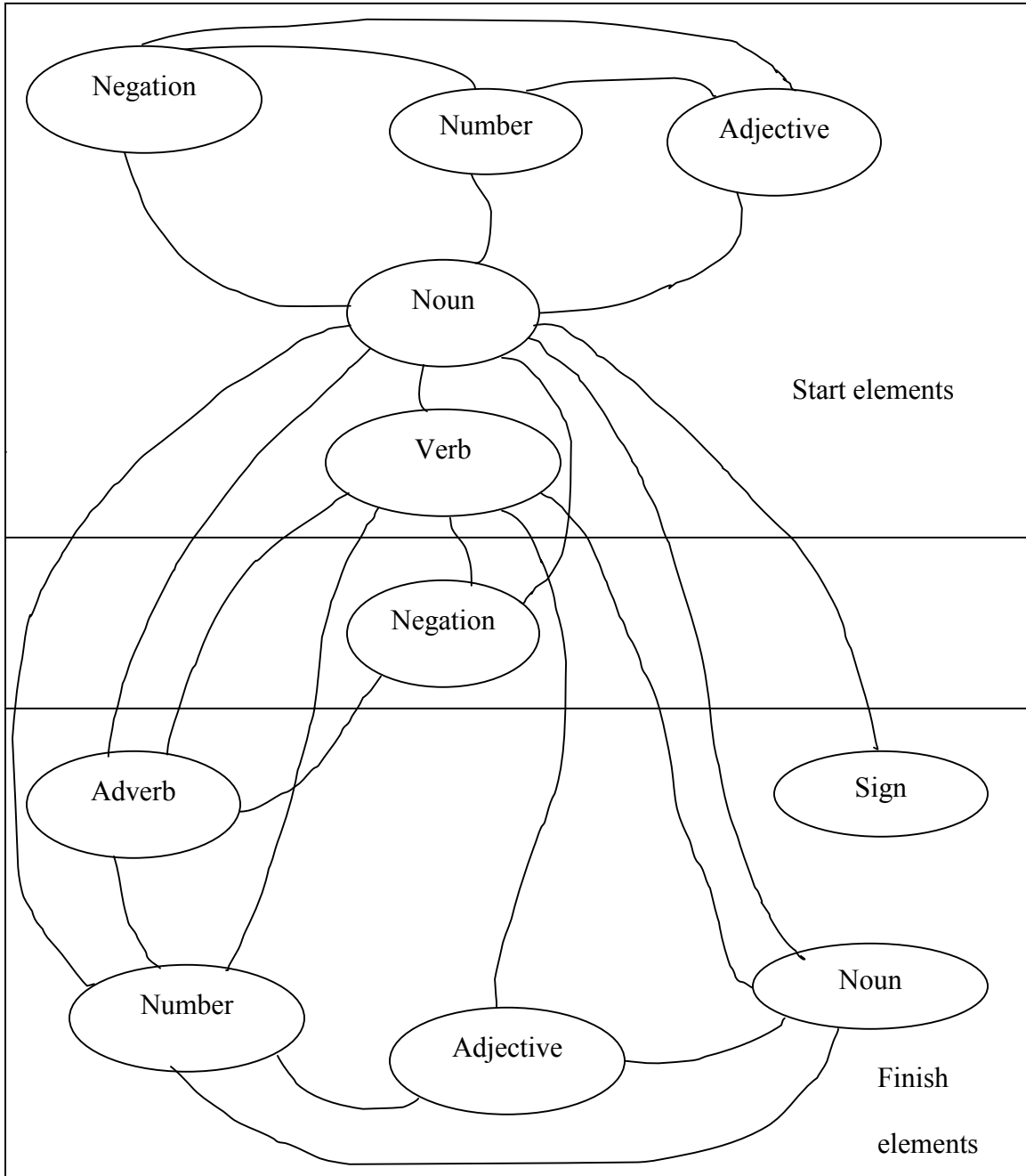


Figure 23. The visual representation of the grammar.

Going back to our grammar, we could see that the elements, which the sentence could start with, are given by the first production rule. So that means that the first element in a valid sequence of icons could be the negation, a number, an adjective, a noun, or a verb.

In the same way we could check which are the characters with which a valid icon sequence could end. These elements can be distinguished by checking the right side of the production rules. We can observe that the production rules are usually formed (except the last one, which is a direct transformation) by a terminal followed by a non-terminal, which alternates either with a non-terminal (which means that the derivation goes on, with no transcription directly to the out-put text), or with a terminal. These particular terminals are actually the ones with which a valid sequence of icons could end with. Specifically speaking, they are: noun, verb, adverb or sign.

There are a lot of 'unexpected' associations between the icons, due to the fact that the bounding words would be defined in the template constructions. For example, we could have the next situation:

<noun> <adverb>

like: The door is blocked. – where 'door' is the noun, and 'blocked' is the adverb. In the same way we could have a more complicate construction:

<noun> <adverb> <number> <noun>

as in: The road is blocked by three cars. The underlined words are the one used in the structure, which was took for example.

We have mentioned above that some of the icons could have had more that one meaning, having the possibility of being part of different grammar groups (i.e. flames vs. to burn). By using the template structure, we could create the sentence without a verb or a noun in it. Even the combination between two nouns is accepted, like in the next example:

The car had an explosion.

4.5 Properties of BNF grammars

For this discussion, we use 'children of X' to mean terminals and non-terminals that are mentioned in the production for X. We use 'string' to mean either a sequence of terminals or a sequence of terminals and non-terminals, and 'matched string' to mean a sequence of terminals and non-terminals that is matched by a particular regular expression, or grammar rules that have been defined above. We use here the concept of regular expression like the way we could combine the lexemes according to the structure of our grammar.

Subj noun subj => The house has 3 doors.

4.5.1 Emptiness

One of the most delicate aspects is the one connected with the possibility of generating the empty string (noted here with epsilon). This situation could cause problems at the parsing step. But let us see first which are the conditions to be fulfilled for producing this particular case. If the non-terminal has no non-terminal children, we can determine the fact that a grammar can generate the empty string by looking at the

form of the regular expression. Terminals cannot be empty; if the non-terminal's production is a terminal, it therefore cannot be empty. Alternations of non-empty things cannot be empty, but alternations including at least one alternative that can be empty can be empty. Sequences are their dual: sequences of possibly empty things can be empty, but if the sequence contains any things that cannot be empty, it cannot be empty.

Adding non-terminal children is easy; since for each leaf of the regular expression, we're only interested in whether or not it can be empty, we can simply compute whether the production of that non-terminal can be empty. But this fails to terminate when the grammar is recursive. But it turns out that if a non-terminal can expand to the null string (in zero or more steps), it can do so non-recursively. Here's why.

Suppose

$$X \Rightarrow^* \epsilon$$

Now, suppose X has to recurse to do this; i.e. all derivations of the form

$$X \Rightarrow^* \epsilon$$

are of the form

$$X \Rightarrow^* AXB \Rightarrow^* \epsilon$$

Since there are a finite number of steps in the derivation, there are a finite number of steps at which point the string being expanded contains X . Take the last one of these, say CXD . We know

$$CXD \Rightarrow^* \epsilon$$

and by hypothesis, there are no steps following CXD during which the string contains X . Now, for

$$CXD \Rightarrow^* \epsilon$$

to be true, all of

$$C \Rightarrow^* \epsilon$$

$$X \Rightarrow^* \epsilon$$

$$D \Rightarrow^* \epsilon$$

must be true. Furthermore, all three of these derivations must involve no X 'es. But now we have a derivation of the form

$$X \Rightarrow^* \epsilon$$

that is not of the form $X \Rightarrow^* \epsilon$, contradicting our original hypothesis.

So we can simply ignore recursive derivations -- we only need to look at non-recursive derivations to determine whether the non-terminal can derive the null string. So our new rules are:

1. Terminals can't be empty;
2. Sequences can be empty if they contain no recursive non-terminal references and if all of their elements can be empty;
3. Alternations can be empty if any of their alternatives (except for recursive non-terminals, which are treated as if they weren't there) can be empty;
4. Non-recursive non-terminals can be empty if their production can be empty.

Here 'can be empty' means 'can eventually derive the null string'.

In our specific case, the generation of the empty string is not only unwanted, but it is a situation that would block the standard implementation procedures. That is why we had imposed that there should be at least one item to be sent and checked, and as we could see from the grammar rules that have been defined, the imposed item is a noun. All the sentences should contain a noun, even if it is the noun from the starting category, or

the one from the finish one. Nevertheless, the first production rule has to be taken into consideration, and one of the items should be selected. In this way we have imposed the presence of the items in the input data, and the possibility to generate the empty string has been eliminated.

4.5.2 Starting tokens

That whole emptiness rigmarole is mostly interesting because we need it for this. To construct a predictive recursive-descent parser, we need to know what tokens can begin sequences derived from particular non-terminals. The rules for computing this are fairly simple:

1. a regular expression consisting of a terminal can derive only strings starting with that terminal. Actually, consisting of one occurrence of that terminal, but that's beside the point.
2. an alternation can start with whatever any of its alternatives can start with.
3. a non-terminal can start with whatever its production can start with.
4. a sequence can start with anything that its n-th element can start with provided all elements before that element can be empty. So a sequence of four items, of which the first and last can be empty, can start with only what its first or second item can start with.

This looks good, but rule 3 can lead to infinite recursion. If it does, that means the non-terminal we are looking at is left-recursive, which is a not wanted situation for constructing predictive parsers. But left-recursion doesn't add anything to the set of items that can occur at the start of the matched string -- it just adds the same items again. So again, we can ignore recursive alternatives. Another aspect of this problem is the fact that the parsing technique that we have chosen to use, the top-down parser, works only non-left recursive grammars, with left most derivation. The way our grammar was defined saves us from this inconvenient. More over, there is no item that can be doubled by the same production rule, the enumeration is not allowed into this grammar, not even for the numbers. Otherwise, there would have been difficult to announce the ending of the items sequence. Not even for the numerals this part is not available. The use of that category is strict to only one element, so the forming of numbers – not digits – won't be possible. The only situation that can look like breaking this restriction is the presence of two nouns – sometimes all alone in the sentence. This fact is due to the part that a noun is compulsory and the other one could be generated from the end category. Like in the sentence:

The house is in flames.

4.5.3 Recursive rules

How do we determine if a production rule is generating a recursion? Finding the set of children of a non-terminal is fairly simple: look at its production and extract the terminals and non-terminals mentioned. The descendants of the non-terminal can be computed by taking the transitive closure of the 'children' operation. A recursive non-terminal is one of its own descendants.

4.5.4 Left-recursive

The above procedure for finding starting tokens works just as well for finding starting non-terminals, i.e. non-terminals that can begin a string derived from a particular non-terminal. We can call this set the 'leftmost descendants'. A left-recursive non-terminal is one of its own leftmost descendants.

4.6 An example

Lets take an example and see how the grammar generation works. Let us consider the input stream (as a 'translation' from their meaning of the icons):

The scared people run.

We redefine the S symbol with <sentence>. In the above example the symbol <sentence> is, as one would expect, the goal symbol. Thus, for example, we could start with <sentence> and from this derive the sentential form

the <adjective> <noun> <verb>

In terms of the definitions of the last section we say that <sentence> *indirectly produces* 'the <adjective> <noun> <verb>'. In terms of the definitions of the last section, <sentence> has produced this sentential form in a non-trivial way. If we now follow this by applying production 1, second option (<adjective> \rightarrow scared) we get the form

the scared <noun> <verb>

Application of production 3, second option (<noun> \rightarrow doctor) gets to the form

the scared people <verb>

Finally, after applying production 4, second option (<verb> \rightarrow run) we get the sentence

The scared people run.

The end result of all this is often represented by a tree, as in the next figure, which shows a **phrase structure tree** or **parse tree** for our sentence. In this representation, the order in which the productions were used is not readily apparent, but it should now be clear why we speak of 'terminals' and 'non-terminals' in formal language theory - the leaves of such a tree are all terminals of the grammar; the interior nodes are all labelled by non-terminals.

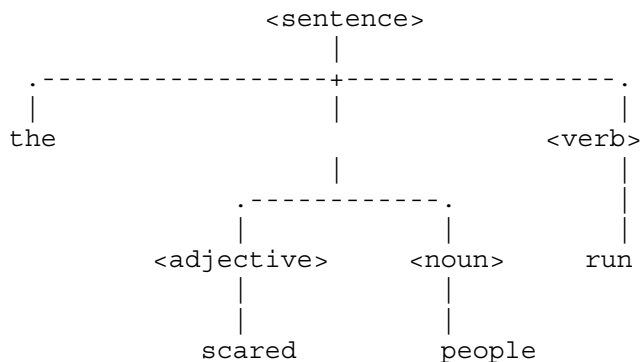


Figure 24. The parse tree for ‘the scared people run’

4.7 Phrase structure and lexical structure

It should not take much time to see that a set of productions for a real programming language grammar will usually be divided into two distinct groups. In such languages we can distinguish between the productions that specify the **phrase structure** - the way in which the words or tokens of the language are combined to form components of programs - and the productions that specify the **lexical structure** or **lexicon** - the way in which individual characters are combined to form such words or tokens. Some tokens are easily specified as simple constant strings standing for them. Others are more generic - lexical tokens such as identifiers, literal constants, and strings are themselves specified by means of productions (or, in many cases, by regular expressions).

As we have already hinted, the recognition of tokens for a real programming language is usually done by a scanner (lexical analyser) that returns these tokens to the parser (syntax analyser) on demand. The productions involving only individual characters on their right sides are thus the productions used by a sub-parser forming part of the lexical analyser, while the others are productions used by the main parser in the syntax analyser.

A moment's thought should reveal that there are many possible derivation paths from the goal or start symbol to the final sentence, depending on the order in which the productions are applied. It is convenient to be able to single out a particular derivation as being *the* derivation. This is generally called the **canonical derivation**, and although the choice is essentially arbitrary, the usual one is that where at each stage in the derivation the left-most non-terminal is the one that is replaced - this is called a **left canonical derivation**. In a similar way we could define a **right canonical derivation**.

Not only is it important to use grammars generatively in this way, it is also important - perhaps more so - to be able to take a given sentence and determine whether it is a valid member of the language - that is, to see whether it could have been obtained from the goal symbol by a suitable choice of derivations. When mere recognition is accompanied by the determination of the underlying tree structure, we speak of **parsing**. A fairly natural way in which we can attempt to solve the problem is to start with the goal symbol and the sentence, and, by reading the sentence from left to right, to try to deduce which series of productions must have been applied.

Let us try this on the sentence
the scared people run

If we start with the goal <sentence> we can derive a wide variety of sentences. Some of these will arise if we choose to continue by using production 1, some if we choose production 2. By reading no further than ‘the’ in the given sentence we can be fairly confident that we should try production 1.

<sentence> → the <adjective> <noun> <verb>.

In a sense we now have a residual input string ‘scared people run’ which somehow must match <adjective> <noun> <verb>. We could now choose to substitute for <verb> or for <adjective> or for the <noun>. Again limiting ourselves to working from left to right, our residual sentential form <adjective> <noun> <verb> must next be transformed.

In a sense we now have to match ‘scared people run’ with a residual sentential form <adjective> <noun> <verb>. We could choose to substitute for any of <adjective>, <noun> or <verb>; if we read the input string from the left we see that by using production 3 and 4 we can reduce the problem of matching a residual input string ‘people run’ to the residual sentential form <noun> <verb>. And so it goes; we need not labour a very simple point here.

The parsing problem is not always as easily solved as we have done. It is easy to see that the algorithms used to parse a sentence to see whether it can be derived from the goal symbol will be very different from algorithms that might be used to generate sentences (almost at random) starting from the start symbol. The methods used for successful parsing depend rather critically on the way in which the productions have been specified; for the moment we shall be content to examine a few sets of productions without worrying too much about how they were developed.

In BNF, a production may define a non-terminal recursively, so that the same non-terminal may occur on both the left and right sides of the \rightarrow sign. But this is not our case. We have imposed the rule of having only one transcription rule for each one of the elements, meaning that we won’t have conjunctions between the nouns for example. This idea was taken into consideration at the beginning, but we have considered the fact that a person, which is in a crisis situation won’t be interested in sending very long and complicate sentences. We only have to assure that he/she will be able to send to the server simple, basic sentences.

4.7.1 Parsing techniques

To examine how the syntactical structure of a sentence can be computed, we must consider two things: the grammar, which is a formal specification of the structures allowable in the language, and the parsing techniques, which is the method of analyzing a sentence to determine its structure according to the grammar.

Top-down methods have the advantage that they will never consider word categories in position where they could not occur in a legal sentence. This advantage stems from the fact that the parser works from a syntactic category and checks that the word fits that category.

To visualize the grammar we will use the notion of transition network consisting of nodes and labeled arcs. Consider the network named NP – each arc is labeled with a word category. Starting at a given node, you can traverse an arc if the current word in the sentence is in the category on the arc. If the arc is followed, the current word is updated to the next word. A phrase is a legal NP if there is a path from the node NP to a pop arc (an arc labeled pop) accounting for every word in the phrase. To get the descriptive power of CGFs, we will use a notation of recursion in the network grammar. A recursive transition network (RTN) is like a simple transition network, except that it allows arc labels that refer to other networks rather than word categories. Any language generated by a CFG can be generated by an RTN, and vice versa.

According to Johnstone [8], when building a new language, the natural design process starts with the language itself, not with the grammar. The situation is particularly difficult in the case of bottom up parsers since, even when a grammar has been accepted

by the parser generator and successfully tested it may break when semantic actions are added.

While designing a grammar, it is convenient to be able to generate a parser to help with the process of checking whether the grammar actually generates the required language. For this process to be effective the parsing technique on which the parser is based should allow easy transfer between parser steps and grammar rules, so that the problems identified using the parser can be related directly to the grammar structure.

Top down, recursive descent based techniques are appropriate in this respect. We will use a top-down approach because it is important for the language design that the parser structure reflects the grammar structure. Although modifications to the standard bottom-up parser generators are available they are not well integrated into the parser's operations and usually require the user to understand internal details of the parser. Top-down parsers allow natural placement of semantic actions, recursive descent parsers allow trivial implementation of both inherited and synthesized attributes in terms of parsers' function parameters. On the other way, a harder method to parse is the one known as shift-reduce or bottom-up parsing, or LR parsing. This technique collects input until it finds that it can reduce an input sequence with a symbol.

4.7.2 Top-down parsing (LL)

The easiest way of parsing something according to a grammar in use today is called LL parsing (or top-down parsing). It works like this: for each production find out which non-terminals the production can start with. (This is called the start set.) Then, when parsing, we just start with the start symbol and compare the start sets of the different productions against the first piece of input to see which of the productions have been used. Of course, this can only be done if no two start sets for one symbol both contain the same terminal. If they do there is no way to determine which production to choose by looking at the first terminal on the input.

LL grammars are often classified by numbers, such as LL(1), LL(0) and so on. The number in the parenthesis tells you the maximum number of terminals you may have to look at a time to choose the right production at any point in the grammar. So for LL(0) we don't have to look at any terminals at all, we can always choose the right production. This is only possible if all symbols have only one production, and if they only have one production the language can only have one string. In other words: LL(0) grammars are not interesting.

The most common (and useful) kind of LL grammar is LL(1) where you can always choose the right production by looking at only the first terminal on the input at any given time. With LL(2) you have to look at two symbols, and so on. There exist grammars that are not LL(k) grammars for any fixed value of k at all, and they are sadly quite common.

4.7.3 Conclusions

For this application we have preferred the LL parser from different reasons. First of it is the simplicity. The workings of an LL parser are much simpler. In case if we have to debug a parser, looking at a recursive-descent parser (a common way to program an

LL parser) is much simpler than the tables of a LALR parser, for example. The second motive was the introduction of new actions, because in an LL parser you can place actions anywhere you want without introducing a conflict. The third criterion is the error repair. LL parsers have much better context information (they are top-down parsers) and therefore can help much more in repairing an error, not to mention reporting errors. Next assuming we write a table-driven LL parser, its tables are nearly half the size. And the last one: the parsing speed – it is very good, but also tool-dependent.

4.8 Competence and performance

Native speakers can differentiate well-formed sentences from ungrammatical sentences. This is the **intuitive grammar** or **linguistic competence**. **Formal grammar** is the explicit **metalinguistic knowledge** of rules of the language. A speaker's linguistic performance may and often does fall short of his/her competence. **Prescriptive grammar** is judgmental whereas **descriptive grammar** simply documents speaker performance.

The value of a defined grammar is given by its general characteristics. The most important one is the completeness of a grammar; this means that one can express anything meaningful using the production rules. In our case, there are of course some imposed limitations, so that we could assure most of it of the semantic approach. But still, with all these constraints, a sentence can start with any icon, and could end with almost any icon (to keep the syntactic correctness). This means that the grammar is a quite open one, things can be explained in different ways, and the way that icons are coming one after another gives a wide variety of ideas, the things to be transmitted.

Another way to take into consideration is usability. Even if we have a rather general grammar, if it is difficult to be used, or if there are inconsistent in it, it is useless. This application was created to be used, not just to have some nice pictures and a lot of theory. And from the given test, all of the sequences of icons that have been introduced were checked both syntactically and semantically in a correct way.

Even if it works with 'unusual' sub-grammar categories, our defined grammar has also the particularity that it operates with special concepts: the images, the icons. The binding between the theoretical approach and the practical one is made with the help of the programming tricks, presented in the next two chapters, the freedom in defining new objects into the running software.

4.8.1 What is a good grammar?

Many grammars may correspond to one programming language, but there could be still defined some criteria for a good grammar:

- Capture the logical structure of the language as the structure carries some semantic information (example: expression grammar);
- Use meaningful names;
- Are easy to read;
- Are unambiguous.

It is difficult to prove the completeness and correctness of a grammar from the theoretical point of view. The only thing that has proven that was the empirical approach

of the application. All the things show that the grammar that we have defined is following these criteria.

Chapter 5

Models

Everyone knows that the interface of an application is extremely important, especially if it is created for human interaction. Even if an application has a very powerful algorithmic background, for example, if the interface is not user friendly, such a product has a lot to lose while being displayed on the market. An interface should be intriguing, smartly created in a more intuitive way than in a logical way, so that the user should be inclined to test it and then, to use it.

In the latest period of time, one of the main goals of the computer science world, was to create artificial systems that could interact more and more with the user, in an order that these system would ‘simulate’ a human behavior.

5.1 Interface

Here is the way that the application looks like:

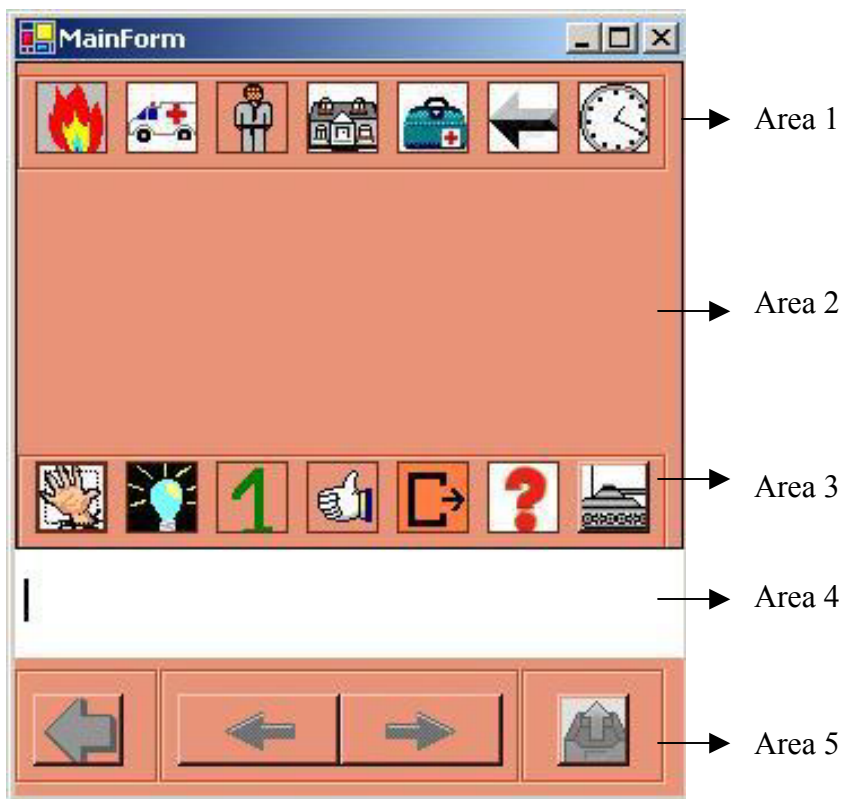


Figure 25. The image of the interface – application

There was of course the restriction of dimension to be applied. The window application has the following dimension: 304 x 352, beginning with these because our

final goal: to make this application work on a PDA, where the space of the screen is not as generous as the one of a desktop computer, or even a laptop.








Area 1

As you can see, the main form is structured in five areas. The icons displayed on the top of the application window form the first one. They represent the index of the category they are defined to be part of it.



Figure 26. The first area of the interface

There are:

- category Crisis represented by the icon 
Flames.ico
- category Cars represented by the icon 
Ambulance.ico
- category People represented by the icon 
Man.ico
- category House represented by the icon 
House.ico
- category First Aid represented by the icon 
Firstaid.ico
- category Directions represented by the icon 
Left.ico
- category Time represented by the icon 
Time.ico

Area 2

The second area, looking from the top to the bottom of the window, is the area where the icons from the chosen category will be displayed. The idea of creating this kind of design was given by the fact that the user should have the possibility to view all the necessary items in the same time, meaning that by displaying the elements from one category, all the other information should be visible, starting with the other index – index icons, ending with the area allocated for the input sequence of icons.

One important criteria for determining this specific order the icons from one category, while being displayed, is that the index – icon should be placed as close as possible to the input OnClick event. The reason is the following: being a ‘definition’ of the group from which it makes part, it is expected that that specific icon should appear in our sight as the first one, like a continuation of its role of ‘idea of the grouping criteria’.

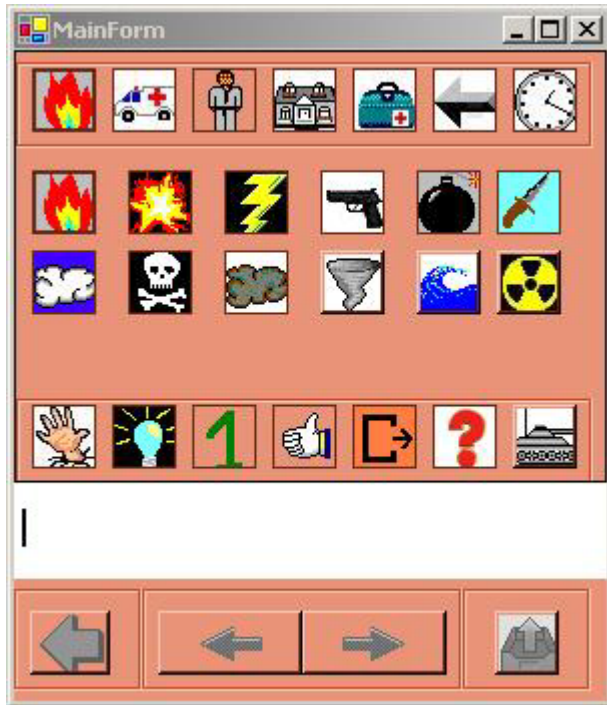


Figure 27. The display of the Crisis elements on the MainForm

Area 3

The third area is reserved to the second sequence of index – icons, having the same expectations like the first level in the application. Lets see here also the grouping in categories:

category Human Actions represented by the icon



Help.ico

category Information represented by the icon



Bulb.ico

category Numbers represented by the icon



One.ico

category Yes/No represented by the icon



Yes.ico

category Special Signs represented by the icon



category Intonation represented by the icon



category military represented by the icon



Area 4

The fourth area is the one in which we will display the sequence of icons that the user will chose. At the first look we have the impression that we are working with images, that we are clicking, choosing, sending icons (correspondent of an image). In fact we are manipulating data from buttons. Everything that seems to be dynamical is just a simple button, and the icon is the background image displayed on the button. As we will see in the implementation part, all the needed information is gathered on the buttons. While selecting a new icon, one could say that we are creating a new button in the display area. This is totally wrong! We have defined from the beginning seven buttons (the maximum number of icons that can be processed at a specific time), which are at start hidden. Whenever the user selects a new icon, the image of that icon is being displayed on a button from area number four, and we make this button visible. The initial area looks like this:

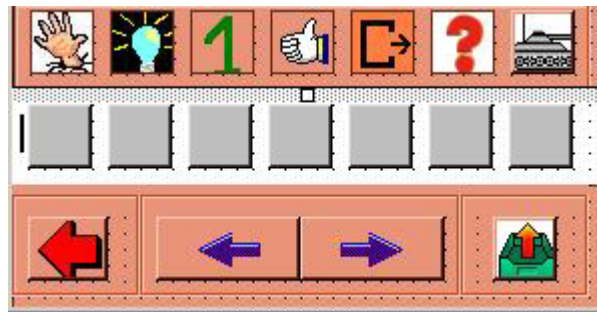


Figure 28. The third, forth and fifth areas of the application

The last area is the one so called ‘the icon editor area’, meaning that the user will have the possibility of navigating through the icons using the purple arrows

to the left with



to the right with



- to delete an icon from the input sequence using the icon



- inserting a new icon in the middle of the sequence by moving the cursor there, and of course – he will generate the processing of the information by pressing the send button, given by the icon



5.2 Testing the application

In order to test this application, we will start by having a good look at the way that the grammar is being defined. It is obvious that not all the sequences of icons are valid, in the same time even if we have the next valid sequence of grammar categories:

<negation> <adjective> <noun> <verb> <adverb>

It doesn't mean that for <noun> all the icons that are taking part in this specific category are right to be chosen. In the same manner, after a particular <noun> not all the verbs are allowed, and so on. This problem of solving and determining the correctness of an input data is debated in the special section dedicated to the syntactic and semantic analysis.

So lets see now the final form of the grammar rules.

S = negation A | number B | adjective C | noun R | verb E | adverb H

A = number B | O

O = verb E | P

P = verb | B

B = adjective C | C

C = noun R | noun

R = sign | D

D = negation F | F

F = verb | G

G = verb E | E

E = adverb | I

I = adverb H | H

H = number J | J

J = adjective K | K

K = noun L | noun

L = adverb

The reason for this final version of the grammar will be proved in the section consecrated especially to the grammar definition. Next, we will create some input data just to see how the application works.

Lets generate the next input sequence of icons:



Figure 29. The sequence of icons representing the sentence: The doctor helps three victims here.

The first icon has the following information: Text = Doctor, Grammar Category = noun. The second icon: Text = help, Grammar Category = verb. The third one: Text = three, Grammar Category = numeral; followed by the icon with Text = victim, Grammar Category = noun; ending with Text = here, Grammar Category = adverb.

From the first production rule, the starting symbol, S, will be rewritten with the rule:

$S \rightarrow \text{noun } R$

We are positioned now on the second element, which is a verb, so we are looking for a production rule that starting with the non-terminal R, will generate the terminal Verb. So the next two rules that will be applied are

$R \rightarrow D$

$D \rightarrow F$

In this point we are checking the next rule more carefully because the actual set key-word is being found in the rule $F \rightarrow \text{verb} \mid G$, but in this case the icon with the grammar category 'verb' is not the last element in the input data, so neither this rule is going to be applied. And we go even further.

$F \rightarrow \text{verb} \mid G$

$G \rightarrow \text{verb } E \mid E$

This last rule is a valid one, because the current element is a verb, and we haven't reached the end of the sentence. So we are now moving forward in the input sequence, 'chasing' a number, starting with the non-terminal E. We are jumping over the next rule, cause they are not fulfilling our requirements.

$E \rightarrow \text{adverb} \mid I$

$I \rightarrow \text{adverb } H \mid H$

The next rule to be applied is:

$H \rightarrow \text{number } J \mid J$

Keeping in mind the non-terminal, J, and the next icon, which is in this case a noun. The next rule is not good for us

$J \rightarrow \text{adjective } K \mid K$

And we have reached the rule

$K = \text{noun } L \mid \text{noun}$

Which is a correct one. We pass next in the input sequence, having to search only for one more element, an adverb, and the last production rule is a correct one.

$L \rightarrow \text{adverb}$

After checking the semantic restrictions, we will generate the output text line, because this is the goal of this application. In this case it will be:

The doctor helps three victims here.

5.3 Programmatic wrapping of the problem

To combine the theory with the practical part, to make things as easy as possible, we have tried to use all the facilities that the software platform puts us at our disposal. The software that we have used is Visual Studio Professional 2003, Visual C#.

As all the controls on the form are buttons, because we needed the OnClick event, we had the idea of extending the basic Button class and add to it some new features. We needed some information about each button in every moment of the application. The most interesting part with this approach is that we do not manipulate directly the buttons, but we are more interested in setting and getting the information contained on a button.

Let us see what are the new elements added to a button, and most of it: why? If we were dealing with grammar syntax, and sub-categories of this grammatical point of view, it would have been useful to know each button which category represents. So, this is why we have optioned for adding a new field to each button that would get this specific information. The field is called *GrammarCat*. That means that each icon is taking part of one, and only one, category: noun, verb, number, adjective, sign, negation, and adverb.

If the first presented field is the one who will be interrogated in the syntactical analysis, for the semantic one we have added a new field, called *Subcategory*. It will memorize the connection between the subcategories, giving us a basic meaning of the sentence.

The other fields are *Category* – for keeping ‘in mind’ the category from which it comes (one of the fourteen ones); *Textt* – is the text that it will be taken in the translation from icon to text; *Index* – an unique index (number) that it can identify an icon.

To have this information modularized, we have created a new class to make them atomically, like another field in the list of attributes for a button.

The IconicInformation class will have only the declaration of the fields, which all of them should be private,

```
private string imagName;  
private int idCat;  
private int indexButton;  
private int subCat;  
private string txt;  
private string grCat;
```

and setter and getter of each one of these elements. We give next the example of the setter – getter for *Category*.

```
public int Category  
{  
    get  
    {  
        return idCat;  
    }  
    set
```

```

    {
        idCat = value;
    }
}

```

In the same way we have defined the others elements of the application.

Another class, called CImageButton, makes the connection between a button and the IconicInformation class. From now on, the buttons that we will have in our project, on the forms, they won't have the System.Windows.Forms.Button type, but they will be instances of CImageButton class. This class is a very special one, using the most used design pattern, singleton, and it will be presented in the next chapter. The connection between the button and the special information is made in a very simple way: the element *iconInf* is considered to be a new self-standing field on the button, behaving in the same manner like any other property. For that, we have to define a setter and a getter, in the unique style of C# development toolkit.

```

public IconicInformation iconInf
{
    get
    {
        return iconInformation;
    }
    set
    {
        iconInformation = value;
    }
}

```

This class extends the Button class, and adds one more element, which is iconInf. This element is an instance of the IconicInformation class and it will bring all those information contained there on the new button.

5.3 The meaning of the sentence - the connection between the user interface and the grammar

From the interface, we have buttons, from the buttons, we go further to their specific information; processing that information, and we get the meaning. But the next question is: what are exactly these subcategories, how are they defined, and why those rules?

Let us take one by one these questions, and give them some answers. The basic idea of this approach, was to define or group the icons in some larger extension in divisions that would prove their behavior in the application. We have chosen first the initial categories, the fourteen of them, and spread the icons in such a manner that they will act alike. What does it mean act alike? Well, it can refer to the fact that they should have, first of all, the same grammar category, plus, they require the same articles, and connection words, and so on. The way they will be used is the same: the same context, only the context differs slightly. Let us take the categories one by one and explain why I did the specific divisions.

Category Crisis Elements

First of all we should observe that in this category we have eleven nouns and one adjective, so, from the start, the adjective, *toxic* defines a self-standing subcategory. *Flames*, the most used word, was at the beginning coupled with *electricity* and *explosion*, as being the major elements in a critical moment, in the way that they are not extremely concrete, but they are more like concepts than things. We had placed the word *flames* in another subcategory, because it requires some specific connection words, for example with the verbs. The *air* and *smoke* are in another category because they are the elements of the etheric world; on the other hand, the words *bomb*, *gun* and *knife*, are the weapons. In the last subcategory are placed the words *flood*, *tornado* and *nuclear threat*, that are less likely to be used, and they have the same demand in the sentence construction. Here is the table with the items from this category:

Table 1. The structure of the subcategories in the category ‘Crisis elements’

Flames	Electricity Explosion	Air Smoke	Bomb Gun Knife	Toxic	Flood Tornado Nuclear threat
--------	--------------------------	--------------	----------------------	-------	---------------------------------------

Category Cars

The elements from this category, they are all nouns, and they have been divided in three subcategories. We can make a difference between the vehicles and the ways they are used on. So, *road*, *railway*, *tunnel* and *bridge*, are grouped together, and the other elements are spread also in two: *bike*, and the rest, because the bike is a very light vehicle and can't be used in all the written sentences. So, in this case the table looks like this:

Table 2. The structure of the subcategories in the category ‘Cars’

Ambulance Car Police car Fire truck Train	Bike	Road Railway Tunnel Bridge
---	------	-------------------------------------

Category Directions

In this situation is elements are all adverbs, adverbs to express distance and place, and they have been divided after the directions: *left*, *right*, *up* and *down*, and after distances: *here* and *there*. The table looks like this:

Table 3. The structure of the subcategories in the category ‘Directions’

Left Right	Here There
---------------	---------------

Up	
Down	

Category Buildings

Also in this case all the elements are nouns, and they have been divided in Buildings: *house*, *building* and *hospital*, elements of the room: *door* and *window*, and the *stairs* and the *elevator*, even if the both represent ways of transporting, they do not have the same behavior in according to the article matrix. The table has the following form:

Table 4. The structure of the subcategories in the category ‘Buildings’

House Building Hospital	Door Window	Stairs	Elevator
-------------------------------	----------------	--------	----------

Category People

In this category we have only nouns, grouped in four subcategories with the next principles. The word *people* should define a self-standing subcategory because it represents a collective noun (the noun that imposes the plural verbal form), the word *dead person* is a special case – the impossibility to act. It wouldn’t make sense to add a moving verb for example after it. Then we have the humans: *man*, *woman*, *handicap person* and *victim*, and the intervention teams: *soldier*, *cop*, *fireman*, *doctor* and *bomb squad*. The form of the table is the following:

Table 5. The structure of the subcategories in the category ‘People’

Man Woman Handicap people Victim	Soldier Cop Fireman Doctor Bomb squad	Dead person	People
--	---	-------------	--------

Category First aid

The elements are in this case nouns too, and they have been placed in different subcategories according to their use: the treatment things – *bandage*, *medicine*, *injection* and *first aid*, elements that sustain life – *food* and *water*, lighting things – *candle*, *matches* and *lantern*, *extinguisher* and *thermo* are separated because they ask for specific connection elements to the other words. This is the table for this category:

Table 6. The structure of the subcategories in the category ‘First aid’

Bandage Medicine Injection First Aid	Candle Matches Lantern	Food Water	Extinguisher	Thermo
---	------------------------------	---------------	--------------	--------

Category Human Actions

This category contains verbs and adjectives, as the state of describing the condition of a person. So, it was natural to group the adjectives together in one subcategory, and the adjectives are: *blind*, *deaf* and *scared*. The other words, the verbs, were very difficult to establish the same way of connecting to the other words. Only two verbs, which are less likely to be used: *note* and *read*, could have been placed in the same subgroup; all the rest of the verbs define a self-standing subcategory. Here is the structure for this category:

Table 7. The structure of the subcategories in the category ‘Human Actions’

Hear	See	Speak	Note Read	Run	Call	Hit	Want	Help	Search	Blind Deaf Scared
------	-----	-------	--------------	-----	------	-----	------	------	--------	-------------------------

Category Information

In this category we can find only nouns, nouns which are divided in two groups: one of them is the word *information*, which is the general description of all the items, and the rest of the words: *computer*, *mobile phone*, *telephone*, *radio* and *tv*. This category has the following construction:

Table 8. The structure of the subcategories in the category ‘Information’

Information	Computer Mobile phone Telephone Radio Tv
-------------	--

Category Special Signs

At the first look it may appear that this category contains everything that didn’t fit in the other categories. This fact is false, due to the fact that these elements, being adverbs, noun, verb or a negation, have the goal of changing totally the meaning of the sentence, like it is with the negation, or it gives the final form of the idea. As we had noticed already, we have four grammatical categories here, and it is obviously that we did the spread in subcategories after this criteria. The table looks like this:

Table 9. The structure of the subcategories in the category ‘Special Signs’

Blocked Unblocked Disorder	Exit	No	Move
----------------------------------	------	----	------

Category Military

The following category is the one of military equipments, which from the grammatical point of view, they are all noun. They were still divided in three parts, according to the medium in which they appear: land, water and sky. So the table has the construction:

Table 10. The structure of the subcategories in the category ‘Military’

Airplane Fighter Helicopter	Tank	Submarine Boat Cruiser
-----------------------------------	------	------------------------------

Category Numbers

In this specific category, we have, of course, only numerals, or as we had defined them: numbers. They are divided in two: number *one* in one subcategory, and the rest in the other part, due to the fact that *one* asks for the singular form of the noun, and the rest words ask for the plural form. So the table looks like this:

Table 11. The structure of the subcategories in the category ‘Numbers’

One	Two Three Four Five Six Seven Eight Nine More
-----	---

Category Intonation

The elements from this category, just two of them, they are some special grammatical element – signs. The behavior between the interrogative and exclamation item are totally different. Only from the fact that we have defined a template construction of building the text for the question sign, gives us a lot of reasons for dividing the two elements in two distinctive subcategories. And the Table has the following form:

Table 12. The structure of the subcategories in the category ‘Intonation’

?	!
---	---

Category Yes/No

In this category, we have only adverbs of time, and they are not usable yet in our application. Why? The presence of this type of adverbs require the changing of the grammar definition, giving the possibility of creating more and more complex sentences, which is out of the range of the goal of this application, until now at least.

Category Time

Even if for this category there is the same situation as the previous one: Yes/No symbols, meaning they are not usable in the application, they were divided in two: *time*, and the two antonyms: *day* and *night*.

5.3.1 Conclusions

We have to admit that all this classifications are pure intuitive, and they were done in a way to help us with the solving of the problem, only in the semantic approach of the project.

Chapter 6 Implementation

The first goal of any approach in solving a problem is to make it work, but the true value of an application, it may be considered the safety of the code and the elegance

of writing it. Usually, when a part of code is being written, there is slight chance that it will remain like that, talking of course of the valuable projects, the large and professional ones. The code says a lot about the application and the one that has build it, and it is also a way of communication between programmers, besides providing the documentation for it. Let us start now the presentation of the small programming tricks.

6.1 Design patterns

Objects, Inheritance, Encapsulation, Abstractions - we've heard a lot about these concepts and orthodoxy use these in our daily work. Object-oriented software is changing the way development is being done and has come into vogue. It is now widely recognized that Object-Oriented software is essential to complex, scaleable software development, and the key to distributed computing. We many-a-times observe that while solving various types of business problems, we come across similar issues that can be solved by using same design methods. Having done it in the past makes it easier for us to solve it the second time. These recurring solutions to Software problems are termed as *Design Patterns*.

But what are exactly the patterns? A definition of this concept could be the following: *A pattern is a named, reusable solution to a recurrent problem in a particular context.*

The goal of patterns within the software community is to create a body of literature to help software architects resolve recurring problems encountered throughout the software development. A Pattern is evolved when we realize that we are trying to find a solution to a problem that has been already dealt with; that is the time when one can make the problem as well as the solution *global* so that it can be applied verbatim to the analogous situations in future. Each patterns has some elements attached to it viz. name, purpose, problem that it solves, constraints and forces that have to be considered etc. The primary focus is not so much on technology as it is on creating a culture to document and support sound engineering architecture and design. In fact, most patterns are discovered rather than written. Instead of reinventing a solution all you have to do is learn the pattern and know when you apply it.

6.1.1 Singleton

There are many patterns discussed, but one of the most widely used and the simplest is the *Singleton pattern*, and we have used it in our application. Its intent is to ensure that a class has only one instance, and to provide a global point of access to it. The situations where this kind of pattern can be applied are many; there can be many ways to implement this pattern. The first and foremost that comes into the mind is to have a global variable that keeps the count of the objects created. But in true Object Oriented paradigm, global variables are not allowed (example Java and C#). We'll take a look on one such way to get the feel of it.

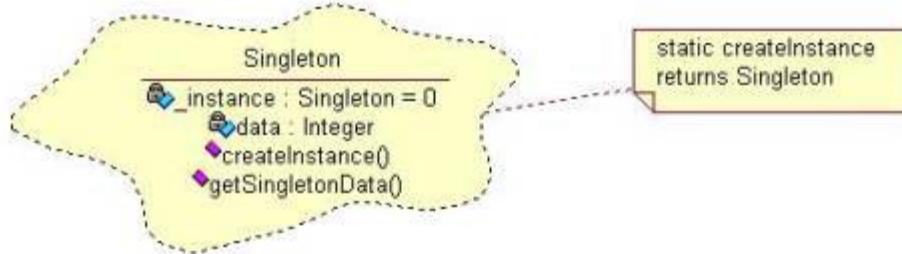


Figure 30. Singleton Class

We can create a static method to create an instance of a class. We make the constructor private so that if someone tried to use the conventional method of creating an instance, that way just won't be possible. This way, the instance can only be created using the static method.

We have used this design pattern twice: once in the Configurator class, the class that provides the input data for the application, that fills the matrixes and the lists, so that the information could be processed; and the second one is the class called CSender, that takes part of the client area in the communication on the sockets, but all this elements will be developed in the next sections.

Let us take like example the definition and implementation of the Configurator class. First we have defined a variable, an instance of the class. It should, of course, be private so that the encapsulation will be respected, and we have also added the fact that this instance is a static one, because we should fill in the data only once, and this data should be global for all the other classes that will use them.

```
//singleton instance
private static Configurator instance = new Configurator() ;
```

The particular element comes with the constructor, which we know that usually it is declared to be *public*.

```
//private constructor, as we have a singleton instance
private Configurator()
{
}
```

Even the name of this specific design pattern suggests the action of the class its self. It gives the idea of being unique – single. By making the constructor *private*, even if it does not do anything in particular (but it might do, eventually), we should have at least a method in which we can get the instance of the class, so that is why we have build a new method to provide this information.

```
//returns the singleton instance
public static Configurator GetInstance()
{
    return instance ;
}
```

```
}
```

By reusing already established designs, one gets the benefit of learning from the experience of others. We do not have to reinvent solutions for commonly recurring problems. If we have the patterns ready on specific domains then we can concentrate on more macro-level problems while creating a design rather than bogging into small-unadorned technicalities. When working in a team, teamwork requires a common base of vocabulary and a common viewpoint of the problem. Design patterns provide a common point of reference during the analysis and design phase of a project.

In our context, sometimes it happens that we are working with great endeavor to accomplish a certain task, that some or the other had already achieved. Thus, we must try to standardize, systematize, and institutionalize the efforts applied on such problems and have some global Design Patterns, those of which can be used by each one of us with much ease, reducing the design and the development time.

6.2 Creating the buttons

We have seen in section 5.3 the definition of the information that we need to build our approach on the problem. But let us see now in fact the way these elements are mixing with the application. We have defined for this a new class, called `CImageButton`, a class that will extend the basic C# class and type – the `Button`. So, obviously, the constructor of our class should call the constructor of the base class.

```
public CImageButton() : base()  
{  
    initializeComponent();  
}
```

In this constructor we call the method `initializeComponent()`, in which we assign all the information defined in the `IconicInformation` to the new `Button`. So, in this moment, our buttons will contain besides the standard information from a normal button, the ones from our special class, encapsulated in the name *iconInf*.

```
public IconicInformation iconInformation;  
private void initializeComponent()  
{  
    iconInformation = new IconicInformation();  
}
```

6.3 Configuration of the data

We have mentioned above that this class is under the singleton design pattern style, so we won't present once again the way that it is defined or used. Why do we need this kind of class, may be a very good question. Before answering to this question, first of all let us see what are the sources of this application. We have a file in which we have the grammar production rules, and this information is the base of our grammar checker, the syntactical part. This file is also listed in appendix????? The next helping element is the file with the network structure of the subcategories, and this will generate in part the semantic part of the problem. You can check the content of this file in appendix ????

Another file we use is the one in which we are memorising the nouns that have a plural verbal form, so that it will be easy just to place the appropriate verbal form, according with the ATN. The last file we use was defined just for the fact that we want to give more meaning to the output sentence, making it more and more meaningful, so that it won't look like we are sending messages in a language used by a five year old kid.

6.3.1 Loading the rules

For this part, which is the first step in the syntactical analysis, we have defined another class that is the template of the construction of the grammar rules, being from now on a new type in our application. The form of the CRule, the class we are referring to, is very simple – it just creates the form of a grammar rule:

Leftside = Terminal Nonterminal.

Or

Leftside = Terminal.

```
public CRule(string leftNT, string rightNT, string terminal)
{
    this.leftNT = leftNT;
    this.rightNT = rightNT;
    this.terminal = terminal;
}
```

The constructor of this class is a trivial one, it is just assigning the elements of the class, elements for which we have also defined a getter, so that we could access them in any moment.

In order not to make the file very long, we have preferred a more condensed form, equivalent with the standard one, in which the first rule, for example, would have had the next form:

S = negation A
S = number B
S = adjective C
S = noun R
S = verb E
S = adverb H

Figure 40. The standard form of the first production rule

But we had preferred a more compact one:

S = negation A | number B | adjective C | noun R | verb E | adverb H

After splitting the file into lines, we will start with the second element of the array of strings, because we will 'jump' over the elements 'S ='. There are different elements that need to take into consideration, being different types of rules that they might be applied. All the conditions are running until we still have elements to check, plus the fact that we test that we have reached the end of the line, or the situation in which we have in the next element the alternation sign, which gives us one more production rule.

```

while( currentPos < tokens.Length )
{
    if( currentPos + 1 == tokens.Length ||
        String.Compare( tokens[ currentPos+1 ], '|' ) == 0 )
    {

```

If we look it more closely, we can observe that the terminals are noted with a low capital letter, and the non-terminals, are starting with a high capital letter. So, for us is very easy to determine if the current element that is to be tested is a terminal or not, just to chose the way to fill the next rule. So, if it is indeed a terminal, it means that we have the next case:

Leftside = terminal

Like for example: $P = verb$, and in this case we do not have a non-terminal in the rule, meaning this a finishing production rule – we can not go any further with our check as we have no other option to follow.

If the current element is a non-terminal, and the next element is the alternation, that means that we have no terminal in the production rule and its form is like this:

Leftside = non-terminal

Like for example, $A = O$, and then we will ‘jump’ over two elements so that we will go on with our checker.

```

if( Char.IsLower( tokens[ currentPos ], 0 ) )
{
    rule = new CRule( tokens[0], null, tokens[ currentPos ] ) ;
}
else
{
    rule = new CRule( tokens[0], tokens[ currentPos ], null ) ;
}
currentPos += 2 ;

```

If none of these cases are fulfilled, that means that we have a full production rule with the form:

Leftside = terminal Non-terminal

Like in the case of: $B = adjective C$, and we will ‘jump’ over three elements.

```

else if( currentPos+ 2 == tokens.Length ||
        String.Compare( tokens[ currentPos+2 ], '|' ) == 0 )
{
    rule = new CRule( tokens[0], tokens[ currentPos+1 ],
        tokens[ currentPos ] ) ;
    currentPos += 3 ;
}

```

For the fact that these information might be required multiple times, so that we have them created and loaded just once, and use them as many times as we want, we have created like a setter and a getter for this section. What we have seen until now is the setter part, and this one will only be executed only once. In case we want to use the elements that have been added to the rules of the grammar, we have defined the next method:

```

public ArrayList GetRules()
{
    if(rules == null) //the rules have not been loaded yet
    {
        LoadRules() ;
    }
    return rules ;
}

```

In the MainForm we call the creation of the rules, so that we have already these elements. But, if we look closer to the getter method, we test once again the loading of that part. This is what is called, a safe code. In each moment a programmer should look very close to all the possibilities that they might appear. In our case, we have doubled check the loading of the information – just in case some other developer of the project would change any part of the code that may generate the lose of consistency of the code.

6.3.2 The semantic elements

This step in defining the elements that we need is trivial, from the fact that from the file that includes the network between the subcategories, after splitting it into lines, we assign the *semanticMatrix* with 1 if there is any connection between two subcategories, and the default value, 0, if there is no such connection. So, if we have in the file the line: *cat1 cat2*, that means that $semanticMatrix[cat1, cat2] = 1$, else $semanticMatrix[cat1, cat2] = 0$. This matrix is not a symmetrical one, meaning that if we have $semanticMatrix[cat1, cat2] = 1$, that doesn't necessarily require that $semanticMatrix[cat2, cat1] = 1$ too. But this case can happen easily.

This matrix, as we have mentioned above, gives us the semantic checker of the input sequence of icons. As we have done with the syntactical approach too, we also check twice the creation of this matrix, like making a setter – which is called in the MainForm, and a getter – that is at the disposal of the other classes.

```

public int[,] GetSemanticMatrix()
{
    if(semanticMatrix == null)
    {
        LoadSemanticMatrix() ;
    }
    return semanticMatrix ;
}

```

6.3.3 Articles between the words

This part is a small extension of the way we have deal with the semantic one. It is also based on the subcategories, adding afterwards, on the same line, the text that should be placed between the two elements from that specific subcategory. So, one line from this file has the following form: *cat1 cat2 text*. The fact that $semanticMatrix[cat1, cat2] = 1$, doesn't imply the fact that the *article Matrix[cat1, cat2]* should contain something apart from the default assignment. The two matrixes are related in the way that if $article Matrix[cat1, cat2] != ''$, which is the empty string, there should definitely be that

$semanticMatrix[cat1, cat2] = 1$. In a different way said: the semanticMatrix includes the *article Matrix*. Also, this last matrix is not a symmetrical one.

6.3.4 Plural nouns

The last part of the configuration step is the one of creating a list out of file that memorize the nouns that have a plural form. This thing is done automatically by splitting the file into words.

6.4 Syntactic analysis

The entire syntactic checker is made in the class CGrammar. This class contains only two methods: one that gives an approximate production rule, *GetRule*, and the next one, *CheckPhrase*, determines if the rule is a correct one.

The first thing we should take into consideration is that the non-terminal that gives the transcription is the same with the non-terminal that we are looking for, element that is memorized in the variable *leftNT*. If we have found such a rule, we check afterwards if the terminal is the same too. It may sound obvious that we will find such a rule according to the imposed context, but we have to remember that a rule has the form

Leftside = non-terminal terminal

For this kind of rules, there is the possibility that either the non-terminal, or the terminal should be *null* – the case in which they are both in the same time null is permitted only by grammars that allow the creation of the empty string, and this is not our case. We have defined in our file the rules in a more ‘relaxed’ way, but in fact, the number of the possible production rules is given by the possibilities that one symbol can produce multiple rules, multiple elements. So, these conditions are not trivial.

```
if ( String.Compare(rule.LeftNT(), leftNT) == 0 )
{
    if (String.Compare (rule.Terminal(), terminal) == 0)
    {
        if(isLastWord && rule.RightNT() == null ||
            !isLastWord && rule.RightNT() != null)
        {
            return rule;
        }
    }
}
```

In case we do not find a rule that transcripts something, that makes us go further with the checker along the sequence of icons, there can be rules that just make a simple movement through the rules of the grammar, having the form:

Leftside = non-terminal

But with the condition that we have passes from our *LeftNT* element, to another one. In this situation we return an auxiliary production rule, also called temporary one.

```
if ( rule.Terminal() == null )
{
    bestMatch = rule;
}
```

The correctness of a sequence is also given by some other factors: the number of elements should conduct to a terminal, there should be another production rule for a non-terminal in case the number of elements are not over, and so on. So, we start the checker from the fact that we have a production rule. In case we do, that means that one element from the input sequence was analysed, the active non-terminal that will create the transcription rule will have now the value of the *RightNT*, the non-terminal that was on the right side of the transcription.

```
while (currentWord < words.Length && (rule = GetRule(leftNT,
words[currentWord], currentWord+1==words.Length)) != null)
{
    if (rule.Terminal() != null)
    {
        currentWord++; // we have a rule of the type A -> bB
    }
    leftNT = rule.RightNT();
    if( leftNT == null)
    {
        break ; //parsing has finished before the words are over
    }
}
```

We should also considerate the fact that the grammar rules might seem ‘insufficient’, meaning that we are still looking for production rules, but the lines from our file are over.

```
if(currentWord != words.Length)
{
    return false; // we have no more rules to check
}
return rule.RightNT() == null ; //we check if the last rule has the
form A -> a
```

6.5 Semantic analysis

The semantic checker is much more simpler than the syntactical one, and we have used the ideas of Schank [12], that proposed the network between certain categories of elements, in order to generate in a way, not a complete one, the meaning of a sentence. This method can be applied only for the cases in which the vocabulary is not very general, having a special topic. Also, the network between the categories should be as robust as possible. For example, if we have three categories: *cat1*, *cat2* and *cat3*, and there is a connection between *cat1* and *cat2*, *cat2* and *cat3*, even *cat1* and *cat3*, and according to our network – everything is correct, but it may be the case that the whole sequence *cat1 cat2 cat3* has no meaning what so ever. So this is just a fragile way of imposing some coherence in the sentence. The reason we went for this approach, was that we have a small vocabulary, and the sequence of icons are imposing some logical constrains, just enough to make it understandable. The only condition that we have to check is that between two icons, there is the network assigned, meaning the value in the *semanticMatrix* should be 1.


```

for(int i = 0; i < subcat.Length-1; i++)
{
    if(matrix[subcat[i],subcat[i+1]] != 1)
    {
        valueToReturn = false;
    }
    if( subcat[i] == 10 && ( i!= 0 || i != subcat.Length - 1))
    {
        valueToReturn = false;
    }
}
return valueToReturn;

```

The only thing that might cause some problems, is the use of the adverb that express space: *here* and *there*. They are one of the most used icons, and this fact was generating the possibility of introducing sentences with no meaning. The constrain that had to be added was that this particular two icons should be used either in the beginning of the sentence, or in the very end. Even the grammar allows us to add first an adverb and then probably a number, or a noun, this is not the case with the icons placed in the subcategory numbered with 10.

6.6 Giving the meaning to the sentence

We have mentioned many times already the fact that our grammar is a very flexible one, that the user can start the sentence with whatever element he/she wants, and ending it almost in the same way. The fact that there are some concepts in the real life that have absolutely no graphical representation, made us use some well-defined structures, also called *templates*.

6.6.1 Templates

6.6.1.1 'There is – There are'

The templates are the constructions that are activated immediately, if there are some conditions to be fulfilled. In our case we have defined three templates. The first of them is the one with the possible following constructions:

<negation> <noun>
 <negation> <number> <noun>
 <negation> <number> <adjective> <noun>
 <negation> <adjective> <noun>

There is the possibility of adding the adverbs *here* and *there* at the end. This particular template has the default beginning: 'There is', 'There are'. For example, for the next sequence of icons, has the meaning 'There are two victims here':



Figure 41. The sequence of icons with the meaning: ‘There are two victims here.’

The conditions that have to be fulfilled are the following:

- Just one noun;
- No verbs;
- Assertive sentence;
- There shouldn't be already an adverb on the first position.

```
if(nounCount == 1 && verbCount == 0 && words[0] != 'adverb'
   && text[text.Length - 1] != '?')
{
    output += 'There ';
    bool aux = false;
    for(int i = 0; i < nounFileElements.Length; i++)
    {
        if(String.Compare(nounFileElements[i], text[text.Length-2])
           == 0)
        {
            output += 'are ';
            aux = true;
        }
    }
}
```

The plural form of the template: There is; There are, is given by the presence of the numerals that are greater than one, or by the nouns that have a plural form.

```
for(int i = 0; i < words.Length; i++)
{
    if( words[i] == 'number' && String.Compare(text[i], 'one') != 0)
    {
        output += ' are ';
        aux = true;
    }
}
if(aux == false)
{
    output += ' is ';
}
```

6.6.1.2 Question template

The second template that we have defined is for the Wh-questions, from which we are creating only the ‘Where’ type question. These are the accepted constructions, where sign is the symbol ‘?’:

<negation> <noun> <sign>
<number> <noun> <sign>
<adjective> <noun> <sign>

<number> <adjective> <noun> <sign>

The main condition for this template is the presence of the question symbol on the last position of the sequence of icons, having in front of it a noun. The case where this element is placed on the first position, or somewhere in the middle, is not a valid one. The default first word is 'Where'. Next, we should establish if the noun is in the plural form, in the same manner as we did in the previous example.

```
if(String.Compare(text[text.Length-1], '?') == 0)
{
    output += 'Where';
    bool aux = false;
    for(int i = 0; i < nounFileElements.Length; i++)
    {
        if(String.Compare(nounFileElements[i], text[text.Length-2])
== 0 )
        {
            output += ' are the ';
            aux = true;
        }
    }
    for(int i = 0; i < words.Length; i++)
    {
        if( String.Compare( words[i], 'number') == 0
        && String.Compare(text[i], 'one') != 0)
        {
            output += ' are the ';
            aux = true;
        }
    }
    if(aux == false)
    {
        output += ' is the ';
    }
}
```

Let us also give an example of the way this template works. Here is the sequence of icons:

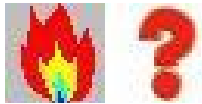


Figure 42. The sequence of icons with the meaning: 'Where are the flames?'

6.6.1.3 First person template

The last template that we have creates is the one of the first person singular. This is the case when we do not have a noun in the beginning of the sentence, having to replace a pseudo-elliptical situation. Why we do that? For the simple reason that the concept of 'I' is very cryptical and abstract to be expressed by a simple icon, without generating confusions. In the same time, we have to define an application for the user, so

that he/she could express and transmit what he/she sees. And the constructions of the type: 'I see', 'I hear', etc. make the communication with the server site in a more human way.

Like in the other cases, there are some situations that are generating this template, and we first present it in a schematic form:

<verb>
<negation> <verb>

The conditions for applying this template are: no noun in the beginning, on the first position there should be either a verb, or a negation followed by a verb. In the negative form of the verb, the default words would be 'I can', and in the other cases, just 'I'.

```
if(words[0].CompareTo('verb') == 0 )
{
    output += 'I ';
}
if(words[0].CompareTo('negation') == 0 && words[1].CompareTo('verb') ==
0)
{
    output += 'I can ';
}
```

6.7 Client – server side

Network programming in windows is possible with sockets. A socket is like a handle to a file. Socket programming resembles the file IO, as does the Serial Communication. You can use sockets programming to have two applications communicate with each other. The applications are typically on the different computers but they can be on same computer. For the two applications to talk to each other either on the same or different computers using sockets one application is generally a server that keeps listening to the incoming requests and the other application acts as a client and makes the connection to the server application. The server application can either accept or reject the connection. If the server accepts the connection, a dialog can begin with between the client and the server. Once the client is done with whatever it needs to do it can close the connection with the server. Connections are expensive in the sense that servers allow finite connections to occur. During the time client has an active connection it can send the data to the server and/or receive the data.

The complexity begins here. When either side (client or server) sends data the other side is supposed to read the data. But how will the other side know when data has arrived. There are two options - either the application needs to poll for the data at regular intervals or there needs to be some sort of mechanism that would enable application to get notifications and application can read the data at that time. Well, after all Windows is an event driven system and the notification system seems an obvious and best choice and it in fact is.

As I said the two applications that need to communicate with each other need to make a connection first. In order for the two application to make connections the two applications need to identify each other (or each other's computer). Computers on

network have a unique identifier called I.P. address which is represented in dot-notation like 10.20.120.127 etc. Lets see how all this works in .NET.

Socket programming in .NET is made possible by the Socket class present inside the System.Net.Sockets namespace. This Socket class has several method and properties and a constructor. The first step is to create an object of this class. Since there is only one constructor we have no choice but to use it.

Here is how to create the socket:

```
m_socListener = new
Socket (AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp) ;
```

The first parameter is the address family, which we will use, in this case, **interNetwork** (which is IP version 4) - other options include Banyan NetBios, AppleTalk etc. (**AddressFamily** is an enum defined in Sockets namespace). Next we need to specify socket type: and we would use reliable two way connection-based sockets (stream) instead of un-reliable Connectionless sockets (datagrams). So, we obviously specify stream as the socket type and finally we are using TCP/IP so we would specify protocol type as **Tcp**.

Once we have created a Socket we need to make a connection to the server since we are using connection-based communication. To connect to the remote computer we need to know the IP Address and port at which to connect. In .NET there is a class under **System.Net** namespace called **IPEndPoint**, which represents a network computer as an IP address and a port number. The **IPEndPoint** has two constructors - one that takes an IP Address and Port number and one that take long and port number. Since we have computer IP address we would use the former

```
public IPEndPoint(System.Net.IPAddress address, int port) ;
```

As you can see the first parameter takes an IPAddress object. If you examine the **IPAddress** class you will see that it has a static method called **Parse** that returns **IPAddress** given a string (of dot notation) and second parameter will be the port number. Once we have endpoint ready we can use **Connect** method of this Socket class to connect to the end point (remote server computer). Here is the code:

```
System.Net.IPAddress ipAdd =
System.Net.IPAddress.Parse('130.161.157.235') ;
System.Net.IPEndPoint remoteEP = new IPEndPoint (iAdd,8221) ;
m_socClient.Connect (remoteEP) ;
```

These three lines of code will make a connection to the remote host running on computer with IP 130.161.157.235, the IPAddress from the computer that I have used at TU Delft university, and listening at port 8221. If the Server is running and started (listening), the connection will succeed. If however the server is not running an exception called SocketException will be thrown. If you catch the exception and check the Message property of the exception in this case you see following text:

```
'No connection could be made because the target machine actively refused it.'
```

Similarly if you already have made a connection and the server somehow dies, you will get following exception if you try to send data.

```
'An existing connection was forcibly closed by the remote host'
```

Assuming that the connection is made, you can send data to other side using the Send method of the Socket class. Send method has several overloads. All of them take a byte array. For example if you want to send 'Hello There' to host you can use following call:

```
try
{
    String szData = 'Hello There';
    byte[] byData = System.Text.Encoding.ASCII.GetBytes(szData);
    m_socClient.Send(byData);
}
catch (SocketException se)
{
    MessageBox.Show ( se.Message );
}
```

Note that the **Send** method is blocking. This means the call will block (wait) until the data has been sent or an exception has been thrown. Similar to Send there is a **Receive** method on the Socket class, but this part will be used only in the next versions of the application, when there will be a communication between the client and the server. Right now, there is only a basic communication on sockets, the client sends the translation of the input sequence of icons, and the server does nothing else but receiving this information.

6.8 Hints

Until now we didn't discuss about the situation when the sequence of icons is not a valid one. What should be done in this situation? One of the ideas was that we should notify the user that the input is not correct, syntactically or semantically, by lightening a warning bulb on the MainForm, like the Windows Warning System works – placing an icon on the task bar, on the very right position. In addition to this thing, there was the possibility of giving to the user some hints, like in Microsoft Word for example. This thing could have been a very difficult task, having three possibilities to do this: deleting the less feasible icon, switching the position of two icons after some criteria, and the last of them is inserting a new icon on a specific position.

All these elements are very useful, and important algorithms could be constructed on them. But our approach is the following: why to cure, than to prevent. We refer now not at the algorithms for giving to the user of some hints, but at the fact that we will display every time for him only the icons that could create a valid sequence of icons.

In this first phase of the project, we will give only the semantic hints, meaning that after choosing an icon, on the moment when the user will browse through the

categories of icons, only the ones which are related semantically with that first icon that has been introduced. And so on. This is a very practical and quite simple method of taking care of the correctness of the input data, but we are also helping a lot the user to choose more easily the icons. In the other approach, with giving the hints in the end, with some very complicated background, the user had to face every time all the icons from all forms, looking in a confused way for the most appropriate one. In this situation, we are showing him what are the options at a very specific moment. Let us see now how we do that.

```
int subCategory = ((IconicInformation)list.ToArray()[list.Count-1]).  
Subcategory ;
```

We start all the time to check the correctness of the input data from the last element that has been introduced, thinking that in a crisis situation the user will have to create fast some sentences, and he will not be interested so much in navigating inside the icons, to change them. It has been proven actually that if one realizes that he introduced an element wrong, he will delete the sequence of icons, or words until it reaches that point, and enters the correct element; instead of preferring to navigate backward, deleting one item, and then navigating back. But my supervisor wished for me to create an icon editor, so in this case, if the user wants to navigate and insert in the middle of the sequence of icons, the basic grammar checker will be done.

At the beginning of the application, we have started the interface part, by creating one MainForm, and fourteen secondary forms for each one of the categories. But this was not an elegant method to solve things, because every time we were clicking on one index – icon, a new forms was opened, and above that another one, and so one. So, to adapt this method, making it more elegant, we have added a panel on the third area of the MainForm. On this panel we will display the icons for a specific category, without needing to create over and over again a new window. It is not only inefficient, but also not nice in appearing.

```
IEnumerator controlEnumerator = pnCurrentCat.Controls.GetEnumerator() ;  
while(controlEnumerator.MoveNext())  
{  
    if(controlEnumerator.Current is CImageButton)  
    {  
        CImageButton currentButton =  
(CImageButton)controlEnumerator.Current ;  
  
        currentButton.Visible =  
semanticMatrix[subCategory, currentButton.iconInf.Subcategory] == 1 ;  
    }  
}
```

Creating an enumerator for the panel, we take one by one all the controls from it, the ones that have CImageButton type, and add to the panel, and make visible only the buttons that have a semantic connection with the last element from the list.

In order to load them practically, we have to make them a copy, and when we refer to them, we are speaking about the buttons. We can't access the buttons from

another form. It is not logical correct that a button should be positioned in two placed at a time.

```
IEnumerator controlEnumerator = form.Controls.GetEnumerator() ;  
  
while( controlEnumerator.MoveNext() )  
{  
    Control c = (Control) controlEnumerator.Current ;  
    if(!(c is CImageButton)) continue ;  
    CImageButton currentButton = (CImageButton)c ;  
    //create a clone of the button  
    CImageButton cib = new CImageButton() ;  
    cib.Size = c.Size ;  
    cib.iconInf = currentButton.iconInf ;  
    cib.Left = c.Left ;  
    cib.Top = c.Top ;  
    cib.Image = currentButton.Image ;  
    cib.Click += new EventHandler( imageButtonClick ) ;  
    //add the button to the panel  
    pnCurrentCat.Controls.Add( cib ) ;  
}
```

Chapter 7

Testing – the user manual

The application can be tested and downloaded from my local web page:

<http://www.kbs.twi.tudelft.nl/People/Students/J.Tatomir/>

The steps to test this application are easily to detect, but they are still constraining the user to pass certain stages. First of all, we have to make sure that we have the proper virtual machine. This project was build like a server – client one, and the server site should be the one who is started the first one. The server has to establish its status as being available, and to be prepared to accept data from the client side. Here is the form that will appear:

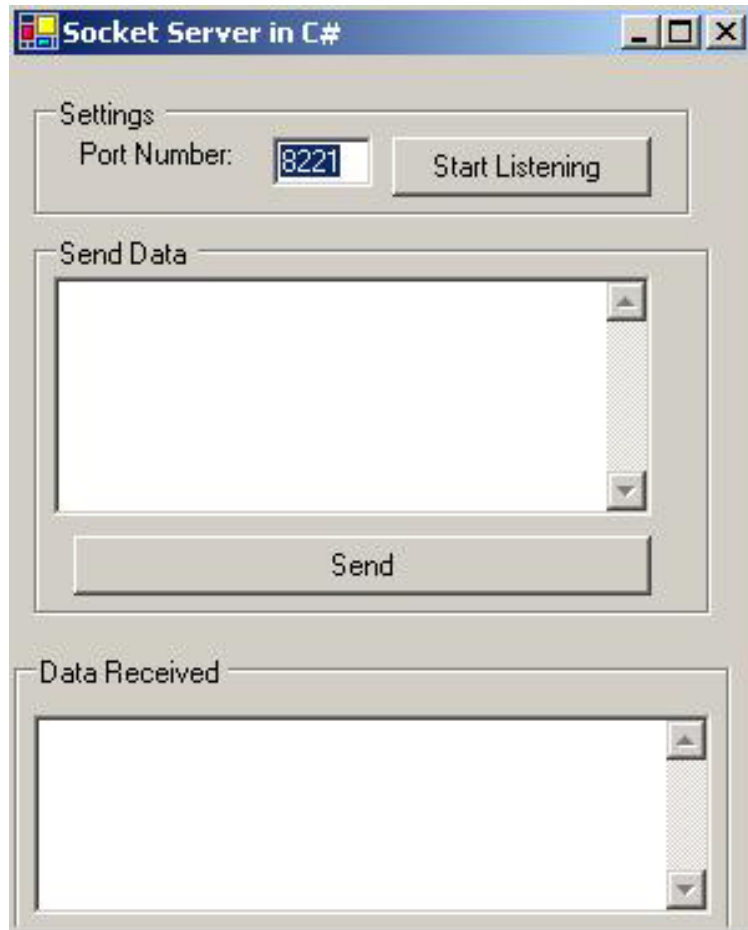


Figure 43. The server form

On this form, press Start Listening. From this moment the server is prepared to listen on the port 8221, the same port that the client application uses to send data. The main form for the server also includes an area that could be used in the future for the real communication with the client, but for the moment, we will use only the area dedicated to the receiving of the data.

After starting the server site, we are prepared to start the client side. While running this application, a window is being displayed asking for the IP address of the server. The whole project can run on the same machine, or on different machines, with the condition that these two ones are connected in the network. Here is the image of the form:



Figure 46. Form asking for IP address

After introducing the IP address, press ok. The next image will appear:

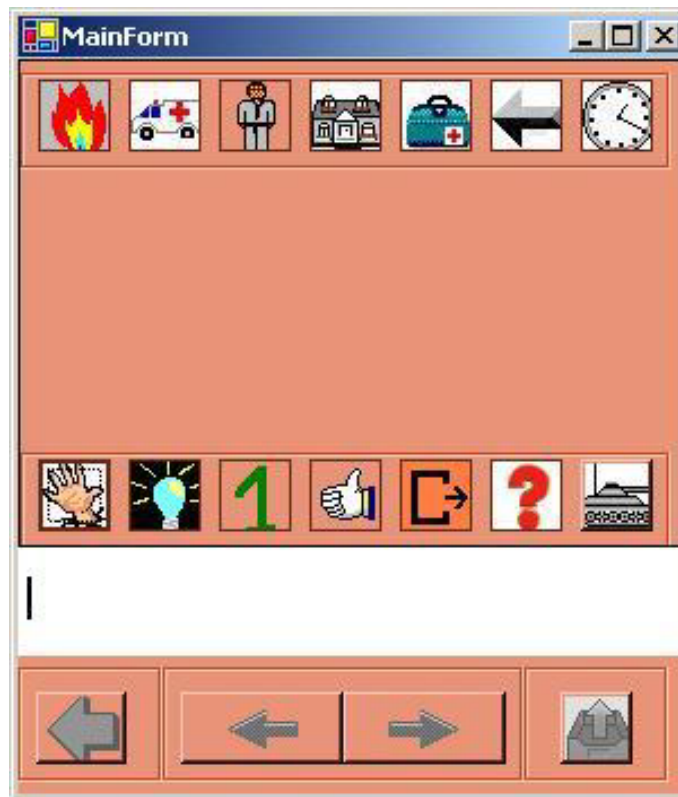


Figure 47. The main application- the starting frame

Starting with this moment, the user can introduce the icons that can form the idea he wants to express. At the beginning, he can choose any icon, all being available; but after choosing one icon, we will display only the icons that can lead to a correct input sequence, speaking from semantic point of view. In this way it is easier for the user to create the input sequence, because he doesn't have to bother with illogical combinations. For example, it is obvious that after a number there cannot follow a verb.

Another step would be to have a minimum base knowledge about the structure of the grammar. One should not be surprised by the fact that by the moment, we are doing only a semantic restriction, and there are still some sequences of icons, that even they have passed with success after the semantic checker, from the point of the syntactical one it will not. But this will be corrected, or continued in the second part of the development of the project.

But let us give a basic example of how the project is working. For example, we will like to start with ‘Man’, and then we will like to add a verb, a human action. We will notice that from the all the initial icons, we will have displayed only some of them – the adjectives: scared, blind and deaf won’t appear. And the form looks like this:



Figure 48. The Human Actions form after inserting the ‘Man’ icon

Chapter 8 Conclusions

Related works

Initially, even these strategies do not always work for those Traumatized by the sudden onset of aphasia, with stiff bodies in a state of shock and depression. A better strategy would be to utilize the

origins of written communication; very simple symbols.

Further work

Language learning through icons.

References

[1] Allen, J. - *Natural Language Understanding* – Benjamin/Cummings Publishing Company, Inc., 1987

[2] Callan, R. – *Artificial Intelligence* – Palgrave Macmillian, 2003.

[3] Castagliola, G., Tortora, G., Arndt, T. – *A Unifying Approach to Iconic Indexing for 2-D and 3-D Scenes* – IEEE Transactions on Knowledge and Data Engineering, vol. 4, no. 3, June 1992

- [4] Chandler, D. - *Semiotics: the Basics (The Basics)* - Routledge, an imprint of Taylor & Francis Books Ltd, 2001
- [5] Cook, G. - *The Discourse of Advertising* - Routledge, London, 1992
- [6] Eco, U. - *A Theory of Semiotics* - Bloomington, IN: Indiana University Press/London: Macmillan, 1976
- [7] Hayes-Roth, F., Waterman, D. A., Lenat, D.B. – *Building Expert Systems* – Addison-Wesley Publishing Company, Inc., Massachusetts, 1983
- [8] Horton, W. – *The Icon Book: Visual Symbols for Computer Systems and Documentation* – John Wiley & Sons, Inc., 1994
- [9] Johnstone, A., Scott, E. – *Generalized recursive descendent, part I Language design and parsing* - CSD – TR – 97 – 18, 1997
- [10] Knuth, D. E. – *On the Translation of Languages from Left to Right* – Inf. Contr. 8, 1965.
- [11] Krulee, G. K. – *Computer Processing of Natural Language* – Prentice-Hall International Editions, 1991.
- [12] Morris, Charles W - *Foundations of the Theory of Signs*. Chicago: Chicago University Press, 1970
- [13] Moyne, J. A. – *Understanding Language, Man or Machine* – Plenum Press, NY&London,1985.
- [14] Naur, P. - *Revised Report on the Algorithmic Language ALGOL 60*. - Communications of the ACM, Vol. 3 No.5, pp. 299-314, May 1960.
- [15] Negnevitsky, M. – *Artificial Intelligence, A Guide to Intelligent Systems* – Addison-Wesley, Pearson Education, 2002.
- [16] Peirce, C. S. - *Collected Writings* (8 Vols.). - Ed. Charles Hartshorne, Paul Weiss & Arthur W Burks, Cambridge, MA: Harvard University Press, 1931
- [17] Russel, S., Norvig., P. – *Artificial Intelligence, a modern approach* – Prentice Hall International Editions, 1995.
- [18] Schank, R. C., Reisbeck, C. K. – *Inside Computer Understanding: Five Programs Plus Miniatures* – Lawrence Erlbaum Associates, Publishers Hillsdale, New Jersey, 1981

[19] Sharma, R., Yeasin, M., Krahnstoeber, N., Rauschert, I., Cai, G., Brewer, I., Maceachren, A., M., Sengupta, K. – *Speech-Gesture Driven Multimodal Interfaces for Crisis Management* – IEEE, vol. 91, no. 9, September 2003.

[20] Vaillant, P. – *Semiotique des langages d'icônes* – Honore champion, Paris, 1999

[21] Vaillant, P. – *Modelling Semantic Association and Conceptual Inheritance for Semantic Analysis* – Mautousek et al., TDS 2001, LNAI 2166, pp. 54-61, 2001.

[22] Woods, W. A. – *Transition Network Grammars for Natural Language Analysis - Readings in Natural Language Processing* – Morgan Kaufmann Publishers, Inc., 1986.