Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

Miroslav Janíček

Správce souborů v přirozeném jazyce
File Manager in a Natural Language

Ústav formální a aplikované lingvistiky

Vedoucí bakalářské práce: RNDr. Ondřej Bojar
Studijní program: obecná informatika

2007

I thank my family for their tireless support and encouragements. I also thank all men and women at ÚFAL, and especially my supervisor, RNDr. Ondřej Bojar, for his helpful comments and advice.

Contents

List of Tables

Název práce: Správce souborů v přirozeném jazyce
Autor: Miroslav Janíček
Katedra (ústav): Ústav formální a aplikované lingvistiky
Vedoucí bakalářské práce: RNDr. Ondřej Bojar
e-mail vedoucího: `bojar@ufal.mff.cuni.cz`

Abstrakt: Práce představuje NLSH, správce souborů pro UNIXové operační systémy s uživatelským rozhraním v psaném přirozeném jazyce. NLSH je dialogový systém schopný zpracovat základní kontextové prvky komunikace, jako je použití zájmen; využívá dostupných nástrojů pro automatický převod vět v češtině do jejich tektogramatické reprezentace v podobě závislostních stromů podle funkčního generativního popisu. Tyto závislostní stromy jsou pak interpretovány za použití formálního dotazovacího jazyka vycházejícího z výrokové logiky. Práce popisuje návrh, architekturu a omezení jak systému jako celku, tak i interpretačního mechanismu a dotazovacího jazyka.

Klíčová slova: dialogový systém, uživatelské rozhraní v přirozeném jazyce, správce souborů, tektogramatická rovina.

Title: File Manager in a Natural Language
Author: Miroslav Janíček
Department: Institute of Formal and Applied Linguistics
Supervisor: RNDr. Ondřej Bojar
Supervisor's e-mail address: `bojar@ufal.mff.cuni.cz`

Abstract: In this work we present NLSH, a file manager for UNIX-like operating systems with user interface in written natural language. NLSH is a context-aware dialogue system capable of basic anaphora resolution; it uses currently-available tools for automatic conversion of sentences in Czech language to tectogrammatical dependency trees based on the framework of Functional Generative Description. The dependency trees are then interpreted using a formal querying language based on propositional logic. We describe the design, architecture and limitations of the whole system as well as of the interpretation mechanism and querying language.

Keywords: dialogue system, natural language user interface, file manager, tectogrammatical layer.

Chapter 1

Overview

In this work, we present NLSH, a file manager for UNIX-like operating systems with user interface in natural language, namely the Czech language.

1.1 Natural language as a user interface

By the term "user interface" we mean a collection of methods using which the user interacts with a computer system, i. e. issues commands and queries its state. There are many interface styles, but currently the most prevalent forms are *graphical user interfaces* and *command-line interfaces*.

To use a natural language to interact with the computer is an appealing perspective: indeed, natural language is a flexible and expressive form of communication we use in our daily lives and therefore we are experienced in it. The users of an application that would allow such an interaction would therefore not be required to learn a new artificial language[1] in order to work with the system, which would render it much more accessible than without such possibility.

Moreover, in combination with speech processing, a system that utilizes natural language as a form of input would not require the user to type her request on a keyboard. This would free her hands and therefore increase her productivity.

1.1.1 Problems

However, according to [5], it is a "conventional wisdom" in the field of human-machine interaction that negatives of the usage of natural lan-

[1]This, of course, includes formal languages such as SQL, but also symbols, icons and colour guidelines in graphical user interfaces.

guage as a user interface overweight its positives, at least in cases where it is *the only* interface style.

There are three main problems which every system that employs natural language as an interface style has to tackle in order to be useful to their users:

- **Not enough information.** When speaking to each other, people convey only so much information to each other that they deem sufficient to successfully decode the meaning. The message itself is thus dependent not only on the context, but also on the knowledge of the world and reasoning skills of the recipient.
- **Conceptual vs. language errors.** If a request from the user is not understood by the system, the user is typically asked to reformulate it and try again. While this may lead the user to believe that the system did only fail to parse the language construction, her query may as well be beyond the conceptual coverage of the the system. From the system's perspective, such error states are very hard to detect.
- **Antropomorphism.** Since intelligence is an essential element in understanding (unrestricted) natural language, any system that would engage in conversation with a human user is likely to be attributed at least some intelligence. This may lead to excessive expectations of system's potential and to a frustrating work experience.

Therefore, in all situations, the user should be encouraged to be verbose and precise in her requests and should be aware of the system's potential. Also, it should be obvious to her that the system is not a real person, but a computer with minimal intelligence and reasoning capabilities.

1.1.2 Implementations

For a thorough survey of available systems that provided natural language interfaces to databases, as well as the summary of problems and approaches to their solution, the reader is referred to [1]. Although the survey is more than 12 years old, the author of this text is not aware of any similar overview that has been made since then.

1.2 A natural language file manager

NLSH is a *file manager*. File managers are software applications that allow the user to manipulate with files and directories. This manipulation includes deletion, copying, moving or creating them.

The basic idea behind NLSH is that the elementary unit of each operation on files[2] is a list of files over which the operation is to be performed. NLSH tries to deliver a novel approach to the creation of such lists. The method that "ordinary" file managers offer the user to create such lists is more or less explicit – the user usually picks target files by a cursor in the file manager window – and is essentially an enumeration.

Instead, NLSH treats the file system as a universe populated by objects that have some properties and provides means for selecting subsets of all objects using propositional logic. This allows the user to construct a list of files merely by describing their properties instead of having to name each member.

For instance, if the user wanted to work with all files in directory X, it would seem natural to describe the target of the operation as *files whose location is equal to X* – i. e. to treat file location as a property of that file. On the other hand, a graphical file manager would force the user to actually *enter directory X* and manually *select all files*.

The advantage of descriptive approach taken by NLSH is that the user can formulate her queries in a complex language that is closer to her *intention* and, importantly, can be further processed by the computer. Looking at the example above, it is considerably easier to interpret the action of getting *files whose location is equal to X* as two operations, *enter directory X* and *select all files*, than to infer that these two operations in this order determine files with the mentioned property.

However, the querying mechanism forms only a half of the whole system. In order to fulfill its function, NLSH must find interpretation of each (suitable) sentence in the input natural language; only after such interpretation is found, the querying mechanism is of some use. Since a file manager must be able to manipulate objects in the system as well as answer questions about their properties, the mechanism for matching interpretations to sentences must be flexible enough to support both of these requirements.

1.3 Structure of the thesis

The structure of the thesis is as follows: Chapter 2 describes the querying mechanism used in NLSH to create lists of objects, its properties and implementation.

[2]What we mean here is an "external" operation, i. e. an operation during which the contents of the manipulated file do not change.

Chapter 3 is concerned with the design, architecture and implementation of the interpretation system that assigns each sentence a meaning and its using the aforementioned querying mechanism.

Chapter 2

Querying mechanism

The querying mechanism of NLSH is a descriptive formal language based on propositional logic, with some extensions that allow formulation of queries beyond its scope. Although designed primarily with a UNIX-like computer system in mind, it is fairly domain-independent and extensible.

This chapter describes in detail the formal properties of the system and its implementation.

2.1 Formal description

Each query is related to a certain *state of the world*; we denote the set of all possible states of the world Ω. Since NLSH is a real-world application, the concept of the state of the world is different from "virtual-world" mechanisms such as the situation calculus in planning – operations in such systems typically take one state of the world and yield another, which is then given to next operation and so forth.

In NLSH, the state of the world is a formal representation of the "real" world state: it comprises not only the internal status of the computer system, such as the current hierarchy of file system or the layout of the user's screen, but also influences of the *outer* world, such as time.[1]

The querying mechanism is therefore unable to answer hypothetical questions. Any effort to add planning and deep inference capabilities would obviously require the addition of a whole new layer of operations.

[1]Generally, every input is related to the outer world – be it the movement of computer mouse or the movement of people in a nearby street in case the computer has a camera attached. We may even extend our definiton of "input" here to include hardware malfunctions etc.

Although it might be a desirable feature for a real-world system, NLSH does not implement such a layer.

Another important note concerning the reality of the world is the lack of atomicity of operations. Most, if not all, operations are sequences of suboperations and it usually does take some time before their execution is done. From the practical point of view, this means that the state of the computer may well be changed before the operation is finished and that it is only a matter of chance whether the change is related to the data the operation works with. In such case, the result will be inaccurate.

We should also take this into account in the following description. However, this would make the text unnecessarily hard to follow; therefore, for the sake of simplicity, we will ignore this fact in the further text and will treat operations and queries as atomic.

2.1.1 Objects, types and attributes

NLSH operates on two entities present in every UNIX-based system: files and users. These are called *objects* in NLSH's terminology. Each object is fully determined by a unique *identifier*; for files, the identifier is the absolute path to them in the file system, users are identified by their UID.[2] From the view of underlying layers, however, identifiers are atomic – given an intentifier, NLSH does not deduce anything about the properties of the object it describes.

Instead, identifiers are "handles" – implementation-dependent values that determine how operations and queries on given objects are to be performed. For instance, a query about the object name triggers different actions for files and users: for files, the name can simply be obtained from the path, but for users, a user database needs to be consulted. This makes it easy to change the underlying implementation of current querying primitives, but also allows a simple addition of new entities such as user groups or open network connections to the system.

In the following, Θ denotes the set of all object identifiers.

Attributes

Every object in NLSH has a set of attributes. Or, more precisely, given an object, an attribute and a constraint, NLSH can decide whether the constraint on the attribute is valid for the given object. This distinction is

[2]UID means *user identifier*. In UNIX and similar operating systems, UID is always a non-negative integer.

more elaborated upon in section 2.2.1; for now, let us just point out that it is not necessary to know the actual *value* of the attribute – for example, if we already know that a file is smaller than 1 kilobyte, we may safely conclude that it also smaller than 2 kilobytes.

NLSH defines 13 attributes (see table 2.1 for an overview). A denotes the set of all attributes.

Table 2.1: Attributes in NLSH

Attribute ($\in A$)	**Description**	**Typical value**
objtype	object type	$t \in T$
name	object name (filename or username)	*string*
location	file location	*string*
ctime	date and time of last change	*timestamp*
mtime	date and time of last modification	*timestamp*
bytesize	size in bytes	*integer*
owner	object owner	*uid*
homeof	users for which this is home directory	*set of uid's*
files	contents of directory	*set of file-id's*
contents	contents of file	*string*
target	symbolic link target	*string*
exists	object existence flag	*boolean*
uid	UID represented by the object	*uid*

Note that none of the attributes is related to file access permissions. This feature of UNIX-like operating systems is not reflected in NLSH.

Object types

Each existing object has a *type*, which is represented in the attribute system as attribute objtype $\in A$. There are 4 object types in NLSH (see table 2.2 for their listing), we denote the set of all object types T.

Define relation $PT \subset A \times T$. If $attr \in A$ and $type \in T$ so that $(attr, type) \in PT$, we say that $attr$ *is compatible with* $type$; object that has type $type$ necessarily has attribute $attr$ and object with attribute $attr$ may be of type $type$.

Relation PT, as implemented in NLSH, is presented in table 2.3.

No hierarchy of types is defined – although types regular_file, directory and symlink share most of the attributes, they do not have a common an-

Table 2.2: Object types in NLSH

Type ($\in T$)	**Description**
regular_file	regular file
directory	directory
symlink	symbolic link
user	user

Table 2.3: Validity of attributes for given types

	regular_file	directory	symlink	user
objtype	×	×	×	×
name	×	×	×	×
location	×	×	×	
ctime	×	×	×	
mtime	×	×	×	
bytesize	×			
owner	×	×	×	
homeof		×		
files		×		
contents	×			
target			×	
exists	×	×	×	×
uid				×

cestor. This is a clear limitation of the system and a candidate for future improvement.

2.1.2 Restrictions

The formal language based on propositional logic that allows the user to describe the properties of objects she wants to work with is called the language of *restrictions*[3] in NLSH's terminology.

The formal language of restrictions

Define *restriction* as follows:

(1) A pair $(attr, rel)$, where $attr \in A$ is an attribute and rel is an unary relation (property) is restriction. We call such restriction an *elementary restriction*.

(2) If R is a restriction, then $\neg R$ is a restriction.

(3) If R_1, R_2 are restrictions, then $R_1 \wedge R_2$, $R_1 \vee R_2$ and $R_1 \Rightarrow R_2$ are restrictions.

Restriction semantics

To assign semantics to restrictions, we define function sat, which, given an object identifier $\theta \in \Theta$ and a state of the world $\omega \in \Omega$, assigns each restriction a truth value:

(1) Let $R = (attr, rel)$ be elementary restriction. Let val be the value of attribute $attr$ of object θ in the world state ω. Then $\mathrm{sat}_{\theta,\omega} R = \mathrm{true}$ iff $val \in rel$.

(2) Let R be restriction and $R = \neg R'$. Then $\mathrm{sat}_{\theta,\omega} R = \neg \mathrm{sat}_{\theta,\omega} R'$.

(3) Let R_1, R_2 be restrictions, $R'' = R_1 \wedge R_2$. Then $\mathrm{sat}_{\theta,\omega} R'' = \mathrm{sat}_{\theta,\omega} R_1 \wedge \mathrm{sat}_{\theta,\omega} R_2$. Similarly for $R_1 \vee R_2$ and $R_1 \Rightarrow R_2$.

Given an elementary restriction R, we may treat $\mathrm{sat}_{\theta,\omega} R$ as a propositional variable. Thus, any restriction may be converted to a formula in

[3]Expressions in this language *restrict* the set of all objects in the world to the desired subset.

propositional logic merely by applying rules (2) and (3) on non-elementary restrictions and substituting propositional variables for elementary restrictions.

Hence, every restriction, being a propositional formula, can be converted to its disjunctive normal form (DNF). Disjunctive normal form of a formula is a disjunction of clauses, where clauses are conjunctions of literals. Literals are either negated or non-negated propositional variables, in our case elementary restrictions.

Observe that for $R = \neg R'$, $R' = (attr, rel)$ and $type \in T$ object type of θ in ω compatible with $attr$, it is true that

$$\operatorname{sat}_{\theta,\omega} R = \operatorname{sat}_{\theta,\omega}(attr, \overline{rel})$$

where $\overline{rel}$ is the inverse unary relation to rel. The compatibility of $type$ with $attr$ is essential: if $(attr, type) \notin PT$, the equation would not hold.

Thus, if we assure this condition, we may treat every literal in a clause of a restriction's DNF as an elementary restriction.

In order to satisfy a DNF clause, all literals (i. e. elementary restrictions) in it must be satisfied. The fact that clauses are in disjunction means that if the formula as whole holds for the objects, at least one of the clauses is satisfied. DNF clause is therefore the basic unit of restriction evaluation.

Possible types of a clause

To make sure that negated literals may be evaluated as elementary restrictions as in the observation above, NLSH assumes that if attribute $attr$ is *mentioned* in an (elementary) restriction, any object that satisfies that restriction is of type compatible with $attr$.

For example, restriction $R = (\text{bytesize}, \neq 0)$ requires that any object that satisfies R has type $type$ so that $(\text{bytesize}, type) \in PT$, that is, $type =$ regular_file. Even though objects with type user do pass the condition that the value of their bytesize attribute is not equal to 0 (for they do not even have it), they are discarded.

Thus, given a DNF clause, this principle does not only provide means for its simpler evaluation, but also allows us to determine the set of possible types of objects that satisfy it, which is essential in order to generate suitable objects efficiently.

Let $C = L_1 \wedge \ldots \wedge L_n$ be a DNF clause. Since we may now safely treat negated literals as elementary restrictions, $L_i = (attr_i, rel_i)$, where rel_i is either the unary relation of the elementary restriction in case the literal is non-negated, or the inverse relation to the underlying restriction if it is negated.

We define function posstypes that assigns each clause a set of possible types of objects that satisfy it as follows:

$$\mathrm{posstypes}\, C = \bigcap_{i=1}^{n} \{type_i \,|(attr_i, type_i) \in PT\}$$

2.1.3 Constructors

At this point, we already have at our disposal a mechanism to create lists of objects by means of merely describing their properties.

However, that mechanism has limits.

Although expressive enough to describe the actual properties of objects, propositional logic is unable to describe their *relations with other objects*. For instance, the language of restrictions as it has been defined in 2.1.2 has no means to describe objects such as "the biggest file" or "the user with the highest UID".

An obvious solution to this problem would be to base restrictions on first-order logic instead of propositional logic. Indeed, "the biggest file" could then be described as:[4]

$$\theta : \forall\theta'(\textsf{bytesize of } \theta' \leq \textsf{bytesize of } \theta) \wedge (\textsf{objtype of } \theta = \textsf{regular\_file})$$

However, NLSH pursues another direction: instead of extending the language, it presents a mechanism to post-process the results returned by "propositional" restrictions.

Orderings and selectors

NLSH comes with two concepts that aim to supplant the extension of restrictions to first-order logic. They do not touch the process of restriction evaluation; their area of operation is the resulting set of object identifiers that satisfy the restriction.

Since the result of the restriction evaluation is a set, in order to be presented to the user or supplied to an operation as an argument, it needs to be serialized. By default, when converting the set to a list, NLSH does not order it in any "sophisticated" way – it is simply sorted by the object identifiers. While this may be sufficient in many cases, NLSH also allows to define the order.

Such facility is called *ordering* in NLSH's terminology. Ordering is a pair $(attr, order)$, where $attr$ is an attribute and $order$ is either ascending

[4]The world state is omitted for the sake of clarity.

or descending. A concept similar to ordering is needed in any system that supports commands such as “list all files sorted by age”.

The second concept is called *selector*. Selector is a function that picks a specific range of objects from the list. It is important to note that selector comes to action *after* the set is converted to a list by an ordering.

By combining ordering with an appropriate selector, the user can formulate queries such as “the biggest file” or even “the second biggest file”. All that is needed to do is to sort the list of files by size in descending order, and pick the first or second element of the list. The second example, in particular, illustrates the simplicity of this approach: such query, albeit possible, would be considerably more complex in first-order logic.

Shapes

A *shape* is an additional constraint on the number of elements in the resulting list. When the user expresses her desire for “the empty file in directory Y”, the restriction itself does not contain the information that the user *expects* that there will be exactly one file with this property. This additional information allows NLSH to appropriately react when e. g. there is no empty file in directory Y or there are more than one such file.

Constructors

Thus, *constructor* is a 4-tuple (S, L, O, R), where S is a shape, L is a selector, O is an ordering and R is a restriction. The shape, selector and ordering do not need to be specified.

2.2 Implementation

Being a real-world application, NLSH has to find compromises between pedantic implementation of the entire system and its practical expressiveness and effectiveness.

2.2.1 Restriction evaluation

Since the functionality of the whole querying mechanism revolves around the formal language of restrictions, we shall start by connecting the theory with the real world.

In the state of world ω, given an object identifier $\theta \in \Theta$ and an elementary restriction $R = (attr, rel)$, how do we determine the value of $\mathrm{sat}_{\theta,\omega} R$?

The answer to this question depends on the type of relation rel: the target of rel may be either a value such as string or integer, or an object identifier (or a set of object identifiers).

Restrictions of values

Each object identifier determines the method of handling the represented object. For instance, the value of attribute exists is obtained by searching file `/etc/passwd` for users and by executing function `lstat()` for files. In NLSH's terminology, these methods are called *property providers*. See table 2.4 for their summary.

Table 2.4: Property providers in NLSH

Attribute	Property provider for files	Property provider for users
objtype	`lstat()` call	*automatically set to* user
name	*from the identifier*	`/etc/passwd` scan
location	*from the identifier*	
ctime	`lstat()` call	
mtime	`lstat()` call	
bytesize	`lstat()` call	
owner	`lstat()` call	
homeof	`/etc/passwd` scan	
files	`readdir()` call	
contents	`grep` execution	
target	`readlink()` call	
exists	`lstat()` call	`/etc/passwd` scan
uid		*from the identifier*

For each object identifier, NLSH stores its properties in a data structure called *object cache*. The object cache keeps the access to I/O at a necessary minimum. For each $attr \in A$, a set of known valid properties (unary relations) is kept; appropriate property providers are consulted only if the value cannot be inferred from the already known properties.

When an object changes, the corresponding item in the cache becomes outdated. However, when the object is accessed the next time, the change is detected and all stored properties are rechecked.

For attributes such as bytesize or owner, the cost of storing of the actual value and its retrieval is negligible, but, the value of attribute contents can be much larger than the volatile memory of the computer. By using this mechanism, NLSH avoids storing the entire files as well as unnecessary executions of `grep`[5].

Comparison restrictions

When the target of the unary relation is an object identifier tgt, objects θ and tgt are compared. The actual method of comparison is again determined by handles of both objects. For most attributes, a simple projection of their values is sufficient; there are, however, attributes that need special treatment.

Table 2.5 lists these special attributes; i. e. the comparison of objects by attributes that are not mentioned in the table is performed by projecting their values and comparing these.

Table 2.5: Special object comparison methods

Attribute	θ	tgt	**Comparison method**
location	file	file	compare value of location to the full path to tgt
owner	file	user	compare value of owner to the UID of tgt
homeof	file	user	compare value of homeof with $\{tgt\}$
contents	file	file	execute `cmp` on the two files
target	file	file	compare value of target to the full path to tgt

2.2.2 Construction

The problem of getting all objects that satisfy restriction R in world state ω is equal to the problem of constructing set

$$S_R = \{\theta \in \Theta |\, \mathrm{sat}_{\theta,\omega}\, R = \mathrm{true}\}$$

Function $g : \Omega \to P(\Theta)$ is called *generator*. Generator $\nu(\omega) = \Theta$ is called the *universal generator*.

We call set

$$C_{g,R} = \{\theta \in g(\omega) |\, \mathrm{sat}_{\theta,\omega}\, R = \mathrm{true}\}$$

the *set constructed using generator* g. Obviously, for each g generator and R restriction, $C_{g,R} \subseteq S_R$. For the universal constructor, $C_{\nu,R} = S_R$.

[5]`grep` is a standard UNIX tool for finding text patterns in files.

For finite Θ, we can simply follow the definition and use the universal generator to obtain S_R.

For infinite Θ, however, the universal constructor cannot be used. That is the case of NLSH – although user identifiers are of limited range[6], the number of possible paths in the file system is virtually infinite.

It is therefore vital to reduce the number of generated object identifiers to a finite number. By default, **only existing objects are generated**. Nonexistent objects may also be generated, but only as a special case.

Separated generation

As noted in 2.1.2, given a DNF clause of a restriction, it is possible to determine all allowed types of objects that satisfy it. Therefore, each object type in NLSH has its own generator.

Since each generator is now concerned only with objects of a specific type, it "knows" which attributes are present and which of them can be used in their conversion to an object identifier. If the supplied attribute values fully determine the object identifier, the generator stops and returns the identifier; if not, the system needs to be traversed. Note that the first case is also **the only way to construct a nonexistent object**.

For example, when constructing objects of type regular_file, the object identifier is formed by concatenating the values of attributes location and name. Thus, if the restriction specifies values of both these attributes, the generator returns the identifier without actually testing whether it is present in the file system.

But what to do if none of the two attributes is specified? At that point, the generator would have to return object identifiers of all existing objects of the given type. For users, this not a serious problem as the number of users in the system is reasonably low. For files, however, traversing the entire file system is a very costly operation. For the sake of efficiency, file generators in NLSH return an empty set in such case and signalize that they *refused to work*. The caller may then try to add another restriction to the clause (such as $(\mathsf{location}, = \mathsf{CWD})$, CWD denoting the current working directory).

The algorithm of construction

Given restriction R, the algorithm of construction of set C_R is as follows (let ω be the state of the world at the moment of construction):

[6]due to their representation in the computer

1. Convert restriction R to its DNF, $C_1 \vee \ldots \vee C_n$.
2. For $i = 1 \ldots n$:
   (a) For each type $t \in \mathrm{posstypes}\, C_i$, call the corresponding object generator and supply it with C_i. Denote the result of generation G_t.
   (b) $G := \bigcup_t G_t$
   (c) $S_i := \{g \in G | sat_{g,\omega} C_i = true\}$
3. $C_R := \bigcup_{i=1}^{n} S_i$

After C_R is constructed, the post-processing takes place in this order:

1. Apply ordering.
2. Apply selector.
3. Apply shape.

2.2.3 Treatment of nonexistent objects

The mechanism described in this chapter is well-suited for querying about the current state of the system, but its performance in tasks pertaining to its change is limited to changes of objects that already exist (such as renaming of files). Adding *new* objects to the system may be problematic.

In general, generating nonexistent objects necessarily poses efficiency problems. If we assume that the number of all existing objects in the system is always finite and that Θ is infinite (as in NLSH), the time to generate "all files that do not exist" is necessarily infinite.

However, some actions, such as the creation of directory[7], *require* an argument that does not exist at the time of interpretation. To assure that, the current implementation changes all elementary restrictions except for location and name to $(\mathsf{exists}, = \mathrm{false})$. (See Section 2.2.2.)

Because object identifiers carry only the information about methods of manipulation with the represented object, they cannot be used for addition of objects that need more information that is contained in such handles – for instance, when adding a user, the user name must be specified. Since all users are represented by their UID, there is no possible way to implement such an action in NLSH. (For directories and files, however, the handle is sufficient.)

[7]Which is also the only such action in the current implementation of NLSH.

Therefore, in order to allow such actions, the identifiers would need to be enriched with more information.

Chapter 3

System architecture

NLSH works with Functional Generative Description (FGD), which is a stratified dependency linguistic framework devised since 1960's by a team led by Petr Sgall at the Charles University in Prague. Thanks to the creation of the Prague Dependency Treebank (PDT) for Czech language, which uses FGD as its theoretical background, tools for automatic annotation of plain text sentences to structures used by the treebank are available.

Each sentence in PDT has four layers of interpretation, each serving as the basis for the next one: the *word layer*, in which the sentence is divided into words and punctuation, the *morphological layer*, in which each word is assigned a morphological tag, the *analytical layer*, in which dependencies between words are linked, and finally the *tectogrammatical layer*, in which the sentence is represented by a dependency structure where each node is assigned a functor, that is, its semantic function in regard to its parent.

NLSH works with the tectogrammatical representation of the sentence and tries to convert it to a symbolic expression of the problem, using the querying mechanism described in Chapter 2. Due to the distance of tectogrammatical layer to the surface representation of a sentence, NLSH needs not to worry about the surface and can focus on the semantics part.

None of the automatic annotation tools is fully accurate, and due to the nature of their usage (the output of each of them being the input of the next one), the error is cumulative. As a result, the output of the last step, the tectogrammatical analysis, is often incorrect. NLSH has to cope with that issue.

Since PDT is a treebank of sentences in Czech language, the automatic annotation pipeline works only for Czech. Thus, the natural language the user communicates with NLSH is the Czech language. However, the method itself is independent of language – if tools for tectogrammatical analysis of other languages were available, NLSH could be ported to these as well.

3.1 Concepts

The following section summarizes the main properties of the method used in NLSH to interpret input sentences.

3.1.1 Targets

Constructors described in Chapter 2 require the user to explicitly specify all the properties of objects she wants to work with. This rarely happens in real-world conversations, where the user may for instance use pronouns to refer to objects she mentioned earlier. Moreover, mere constructors are not capable of expressing sets of objects such as "all files except file X" or "all files except the biggest one".

NLSH therefore defines additional types of expressions that can be converted to a list of objects. These are called *targets*. There are four types of targets:

- **Constructors.** Constructors are 4-tuples (S, L, O, R), where S is a shape, L is a selector, O is an ordering and R is a restriction (the definition is identical to that in Section 2.1.3).

  Corresponding example: *"the biggest file in directory X"*.

- **References.** These constitute references within the context of the dialogue and always represent a target that has been used earlier in the discussion. The process of resolving references is described in 3.1.2.

  Formally, references are 4-tuples (S_r, L_r, O_r, R_r), where S_r, L_r, O_r and R_r are constraints on shape, selector, ordering and restriction, respectively. None of the constraints is required, which means that each of them may be left unspecified.

  Corresponding example: *"that file"*.

- **Filters.** Filter consists of constraints on restrictions, ordering, selector and shape. Given a target, the result of the construction of a filter is a list of objects obtained by constructing the target and keeping only objects that satisfy the additional conditions.

  Filters are 5-tuples $(S_f, L_f, O_f, R_f, T_f)$, where S_f, L_f, O_f and R_f are constraints on shape, selector, ordering and restriction, respectively;

T_f is the underlying target. As in references, constraints S_f, L_f, O_f and R_f are allowed to be unspecified.

Corresponding example: "*the oldest of them*".

- **Exceptions.** Given targets A and B that represent objects S_A and S_B, respectively, the result of the construction of this target is $S_A \setminus S_B$.

  Formally, exception is a pair (A, B).

  Corresponding example: "*all of them except the oldest*".

Targets are independent of the system context (that is, the state of the world and system settings). Because of that, it would not be possible to use the "current user" and "current working directory of the system" as arguments of other targets or even as separate targets.

Since such feature would severely limit the usability of the system, NLSH defines both of them as *symbolic targets*; their construction (which always occurs within a context) is done simply via their conversion to the corresponding constructors.

After that, since they are no more than "ordinary" constructors, the special information that they denote – for instance, the current working directory – is lost and cannot be retrieved again.

3.1.2 Context

The user's feeling of dialogue with NLSH is mostly an illusion; NLSH has no notion of the dialogue context except that it remembers targets that have been mentioned in the conversation.

The items stored in the dialogue memory are constructors, filters and exceptions – references and symbolic targets are converted to these types of targets (*instantiated*) prior to the point at which they are put into the memory and thus do not occur in it.

Structure of the dialogue memory

The actual usage of dialogue memory in conversation is inspired by [2]. Referenced targets are "shifted up" to reflect the fact that they were just used and have been refreshed in the dialogue.

The dialogue memory in NLSH is divided to a *global* and a *local* part. References search only the global memory. When a reference is successfully resolved, the referent is removed from the global memory and is pushed to the local memory. This means that no target can be referenced to more than once during the construction phase, i.e. within one request from the user.

After the construction is finished, all items from the local memory are pushed to the global memory in the same order as in which they were inserted. (The local memory is therefore a FIFO.)

This effectively implements the "chaining" of utterances where the focus of one utterance becomes the topic of the next utterance (for definition of the terms topic and focus see [8]), given the left-to-right order of traversal of input tectogrammatical trees where nodes are ordered from topical to focal by definition of the tectogrammatical layer. In other words, if two objects from one utterance could be considered as antecedents of a reference, NLSH assumes that the user refers to the less topical one of them.

Reference resolution

As defined in 3.1.1, reference is a 4-tuple (S_r, L_r, O_r, R_r) that constitutes constraints on restrictions, ordering, selector and shape of the intended target. None of these constraints is required, i. e. it is allowed that some or even all of them are not specified.

How do we decide whether a target T from the (global) memory satisfies these constraints, and can therefore be the antecedent?

- Let $T = (S, L, O, R)$ be a constructor. T satisfies the constraints if and only if (for constraints specified in the restriction) it is true that S_r covers S, $L_r = L$, $O_r = O$ and for each $\theta \in \Theta$ for which $\operatorname{sat}_{\theta,\omega} R = true$ also $\operatorname{sat}_{\theta,\omega} R_r = true$.

- Let $T = (S_f, L_f, O_f, R_f, T_f)$ be a filter and let T_f be a target storable in the memory (i. e. no reference or symbolic target). Denote $i(T)$ constructor obtained from T followingly:

  - If $T_f = (S'', L'', O'', R'')$ is a constructor, then
  $$i(T) = (S', L', O', R')$$
  where each of S', L' and O' equals S'', L'' and O'' if the constraints corresponding to them are not defined, and is equal to the (defined) constraints S_f, L_f, O_f otherwise. If R_f is defined, $R' = R_f \wedge R$; $R' = R''$ otherwise.
  - If $T_f = (S''_f, L''_f, O''_f, R''_f, T''_f)$ is a filter, then
  $$i(T) = i((S_f, L_f, O_f, R_f, i(T''_f)))$$

  – If $T_f = (A, B)$ is an exception, then

$$i(T) = i((S_f, L_f, O_f, R_f, i(A)))$$

  Then T satisfies the constraints if and only if $i(T)$ satisfies the constraints.

- Let $T = (A, B)$ be an exception. T then satisfies the constraints if and only if A satisfies them.

The most recently inserted target of those that satisfy the constraints is deemed to be the antecedent. In further processing, it is used as if the user has mentioned it directly.

3.1.3 Readings

NLSH assigns an interpretation, or *reading* in its terminology, to each node of the tectogrammatical tree. Such reading depends on the readings of the node's children and therefore is equal to the reading of the entire subtree. The reading of the root node is then the interpretation of the whole sentence.

We distinguish *inner* and *root* readings. Inner readings are a matter of implementation and are not directly visible to higher layers of the mechanism. However, it is the accuracy and quality of these readings, on which depends the functionality of the system.

Root readings, which are the interpretations of sentences, are of three types: *actions*, *queries* and *replies*. Actions are commands that change the state of the world and are typically represented by imperative sentences. Queries are commands that do not change the world, but merely examine it; most often, they correspond to sentences in interrogative mood. Replies are the user's answers to questions posed by the server.

Inner readings

Inner readings are of diverse types, ranging from flags denoting that the parent node is a reference or that the given node is the focus of the sentence, to elementary restrictions and targets.

The current implementation of NLSH is tailored to handle a test suite of 7 scenes containing 44 user utterances in total (see Appendix A for a complete overview), which, when converted to tectogrammatical trees, contain 132 inner nodes. There are 49 rules for inner readings; therefore roughly each third node has its own rule.

Root readings

The number of rules that match root readings is considerably smaller, as there are only 44 sentences in the test suite and thus only 44 root nodes.

Root readings are also much simpler than rules for inner nodes – for instance, the rule for matching the action of deleting a directory merely checks the sentence mood (which must be imperative), that the actor's reading is "you" and that the patient is a target. This target is then the argument of the operation.

In total, there are 16 rules for queries, 5 for actions and 1 for replies.

Rule matching

Each rule that assigns reading R to a tectogrammatical node N can be written as

$$(P, C) \to R$$

where P are necessary tectogrammatical properties of the node (such as the required values of grammatemes) and C is a set of necessary readings that have been assigned to the node's children.

Let C_N denote the readings assigned to children of N. Then the rule is said to *match node* N (N has reading R) if and only if N satisfies P and $C \subseteq C_N$.

There may be more than one rule matching the node and the readings of its children; in such case, multiple results are returned, but the rules cannot be combined.

Greediness

In order to allow the disambiguation of root readings in case there is more than one such reading, a "quality" measure of readings needs to be defined. The metric used in NLSH is called *greediness* and is equal to the sum of elements in C of each rule applied in process of obtaining the given reading. Thus, greediness of each reading is the number of tectogrammatical nodes it is based on.

Formally, let N be a tectogrammatical node, R reading and U the set of all rules for matching of readings. Let $u = (P, C) \to R$, $u \in U$ be a rule that matches N. For each $c \in C$, let $n(c)$ denote the child node of N that corresponds to reading c.

Then, the function gr that assigns greediness to each each pair (N, u) is defined as follows:

$$\mathrm{gr}(N,u) = 1 + \sum_{c \in C} \max \{\mathrm{gr}(n(c),v) | v \in U, v \text{ matches } n(c)\}$$

3.1.4 Instructions

The ultimate goal of the processing is to convert every reading to a sequence of *instructions*. It is assumed that every sentence consists of one or more instructions; in this model, compound sentences are allowed, but since they introduce a new array of problems, they had not been the focus of NLSH and are not supported. (The test suite contains a mere 1 compound sentence, #15).

Table 3.1 summarizes the instructions (and therefore the possible scope of operations) in NLSH.

Table 3.1: Instructions in NLSH

Instruction	**Description**
output	Send formatted output to the user
change_wd	Change working directory
create_dir	Create a directory
delete_file	Delete a file
move_file	Move a file
copy_file	Copy a file

3.2 Mechanism of action

NLSH operates in two input modes:

- **Command mode.** The user is expected to enter a command or query. This is the standard input mode.
- **Acknowledge mode.** In this mode, NLSH expects the user to express a reply to a question posed by the server earlier in the discussion (in the previous step). The current implementation of NLSH asks the user only boolean questions, e. g. if it really should delete the files the user had asked for. The user is therefore expected to reply either positively or negatively.

NLSH processes one user request at a time and works in loop until it is signalled to shut down. The loop goes as follows:

1. Read input from the user.
2. Convert the input to a tectogrammatical tree.
3. Apply structural transformations.
4. Find root readings.
5. Pick the best reading. If this fails, signalize that the system did not understand the input and go to 1.
6. If the system is in Command mode:

   (a) Construct targets and convert the reading to a list of instructions.

   (b) If there is an instruction that needs the user's confirmation, switch to Acknowledge mode and store the list of instructions. Send a request for confirmation to the user and go to 1.

   (c) Otherwise, execute the instructions and go to 1.

   If the system is in Acknowledge mode:

   (a) If the reading is a reply, switch to Command mode and evaluate it: if it is positive, execute the previously stored instructions and go to 1, if negative, go to 1 without doing anything.

   (b) If the reading is an action or query, switch to Command mode and go to step 6 (re-evaluate the reading in Command mode).

   (c) Otherwise, send a request for confirmation to the user and go to 1. Stay in Acknowledge mode.

The following sections describe some parts of the process in detail.

3.2.1 From plain text to a tectogrammatical tree

The automatic annotation tools have been designed and trained on data from the Prague Dependency Treebank and as such, they follow its interpretation of FGD as having four layers. Each of them serves as the input to the next one, starting with segmentation of the plain text string to words.

NLSH treats text enclosed within apostrophes as a literal string. Therefore, the tokenization of the sentence needs to be slightly changed so as to treat such portion of text as a single word.[1]

The morphological tagging is provided by a statistical tagger (see [3]). Since literal strings cannot be expected to be assigned any reasonable tag from the tagger, there is a post-processing phase, during which all literal strings are tagged as singular masculine inanimate nouns that may be of any case.

Tagged output of the morphological analysis is then passed on to analytical and then tectogrammatical analyzer (for properties of both systems, see [6] and [4], respectively). Since no grammatemes except for `sempos` are filled in the resulting trees, they are then processed with a script that assigns grammatemes using manually written rules. (For more about the grammatemes, see [7]).

3.2.2 Structural transformations

There are basically two reasons why the structure of a tectogrammatic tree returned by the automatic tools should need to be changed:

- The annotation is not errorless. In some cases, the structure of the tree is so severely malformed that in order to assign an interpretation to it, the design of rules for subsequent reading assignment would have to be cluttered with exceptions and thus hard to maintain.

- The structure can also be systematically improved in order to better suit the model for the assignment of readings.

  For instance, sentences such #21, *"Které končí na '`rpm`'?"* ("Which end with '`rpm`'?") are transformed to *"Které jsou končící na '`rpm`'?"* ("Which are ending with '`rpm`'?") – the information about a quality of the patient is moved from predicate to a new child.

There is a special transformation rule, the *null* rule, which performs no transformations at all. Its presence assures that the original sentence (no matter how malformed) is always passed on to the reading assignment phase. This assures that no possible interpretation is ever lost.

In the current implementation, there are 9 rules for structural transformations.

[1]Note that by observing this convention, NLSH needs not to concern itself with recognition of named entities – this task is performed by the user.

3.2.3 Picking the best reading

If there were no suitable root readings, the user is signalled that her input was not understood and is asked to choose a different wording for the request.

If there was at least one root reading, the one with the highest greediness (see 3.1.3) is selected. If there were more than one such readings, the sentence is deemed ambiguous and the user is asked to reformulate the command.

3.2.4 Conversion to instructions

In this phase, we instantiate and construct targets and check all constraints of the selected root operation. For instance, when moving files to a certain target, that target must denote exactly one object.

When a generator refuses to provide any objects in the process of construction (see 2.2.2), NLSH tries to mend the problem by adding an elementary restriction $(\text{location}, = \text{CWD})$, where CWD denotes the current working directory of the system, to the constructed clause. This problem could be probably solved better, such as by allowing the dialogue memory to store locations that have been under discussion.

3.3 Possible improvements

An obvious way to improve the system's accuracy, and to simplify the set of rules for the assignment of readings, would be to improve the accuracy of at least one of the automatic tools. With more reliable and richer tectogrammatical structures at its disposal, NLSH could even use some of its tectogrammatical properties such as coreferences. As noted in 3.1.2, in the current implementation, each target stored in the dialogue context can be referenced to only once in each sentence and cannot be referenced from within it at all. The availability of coreferences would allow the implementation of "local" references, which would allow, or at least ease, the interpretation of compound and complex sentences.

The dialogue memory could also be improved. Currently, the only items stored in it are targets; in order to allow for referencing other elements in the conversation, such as values, restrictions or even replies the system has sent to the user, it could be extended to store these as well. However, it is likely that such a change would require a thorough revision of the entire interpretation mechanism and the system architecture.

Chapter 4

Conclusion and future work

We have presented a file manager for UNIX-like operating systems with a user interface in Czech language. The application is capable of performing basic operations such as copying or deleting files and creating directories, and provides means for examining the system and and querying the properties of its contents.

The system employs tools for automatic annotation of Czech to convert plain text sentences to tectogrammatical dependency trees. The trees are then interpreted and converted to instructions that are executed. The querying language, which is the heart of the interpretation mechanism, is a formal language based on propositional logic with some extensions that allow it to describe basic relations between objects.

The future work should concentrate on removing the most obvious limitations of the system, which is the support of nonexistent objects, which affects the addition of new entities to the world, and the extremely simplified model of dialogue with the user.

Bibliography

[1] ANDROUTSOPOULOS, I., RITCHIE, G. D., AND THANISCH, P. Natural Language Interfaces to Databases–An Introduction. *Journal of Language Engineering 1* (1995), 29–81.

[2] BOJAR, O., BROM, C., HLADÍK, M., AND TOMAN, V. The Project ENTs: Towards Modelling Human-like Artificial Agents. In *SOFSEM 2005 Communications* (Jan. 2005), P. Vojtáš, M. Bieliková, B. Charron-Bost, and O. Sýkora, Eds., Society for Computer Science, pp. 111–122.

[3] HAJIČ, J. *Disambiguation of Rich Inflection (Computational Morphology of Czech)*. Karolinum, Charles University Press, Prague, Czech Republic, 2004.

[4] KLIMEŠ, V. *Analytical and Tectogrammatical Analysis of a Natural Language*. PhD thesis, Faculty of Mathematics and Physics, Charles University, Prague, 2006.

[5] LONG, B. Natural Language as an Interface Style. Available on-line at `http://www.dgp.utoronto.ca/~byron/papers/nli.html`. Retrieved on August 3, 2007.

[6] MCDONALD, R., PEREIRA, F., RIBAROV, K., AND HAJIČ, J. Non-Projective Dependency Parsing using Spanning Tree Algorithms. In *Proceedings of Human Langauge Technology Conference and Conference on Empirical Methods in Natural Language Processing (HTL/EMNLP)* (Vancouver, BC, Canada, 2005), pp. 523–530.

[7] RAZÍMOVÁ, M., AND ŽABOKRTSKÝ, Z. Annotation of Grammatemes in the Prague Dependency Treebank 2.0. In *Proceedings of the LREC Workshop on Annotation Science* (2006), pp. 12–19.

[8] SGALL, P., HAJIČOVÁ, E., AND PANEVOVÁ, J. *The Meaning of the Sentence in Its Semantic and Pragmatic Aspects*. Reidel Publishing Company, Dordrecht and Academia, Prague, 1986.

Appendix A

Test suite sentences

Testing data used for the development of NLSH consisted of 44 sentences in 7 dialogues ("scenes").

Sentences that failed to match to an operation or were interpreted incorrectly are marked with † (there were 4 such failed sentences: #13, #14, #15 and #32).

1. Kolik je souborů v mém domovském adresáři?
   How many files are there in my home directory?

2. Který z nich je nejstarší?
   Which of them is the oldest?

3. Vytvoř adresář '`archiv`'.
   Create directory '`archiv`'.

4. Přesuň do něj ty soubory.
   Move the files into it.

5. Jdi do toho adresáře.
   Go to that directory.

6. Je v něm soubor '`README`'?
   Is there file '`README`' in it?

7. Smaž ho.
   Delete it.

8. Ano.
   Yes.

9. Jdi do domovského adresáře.
Go to the home directory.

10. Je tady adresář '`devel`'?
Is directory '`devel`' here?

11. Je v adresáři '`~/mnt/usb`'?
Is it in directory '`~/mnt/usb`'?

12. Které adresáře v '`~/mnt/usb/`' začínají na '`d`'?
Which directories in '`~/mnt/usb/`' begin with '`d`'?

13. † Co je v tom '`dev`'?
What is in that '`dev`'?

14. † A v '`d`'?
And in '`d`'?

15. † Zkopíruj ten adresář ke mně domů a přejmenuj ho na '`devel`'.
Copy the directory to my home and rename it to '`devel`'.

16. Je '`devel/network.c.old`' starší než '`devel/network.c`'?
Is '`devel/network.c.old`' older than '`devel/network.c`'?

17. Tak ho smaž.
(So) delete it.

18. Ano.
Yes.

19. Jdi do '`install`'.
Go to '`install`'.

20. Kolik je tu souborů?
How many files are there here?

21. Které končí na '`rpm`'?
Which end with '`rpm`'?

22. Jakou mají dohromady velikost?
What is their total size?

23. Jak velký je poslední z nich?
How big is the last of them?

24. Tak ho smaž.
    (So) delete it.

25. Ano.
    Yes.

26. Kde jsem?
    Where am I?

27. Jsou tu nějaké soubory?
    Are there any files here?

28. Které?
    Which?

29. Smaž všechny kromě nejnovějšího.
    Delete all except the newest.

30. Ne.
    No.

31. Který ze souborů začínajících na '`crc`' je nejnovější?
    Which of the files that begin with '`crc`' is the newest?

32. † Smaž všechno kromě něj.
    Delete everything except it.

33. Ano.
    Yes.

34. Je tu nějaký soubor o velikosti 0 bytů?
    Is there any file with size 0 bytes here?

35. Který?
    Which one?

36. Který z nich je novější?
    Which of them is newer?

37. Smaž ten starší.
    Delete the older.

38. Ano.
    Yes.

39. Kolik souborů je novějších než soubor ‘`timestamp`’?
    How many files are newer than file ‘`timestamp`’?
40. Vypiš všechny soubory začínající na ‘`time`’.
    Print all files that begin with ‘`time`’.
41. Kolik je tu souborů?
    How many files are there here?
42. Existuje adresář ‘`archiv`’?
    Does directory ‘`archiv`’ exist?
43. Tak ho vytvoř.
    (So) create it.
44. Zkopíruj do něj všechny soubory z adresáře ‘`todate`’.
    Copy all files from directory ‘`todate`’ into it.