NICE project (IST-2001-35293)

Natural Interactive Communication for Edutainment

Natural language understanding for the NICE fairy-tale game

NICE Deliverable D3.5b

28 November 2004

Authors

Johan Boye, Joakim Gustafson and Mats Wirén

Voice Technologies, TeliaSonera, Sweden

Project ref. no.	IST-2001-35293
Project acronym	NICE
Deliverable status	Restricted
Contractual date of delivery	1 November 2004
Actual date of delivery	28 November 2004
Deliverable number	D3.5b
Deliverable title	Natural language understanding for the NICE fairy-tale game
Nature	Report
Status & version	
Number of pages	19
WP contributing to the deliverable	WP3
WP / Task responsible	LIMSI/TeliaSonera
Editor	
Author(s)	Johan Boye, Joakim Gustafson and Mats Wirén
EC Project Officer	Mats Ljungqvist
Keywords	Natural language understanding, parsing, semantic interpretation
Abstract (for dissemination)	

Table of Contents

1 Introduction

1.1 Definitions and scope

This deliverable describes the natural-language understanding component for the NICE fairy-tale game. For the overall scenario of the game, see deliverable D1.2b, Section 2.

By "natural-language understanding" we mean the interpretation of user utterances, one by one, in the course of a dialogue with an embodied character of the system. Since the dialogue involves references to characters, objects and relations in a 3D world, each utterance must ultimately be interpreted both in its relevant dialogue and fairy-tale world context. Hence, natural-language understanding is a mapping from an input utterance to a context-dependent representation.

More specifically, each input utterance is available in the form of an *N*-best sentence hypothesis provided by the speech recognizer. We divide the process of natural-language understanding into *parsing*, which we take to be a mapping of the utterance to an intermediary, context-independent semantic representation, and *contextual interpretation*, in which references pertaining to the dialogue context are resolved. The latter phase involves trying to resolve anaphoric expressions such as pronouns and elliptic utterances in a specific dialogue context. It requires an algorithm for maintaining and updating the set of most salient objects that can be the targets for anaphora resolution. Such an algorithm is often referred to as a *focus management* algorithm.

This deliverable describes the semantic representation formalism, the parsing algorithm, and the principles for contextual interpretation. However, since focus management is tightly integrated with dialogue management, we have chosen to present our method for focus management in deliverable 5.2b (the deliverable on dialogue management in the NICE fairy-tale game). In addition, contextual interpretation takes into account the result of input fusion of linguistic and graphical input. This aspect is described in deliverable D3.6 (the multimodal input understanding module).

Before proceeding, we should note that the terminology used here is not altogether compatible with Figure 1.1.2 in Deliverable D3.3, showing the overall NICE system architecture: What is there referred to as "natural-language understanding" corresponds to our notion of parsing above. Consequently, our notion of contextual interpretation is part of the dialogue manager's tasks as depicted in Figure 1.1.2.

1.2 Robustness

The input to the parsing algorithm is the output from the recognizer. Because the dialogue scenario in the NICE fairy-tale game encourages the user to speak spontaneously, the input to the parser is noisy, i.e. it contains erroneous words due to disfluent speech or leakage in the recognizer's langague model. An important consideration in the design of the parsing algorithm was that it be *robust*, i.e. it should capitalize on whatever structure is present in the input, and degrade gracefully on noisy input.

1.3 Structure

The rest of this deliverable is organized as follows: The semantic formalism to which utterances are mapped is described in Section 2, parsing in Section 3, and multimodal parsing in Section 4.

2 Semantic representation formalism

2.1 Requirements and basic approach

To begin with, we are interested in defining a language L for representing the semantic–pragmatic contents of utterances. As outlined above, we assume that user utterances are analysed in two steps:

1. ***Parsing***: The surface structure of an utterance is mapped to an expression $a \in L$ which represents its meaning independently of the (discourse and world) context.
2. ***Contextual interpretation***: a is mapped to another expression $b \in L$ which is a function of a and the *context* C (which in turn is a function of the incoming event-stream S).

It is convenient to define L to be able to represent both the context-independent meaning of an utterance (the output of step 1 above) and the result of contextual interpretation of the same utterance (the output of step 2). In this way, contextual interpretation can be defined as rewriting rules operating on expressions in L.

There are several requirements that we would like to put on L:

1. ***Adequately expressive with respect to the task***: If u_1 and u_2 are different user utterances, and there exists at least one context C in which u_1 and u_2 should result in different responses from the system, then u_1 and u_2 should be parsed to different expressions in L. Otherwise, they should be parsed to the same expression. The reason for this is that we want to avoid an overly expressive semantics in order to facilitate robust parsing.
2. ***Structured***: Expressions in L should be tree-structures so that expressions can be nested and subexpressions identified within bigger expressions. For instance, one dialogue participant might accept a request from the other participant; then this nesting (of a *request* speech act within a *accept* speech act) should be reflected in the structure of the expression representing the utterance.
3. ***Partially ordered according to information content***: The result of the parsing procedure should be able to degrade gracefully on noisy input. That is, L must not be an "all-or-nothing" language. In particular, L must be able to represent also incomplete, fragmentary utterances by allowing for underspecification of bits of the semantic analysis. To this end, we assume an "information"-ordering < defined on all expressions of L such that $a < b$ iff a contains less information than b.
4. ***Minimal commitments with respect to parsing approach***: Basically, we would like L to be designed in such a way that it allows for fast and robust parsing of input utterances. However, we want to be able to experiment with different approaches to robust parsing, and we also do not want to exclude the use of domain-independent, general-purpose techniques. We would thus like L to make minimal commitments with respect to the design of the parser.
5. ***Suitable also as a representation of system utterances***: When trying to resolve anaphoric expressions such as pronouns and ellipses, it is an advantage if all utterances of the

evolving dialogue context (those of the user as well as those of the embodied characters) have a common representation. Hence, we would like L to be able to represent system utterances as well. Taking this one step further, we would also like L to be able to serve as a source representation for surface text generation.

To systematically assemble the meaning of an utterance from the meanings of its parts, and to do so while taking the context into account, we will use lambda calculus as "glue language". Basically, the usefulness of the lambda calculus stems from the fact that it allows us to treat constituents as either functions or arguments in building up a meaning representation. As we shall see later on, it is also compatible with all of the requirements above. In particular, it is highly useful from the point of view of underspecification of elliptic, contextual and other information, and thus fits well both with handling of utterances in continuous dialogue and with robust parsing of potentially noisy input. For an additional argument for favouring an approach based on lambda calculus instead of unification-based grammar, see Blackburn and Bos (2003).

2.2 User utterances

Another set of requirements arises from the repertoire of user utterances which the system must be able to interpret and distinguish from one another. They can be classified as follows:

- **Instructions**: "Go to the drawbridge", "Pick up the sword", etc.
- **Domain questions**: "What is that red object", "Where is the sword", "How old are you", etc.
- **Giving information**: "I'm fourteen years old", etc.
- **Stating intentions**: "I will give you the ruby", etc.
- **Confirmations**: "Yes please", "Ok, do that", etc.
- **Disconfirmations**: "No", "Stop!", "I didn't say that", etc.
- **Problem reports and requests for help**: "Help", "What can I do?", "I don't understand", "What should we do now?", "Do you hear me", etc.
- **Requests for explanation**: "Why did you say that?", "Why are you doing this", etc.

2.3 Dialogue acts

Utterances will be represented by tree-strucured expressions, called *dialogue acts*. As an example, the dialogue act representing the user saying to Cloddy Hans; "Pick up the axe":

request(user, cloddy, pickUp(cloddy, axe))

Here, the topmost symbol (request) indicates the type of dialogue act, the first argument (user) indicates the character issuing the dialogue act, whereas the second argument (cloddy) indicates the intended recipient of the dialogue act. These components are present for all types of dialogue acts. The third component (pickUp(cloddy, axe), in this case) indicates the propositional contents of

the dialogue act, in this case the action of picking up the axe. The general form of a request takes the form:

$$\text{request}(\ x^{\text{character}},\ y^{\text{character}},\ z^{\text{action}}\ )^{\text{dialogueAct}}$$

where the superscripts indicate type constraints on the subexpressions. The pickUp action can be further decomposed into

$$\text{pickUp}(\ x^{\text{character}},\ y^{\text{thing}}\ )^{\text{action}}$$

i.e. the first argument must be a character (who is doing the picking up), and the second argument is a thing (which is picked up).

Anaphoric utterances are represented by means of typed lambda abstractions. For instance, consider the utterance "Pick it up". The meaning of this utterance is obviously depending on the context in which it is said (i.e. what “it” is referring to). Therefore it is reasonable to assert that the meaning of the utterance “Cloddy Hans, put it down on the table” is a function, mapping the relevant part of the dialogue context to an expression of the type dialogueAct. Thus:

$$\lambda y^{\text{thing}}.\text{request}(\ \text{user},\ \text{cloddy},\ \text{pickUp}(\ \text{cloddy},\ y\ ))$$

This expression denotes a function taking a thing as argument returning a character as the result. (its type is written thing→dialogueAct). Functions of several arguments are represented with nested lambda abstractions, e.g. “Put it down” is

$$\lambda x^{\text{thing}}\ \lambda y^{\text{location}}.\text{request}(\ \text{user},\ \text{cloddy},\ \text{putDown}(\ \text{cloddy},\ x,\ y\ ))$$

Domain questions are represented by means of ask expressions, e.g. "What color is the ruby?" is:

$$\lambda x^{\text{color}}.\text{ask}(\ \text{user},\ \text{cloddy},\ x\ [\text{ruby.color}=x])$$

Here the expression within square brackets indicates domain constraints imposed on the possible instantiations of x (in this case that x should be the color of the ruby).

Granting of information is represented by tell expressions, e.g. "I'm fourteen years old" is:

$$\text{tell}(\ \text{user},\ \text{cloddy},\ 14\ [\text{user.age}=14]\ )$$

The tell construction is also used for representing statements of intent, e.g. the user saying to Karen "I will give you the ruby" is

$$\text{tell}(\ \text{user},\ \text{karen},\ \text{intend}(\ \text{user},\ \text{giveTo}(\ \text{user},\ \text{ruby},\ \text{karen}\ )))$$

Confirmations and disconfirmations are represented by confirm and disconfirm expressions, respectively, e.g. "Yes, do that" is:

$$\lambda x^{dialogueAct}.confirm(\ user,\ cloddy,\ x\ )$$

Requests for help and explanations are represented by askForSuggestion and askForExplanation expressions, e.g. "What should we do now?" is

$$\lambda x^{dialogueAct}\ .askForSuggestion(\ user,\ cloddy,\ x\ )$$

The table below summarized the types of dialogue acts to which user input will be mapped used in the fairy-tale game, and the types of their arguments. The type niceObject is a superset of all other types in the system.

Name	**Argument structure**
request	request($x^{character}$, $y^{character}$, z^{action})
ask	ask($x^{character}$, $y^{character}$, $z^{niceObject}$)
tell	tell($x^{character}$, $y^{character}$, $z^{niceObject}$)
confirm	confirm($x^{character}$, $y^{character}$, $z^{dialogueAct}$)
disconfirm	disconfirm($x^{character}$, $y^{character}$, $z^{dialogueAct}$)
askForSuggestion	askForSuggestion($x^{character}$, $y^{character}$, $z^{dialogueAct}$)
askForExplanation	askForExplanation($x^{character}$, $y^{character}$, $z^{dialogueAct}$)

Figure 1. Types of user dialogue acts

The possible actions the system can reason about is listed in the table below. The first argument is always the character performing the action; the remainder of the arguments are the other role-players of the action:

Name	**Argument structure**
goTo	goTo($x^{character}$, y^{place})
pickUp	pickUp($x^{character}$, y^{thing})
putDown	putDown($x^{character}$, y^{thing}, $z^{location}$)
giveTo	giveTo($x^{character}$, y^{thing}, $z^{character}$)
raiseDrawbridge	raiseDrawbridge($x^{character}$)
lowerDrawbridge	lowerDrawbridge($x^{character}$)

Objects of other types (character, place, thing, location, etc.) are represented by argument-free terms (e.g. cloddy, knife, atMachine).

As seen above, the semantic expressions may also include expressions that constrain the set of possible values for a variable or a set of variables, for example:

x.color = red

In general, if a is an expression of type t, and objects of type t have an attribute att of type s, and b is an expression of type s, then

a.att = b

is a well-formed constraint. See deliverable 1.2 for a listing of the attributes of various classes of objects.

2.4 Formal syntax

The semantic dialogue act expressions informally discussed in the previous section can be gievn a rigorous definition, as follows. We assume the following sets to be given:

- A set T of *type names*. Each type name is a constant that corresponds to a set of semantic or pragmatic concepts relevant to the domain. Examples are thing, character, action, location, place
- For each symbol t in T, we assume the existence of a set V_t of variables.
- A set F of semantic constructor symbols. Each symbol $f \in F$ has a *type* and a fixed number of typed arguments. We use the notation $\sigma(f) = (t_1, \ldots, t_n, t)$ to indicate that f has the type t, and n arguments with types $t_1 \ldots t_n$.
- A set A of attribute names. Each symbol $att \in A$ has a domain type and a range type. We use the notation dom(a) to indicate the domain type, and ran(a) to indicate the range type.

The semantic representation language L is now defined as the smallest set that satisfies:

1. If $v \in V_t$ then $v \in L$ and is said to be of type t.
2. Suppose $f \in F$ and $\sigma(f) = (t_1, \ldots, t_n, t)$, and suppose $a_1,\ldots,a_n$ are expressions in L of types $t_1, \ldots, t_n$ respectively. Then $f(a_1,\ldots,a_n) \in L$ and is said to be of type t.
3. Suppose $x \in L$ and is of type t, and $y \in L$ and is of type s. Suppose further that $att \in A$ and ran(att)=t and dom(att)=s. Then x[y.att=x] is in L and is of type t, and y[y.att=x] is in L and is of type s.
4. Suppose $a \in L$ and a is of type t, and suppose $x \in V_s$ for some type s. Then $\lambda x.a \in L$ and is said to be of type $s \rightarrow t$.
5. Suppose $a \in L$ and a is of type $s \rightarrow t$, and suppose $b \in L$ and b is of type s. Then $(a\ b) \in L$ and is of type t.

2.5 Normalization

We assume familiarity with the typed lambda calculus (see e.g. Hindley and Seldin 1986), and will here only repeat the basic concepts. An expression of the form

$$(\lambda x.a\ b)$$

where $\lambda x.a$ is of type $s \rightarrow t$ and b is of type s, reduces to the expression a in which all free occurrences of the variable x have been substituted for b. The resulting expression, written as $a[x{:=}b]$, is of type t. The process is called β-*reduction*, and is written

$$(\lambda x.a\ b) \rightarrow_\beta a[x{:=}b]$$

The expression $(\lambda x.a\ b)$ is called a *redex* and the expression $a[x{:=}b]$ is its *contractum*. An expression without redices is said to be in *normal form*. Sometimes several reduction steps are needed to reach a normal form, as in the example

$$((\lambda x \lambda y.\text{putdown}(\text{cloddy}, x, y)\ \text{hammer})\ \text{table}) \rightarrow_\beta$$
$$(\lambda y.\text{putdown}(\text{cloddy}, \text{hammer}, y)\ \text{table}) \rightarrow_\beta$$
$$\text{putdown}(\text{cloddy}, \text{hammer}, \text{table})$$

The inverse of β-reduction is called β-*abstraction*. Any given expression has an infinite number of different β-abstractions; however, we will only be interested in such β-abstractions that replace a subexpression with a new bound variable. For example, there are two such possible β-abstractions from the expression $\text{putdown}(\text{hammer}, \text{table})$, namely

$$\text{putdown}(\text{cloddy}, \text{hammer}, \text{table}) \rightarrow^{-1}_\beta \lambda x^{\text{location}}.\text{putdown}(\text{cloddy}, \text{hammer}, x)$$
$$\text{putdown}(\text{cloddy}, \text{hammer}, \text{table}) \rightarrow^{-1}_\beta \lambda x^{\text{thing}}.\text{putdown}(\text{cloddy}, x, \text{table})$$
$$\text{putdown}(\text{cloddy}, \text{hammer}, \text{table}) \rightarrow^{-1}_\beta \lambda x^{\text{character}}.\text{putdown}(x, \text{hammer}, \text{table})$$

A lambda expression without free variables is called a *combinator*. In the following, we will only consider expressions without free variables.

2.6 Higher-order functions and representation of ellipsis

Ellipses are are represented by means of higher-order functions. Consider the example:

1. *User:* "Cloddy Hans, please pick up the axe."
2. *Cloddy Hans:* "OK" (*picks up the axe)*
3. *User:* "Now the hammer".

In utterance 3, the user wants Cloddy Hans to do something with the hammer, but it is not possible to infer what dialogue act the user is performing without taking the dialogue context into account. Thus a context-independent representation of this utterance must represent the dialogue act by a function, as follows:

$$\lambda f^{\text{thing} \rightarrow \text{dialogue\_act}}.(f\ \text{hammer})$$

The parameter f is to be bound to a function that takes as argument the information present in the utterance (hammer), and returns the appropriate dialogue act. Constructing this function is the task of contextual interpretation, and how this is done is explained in deliverable 5.2b, chapter 4. Just to that such a function actually exists, we will stipulate that it is

$$\lambda y^{\text{thing}}.\text{request}(\ \text{user},\ \text{cloddy},\ \text{pickUp}(\ \text{cloddy},\ y\ ))$$

since

$$(\lambda f^{\text{thing} \rightarrow \text{dialogue\_act}}.(f\ \text{hammer})\ \ \lambda y^{\text{thing}}.\text{request}(\ \text{user},\ \text{cloddy},\ \text{pickUp}(\ \text{cloddy},\ y\ )))\ \ \rightarrow_\beta$$
$$(\lambda y^{\text{thing}}.\text{request}(\ \text{user},\ \text{cloddy},\ \text{pickUp}(\ \text{cloddy},\ y\ ))\ \ \text{hammer})\ \ \rightarrow_\beta$$
$$\text{request}(\ \text{user},\ \text{cloddy},\ \text{pickUp}(\ \text{cloddy},\ \text{hammer}\ ))$$

i.e. “Pick up the hammer”.

3 Parsing

The robust parsing algorithm[1] consists of two phases, a pattern matching phase and a rewriting phase. In the first phase, a string of words[2] is scanned left-to-right, and a sequence of semantic constraints, triggered by syntactic patterns, are accumulated. In the latter phase, heuristic rewrite rules are applied to the result of the first phase. When porting the parser to a new domain, one has to rewrite the pattern matcher, whereas the rewriter can remain unaltered.

3.1 Semantic constraints

The most common kind of semantic constraint simply stipulates that the existence of certain objects of certain types can be inferred from the user's utterance. Such constraints are written on the form

$$\text{object}^{\text{type}}$$

For instance, the word "hammer" would trigger the constraint

$$\text{hammer}^{\text{thing}}$$

whereas the phrase "pick up" would trigger the following conjunctive constraints:

$$\text{pickUp}(x, y)^{\text{action}},\ x^{\text{character}},\ y^{\text{thing}}$$

Disequalities are used to express that two objects (of the same type) are necessarily different. For instance, the initial phrase "What is..." indicates that the user is asking a question. Thus it results in the following list of constraints:

$$\text{ask}(\text{user}, x, y)^{\text{dialogue\_act}},\ \ \text{user}^{\text{character}},\ x^{\text{character}},\ x \neq \text{user},\ y^{t}$$

Obviously, the user is asking someone else than himself; hence the disequality $x \neq \text{user}$. As "What is..." does not give any clue to what the user is asking about, the type of the third argument is a variable t.

Equality constraints are used to relate objects with attributes of other objects. For example, the initial phrase "Where is..." indicates that the user is enquiring about the position of some object. The list of constraints triggered by the syntactic pattern "Where is..." is:

$$\text{ask}(\text{user}, x, y)^{\text{dialogue\_act}},\ \ \text{user}^{\text{character}},\ x^{\text{character}},\ x \neq \text{user},\ \ y^{\text{location}},\ \ y = z.\text{position},\ \ z^{t}$$

[1] The algorithm presented here is an extension of that presented in Boye and Wirén (2003 a, 2003 b).
[2] We assume 1-best output from the speech recogniser to be used.

Here it is possible to infer that the object asked about is a location; hence the type of y is location rather than a variable t. Furthermore, it is assumed that this location is the position of some object z, whose type we do not know (and therefore its type a variable t). However, z must be an object that has a position attribute.

3.2 Pattern-matching phase

The purpose of the pattern matching phase is to generate a list of semantic constraints on the basis of the syntactic patterns that appear in the input. Such rules are coded by means of a definite clause grammar (see e.g. Sterling and Shapiro 1994, chapter 19). An example of such rules are:[3]

```
pickUp_hints( [ pickUp(X, Y)action, Xcharacter, Ycharacter | MoreHints ], Tail ) -->
        [ take, the ],
        thing_hints( [ Ycharacter | MoreHints ], Tail ).

pickUp_hints( [ pickUp(X, Y)action, Xcharacter, Ycharacter | Tail ], Tail ) -->
        [ take ].

thing_hints( [ hammerthing | Tail ], Tail ) -->
        [ hammer ].

thing_hints( [ swordthing | Tail ], Tail ) -->
        [ sword ].
```

The algorithm is simply to try to match an initial segment of the input with the right hand side of such a rule. The rules are tried in the order they are written. If a match is possible, the semantic constraints on the left hand side are appended to the result list, the matched input segment is discarded, and the process is repeated with the remaining input. If a match is not possible, the first word of the input is discarded, and the process is repeated with the remaining input.

For instance, suppose the input is "take the ehh hammer". The first rule is not applicable in this case because of the inserted "ehh", but the second rule is applicable, since the input begins with "take". The following two words ("the" and "ehh") are discarded as they do not match any rule. Finally, the last word "hammer" matches the third rule. The accumulated sematic constraints are:

$$\text{pickUp}(x, y),\ x^{\text{character}},\ y^{\text{thing}},\ \text{hammer}^{\text{thing}}$$

In case the input is "take the hammer", without the inserted hesitiation "ehh", the first rule matches the whole input string. In this case, the variable Y is set to hammer, and the output is:

$$\text{pickUp}(x, \text{hammer}),\ x^{\text{character}},\ \text{hammer}^{\text{thing}}$$

[3] For these rules, we adopt the standard logic programming convention that expressions with an initial capital letter are variables.

As can be seen from these examples, longer syntactic patterns are likely to convey more precise semantic information, but on the other hand they are more brittle, as the probability increases that recognition errors and disfluencies like "ehh" prevent matching. Moreover, longer patterns are less likely to occur in the input anyway. Therefore rules should be ordered as in the example, with longer patterns appearing before shorter patterns, so that the parser can capitalize on structure whenever present in the input, and degrade gracefully on noisy input.

In the example above, the presence of the filler word "ehh" made the parser miss the link between the hammer and the second argument of pickUp. However, this link will be recovered in the second phase of the parsing algorithm, presented next.

3.3 Rewriting phase

In the rewriting phase, the list of constraints aggregated in the pattern-matching phase is rewritten using four rewrite rules: *object merging, constraint inference, filtering* and *abstraction.*

3.3.1 Object merging

The first rewriting step, object merging, amounts to unifying objects of the same type. The rewriting rule can be formulated generally as follows:

Starting from the left, terms are unified with their nearest unifiable neighbour to the right.

Here “unifiable” means that the ensuing list of semantic constraints (after unification) must be compatible. For instance, in a list containing the three constraints

$$x^{\text{character}},\ y^{t},\ y.\text{nextTo}=z$$

x and y are *not* unifiable, even though the type of y is a variable, since a character does not have a nextTo attribute. However, in the example of the previous section:

$$\text{pickUp}(x, y),\ x^{\text{character}},\ y^{\text{thing}},\ \text{hammer}^{\text{thing}}$$

y and hammer can be unified, resulting in

$$\text{pickUp}(x, \text{hammer}),\ x^{\text{character}},\ \text{hammer}^{\text{thing}}$$

The object merging process can be controlled by properly ordering the constraints in pattern matching rules, and by the use of disequality (≠) constraints. This was demonstrated previously in the example:

$$\text{ask}(\text{user}, x, y)^{\text{dialogue\_act}},\ \ \text{user}^{\text{character}},\ x^{\text{character}},\ x\neq\text{user},\ y^{t}$$

where the disequality constraint x≠user prevents unification of x and user.

3.3.2 Constraint inference

Consider the utterance "Go to the hammer", giving the following list of constraints:

$$goTo(x, y)^{action},\ x^{character},\ y^{place},\ hammer^{thing}$$

At first, it seems as uncomplicated a sentence as "Take the hammer", discussed previously. But "Go to the hammer" actually poses bigger natural language understanding problems, because the domain encoding is strictly typed so that characters cannot go to things, only to places. Essentially the system must reason as follows:

> The user wants me to go to some place x.
> The hammer is at location y.
> So x should be the place which is next to y.

This kind of reasoning is embodied in the the following graph algorithm. First create a list of sets where every expression is put in a set of its own:

$$\{ goTo(x, y)^{action} \},\ \{ x^{character} \}, \{ y^{place} \},\ \{ hammer^{thing} \}$$

Then sets are merged according to the following rule:

> ***Set merging rule:*** Two sets X and Y should be merged if X contains an expression x which is a subexpression of some expression $y \in Y$.

This leaves the following list of sets:

$$\{ goTo(x, y)^{action},\ x^{character}, y^{place} \},\ \{ hammer^{thing} \}$$

If there is only one remaining set at this stage, the algorithm halts. If there is more than one set, We choose the smallest set and apply the following rule:

> ***Constraint adding rule:*** Given a set X, choose an object x and one of its attributes att, and add to X the expressions $x.att = y$ and y^t (where att 's values are of type t).

If the object denoted by this expression has an attribute att, we introduce the value of att as a new expression. In the example, objects of class thing have an attribute position, whose value is of type location. This gives us:

$$\{ hammer^{thing},\ hammer.position = l,\ l^{location} \}$$

This set cannot still be merged with the other set in the list, so we choose the same set again and re-apply the constraint adding rule. Objects of type location have an attribute nextTo whose value is of type place. Adding this link gives us:

$$\{\ \text{hammer}^{\text{thing}},\ \text{hammer.position=l},\ \text{l}^{\text{location}},\ \text{l.nextTo=p},\ \text{p}^{\text{place}}\ \}$$

Now the full list of constraints, after applying object merging (section 3.3.1), is:

$$\text{goTo(x, y)}^{\text{action}},\ \ \text{x}^{\text{character}},\ \text{y}^{\text{place}},\ \text{hammer}^{\text{thing}},\ \text{hammer.position=l},\ \text{l}^{\text{location}},\ \text{l.nextTo=y}$$

The set merging rule would place all these expressions in the same set, and therefore the algorithm terminates, returning the list of constraints above as the result. There is now a link from the second argument of goTo to the hammer; "the place y which is next to the location l where the hammer is".

A depth-first version of the algorithm can be concisely formulated as follows:

```
Given a list L of constraints:
while ( true ) {
        perform object merging (section 3.3.1);
        put each constraint in L in a set of its own, producing a list L' of sets;
        apply the set merging rule to L', producing L″;
        if L″ contains a single set,
                return this set as the result;
        else {
                choose a set in L″, and an expression in this set, and apply the
                        object adding rule;
                Let L be the list of all the constraints in all the sets in L″ ;
        }
}
```

The actual implementation is breadth-first rather than depth-first, in order to find the shortest path connecting all constraints. Moreover, the algorithm only proceeds to a certain depth, to prevent looping.

3.3.3 Filtering

The next step is to filter the list of semantic constraints by removing all *implied* constraints. A constraint c in the list L is implied if

- c is a variable-free expression of the form a=b or a≠b.
- c is an non-variable expression of the form a^t, appearing as a subexpression of some other constraint b^s in L, or

In the first case, trivially true facts like axe=axe or axe≠hammer are removed. In the second case, the existence of the object a^t is implied by the existence of the object b^s. So for instance, in the list

$$\text{request(user, cloddy, goTo(cloddy, y))}^{\text{dialogueAct}},\ \text{goTo(cloddy, y)}^{\text{action}},\ \text{cloddy}^{\text{character}},\ \text{user}^{\text{character}},\ \text{y}^{\text{place}}$$

the three constraints $\text{cloddy}^{\text{character}}$, $\text{user}^{\text{character}}$ and $\text{goTo(cloddy, y)}^{\text{action}}$ are implied by the constraint $\text{request(user, cloddy, goTo(cloddy, y))}^{\text{dialogueAct}}$, and are therefore removed. However, the expression y^{place}, being a variable, is kept. This results in:

$$\text{request(user, cloddy, goTo(cloddy, y))}^{\text{dialogueAct}},\ \text{y}^{\text{place}}$$

3.3.4 Abstraction

The point of the abstraction step is to transform the list of semantic constraints into a combinator by binding all free variables. When the dialogue act is known, this is straightforward. So is, for instance, the list of constraints above transformed to the following combinator by abstraction on y:

$$\lambda y^{\text{thing}}.\ \text{request(user, cloddy, goTo(cloddy, y))}$$

This expression, of type thing→dialogueAct, is returned as the final answer of the parsing process.

A slightly more complex situation arises if the dialogue act is not known (i.e. there is no constraint of type dialogueAct in the list of constraints). Consider, for instance, the elliptical utterance “the hammer”, leading to the singleton list

$$\text{hammer}^{\text{thing}}$$

Here, a new function symbol $\text{f}^{\text{thing} \rightarrow \text{dialogue\_act}}$ has to be introduced, as explained in section 2.6. The final result is:

$$\lambda \text{f}^{\text{thing} \rightarrow \text{dialogueAct}}.(\text{f hammer})$$

If the list of semantic constraints contains several expressions, the same process is used, only that the introduced function st with several objects process is repeated. So is, for instance, the list

$$\text{hammer}^{\text{thing}},\ \text{axe}^{\text{thing}}$$

represented as

$$\lambda \text{f}^{\text{thing}\rightarrow(\text{thing} \rightarrow \text{dialogueAct})}.((\text{f hammer}) \text{ axe})$$

That is, f should be boud to a function which is applied to hammer, returning a function which is applied to axe, returning a dialogue act.

3.4 Domain-dependent rewriting phase

We started section 3 by claiming that the rewriting phase is domain-independent, and thus does not need modification when moving to a new domain. Nevertheless, it can be very useful also to be able to define domain-dependent rewriting rules used for resolving those types of underspecifications that are always resolved in the same way in the domain.

Such heuristic rewrite rules are expressed as combinators a^s. If the resulting expression b from the previous rewriting process is of type $s \rightarrow t$, then b will be applied to a. As an example, consider the utterance:

"Ehh... put down ehh... let's see the pencil"

The parsing algorithm just presented yields the following result:

$$\lambda f^{action \rightarrow dialogue\_act}\ \lambda x^{character}\ \lambda z^{location}.(f\ putDown(x, pencil, z))$$

This expression adequately represents all underspecifications in the utterance: Someone should put down a pencil somewhere, and the user is saying something about it. However, there are several reasonable assumptions we can make in order to simplify this expression, namely:

1. The user is making a request to Cloddy Hans
2. Cloddy Hans is the one who should put down the pencil

The point here is that these assumptions are made without considering the dialogue context. This can be done, since at least in the first scene of the game (see deliverable 1.2b, section 2), Cloddy Hans is the only character present, and the scenario is all about the user instructing him where to put various things. So the two heuristics (1) and (2) above are domain-specific rather than dialogue-context-specific.

The first heuristic, that an utterance about an action is a request to perform that action, can itself be expressed by a combinator:

$$\lambda x^{action}.request(user, cloddy, x)^{dialogue\_act}$$

Applying our expression to this heuristic combinator yields:

$$(\lambda f^{action \rightarrow dialogue\_act}\ \lambda x^{character}\ \lambda z^{location}.(f\ putDown(x, pencil, z))\quad \lambda x^{action}.request(user, cloddy, x)) \rightarrow_\beta$$
$$\lambda x^{character}\ \lambda z^{location}.\ (\lambda x^{action}.request(user, cloddy, x))\ \ putDown(x, pencil, z)) \rightarrow_\beta$$
$$\lambda x^{character}\ \lambda z^{location}.request(user, cloddy, putDown(x, pencil, z))$$

The second heuristic, that the user is talking to Cloddy Hans, can be expressed simply as the following combinator:

$$cloddy^{character}$$

Applying our expression to the cloddy combinator yields:

$(\lambda x^{character}\ \lambda z^{location}.\text{request}(\text{user}, \text{cloddy}, \text{putDown}(x, \text{pencil}, z))\ \text{cloddy}) \rightarrow_{\beta}$
$\lambda z^{location}.\text{request}(\text{user}, \text{cloddy}, \text{putDown}(\text{cloddy}, \text{pencil}, z))$

The final expression is taken to be the (context-independent) interpretation of the user's utterance. The last parameter z might be bound as a result of context-dependent processing (see deliverable 5.2b, section 4).

4 Parsing multimodal input

The NICE fairy-tale system is a multimodal system. In particular, the user can both speak and gesture at the screen, either simultaneously or sequentially. To this end, the system contains modules for capturing and interpreting gestures as well as performing input fusion, i.e. constructing a combined interpretation of the user's gestures and spoken input (see further deliverable 3.6).

The parsing algorithm just presented can actually parse some multimodal input directly. If the gestural input amounts to simple references to objects, such references can be turned into semantic constraints and parsed along with the verbal input. As an example, consider the multimodal utterance:

<*user clicks on the hammer*> "Pick this up"

The click on the hammer can be translated into the semantic constraint $\mathsf{hammer}^{\mathsf{thing}}$. Together with the semantic constraints generated from analysing "Pick it up", we have

$$\mathsf{hammer}^{\mathsf{thing}},\ \mathsf{pickUp}(x, y),\ x^{\mathsf{character}},\ y^{\mathsf{thing}}$$

that is, just as if the user had said "pick up the hammer".

This is not to say that the parser is a replacement for the input fusion module in the system. For instance, the parser disregards issues that concern the relative timing of verbal and gestural input. When handed a piece of gestural input and a piece of verbal input to analyse, it can construct a combined interpretation, but the parser relies on other modules to decide which pieces of input to analyse together. (This task is performed by the Dispatcher module, see deliverable 3.6).

5 References

Blackburn, P. and Bos, J. (2003): Computational semantics. *Theoria* 18(1): 27–45.

Boye, J., Gustafson, J. and Wirén, M. (2004a) Formal representation of domain information, personality information and dialogue behaviour for the NICE fairy-tale game. NICE deliverable D1.2b.

Boye, J., Gustafson, J. and Wirén, M. (2004b) Dialogue management and response planning for the NICE fairy-tale game. NICE deliverable D5.2b.

Boye, J. and Wirén, M. (2003 a): Robust parsing of utterances in negotiative dialogue. *Proc. Eurospeech*, Geneva, Switzerland.

Boye, J. and Wirén, M. (2003 b): Negotiative spoken-dialogue interfaces to databases. *Proc. Diabruck* (*7th workshop on the semantics and pragmatics of dialogue*), Wallerfangen, Germany.

Hindley, R. and Seldin, J. (1986): *Introduction to combinators and λ-calculus.* Cambridge University Press.

Sterling, L. and Shapiro, E. (1994) *The art of Prolog,* 2nd edition. The MIT Press.