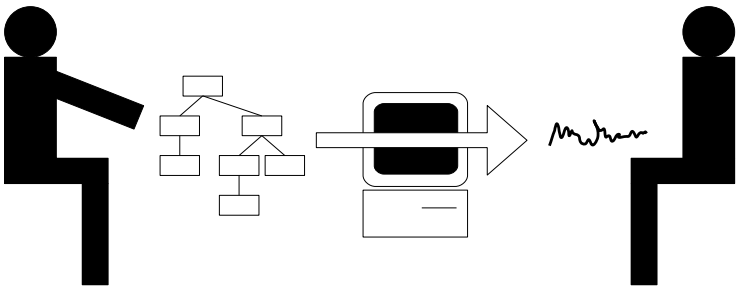
David Stark



A Natural Language Generator for Software Translation

Computer Science Tripos Part II

Girton College

May 22, 2006

Proforma

Name:	**David Stark**
College:	**Girton College**
Project Title:	**A Natural Language Generator for Software Translation**
Examination:	**Computer Science Tripos Part II**
Word Count:	**10545**
Project Originator:	David Stark
Supervisor:	Anna Ritchie

Original Aims of the Project

To produce a basic implementation of a natural language generator. The generator should be capable of generating text in at least two different languages from the same input. The range of text it can produce should be of the type used in graphical user interfaces.

Work Completed

Implemented a generator able to produce both English and German text. Additionally, implemented a graphical user interface for manipulating the input of the generator where the output is displayed in real-time.

Special Difficulties

None

Declaration

I, David Stark of Girton College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Chapter 1

Introduction

1.1 Aims

The aim of this project was to design and implement a natural language generator, capable of generating text in multiple languages from the same meaning representation. The purpose of the generator was to be a tool for making translation of software easier and more affordable. As such, the range of text it can produce (its *domain*) is limited to the kind of text found in user interfaces.

1.2 Motivation

Most software programs interact or at least communicate with the user using natural language. Even programs that are not end-user applications, like background processes, drivers, and operating system kernels need to be able to convey information to the user, and do so by displaying natural language strings.

It is (still) the case that the most visible software programs are written in English, with the roots of computing arguably located in the UK and the US. But nowadays, computers are used all over the world, by people speaking a large number of different languages. The majority of the world's population does not speak English and still more speak it only as a second language.

Thus, in order to be accessible to the non-English-speaking world, the natural language strings used by a computer program need to be translated into many different languages. This is a time-consuming and expensive process, and significantly increases the length of the software release cycle. That is, whenever a new version of a software program has been completed, its makers need to have any alterations to its strings re-translated. As a result, software updates for non-English versions of software are often significantly delayed. Also, translation is

often too expensive to contemplate for small software developers. As a result, non-English speakers have a greatly reduced choice of software, and the software that is available to them is less up-to-date.

1.3 Machine Translation

One option for making translation of software fast and easy would be to use machine translation. However, machine translation is very much an area of active research, and is arguably neither reliable nor accurate enough for this purpose. For example, I translated the following sentences found in OpenOffice.org[1] into German and then back into English, using Google's translation services[2]:

> "This is where you configure OpenOffice.org for the Internet."
>
> "This is where you select the print format and print options for all newly saved formula documents."

The resulting sentences were:

> "This is, where you assemble OpenOffice.org for the InterNet."
>
> "This is, where you preselect the pressure format and print elections for all evenly stored formula documents."

Such translation is not quite accurate enough for use in user interfaces. While the meaning of the resulting phrases can be inferred, translating software using machine translation would result in the user having to guess at the precise meaning of text in the user interface.

So why is machine translation so hard? And what can be done to make it easier for the specific case of user interfaces?

1.3.1 Ambiguity and Vocabulary Size

One of several approaches to machine translation is the following one, based on the concept of a language-independent *meaning representation*. It consists of two rough stages: First, the translation program parses the source text into a syntax tree, which it then interprets into a meaning representation. The second step reverses this process for the target language, generating text from the meaning representation.

[1] http://www.openoffice.org
[2] http://www.google.com/translate_t/

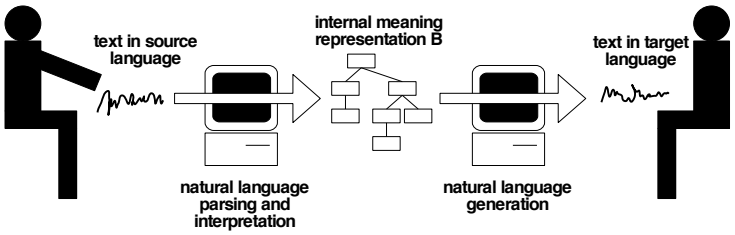


Figure 1.1: Machine Translation

Many problems with machine translation appear in the parsing and interpretation stages, and stem from the fact that natural language is very ambiguous in both syntax and semantics. Resolving these ambiguities is AI-complete: the machine needs to "understand" the meaning of text, which often requires local contextual or world knowledge combined with complex inference.

Syntactical ambiguity occurs when multiple different syntactical trees map to the same text. Hence, given a text, a natural language parser will produce multiple syntax trees varying wildly in their structure. A translation system has to somehow choose which syntax tree is the "correct" one. Failure to do so is liable to result in a great deal of misunderstanding.

The most famous example of such a misunderstanding is the phrase "Time flies like an arrow". In 1963, Susumu Kuno, a researcher at Harvard University, fed this phrase to a natural language parser, which came up with five different parse trees with the following meanings:

- Time proceeds as quickly as an arrow proceeds.
- Command: Measure the speed of flies in the same way that you measure the speed of an arrow.
- Command: Measure the speed of flies in the same way that an arrow measures the speed of flies.
- Command: Measure the speed of those flies that resemble an arrow.
- Flies of a particular kind, i.e. time-flies, are fond of an arrow, that is, they like arrows.

Apart from syntactic ambiguity, machine translation also suffers from semantic ambiguity - unclearness about the meaning of words.

Depending on the domain in which they are used, words in a language can acquire entirely different meanings. Consider, for example, the phrase "The mouse eats a cookie." Under the most common meaning of the words "mouse" and "cookie", this phrase makes perfect sense - a small rodent is consuming a food item. However, in the domain of computer science, this phrase becomes deeply nonsensical: A mouse is a type of peripheral, and the word "cookie" most commonly refers to a "HTTP cookie", a piece of text stored by a web browser for user tracking and authentication purposes.

Without knowing the domain of the source text, a machine translation program is forced to guess as to the meaning of words with different domain-specific meanings. As a result, open-domain machine translators tend to perform much worse than domain-specific ones.

In addition to word sense disambiguation, machine translation also has to deal with inter-language word ambiguity. When translating a word to another language, that language often offers several different words to choose from. This is because the target language makes distinctions the source language does not make, and conversely lumps together meanings that are separate in the source language. For example, when translating the English word "wall" into German, one encounters the problem that German makes a distinction between a wall as seen from the inside ("Wand"), and from the outside ("Mauer"). A machine translator would have to make a choice from those two options, and the correct choice depends on the wider context of the text being translated: Is the point of view inside or outside a building?

As with the above problem, a machine translation system would have to be able to analyse the meaning of the text being translated, and apply real-world knowledge to make correct word choices. Doing this arguably requires artificial intelligence beyond what is currently available.

A further issue is that of vocabulary size. A working vocabulary in English is estimated to contain about 50 000 words. In addition to this, there are many more obscure and specialised terms. What is more, new words are constantly being invented. As a result of this, implementing a robust, open-domain machine translation system involves teaching it many tens of thousands of words. What is more, these words then have to be correctly mapped onto another equally large set of target language words.

Note that the problems mentioned above do not constitute a comprehensive overview of the challenges inherent in machine translation, but are merely intended to illustrate some of the particular difficulties that must be overcome in

the domain of translating user interfaces[3]. In order to circumvent these difficulties for this project, I adapted the approach described in the next section.

1.4 User-Encoded Meaning Representations

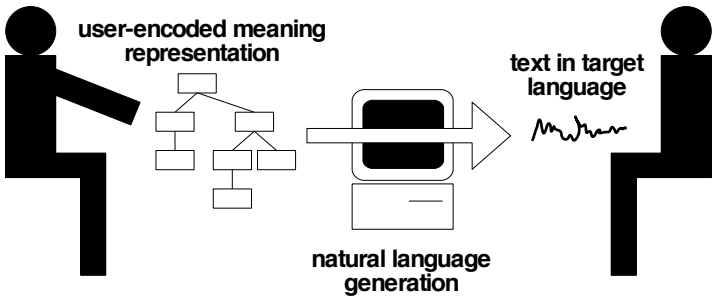


Figure 1.2: Natural Language Generation

Many of the problems for full machine translation appear in the parsing and interpretation stages, when trying to determine the meaning of natural language. By instead starting directly from the meaning of the language, these stages and their inherent problems are avoided. The user input becomes a meaning representation, rather than natural language, and the task becomes *generation*, rather than machine translation. The aim of this change is to offload the work of disambiguation onto the human user who is much better suited for this task. The advantages of using a meaning representation as input are as follows:

- Since the meaning representation can be more structured than a text string, syntactic ambiguity can be effectively eliminated.
- The domain of the translation problem is restricted to text found in user interfaces. This reduces the number of different meanings that have to be dealt with. In the example case of "mouse", it is now clear that "mouse" refers to a peripheral, and not a rodent.
- A meaning representation can be composed out of atomic meanings chosen to minimise ambiguity issues. Words that have ambiguous translations can be split into multiple, more fine-grained meanings. In the case of "wall", the word could be split into two meanings:

[3]See[2] for a more complete and detailed overview of the challenges of machine translation.

"wall (seen from outside)"

"wall (seen from inside the building)"

- By restricting the domain, the sheer size of the required vocabulary is also reduced, as many words not liable to turn up in user interfaces can simply be left out.

- Since the input of the generator is now defined by the user instead of being interpreted (and potentially misinterpreted) by the computer, the output should more closely match what the user intends to express.

1.5 Comparison of Approaches

1.5.1 Advantages of this Approach

The prime advantage of this approach is that it allows users who do not at all speak a given language to express themselves in that language. The main obstacle is that users have to learn how to encode the meaning representations for what they wish to express. However, if learning how to encode the meaning representation is easier than learning a natural language, let alone as many languages as the generator can produce, using the generator becomes an attractive proposition. Encoding meaning representations can also be made easier by giving the user feedback. This can be done by expressing the meaning representation the user is working on in a language the user does speak.

Storing the text for user interfaces in meaning representations instead of natural language also has the advantage of making it easier to change the text. If human translators are used, even the most minor changes have to be sent off for re-translation. Using a meaning representation, any changes are instant across all languages the generator can produce.

1.5.2 Quality of Translation

A natural language text tends to contain more meaning than what is presented at face value. Understanding the meaning of text fully means seeing it in the social and cultural context it is found in. Depending on context, the same phrase can be considered to be praise, or considered an insult. The reader has to understand the motivations of the author, and catch the emotional undertones implicit in the text.

The generator outlined above would not understand any of these finer points unless explicitly told. But thankfully, user interfaces are as a rule not concerned with emotional undertones or cultural circumstances.

Ideally, translation should also take into account the elegance of the translated text, but because elegance is not something easily defined algorithmically, the generator cannot aim to produce particularly elegant output, only accurate output. This also is not much of a problem since accuracy is greatly more important than elegance in the case of user interfaces.

These limitations do mean that the translation produced with the help of the generator will be inferior to the translation a competent human translator could make. However, such a human translator is not the yardstick with which we wish to measure the success of the translation: What is important is not whether the translation is perfect, but whether it is good enough for its intended purpose - communicating factual information to users.

Chapter 2

Preparation

2.1 Requirements

The aim of this project was to develop a natural language generation system suitable for translating user interfaces. To achieve this aim, the base requirements were as follows:

- The system should take as an input a meaning representation and generate appropriate natural language output.
- The system should be able to generate output in at least two natural languages.

Additional Requirements:

- The format of the meaning representation should be such that the information contained be as precise as possible. Similar information should be represented in similar ways, and as few exceptions to the structure should be made as possible without reducing the range of expression (domain) of the meaning representation.
- The generator need not be able to generate any specific natural language text, but be able to express any information encoded in a valid meaning representation.
- It should be easy to extend the range of meanings the generator is able to process, and to add new target languages for it to generate.
- The generator is not required to have a domain large enough to make it usable for real-world translation purposes.

To this end, the main components of the system to be created were as follows:

- Meaning Representation Specification
  The meaning representation is composed of individual meanings in relation to one another. The meaning representation specification describes the kind of relations possible between meanings.

- Vocabulary of Meanings
  A set of meanings out of which to compose meaning representations.

- Transformation Rules
  Both output languages required a set of rules defining how to transform a meaning representation into a natural language text expressing the intended meaning.

- Generator
  A computer program able to apply a set of transformation rules onto a meaning representation, returning the resulting natural language text.

2.1.1 Possible Extension: A Graphical User Interface

I was aware that having to manually type in the meaning representations would make the system difficult to evaluate, as evaluation subjects would first have to learn how to write valid meaning representations.

A graphical user interface would serve to abstract away from the syntax of the meaning representation, with users being able to manipulate a graphical representation of the meaning representation. This would eliminate syntax errors. Semantic errors could be eliminated by only allowing the user to combine meanings in valid ways. Furthermore, the user could be given feedback by continually running the meaning representation through the generator, and so displaying the natural language output of the user's input thus far.

Therefore, I set down the implementation of a GUI as a possible project extension, with the following base requirements:

- The graphical user interface should allow the user to manipulate meaning representations through a graphical representation.

- It should not be possible for the user to produce syntactically or semantically malformed input using the graphical user interface.

Additional requirements:

- If possible, the GUI should use the generator to display the natural language output of partially completed user's input in real-time.

2.2 Choice of Programming Language

I chose to implement the generator in Java. The project had no special requirements in terms of programming language structure or required libraries, so I defaulted upon the general-purpose language I had the most experience with. This also allowed me to easily implement a graphical user interface for the generator using AWT/Swing.

2.3 Choice of Output Languages

I chose English and German as the two languages to write generation rules for, since I speak both at a native level. I would ideally have liked to use two less similar languages, as the close relation of the two languages likely meant that the meaning representation was overly influenced by their shared idiosyncrasies. If the pairing had instead been English and Chinese, or two equally dissimilar languages, this would have enforced the design of a more abstract meaning representation, in order to deal with the differences between the two languages. Unfortunately, I have only basic knowledge of any non-Indo-European language.

Even with this local bias, the generator is still useful as a translation aid, as it should serve adequately to generate such closely related languages as English, German, French, Italian, Spanish, Portuguese, etc. Also, a more abstract meaning representation could easily be substituted by re-writing the meaning representation data file. Thanks to the modular design of the generation system, there would be no need to alter the generator itself.

2.4 Collecting a User Interface Text Corpus

2.4.1 Motivation

I needed to know what kind of text appears in user interfaces, so I compiled a corpus of such text. Its purpose was twofold:

- It would serve as a reference for designing the meaning representation and provide me with a vocabulary to implement.

- It would allow me to test the generator during system development, since I could test if the generator was able to generate the text found in the corpus.

2.4.2 Corpus Contents

The corpus consists of 67 items harvested from the user interfaces of three popular software applications: OpenOffice.org, Apple's iTunes, and Mozilla Firefox. I chose those three because they cover a range of commonly used application types, and because it was easy to obtain both an English and a German version of each program. This allowed me to make the corpus bilingual, i.e. each item in the corpus is available both in English and in its German translation. An excerpt from the corpus follows:

1. "<website> could not be found."
   "<website> konnte nicht gefunden werden."

2. "With Shopping Cart selected, music is added to your cart when you click an "add" button."
   "Wenn Sie den Einkaufswagen auswählen, wird die Musik zu Ihrem Einkaufswagen hinzugefügt, sobald Sie in "Hinzufügen" klicken."

3. "Text Document"
   "Textdokument"

4. "Show Download Manager window when a download begins."
   "Den Download-Manager anzeigen, wenn ein Download startet."

5. "Do you want to save your changes?"
   "Sollen die Änderungen gespeichert werden?"

The corpus contained 16 statements, 17 noun phrases, 32 command labels, one command and one adjective.

2.4.3 Commands versus Command Labels

One important distinction I had to draw was between commands and command labels. The former are commands from the computer to the user, such as "Please check the name and try again.", whereas command labels are potential commands from the user to the computer, as you would find in menus or on buttons, such as "Increase Indent". The distinction is necessary because, while in English, both are written in same manner, in German this is not the case: only commands from the computer to the user are in the imperative, while command labels are in the infinitive.

For example:

- Command
  English: “Close the file.”
  German: “Schliessen Sie das Dokument.”
  Word-by-word English translation: “Close you the file.”

- Command Label
  English: “Close the file”
  German: “Das Dokument schliessen”
  Word-by-word English translation: “The file close”

2.4.4 Unsuitable Corpus Items

I felt it necessary to ignore four items in the corpus due to problems with encoding them in a coherent meaning representation. This is one of them:

> “For example, you decide which contents should be displayed on the screen or printed, how the pages are scrolled on the screen, in which color keywords are highlighted in the source text and much more.”

I thought it wiser to concentrate on getting robust behaviour with simpler sentences than to invest great effort into making an overly complicated minority of the corpus work.

2.5 Backups

To protect myself against data loss, I wrote a shell script that made a scheduled daily backup of the entire project to the Pelican backup system, and also made a backup onto another server whenever a major project milestone had been reached.

Chapter 3

Implementation

3.1 The Meaning Representation

The first decision I took was to structure the meaning representation as a tree. Since the syntax and semantics of natural language is hierarchical in nature, tree structures lend themselves to representing natural language. This meant that the meanings most broadly defining the overarching meaning of a meaning representation would be found at the top of the tree, with the details of those meanings attached as child nodes.

The structure of a meaning-node object is described in Figure 3.1. What kind of meaning a node object represents is defined by its *type*: a reference to a *type object* which defines a specific meaning in the vocabulary. Using object orientation terms: a node object is an instance of a type object.

The node's children are stored in a list of tuples. One part of the tuple is a reference to the child itself, while the second references the *field* the child is in.

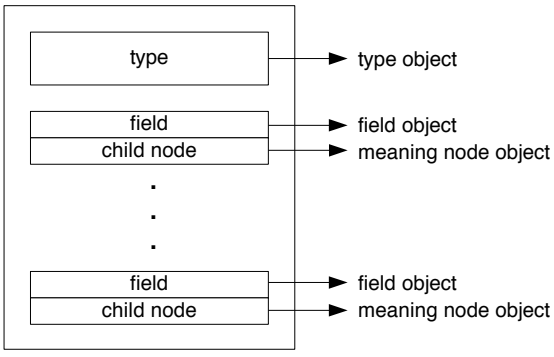


Figure 3.1: Structure of a Node Object

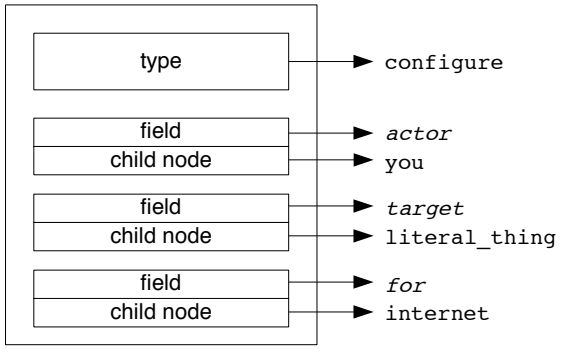


Figure 3.2: Node of Type `configure`

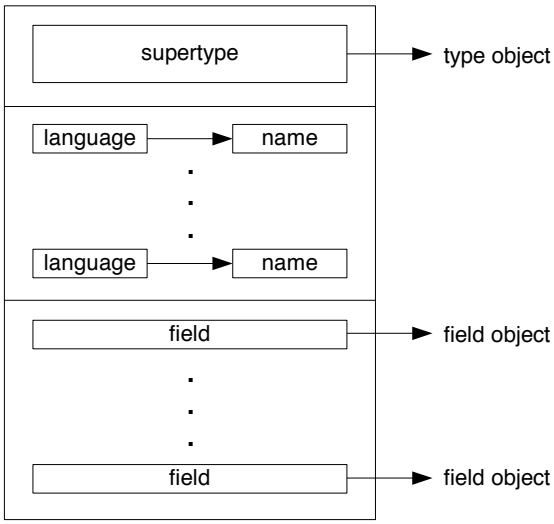


Figure 3.3: Structure of a Type Object

The fields that are available for putting children in are defined in a node's type.

Figure 3.2 is an example of a node in a meaning representation. It is of type `configure`, and has three fields containing child nodes: The *`actor`* field containing a node of type `you`, the *`target`* field, containing a node of type `literal_thing`, and the *`for`* field, containing a node of type `internet`.

A type object (Figure 3.3) represents a specific meaning in the vocabulary, for example the word "configure". To facilitate identification in multiple languages, a type object contains a mapping of language codes like "en" or "de" to natural language names identifying the type. A type object also contains a list of field objects. These fields define what kind of children a node of that type may have.

To continue the example, Figure 3.4 shows the type object for the meaning

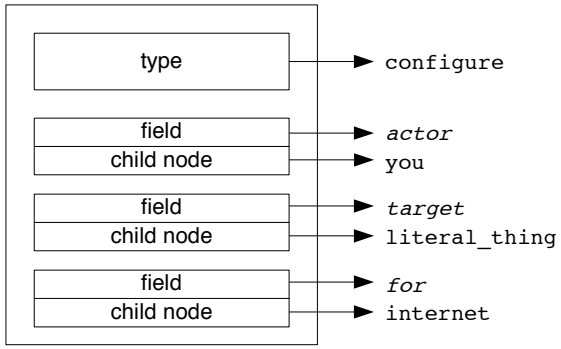


Figure 3.4: Type Object for `configure`

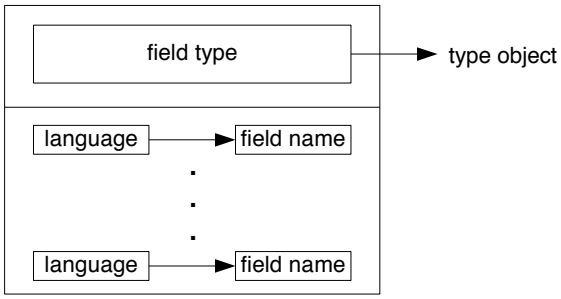


Figure 3.5: Structure of a Field Object

`configure`. It contains a name for itself in English and German, and has a single field called *`for`*. Its super-type is `action`, since configuring is a kind of action that can be performed.

Figure 3.5 shows a field object. It contains a mapping of language codes onto field names used to describe the relationship of a node to a child node in that field. It also contains a reference to a type object that constrains what kind of nodes may be put into that field.

Specifically, a field may only contain nodes of the specified type, or nodes whose super-type is of the specified type. This does not only include the immediate super-type, but also any other types that can be reached by following the chain of super-type references, including the super-super-type, etc. (Note that when referring to the super-type of a node, I always mean any and all types in this chain of references.)

This explains one effect of the type hierarchy. The other is that a type inherits the fields of its super-types: For example, the type `configure` inherits the *`actor`* and *`target`* fields from its super-type, `action`, and so, in the example

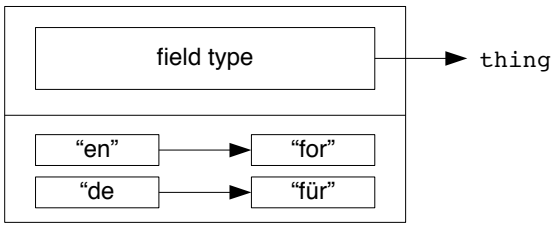


Figure 3.6: Field Object for the Field *for*

node, those two fields and the configure type's own *for* field are filled with nodes.

A last example: The *for* field of the configure type, shown in Figure 3.6. lt is of type thing, which means that nodes with a type whose super-type is thing may be entered into the for field of a node of type configure.

Meaning representations are hence tree structures with constraints on what children nodes may have, and with the children contained in labelled fields.

Figure 3.7 gives a more complete example. In this figure, nodes are displayed as thick-bordered boxes, and their associated fields are displayed immediately below them as thin-bordered rounded boxes. The contents of fields are linked to by lines.

Here, the root node of the tree is of type statement. Its *action* field requires a node of type or super-type action. In this case, it contains a node of type configure. That node in turn contains other nodes for specifying who is doing the configuring, what is being configured, and what it is being configured for.

One node bears special mention, as it is of a kind not yet mentioned: the one with the dotted border labelled “OpenOffice.org”. It is a string container used for storing literal string values in the meaning representation. During generation, literal string values are substituted in without any translation. They are useful for specifying words that should not be translated, like product or company names.

3.1.1 Language Independence

An important thing to understand about the meaning representation is that it is not bound to a specific language. While in Figure 3.7, the tree was labelled in English, the same data structure can just as well be labelled in German, as Figure 3.8 shows. The names of the nodes in different languages are simply identifiers pointing to the same meaning type, and to the same data structure in memory.

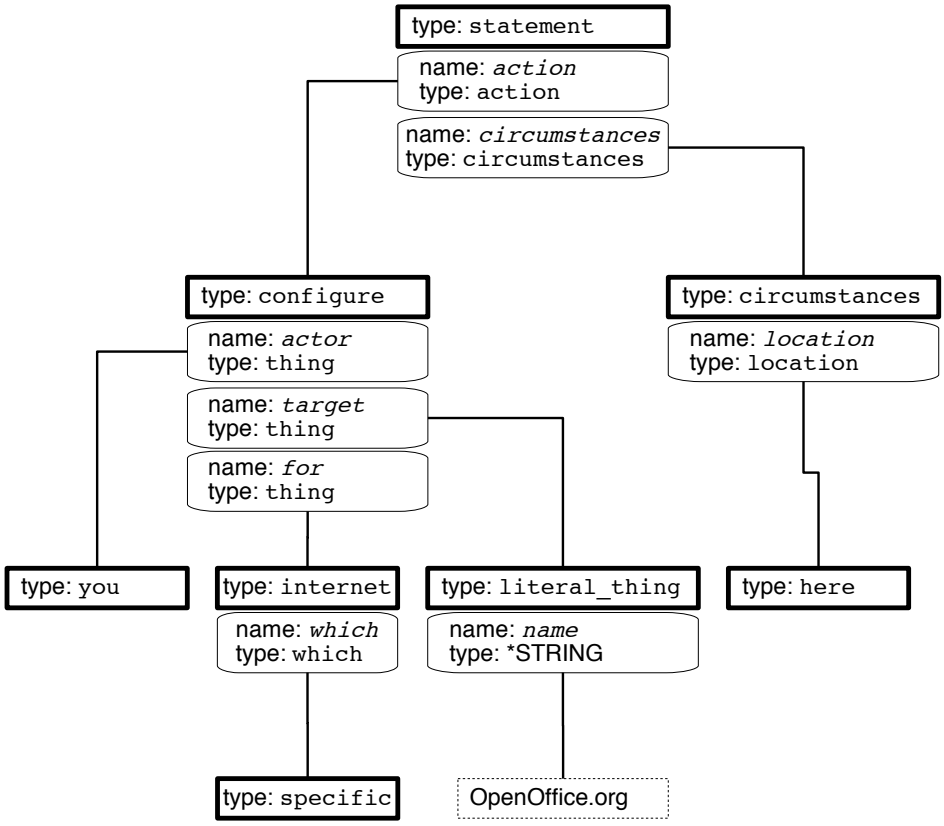


Figure 3.7: Meaning Representation for "This is where you configure OpenOffice.org for the Internet."

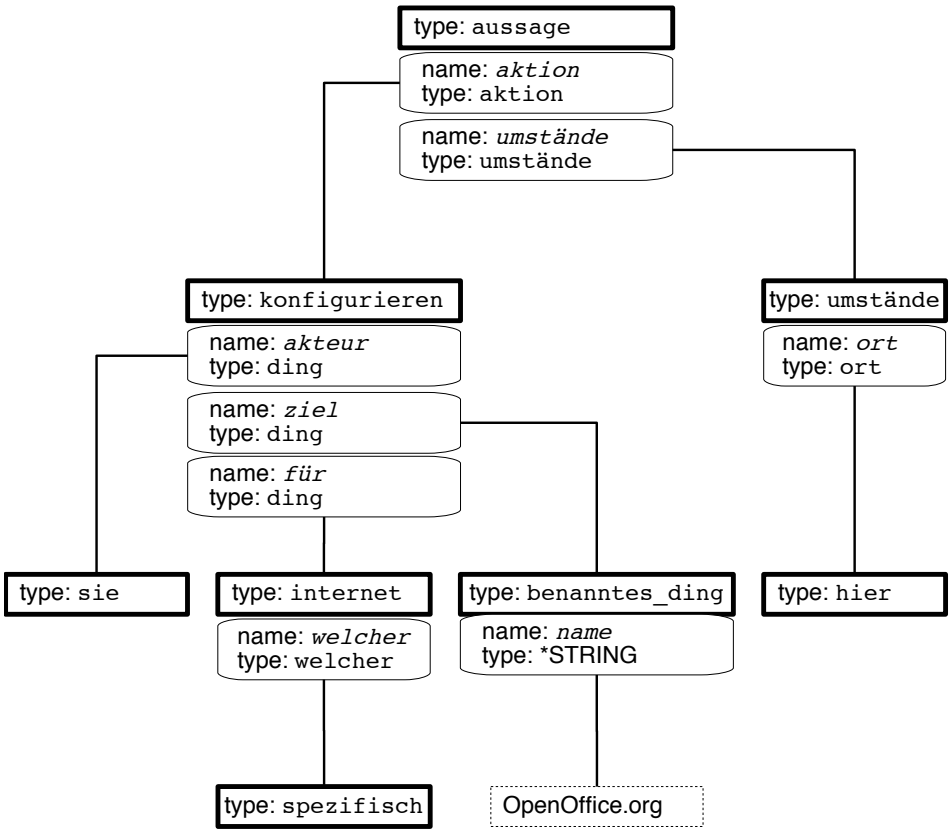


Figure 3.8: German Meaning Representation

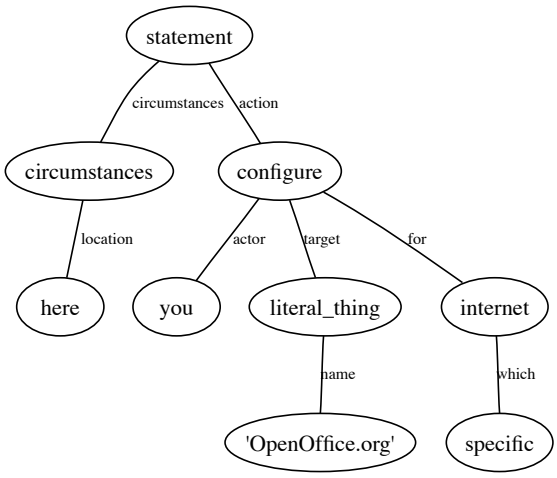


Figure 3.9: Meaning Representation graph drawn by GraphViz

3.1.2 Loading and Displaying

The vocabulary of types, the generation rules for a language, and the actual input are each stored in a different kind of file. The first step of the generator's operation is to load those files into memory. Hence, the first thing I implemented was the code for loading those three types of file into memory, linking nodes to their types, and types to their rule-sets. In order to test whether the data had been loaded and linked correctly, I implemented functions that allowed me to display the contents of the data structures using GraphViz [1]. (Figure 3.9)

GraphViz is an open source graph-drawing program that takes standardised text files as its input. To draw a graph, it is sufficient to list the name of nodes and their connections, and GraphViz will attempt to arrange them in a suitable and legible way. Thanks to this, I could easily produce highly informative graphs, showing meaning representations, the inheritance relations and fields of types, and the inheritance relations of rulelists.

3.1.3 Iterative Vocabulary Development

Once assembled, the corpus represented potential output from an underlying set of meanings that defined the generator domain. Thus, I could use the corpus to inform my design decisions for the meaning representation. I started the design process by encoding the first item using a putative meaning representation, and then continued to encode one item after the other, evolving the meaning repre-

[1] http://www.graphviz.org/

sentation to be able to cope with expressing the varying corpus items. Keeping the design coherent involved continually re-formatting earlier putative meaning representations, and as I had expected, I initially needed to re-write all previous meaning representations with nearly every new one encoded. But soon, the design of the meaning representation stabilised, making re-writes less frequent to the point where encoding the last third of the corpus felt quite routine.

Once I had implemented the generation algorithm, I compiled the final version of the vocabulary. This was a relatively quick process: I went over the putative meaning representations again, and fixed a number of inconsistencies. By listing all the node types that appeared in the corpus meaning representations, and the fields they used, I was then able to compile the vocabulary. The final vocabulary file consists of about 1700 lines, describing about 210 meaning types, naming them in both English and German.

3.2 Generation Rules and Algorithm

The generator is a system for applying transformation rules to a meaning representation, which transform it into natural language text. For a given output language, each type in the vocabulary maps to a list of generation rules. During generation, the generator invokes the *rulelist* of a node's type to produce the appropriate output text for that node. The generator chooses which specific generation rule out of the list to use, depending on what other nodes there are in the meaning representation.

Figure 3.10 shows the structure of a rulelist object. Its name field contains a text string used to identify a rulelist. The name of a rulelist is the same as the name of the type that maps to it. So for example, in the English generation rules, the type `configure` maps to the rulelist **`configure`**.

Optionally, a rulelist object contains a reference to a super-rulelist, which is a list of rules used for when no appropriate rule can be found within the rulelist itself. It also contains a set of tags, text strings used to specify properties the word associated with these rules has in the output language. Most importantly, a rulelist object contains an ordered list of individual generation rules. When the rulelist of a node is invoked, one generation rule is chosen to be used.

Figure 3.11 shows the structure of an individual generation rule. It consists of three parts: its *name*, its *condition* and its *output string*. The name is a text string, the condition is a *path* or a combination of paths, and the output string is a text string optionally containing further rulelist invocations. The name and condition are used to determine which rule out of the rulelist is used. Rulelist

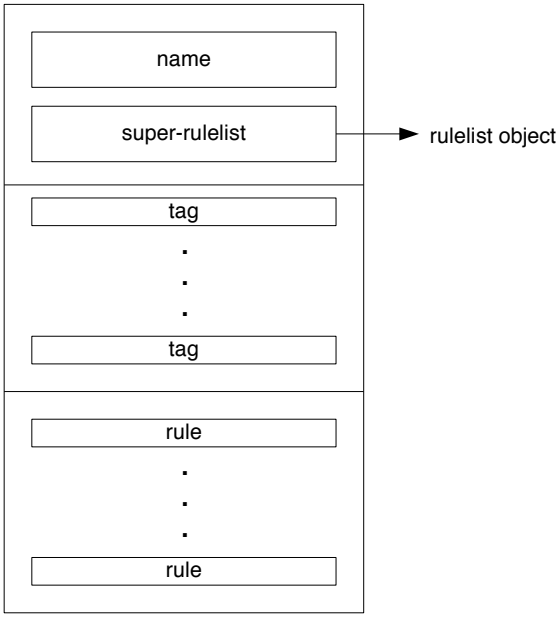


Figure 3.10: Structure of a rulelist object

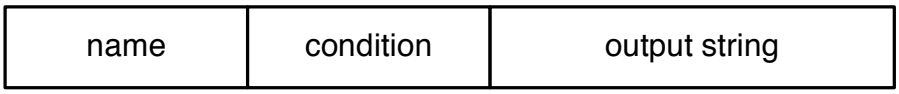


Figure 3.11: Structure of a rule object

invocations can be optionally supplied with a name string, and this name must match the name of a rule for it to be considered for use. If no name is specified, only rules with the default name of * (an asterisk) are considered. The first correctly named rule on the list whose condition evaluates to true is used. If no rule can be found where this is the case, the super-rulelist is invoked with the same name passed.

A path is a series of instructions for traversing a meaning representation tree, starting at the node whose rulelist has been invoked. There are three types of instruction:

- Go into field
  Instructs the generator to traverse the link from a node to a node's child. The field the destination child is in has to be specified. If that field does not contain a node, evaluation of the path fails. This instruction is written as a period followed by the field name.

- Go to parent
  Instructs the generator to traverse the link from a node to that node's parent. Optionally the destination node's field in which the origin node must be contained can be specified. If the node is contained in a different field, the path evaluation fails. The instruction is written as a caret symbol followed by the field name.

- Stay at this node
  Instructs the generator to stay at the current node. This instruction is written as an "equals" symbol.

An instruction can be optionally followed by a tag specified within square brackets. This tag is matched against the destination node of the preceding instruction in the following way: It is matched against the name of that node's rulelist, the rulelist's set of tags as well as the names of its super-rulelists and their tags. If no match can be found, evaluation of the path fails. This allows the path to specify that visited nodes must be of a certain type or have certain properties, as described by their rulelists' tags.

If a path evaluation succeeds, the path's value is "true", otherwise it is "false". The boolean results of evaluating multiple paths can be combined using basic boolean logic operators (and, or, not) to form more complex conditions.

Figure 3.12 shows a generation rule from the rulelist **`statement`**, which is invoked during the generation of the example sentence "This is where you configure OpenOffice.org for the Internet." Figure 3.13 describes how the path

nocapsorperiod	.circumstances.location[here]	"this is where <.action>{dealwithtime}{dealwithcondition}"

Figure 3.12: A generation rule from the rulelist **`statement`**

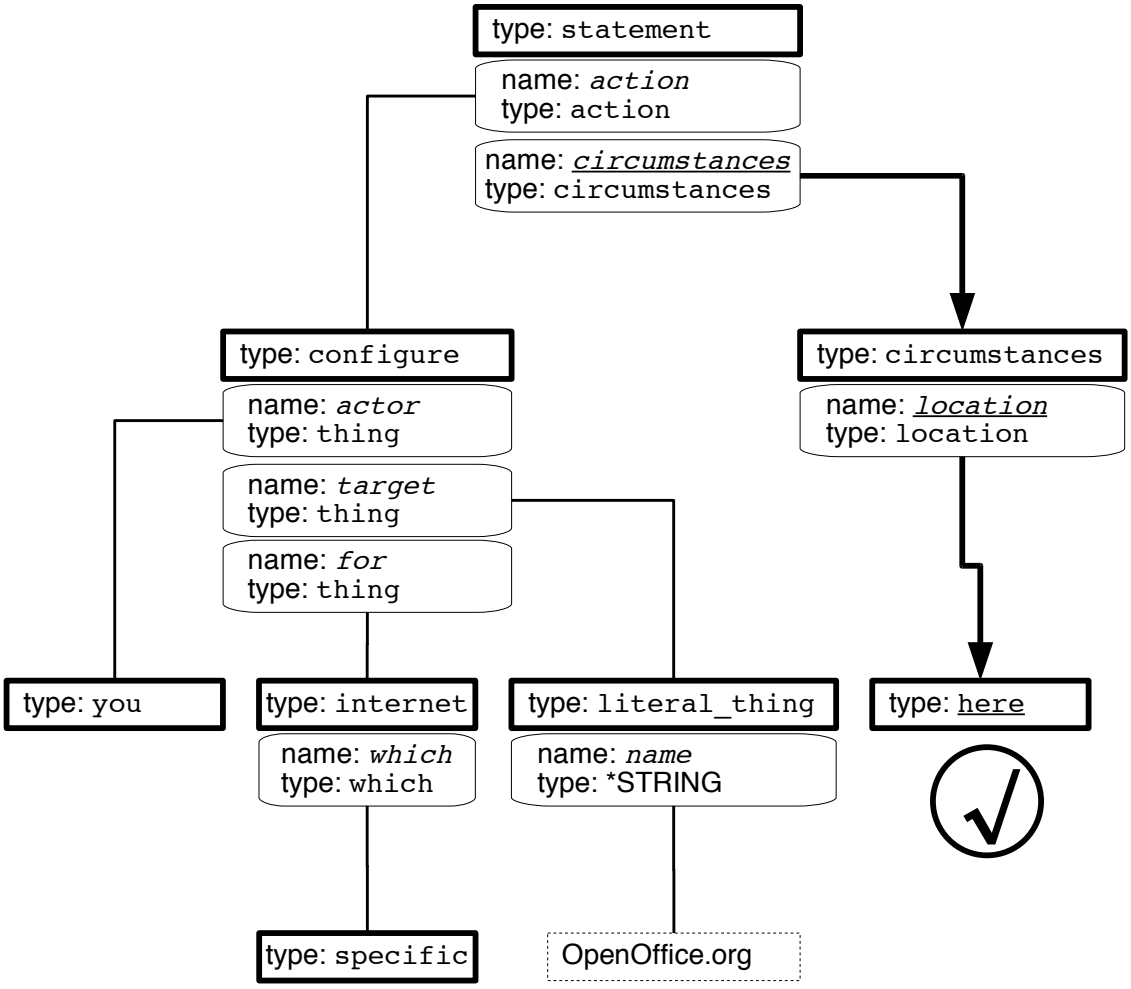


Figure 3.13: Following the path in a condition

in that rule's condition is followed. The generator starts out at the `statement` node, and is first instructed to go into the node's *`circumstances`* field, where it encounters a node of type `circumstances`. Next, it is instructed to go to that node's *`location`* field, where it encounters a node of type `here`. Last, it is instructed to match the tag [here] against the current node. This match (obviously) succeeds, and so the whole path's value is "true", telling the generator that this rule is the correct one to use.

When a generation rule is chosen, the generator looks at its output string for any instructions to invoke other nodes' rulelists. When it finds such an instruction the generator invokes it. This invocation will return a text string which is then substituted for the text of the instruction. Doing this to all invocation instructions results in a natural language text string which is then returned, to be substituted for the rulelist invocation instruction that was responsible for this rule being used in the first place.

An invocation instruction is an expression in angle brackets, comprised of a path to the node whose rulelist should be invoked. Optionally, a name string to be passed along with the invocation can be specified after a colon. Furthermore, if a plus symbol precedes the path, the first letter of the string returned is capitalised.

There is also another syntax specifically for invoking the rulelist of the current node using a different name string. This syntax simply consists of the name string within curly brackets.

Thus, the expression

```
{rulename}
```

is equal to the expression

```
<=:rulename>
```

To complete the example, Figure 3.14 shows the rulelist **`configure`** for the type `configure`. It contains only three very simple rules, since the bulk of conjugation and word order is handled by its **`simpleverb`** super-rulelist, and its super-super-rulelist **`verb`**, which handle the common behaviour of most English verbs. As such, only three rules for producing the stem, the participle, and the gerund form are needed. These three rules are invoked by **`simpleverb`** and **`verb`** as needed, to produce the text specific to the meaning `configure`.

In summary, the generator begins generation by invoking the rulelist of the root node of the meaning represenation. The selected rule of that rulelist then invokes the rulelists of other nodes deeper in the tree, which in turn do the same. As the recursion returns, the returned text strings are assembled into bigger and bigger pieces of the final output, till the root node's rule assembles the final

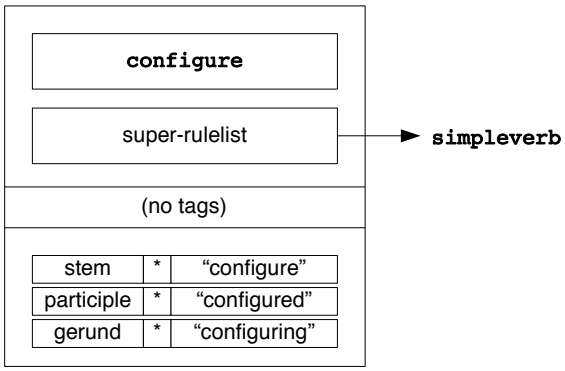


Figure 3.14: The rulelist **configure**

text, which is then returned by the generator as the output for that meaning representation.

3.2.1 Example Sentence Generation

To illustrate the generation algorithm, this section describes how the generator produces the sentence “This is where you configure OpenOffice.org for the Internet” from the meaning representation shown in Figure 3.7.

The generator starts the generation process by invoking the rulelist of the root node of the meaning representation using the default name string, “*”. Here, the root node is of type `statement`, so the rulelist **statement** is invoked. **Statement** only has one rule with the default name, which is

```
*, *, "<+=:nocapsorperiod>."
```

The condition of this statement is “*”, which means that it is automatically true. Hence, this rule is used. Its generation string consists of a rulelist invocation instruction, followed by a period. Since the path of the instruction is just an equals sign, the instruction is to invoke the rulelist of the root node again, this time using the name string “nocapsorperiod”, and to capitalise the first letter of the returned string.

So the generator invokes the rulelist **statement** again, using the name string “nocapsorperiod”. This time, there are six rules with the correct name, with the following conditions specified:

1. `.circumstances.time[currently]^.location[here]`
2. `.circumstances.time[past_affecting_present]^.location[here]`

3. `.circumstances.location[here]`
4. `.circumstances.time[past_affecting_present]`
5. `.circumstances.time[currently]`
6. `*`

Evaluating the condition of the first rule on this list, the generator attempts to follow the path `.circumstances.time[currently]^.location[here]`. The first instruction works, as the `statement` node's *`circumstances`* field contains a node of type `circumstances`. But the attempt to follow the next instruction fails, as the `circumstances` node's *`time`* field is empty. Resultingly, the rule's condition evaluates to "false".

Next, the generator attempts to evaluate the path in the second rule's condition but fails there too, as the path also attempts to access the *`time`* field.

But with the third rule, the condition evaluates to true, since the `circumstances` node does indeed contain a node of type `here` in its *`location`* field. Hence, the third rule is used. Its output string is as follows:

```
"this is where <.action>{dealwithtime}{dealwithcondition}"
```

It is more complex than the previous one, containing three rulelist invocation commands. However, it turns out that the latter two invocations only return empty strings, and so nothing more needs to be said about them. The first rulelist invocation contains a path to the `statement`'s *`action`* field, invoking its rulelist with the default name string.

The node in the *`action`* field is of type `configure`, so the rulelist **`configure`** is invoked. **`Configure`** only contains three very simple rules:

1. `stem, *, configure`
2. `participle, *, configured`
3. `gerund, *, configuring`

None of these rules have the correct name, which would need to be "*". So the generator invokes **`configure`**'s super-rulelist, **`regularverb`**. But there are no rules with the correct name there either, so **`regularverb`**'s super-rulelist is invoked: **`verb`**. There, finally, two rules names "*" can be found. The first rule is for expressing the passive voice, and hence requires in its condition that the *`actor`* field be empty. Since this is not the case, the second rule, which has no condition, is used. Its output string is as follows:

```
"{actor}{timebeforeverb_}{v_normal}{target}{aftertarget}{for}{modifier}"
```

This is where the word order of the sentence is mainly described, in seven invocations of rules within **verb** tasked with generating different elements of the sentence. Describing these invocations in detail would be merely tedious without adding to the example, and as such, only the return values are listed here:

- The first invocation, `{actor}`, returns the string "you ".
- The second one, `{timebeforeverb_}` returns an empty string, as no time is specified in the *circumstances*.
- `{v_normal}` invokes **configure**'s "stem" rule, producing the string "configure".
- `{target}` returns the literal string " OpenOffice.org".
- `{aftertarget}` returns an empty string, since it is only used by a handful of verbs to add extra information to the sentence.
- `{for}` returns " for the Internet".
- Finally, `{modifier}` also returns an empty string, since no *modifier* to `configure` is specified.

Together, these invocations assemble the text string "you configure OpenOffice.org for the Internet". This string is then returned to the first rule to be invoked, which capitalises the first letter and adds a period, returning the final phrase as the output of the generation algorithm:

> "This is where you configure OpenOffice.org for the Internet."

3.2.2 Writing the Generation Rules

Having defined what vocabulary the generator should process, I had two sets of generation rules to write, one for English and one for German. I chose to write the English rules first, as I assumed that English was a somewhat less complex language. This was on account of there being less *agreement* in English than in German.

Agreement is a form of linguistic connection in-between words of a phrase, resulting effects of one word changing in form depending on others it is related to. It can be explained with the following example: Consider replacing the `you` node

in Figure 3.7 with a `computer` node. This will result in the word "computer" being output by the generator instead of the word "you". But it also signifies a change from the second person to the third. Due to the effects of agreement in the English language, the verb of a sentence must agree in person with the subject. In our example, this means that the node `configure` is now output as "configures" instead of "configure", resulting in the sentence "This is where the computer configures OpenOffice.org for the Internet."

Since the effects of agreement in English are relatively simple, tt turned out that the most complex thing was the word order in a sentence. Depending on the tense, the presence of the object and subject, and other information, the order of words changes quite a bit. Out of the approximately 900 lines of generation rules, nearly 170 were alone required for sentence-level word order, and adding helper verbs such as "can" or "has been".

German did turn out to be a more difficult language to write rules for. This can be most readily exemplified by there being about 1500 lines in the file for German, as opposed to only about 900 for English. Thankfully, having already written the English rules and having learnt from the experience, I could make the German rules much more structured from the beginning, resulting in a cleaner division of functionality. There emerged four distinct levels of rules to deal with different aspects of the language, each level invoking the one below.

1. The top-level rules deal with word order alone, and invoke one rule for each possible word in a sentence.

2. The second-level rules deal with whether a word is actually present. For example, the rule for the object of a sentence only produces anything if there is indeed an object present, otherwise it returns an empty string.

3. The third-level rules are for adding articles, adjectives, adverbs, suffixes, etc. and call the fourth-level rules to fill in the actual word itself.

4. There is a list of fourth-level rules for each meaning in the vocabulary, and that rulelist is responsible for actually producing the word (or the stem of the word) that expresses that meaning.

This structure is also present in the English rules, but is less clear-cut.

The greatest difficulty in both languages was to keep down the number of agreement cases that had to be addressed. It turned out to be a good idea to give each individual word generated by a rule its own set of sub-rules that governed its agreement. This became especially important in German, where there is a great deal more agreement than in English. For example, there are four

factors determining the article of a noun: which case the noun is in, the gender of the noun, whether the noun is singular or plural, and whether its article is definite ("the") or indefinite ("a"). With four cases, three genders, and the other two binary distinctions, this results in 48 different combinations to consider, each potentially requiring a different article!

3.3 Modularity and Extensibility

Since the generator was designed to be solely a rules application engine, all information about the meaning types and their associated generation rules for each output language is contained in external text files. This means that adding new vocabulary, adding new languages, or even substituting an entirely different tree structure for the meaning representations is merely a matter of changing text files.

3.4 Testing

While encoding the generation rules, I used the corpus meaning representations as a way to test the rules. I encoded the rules in the order in which they were needed by one corpus item after the other. Having completed implementing the rules for generating an item in the corpus, I would then run all items before that one through the generator. This was to make sure my alterations had not impaired the generator's ability in some other area I had already implemented.

3.4.1 Line Tracking and Debug Tracing

Once enough of the generation algorithm was implemented to start running tests, I realised that I needed some way of tracing a generator crash to the exact step in the generation it happened.

The first attempt at achieving this was line tracking. This meant that in all three main data structures, each piece of information was tagged with the line number and file name of where it had been read in from. This meant that when, for example, a rulelist invocation failed because of a malformed condition expression in one of the rules, the line number of that rule would be displayed in the resulting error message. This allowed me to quickly find errors in the generation rules as well as the vocabulary and the input meaning representation.

This however turned out not to be sufficient to track the cause of crashes due to more complex circumstances, such as a rule's output string trying to refer to

a field that was empty. The error messages, while telling me precisely where the rule that had crashed was, could not tell me under which circumstances the crash had occurred. During the generation of a more complex meaning representation, a rule may well be used several times, and how was I to know which rulelist invocation caused the problem?

In order to find this out, I implemented an option to print a large quantity of debug information that detailed each step of the generation. This would allow me to follow the generation up to the point where it had failed, giving me a much better idea of why it had failed.

I implemented this by adding a command line option called "--debugtrace". When this option is turned on, the generation algorithm prints lines explaining what it is doing at every step. The number of lines thus generated tends to be very large. Even a modest example like the one above generates more than a thousand lines of debug information. By using indentation as a way to distinguish in between the more and less important steps, I was able to make this a bit more legible.

3.5 The Graphical User Interface

Once I had finished implementing the generator, I found myself with a bit of spare time, in which I decided to implement the user interface I had conceived of. I hoped that this would make the evaluation phase a lot faster and easier. Instead of having to write meaning representations in text form, and explicitly running them through the generator to get feedback, evaluation subjects could instead manipulate a graphical representation of the meaning representation and get continual feedback.

I had not previously done any Swing programming before, but I managed to produce the user interface within a few days of intensive programming. A side effect of this is that the GUI code is not as well structured as it could be, but it was more than adequate for the purpose of helping with evaluation. The hardest part about writing the GUI was implementing the tree drawing algorithm, which had to correctly position nodes in the horizontal without them overlapping or taking up too much space.

Figure 3.15 shows a screenshot of the GUI. Its main area displays the meaning representation being worked on as a tree structure. Each node in the tree is labelled two lines of text, the upper one being the name of the field and the lower the type of the node in the field. The exception to this is the node at the root, which is not contained in a field.

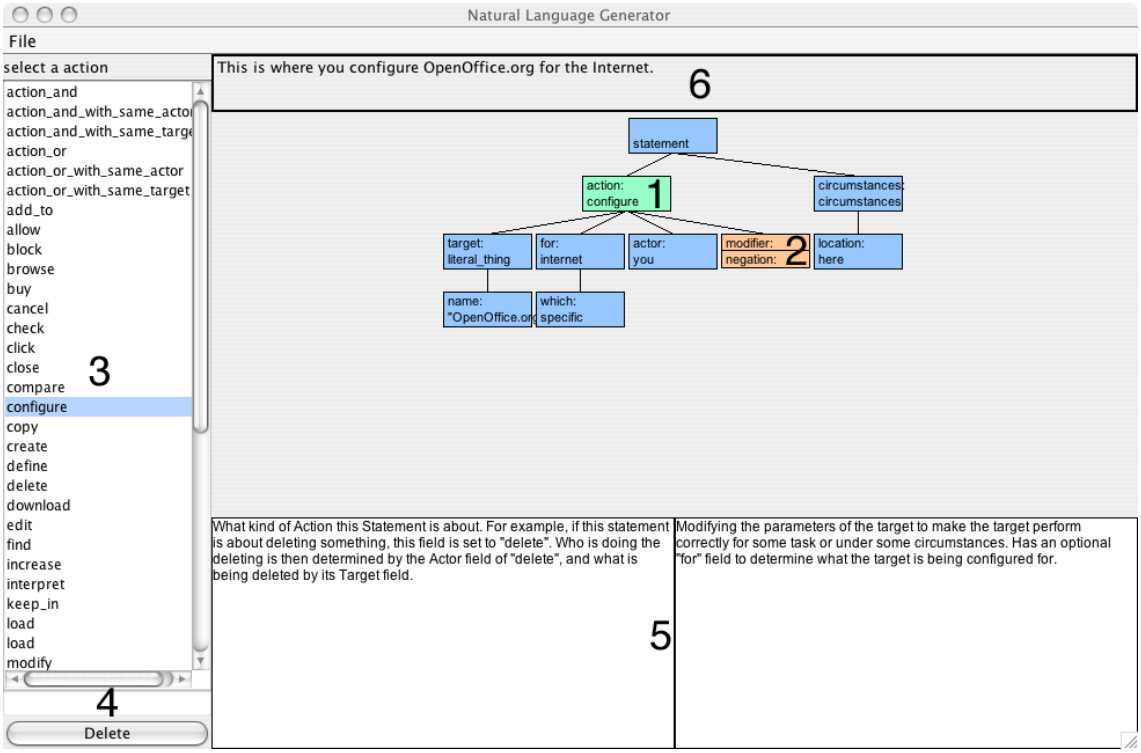


Figure 3.15: Screenshot of the GUI

The user can select (1) nodes by clicking on them. If the selected node has any fields that are not filled in, a list of them appears below the node (2). When the user selects one of those fields, they can then choose with which type of node to fill the field with, by selecting an option from the list on the left (3). If the node selected is a container for literal text or numbers, its contents can be edited via the text box below the list. (4)

The GUI also displays information about the selected node and the field it is in (5). This makes it easier for people to understand what meaning each type in the vocabulary stands for. Finally, the GUI displays (6) the text generated from the meaning representation in its current state. This text is automatically updated whenever the user changes the meaning representation.

Since the list on the left (3) only displays valid options, the user can quickly construct complex meaning representations.

Chapter 4

Evaluation

4.1 Aims

The purpose of the generator is to act as a translation aid. Users speaking one language can use the system to express themselves in a language they do not speak. The degree to which the translation is accurate determines how well the project has succeeded. The translation need only be accurate within the domain of text found in graphical user interfaces. Precision is important, whereas good style is not.

4.2 Evaluation Criterion

The circumstances under which the generator would be used in the real world involve it being used for translation into another language which the user does not speak. I sought to reproduce these circumstances by having a number of English-speaking evaluation subjects produce a corpus of meaning representations for sentences of their choice. The meaning representations in the corpus would then be run through the generator to produce both English and German sentences. A second group of evaluation subjects, fluent in both languages, would then assess the success of the translation.

4.3 Evaluating the Generator

I recruited two groups of evaluation subjects. The first group only spoke English, whereas the second group spoke both English and German fluently. Two groups were used instead of one so that the subjects who produce the evaluation corpus had no knowledge of the target language of the translation. It was also much

easier to find English-speakers, so I did not want to overburden the bilingual speakers I did find.

I recruited seven evaluation subjects for the first group. I gave each evaluation subject a copy of the generator running its graphical user interface and verbally instructed the subject in its use. The generator was set to only produce English output, in order to keep the evaluation subjects ignorant of what their meaning representations would look like in German. Once an evaluation subject was satisfied that they understand how to use the generator, I asked them to encode encode the meaning representations of ten different sentences. These sentences had to be be grammatically correct, and their meaning clear to me.

This resulted in a set of seventy meaning representations. Unfortunately, the evaluation subjects had a tendency to use the `literal_thing` construction to add extra words they were missing from vocabulary given. These extra words would of course not be translated into German, giving the appearance of incomplete translation in the German output. As a result of this, and due to some sentences being ungrammatical, I had to discard eleven items, leaving me with an evaluation corpus of 59 items.

I then ran the meaning representations in that corpus through the generator, producing pairs of English and German sentences. It is the similarity of meaning in between those pairs that determines the quality of the translation. To that end, I gave the members of the second evaluation group a web form containing all the sentence pairs, and asked them to rate each pair according to how closely they matched in meaning. Three different answers were possible:

- "Perfect Match"
  The two pieces of text have exactly the same meaning - the translation is as precise as possible. Both pieces of text are grammatically correct.

- "Close Enough"
  Although a more precise translation is possible, the two pieces of text have a meaning close enough for the purposes of user interfaces. That is, given the context of seeing the text in a menu, on a button, or in a tool-tip or other explanatory text, the two pieces of text can be said to mean the same thing. Any grammatical errors in are minor enough not to impair understanding. Any information missing from one sentence that is present in the other is superfluous.

- "Do Not Match"
  The translation is incorrect. That means that the information content of the

two pieces of text does not match, due to different information presented, or information missing in one of them that is present in the other.

I found seven English / German bilingual evaluation subjects who agreed to rate the sentence pairs.

4.4 Results

From the results of the evaluation, I could assign each item in the evaluation corpus a score from zero to seven depending of how many of the evaluation subjects had rated the translation to be "Perfect" or "Close Enough". The most useful numbers I extracted from these raw data was that 39 out of 59 (59%) of items had achieved a perfect score, and 46 (78%) of items had achieved a score of at least four. In other words, four-fifths of the corpus items had been translated at least "well enough" according to the majority of the testers. (Figure 4.1) A complete breakdown of the score distribution is given in Figure 4.2. Interestingly, there were few items that had been unanimously marked as being mis-translated.

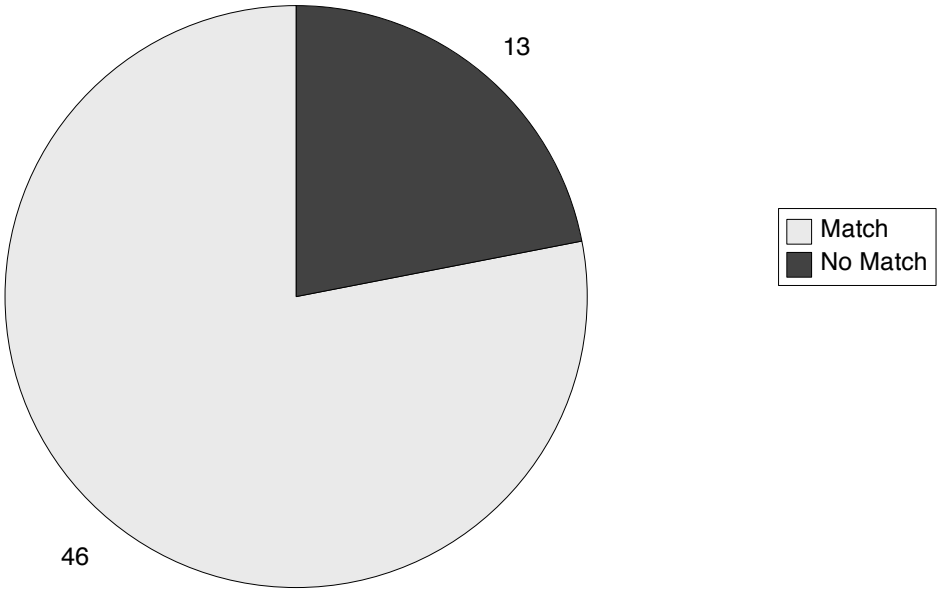


Figure 4.1: Evaluation scores for each corpus item: Majority decision

I also calculated the Kappa statistic [1] for measuring the level of agreemeent for the case of all seven evaluation subjects agreeing. The resulting Kappa value was 0.64, showing a reasonable level of agreement between evaluation subjects.

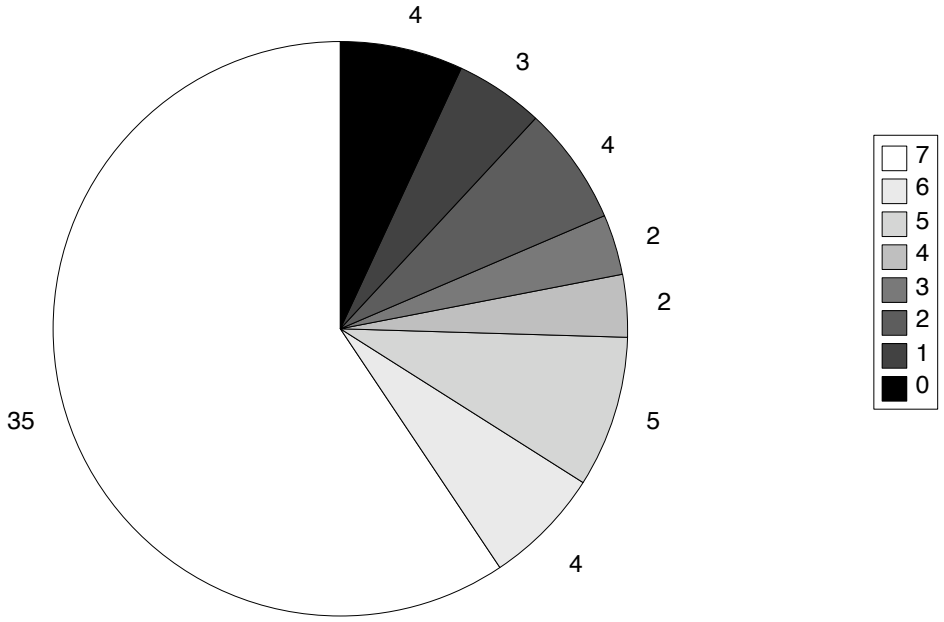


Figure 4.2: Evaluation scores for each corpus item: Detailed breakdown

Of those corpus items that had not generated to correct German, the generation errors were overwhelmingly due to bugs and omissions in the generation rules which could be corrected with a small amount of debugging work. Unfortunately, there was not enough time to fix these problems and re-do the evaluation. Nevertheless it is quite clear that given more work in completing the rules, the generator could achieve a near-perfect translation ability within the very limited domain of the few hundred meanings implemented.

There is one item in the evaluation corpus which is of particular interest, due to it being indicative of a greater problem with the taken approach to software translation. It is the sentence

"This is where your file is found."

The problem with this sentence is that it uses a subtly idiomatic form of the verb "find". Contrary to a very strict reading of the sentence, it does not indicate that there is some unnamed entity which is finding the file. The sentence merely intends to indicate that "this is the location of the file". Unfortunately, this distinction is lost in translation due to German using the literal meaning of the verb "found". Hence, the translation

"Hier wird ihre Datei gefunden."

implies that there is some unnamed entity that finds, or is finding the file. Which is not what the author of the meaning representation intended to say. The reason why this occurs is that the evaluation subject encoding the meaning representation of the sentence relied overly much on the feedback output of the generator, instead of thinking about what they were actually expressing. They were merely manipulating the meaning representation until the output sentence looked correct to them.

The problem here is that there are many subtle idioms in the English language, and that the user cannot be expected to be aware of the fact that they are using a word or a construction in a not entirely literal sense. When such a construction is generated to a different language, its intended meaning is lost and replaced by a more literal, unintended meaning. Ways of dealing with these problems are discussed in the next chapter.

Considering that this one sentence was the only item within the evaluation corpus which posed a significant problem for the generator, the evaluation shows that given trivial debugging work, the generator could achieve a near-perfect translation precision for its domain. This is an extremely good result that can be partly attributed to the very constrained domain. But arguably, a much larger set of meanings, constructed according to the principles described in the Introduction chapter, would yield a similarly high translation accuracy. As such, the generator's performance can be considered to be extremely good.

4.5 Generation Speed

When implementing the real-time feedback of the user interface, I noticed that generation appeared to be very fast, to the point where there was no perceptible delay between changing the meaning representation and the result of the changes becoming visible in the generated output. I hence set out to investigate the speed at which the generation algorithm runs by using the JiP profiler[1]. I used JiP to profile the generator starting up, loading its data files, and then producing the example sentence in English. According to the trace produced by JiP, the entire execution took 2756.3 ms. However, only 58.8 ms of that was spent on generating the output, with the remaining time spent on loading data files! I had not specifically aimed to make the generation algorithm efficient, so this came as a pleasant surprise. This result means that even with massively more complex generation rules and a much larger vocabulary, the generator would still run at a decent speed even without optimisations.

[1] http://jiprof.sourceforge.net/

Chapter 5

Conclusions

5.1 Proposed Further Work

5.1.1 Implementing More Output Languages

Since each language's generation rules are enclosed in a single text file, adding new output languages is an easy proposition. It would be very interesting to see whether the current form of the meaning representation is amenable to be processed into a language not related to English and German, such as Chinese or Japanese. It may turn out that there is important information missing from the meaning representation that is necessary to correctly express sentences in other languages. If that is the case, the meaning representation itself has to be re-worked to include the added information.

5.1.2 Implementing More Input Languages

This is simply a question of adding labels in another language to all the types in the vocabulary. In order to make it easier for people to create meaning representations in the new language, it is also suggested that translated information text about all the types and their fields is added. The generator GUI itself also has a small amount of hardcoded text, which would have to be localised. This could obviously be done using the generator itself!

5.1.3 Designing a More Powerful Generator

There are a number of ways in which the current implementation of the generator could be improved.

The most straightforward improvement is to re-work the format of the meaning representation into a more abstracted and expressive form. Its current form is the result of only looking at two closely related languages, and hence it shares many structural commonalities with English and German that are not present in other languages. Some of those structural commonalities may turn out to be a good way of representing information, whereas many will likely turn out to be flawed, and will be replaced by less language-bound structures.

Looking at the more fundamental structure of the meaning representation, it could be turned from a tree structure into a directed graph, allowing for multiple nodes to make direct reference to the same node. Also, fields could optionally be allowed to contain multiple nodes, allowing for easy description of lists, multiple adjectives, etc. Doing so would require a substantial re-write of the generator and its GUI.

A more powerful generation rules engine could also be implemented at the same time. The most valuable addition to the rules engine would be to implement support for generation rule "functions". These would have the same basic layout as generation rules, but instead of returning text strings, they would return references to nodes. This way, nodes could ask other nodes for where to find information that could be stored in multiple places in the meaning representation, instead of having to search for this information through listing all possible paths to the information. This also allows for the generator to deal with data structures of an unbounded size, instead of being constrained to a specific number of path hops through the tree. For example, a node may want to refer to the root node. But since the meaning representation includes types with fields of the same type, the size of a meaning representation is unbounded. Using the current system of specifying paths, only paths of a finite length can be followed. Using functions, a node could ask its immediate parent for the root node. If the parent was the root node, it would return a reference to itself. If it was not, it would make the same function call to its own parent. Eventually, the root node would be reached, and a reference to it could be returned through the function chain.

5.1.4 Integrating the Generator for Practical Use

In real-world usage, it would be rather inconvenient to have to copy and paste the strings produced by the generator into the source code of the program being translated.

Thankfully, there are already a number of tool kits available to deal with translation, the most important of which is arguably the GNU foundation's "gettext" library. It operates by providing a service for mapping English text onto

foreign-language text contained in an external data file. For example, in order to make available a translation of our example phrase, the following function call is made:

```
gettext("This is where you configure OpenOffice.org for the Internet.")
```

The gettext function will then check at runtime which language the program is set to, and return either the English phrase, or perhaps its German translation: "Hier konfigurieren Sie OpenOffice.org für das Internet."

With minor modification, the generator could be made to output the format of data files gettext uses. However, a change in a meaning representation would lead to different generated text in both English and German, with the result that the parameter given to the gettext function would no longer find a matching entry in the data file. This could potentially be fixed by allowing the generator to keep the source code itself up to date with the newest generated output.

Alternatively, the generator could be more closely integrated with an IDE and a pre-processor. Here, the meaning representations would be stored directly in the source code, but the IDE would display them as blocks of generated natural language text. Clicking on one of those blocks would launch the generator's GUI, allowing the user to change the meaning representations.

The IDE would then run the source code through a pre-processor before compiling, which would replace the meaning representations with automatically generated functions. At runtime, these functions would operate like the gettext function, returning the appropriate generated text for the currently selected language.

Pseudocode for such a function:

```
function gettextAutogenerated271() {
switch getCurrentLanguageCode() {
"en": {
return "This is where you configure OpenOffice.org for the Internet.";
break;
}
"de": {
return "Hier konfigurieren Sie OpenOffice.org fuer das Internet.";
break;
}
}

throw LanguageNotFoundException;
}
```

5.1.5 Detecting Idioms

One problem mentioned in the previous chapter is that users may unknowingly use idiomatic expressions when encoding meaning representations, which leads to the intended meaning to be lost in translation. If the generator is to be used as a real-world system for translating software, this problem needs addressing in order to improve reliability. Unfortunately, it is not clear how to solve this problem, due to there being a plethora of different idiomatic expressions, the identification of which requires an enormous amount of cultural knowledge to be processed. Nevertheless, the problem could be partly addressed in a very brute-force manner by producing a list of idioms in the form of fragments of meaning representations, and having the GUI alert the user if the user has created a meaning representation that includes such a fragment. It would then still be up to the user to fix the problem, but at least they would be aware of it. However, such a list of idioms could never hope to be entirely exhaustive, and could only at best reduce the problem, not eliminate it.

5.2 Conclusions

To reiterate, the aim of this project was to implement a natural language generator capable of producing output in multiple languages, for use as a translation mechanism for graphical user interfaces.

The project was definitely a success in the sense that the generator implemented is able to produce grammatically correct English and German sentences. The evaluation results show the generator's potential for producing reliable and accurate translation from English to German. Obviously, the generator's vocabulary of meaning is insufficient for real-world use, but given more leg-work in implementing further vocabulary and associated generation rules, the generator could be used for its intended purpose of translating user interfaces.

Appendix A

Original Proposal

(see next page)

Bibliography

[1] Jean Carletta. Assessing agreement on classification tasks: The kappa statistic. *Computational Linguistics*, 22(2):249–254, 1996.

[2] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*, chapter 21. Prentice Hall, pearson international edition, 2000.