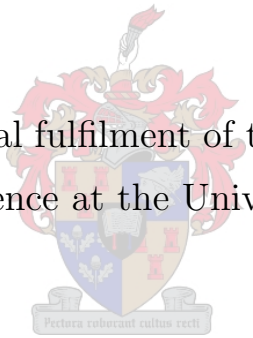
A Prototype System for Machine Translation from English to South African Sign Language using Synchronous Tree Adjoining Grammars

Johan Welgemoed



Thesis presented in partial fulfilment of the requirements for the degree of Master of Science at the University of Stellenbosch.

Supervisor: Dr. L. van Zijl

March 2007

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously, in its entirety or in part, submitted it at any university for a degree.

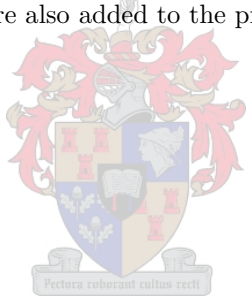
Signature: _____ Date: _____

Abstract

Machine translation, especially machine translation for sign languages, remains an active research area. Sign language machine translation presents unique challenges to the whole machine translation process. In this thesis a prototype machine translation system is presented. This system is designed to translate English text into a gloss based representation of South African Sign Language (SASL).

In order to perform the machine translation, a transfer based approach was taken. English text is parsed into an intermediate representation. Translation rules are then applied to this intermediate representation to transform it into an equivalent intermediate representation for the SASL glosses. For both these intermediate representations, a tree adjoining grammar (TAG) formalism is used. As part of the prototype machine translation system, a TAG parser was implemented.

The translation rules used by the system were derived from a SASL phrase book. This phrase book was also used to create a small gloss based SASL TAG grammar. Lastly, some additional tools, for the editing of TAG trees, were also added to the prototype system.



Opsomming

Masjienvertaling, veral masjienvertaling vir gebaretale, bly 'n aktiewe navorsingsgebied. Masjienvertaling vir gebaretale bied unieke uitdagings tot die hele masjienvertalingproses. In hierdie tesis bied ons 'n prototipe masjienvertalingstelsel aan. Hierdie stelsel is ontwerp om Engelse teks te vertaal na 'n glos gebaseerde voorstelling van Suid-Afrikaanse Gebaretaal (SAG).

Ons vertalingstelsel maak gebruik van 'n oorplasingsbenadering tot masjienvertaling. Engelse teks word ontleed na 'n intermediêre vorm. Vertalingreëls word toegepas op hierdie intermediêre vorm om dit te transformeer na 'n ekwivalente intermediêre vorm vir die SAG glosse. Vir beide hierdie intermediêre vorms word boomkoppelingsgrammatikas (BKGs) gebruik. As deel van die prototipe masjienvertalingstelsel, is 'n BKG sintaksontleder geïmplementeer.

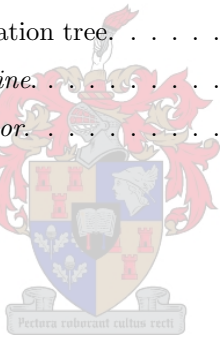
Die vertalingreëls wat gebruik word deur die stelsel, is afgelei vanaf 'n SAG fraseboek. Hierdie fraseboek was ook gebruik om 'n klein BKG vir SAG glosse te ontwikkel. Laastens was addisionele nutsfasiliteite, vir die redigering van BKG bome, ontwikkel.

Acknowledgments

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily those of the NRF.

Contents

List of Figures



Chapter 1

Introduction

Machine translation has been an important subject in computer science for almost sixty years. During most of this time, however, machine translation for sign languages has been largely overlooked. It is only during the last decade that machine translation for sign languages received renewed interest. This is due both to advances in sign language linguistic research [24, 33] and to an increase in computing power. Sign languages are visual languages and have no widely used or accepted written form. Therefore a machine translation system for sign languages needs to represent the translation in a visual way. To do this we need adequate computing power to render an avatar[1] signing the translation. Up until the 1990's this computing power was not widely available.

The South African Sign Language Machine Translation (SASL-MT, see Figure 1.1) project is an ongoing project at the University of Stellenbosch. The aim of this project is to develop practical tools for English to South African Sign Language (SASL) translation. To help achieve this goal, we developed a prototype machine translation system. The aim of our prototype machine translation system is to parse English text and then translate it into a gloss based form which can be used as input for a signing avatar. Glosses are English representations of signs, with each sign represented by its nearest spoken counterpart [50].

Various methods and techniques exist to develop such a prototype machine translation system. Some of these methods and other machine translation projects are discussed in Chapter 2. The approach taken in our machine translation system makes use of tree adjoining grammars (TAGs) to represent both the English input and the gloss based output. The parser described in [28] was implemented to parse the English input. This parser made use of a TAG that was contained within a syntactic database used by the parser. A morphological database was used to determine different word categories in an input sentence. Both the syntactic and morphological databases were developed at the University of Pennsylvania as part of their XTAG project [15, 18].

Using the syntactic database allowed us to shorten the development time of our machine translation system. This was an important consideration in the decision to use TAGs for our machine translation system, as one of the aims of this thesis was to develop a prototype system in as short an amount of time as possible. After parsing is finished, translation rules are used to translate the input representation into the output representation.

The translation rules used in our machine translation system were derived from a small set of translated sentences. These simplistic rules were derived in order to demonstrate the capabilities of the system, and are by no means comprehensive or fully representative. The translated sentences were also used to build a small TAG for SASL. This small TAG was used in representing the gloss based output of the system.

The rest of this thesis is organised as follows:

Chapter 2 provides an overview on general machine translation techniques, and some of the more recent and successful sign language machine translation systems are reviewed. The chap-

[1] An avatar is a 3D virtual representation of a person or entity.

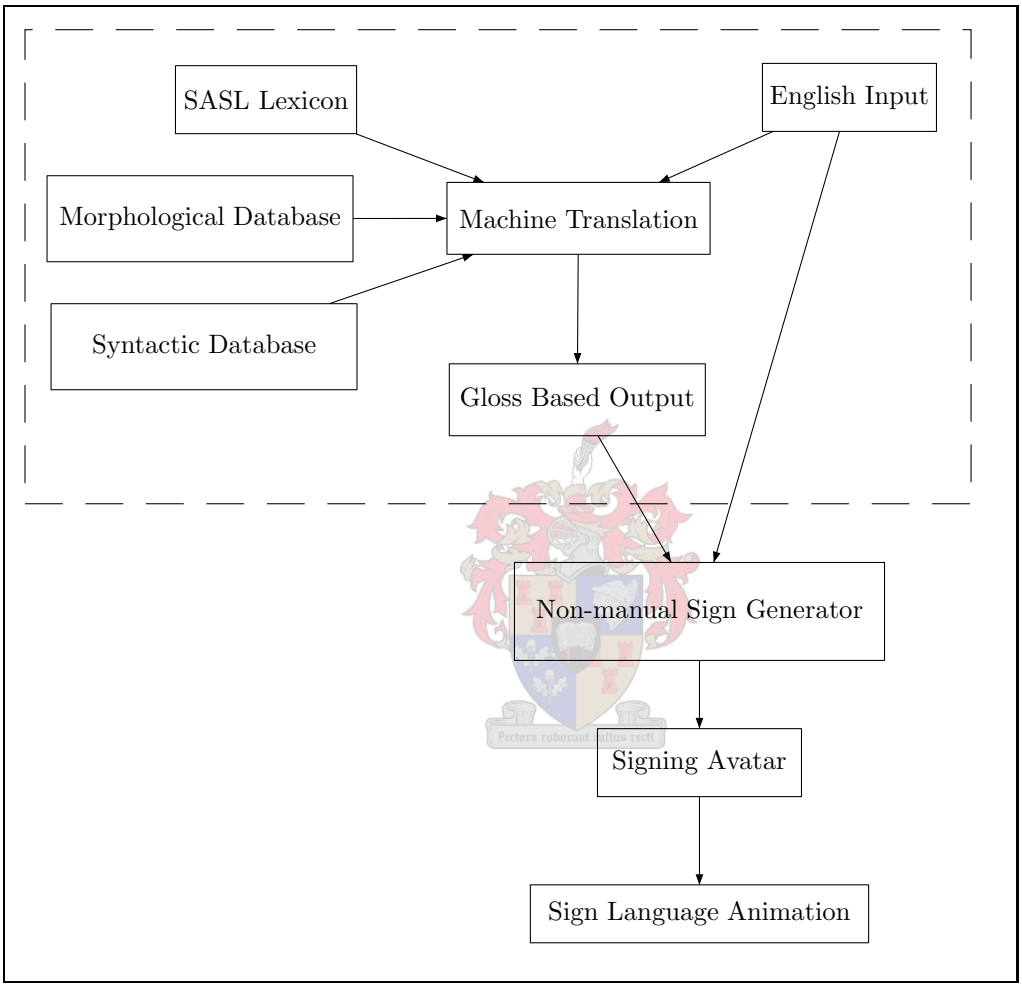


Figure 1.1: An outline of some of the activities of the South African Machine Translation project. The area enclosed by the dashed box corresponds to some of the work covered by this thesis.

ter ends by discussing parsing strategies that could be used in some of the machine translation techniques.

Chapter 3 introduces TAGs and discusses the parsing algorithm that was implemented for the system.

Chapter 4 describes the different components of the prototype machine translation system, as well as how data used in the translation system was gathered. This data consisted of a set of sentences and their SASL translations (in gloss form). These translated sentences were used to derive some rudimentary translation rules that were used in the system. Finally, Chapter 5 presents our conclusions and discusses future work.

Chapter 2

Background

In this chapter we look at various machine translation techniques. We also look at how these different techniques were incorporated in some sign language machine translation systems. The last section of this chapter discusses different parsing strategies that can be used for parsing input in a machine translation system.

2.1 General Machine Translation Techniques

This section discusses two different approaches taken toward machine translation. The first approach is transfer based machine translation. We look at this approach in some detail in the next section. The second approach we consider is statistical machine translation in Section 2.1.2.

2.1.1 Transfer Machine Translation Methods

The goal of any translation is to transfer the meaning of a text from one language to another. This goal is made explicit in transfer based translation methods. The first step in the translation process is to parse the input into an intermediate representation. This intermediate representation is transformed into a representation forming the basis of the translated output. The transformation is usually accomplished through the use of transfer rules. From the representation of the translated output, the final translated text is generated.

The different methods that fall under transfer based machine translation are often visualised as a pyramid (Figure 2.1.1, adapted from [17]). The pyramid represents the amount of analysis required to obtain an intermediate representation of the input. It also represents how close the representation between the input and output is; that is, how much additional effort is required to transform the input representation into the output representation. Finally, the pyramid also represents how much work must be done in order to generate the final translated text from the intermediate representation of the output.

At the bottom of the pyramid we find the direct transfer methods. These methods require the least amount of analysis of the input. Moving towards the top of the pyramid the gap between the source and target representations closes until we reach the apex. Here the intermediate representations of the source and target languages are the same. The methods found here are the so-called interlingua machine translation methods. These methods require the most effort in terms of analysis of the source text and generation of the target text.

The different methods found at the various levels of the machine translation pyramid (starting from the bottom) is described in the following sections.

Direct Transfer

Direct transfer methods require the least amount of analysis of all the transfer based methods. The reason is that, direct transfer methods only analyze the input at a morphological (or word)

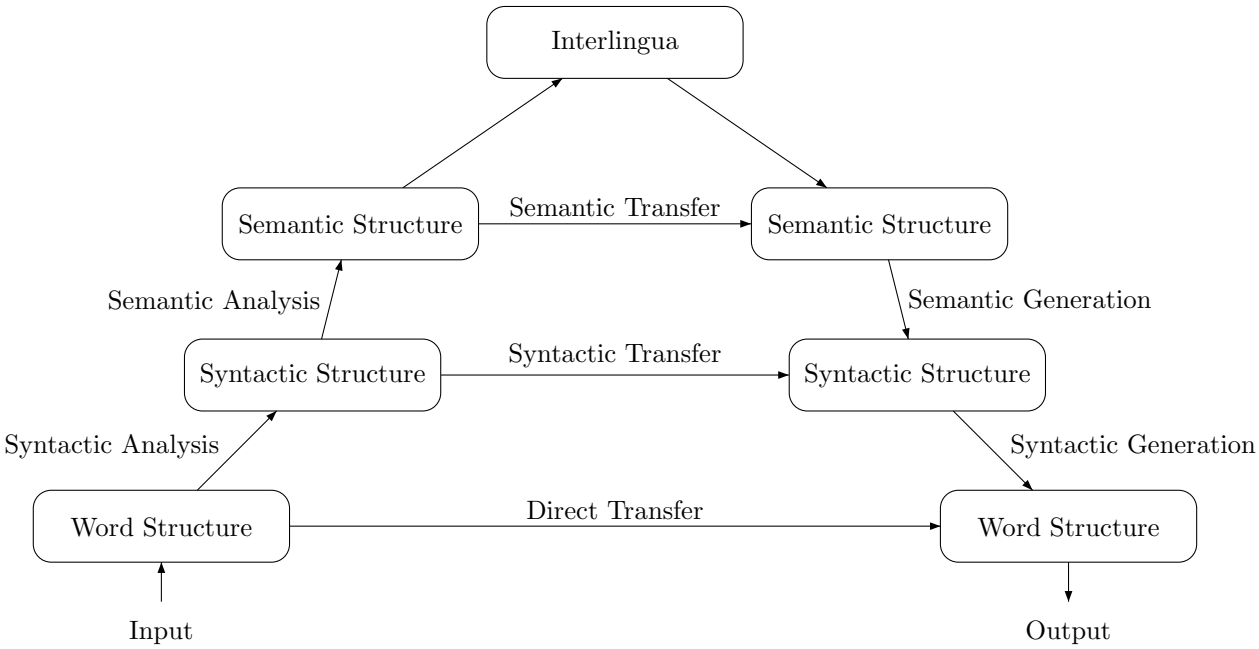


Figure 2.1: The machine translation pyramid.

level. The analysis is only done so far as to use the individual words of the input to look up entries in a bilingual lexicon. These entries form the basis for the translated output.

There are two main advantages to using direct transfer methods. The first advantage is that, because no parsing of the input is required, direct transfer methods are fast and easy to implement. The second advantage is that the only language resource required is a bilingual lexicon.

Direct transfer methods however do have many disadvantages. The 'lookup and replace' approach taken in direct transfer methods leads to literal translations [17]. This means that alternative meanings for sentences, such as with idioms, are not translated. A second problem that can arise with direct transfer methods is in choosing the correct meaning of a word. This occurs because direct transfer methods lack a grammatical structure for the input. This means we do not know what the relationship between words in the input are or how they were used. For example, in the sentence, *The man left the bank* the word *bank* is a noun and can either refer to a river bank, or a financial institution. Determining which meaning is correct is a difficult problem to solve in a direct transfer system [17]. One solution is to restrict the domain of translation. By restricting the domain over which a translation system operates, we can ensure that each word is unambiguously specified in the lexicon.

Apart from these problems, mechanisms are needed to handle at least some grammatical issues. One such an issue is differing word orders between the source and target languages. Therefore, most direct transfer systems incorporate some form of a word re-ordering component.

Some of the first successful machine translation systems were based on direct machine translation methods. Such systems include the Systran (1968), LOGOS (1969-1970), Xonics (1970-1976) and PAHO (1976) systems [26]. The Systran system is still a widely used commercial system even though it is essentially based on a direct transfer based system [27].

In the next section we look at syntactic transfer methods. These methods try to address some of the problems of direct transfer methods.

Syntactic Transfer

Syntactic transfer methods analyze the surface structure of an input sentence. To do this, a formalism with which to represent the surface structure is required. This formalism takes the

form of various types of formal grammars. Such grammars can include context free grammars, head driven phrase structure grammars [34] or TAGs [50]. Language based transfer rules transform the formal representation of the input into an equivalent representation for the output. When the transformation is complete, a language generation module generates a target language sentence from the output representation.

Our prototype machine translation system is an example of a syntactic transfer machine translation system, as we use a TAG to represent both the input and the output and the translation is done through translation rules.

Syntactic transfer methods are able to handle some of the problems of direct transfer methods mentioned in the previous section. Since a syntactic structure now exists for the input sentence, all the words in the input will have a syntactic category assigned to it. This will help with some ambiguities which direct transfer systems find problematic [17]. The problem of word reordering can be incorporated into the translation rules of the machine translation system.

We next consider semantic transfer methods.

Semantic Transfer

Semantic transfer methods attempt to derive meaning from the source text. Just as for syntactic transfer methods, semantic transfer methods analyze the input to determine a semantic representation of the source text. The following example illustrates a simplistic semantic transfer system. This system assumes that each sentence consists of a subject, a verb and an object. For each input sentence, it will try to identify the subject, the verb and the object. This semantic representation of the input is stored in a table-like structure with entries for the subject, verb and object. The sentence *John kicked the ball* will be represented as the following:

SUBJECT	OBJECT	VERB
John	*the ball*	*kick* (PAST)

.

The system now applies translation rules to map this structure onto an equivalent type of structure for the output. Lastly, language generation modules generate a complete output sentence from such a semantic structure.

Semantic transfer systems can produce good translations, given that the semantic analysis and transfer rules of the system is complete, and that the lexicon that the system uses covers the domain over which the translation system is supposed to operate [17].

Many systems are often based on a blend of syntactic and semantic transfer methods. This means that a syntactic representation of the source text is augmented with semantic information. An example of such an approach is when a system adds a discourse representation of the source text to the syntactic representation. Discourse representation structures provide mechanisms to represent the entire source text, and not just individual sentences [22].

Discourse structures allow machine translation systems to address problems such as anaphora resolution and other semantic ambiguities [34], problems which a direct or syntactic transfer system would not have been able to cope with. One example of a system where both syntactic and semantic methods are used is the ViSiCAST translation system [34]. The translation system described in this thesis, only uses a syntactic transfer method. Additional semantic analysis does however form part of the larger SASL-MT project [14]. The translation system discussed in this thesis will be incorporated with the semantic analysis component discussed in [14] to form a more complete machine translation system.

The final transfer based methods we look at are the interlingua machine translation methods.

Interlingua

With interlingua machine translation methods, the intermediate representations for both the source and target languages are the same. This means that no transfer mechanism is required for interlingua methods. However, these methods require the most language resources of all the methods discussed so far. These resources often take the form of world knowledge included in

a system that uses the interlingua approach. Obtaining these resources tend to be a difficult and time consuming task. Therefore, most interlingua systems operate only over a well defined and limited domain. An example of a system where an interlingua approach was taken, is the ZARDOZ system [47].

In the next section, we look at an entirely different method of machine translation, namely, statistical machine translation.

2.1.2 Statistical Machine Translation Methods

Statistical machine translation methods are based on an entirely different premise than the transfer based methods discussed in the previous section. In statistical methods, the source text is seen as an encoding of the target text, and the problem of machine translation becomes one of "decrypting" the source text.

Let $\mathbf{s}$ denote a sentence in the source language and $\mathbf{t}$ a sentence in the target language. The probability that $\mathbf{t}$ is the correct translation of $\mathbf{s}$ is denoted by $Pr(\mathbf{t}|\mathbf{s})$. From Bayes' theorem [12] we have that,

$$Pr(\mathbf{t}|\mathbf{s}) = \frac{Pr(\mathbf{t})Pr(\mathbf{s}|\mathbf{t})}{Pr(\mathbf{s})} \quad .$$

Finding the string $\mathbf{t}$ which has the greatest probability of being the correct translation of $\mathbf{s}$ is the same as finding the $\mathbf{t}$ that maximizes the equation above. However, because $Pr(\mathbf{s})$ is independent from the product $Pr(\mathbf{t})Pr(\mathbf{s}|\mathbf{t})$, the problem can be reduced to searching for that $\mathbf{t}$ that maximizes the product $Pr(\mathbf{t})Pr(\mathbf{s}|\mathbf{t})$. This produces the so-called Fundamental Equation of Machine Translation [12]:

$$\hat{\mathbf{t}} = argmax_{\mathbf{t}}(Pr(\mathbf{t})Pr(\mathbf{s}|\mathbf{t})) \quad .$$

In this equation $Pr(\mathbf{t})$ is the probability that $\mathbf{t}$ is a valid sentence in the target language. This is called the language model. $Pr(\mathbf{s}|\mathbf{t})$ is the probability that $\mathbf{t}$ is a valid translation for $\mathbf{s}$. This is called the translation model. Lastly, $argmax_t$ is a search algorithm that searches for that sentence $\hat{\mathbf{t}}$ that maximizes the probability of the product $Pr(\mathbf{t})Pr(\mathbf{s}|\mathbf{t})$. The most correct translation of $\mathbf{s}$ will then be $\hat{\mathbf{t}}$.

The problem of statistical machine translation comes down to effectively modeling the probabilities of the language and translation models and finding an efficient search algorithm. For the language model, n-gram models with smoothing are generally used [42]. In [12], five translation models and algorithms for estimating their parameters are described. The search algorithm that is used depends on the translation model of the machine translation system in question.

The accuracy of statistical machine translation depends on the size of the translated corpus available for the system. The most important reason why statistical machine translation is not generally used for sign language translation is the fact that usually no corpus large enough is available for such a system [25].

It is interesting to note that in the previously mentioned machine translation techniques, researchers have to design translation rules manually. However, when using a corpus based translation system, the translation rules are expressed implicitly in the translated corpus. The success of a statistical machine translation system can then be thought of in terms of how well these implicit rules are revealed through translation.

Currently the only sign language machine translation system we are aware of that uses a statistical approach is proposed by H. Ney and J. Bungeroth for German Sign Language at RWTH Aachen [13]. Another system using an example-based approach to machine translation, is proposed by Morrisey [30] for Irish Sign Language.

2.2 Other Sign Language Translation Projects

In this section four different sign language machine translation systems are discussed. Each of these systems are based on transfer methods. The first system we look at is the ViSiCAST Project.

2.2.1 The ViSiCAST Project

The ViSiCAST system [34] formed part of the Framework V programme of the European Union. It was originally intended to be used as a multilingual sign language translation system supporting translation into a variety of sign languages including British, Dutch and German Sign language.

The ViSiCAST system consists of various stages. The first stage takes English input and parses it using a link grammar formalism [39]. For this the Carnegie Mellon University (CMU) link grammar parser [21] was used. The user can select between parses generated by the CMU parser and change any part of speech assignments that were made during parsing.

The second stage of the ViSiCAST system builds a semantic representation of the input. It is important during this stage that the correct meaning for each word in the input is selected. The system ensures this correctness through either WordNet [1] or user intervention. The semantic representation takes the form of a Discourse Representation Structure (DRS). The DRS is built by applying definite clause grammar [2] rules on the link grammar parse generated in stage one.

The DRS representation allows the system to order events chronologically. It also allows the system to determine relationships between different entities in the input. For example, in the input *John kicked the ball. The ball went out of bounds.*, the DRS representation will be able to show that the ball spoken of in both sentences is the same entity. It will also show that John first kicked the ball before it went out of bounds.

The third stage translates the semantic representation generated in stage two into a sign language representation. To represent the sign language a head-driven phrase structure grammar (HPSG) [29] is used. ViSiCAST uses feature structures to represent different aspects of individual signs. These include non-manual and grammatical aspects as well as any semantic roles the signs may play. The signs themselves are represented using the HamNoSys [3] notation.

The last stage of the ViSiCAST system generates a Signing Gesture Markup Language (SiGML) representation from the HPSG grammar used in stage three. This representation is then interpreted by a signing avatar [11] that signs the final translated output.

It is claimed that ViSiCAST could handle about 50% of the CMU link types [29]. This 50% includes a wide variety of linguistic phenomena, a list of which can be found in [29].

2.2.2 The TEAM Project

The TEAM (Translation from English to ASL by Machine) project was developed at the University of Pennsylvania [50] to translate English text into American Sign Language (ASL). English input is parsed into an intermediate representation which is then used by a sign synthesizer to generate the final avatar animation.

The intermediate representation of the translated sign language is a so-called gloss notation. With a gloss notation, each sign is represented by its nearest spoken counterpart [50]. The translation process from the English input to the gloss notation occurred within the framework of a synchronous tree adjoining grammar (STAG) (see Chapter 3).

In a STAG formalism, each word (both the English word and the sign gloss) has an elementary tree associated with it. The translation process starts by parsing the English input into a derivation tree. The derivation tree shows how each of the elementary trees was used to build the final parse tree. The translation process takes the derivation tree of the input and looks up all the gloss elementary trees in a word-gloss lexicon. A new derivation tree is then built, using the gloss elementary trees. This new derivation tree can be used to build a parse tree consisting of the glosses instead of the original English words.

The final parse tree is passed on to the sign synthesizer which animates the translated input. The avatar which signs the different signs in the output returns to a neutral position between signs. This looks unnatural, and therefore various smoothing techniques have been developed to smooth the transitions between signs. The TEAM project uses parallel transition networks (PaT-Nets) to solve this smoothing problem [50].

The TEAM project is no longer under development, and because of its prototype nature it only has a small number of demonstration sentences which it can translate [24]. However, research has

continued based on the TEAM project [24]. Our prototype system is based on the same principles as the TEAM project, in that a STAG grammar is used as the formalism in which translation takes place.

2.2.3 Thetos

Thetos (formerly TGT) is an experimental system for translating Polish into Polish Sign Language [43]. The Polish language presents interesting challenges to machine translation. One challenge is the fact that the Polish language has a free word order [44]. These challenges are reflected in the way that the Thetos system was designed.

The Thetos system starts by dividing any compound sentences in the input into separate primitive sentences. Morphological and syntax analysis is then performed on the individual primitive sentences. Syntax analysis involves building a structure of the sentence consisting of a set of syntactic groups (SG). Syntactic groups are sets of words in the sentence. The syntactic representation of the sentence is then represented as a graph (the so-called syntactic graph). Each node of this graph corresponds to a SG and the arcs of the graph represent the syntactic relationships between the different syntactic groups. The syntactic group at the root node of the graph is called a verb group. This is because each of the primitive sentences will always contain a verb (due to the nature of the primitive sentences).

The next phase in the system is to assign semantic roles to each node (or syntactic group) of the syntactic graph generated by the syntactic analyzer. Each syntactic group can only have one of three different types of semantic roles. These roles are the *Action*, *Agent* and *Object* roles. We now have a semantic representation of the primitive sentence with the root node of the graph corresponding to a predicate and the other syntactic groups in the graph corresponding to the arguments of that predicate.

Thetos uses a dictionary to translate the predicates and the arguments into the equivalent Polish Sign Language glosses. The semantic graph however remains unchanged. An output sentence is generated from the graph, which is then passed on to a signing avatar. The avatar interprets each gloss as a gesture, where each gesture is described by a so-called gestogram [19]. The gestograms drive the animation of the avatar.

The designers of Thetos mentions one specific problem which Thetos faces. This problem occurs when compound sentences are divided into primitive sentences. During the partitioning of sentences, reference information between created primitive sentences may be lost [43]. This problem arises because Thetos looks at individual sentences separately. Say we have the compound sentence, *John scored the goal and he was finally happy.* This sentence will be divided into the two primitive sentences, *John scored the goal.* and *He was finally happy.* Thetos will look at these two sentences independently, and therefore it will not know to what *He* refers to in the second sentence. Thus it can happen that the system can not fill in the argument(s) of a predicate in the semantic representation of a primitive sentence. Anaphora resolution techniques may help to solve this problem [43].

The designers of Thetos state that in order to obtain better results for primitive sentences, the number of semantic roles should be increased from its current number of three [43]. At the moment the Thetos system is restricted in that only primitive sentences and some compound sentences can be correctly translated.

2.2.4 Greek to GSL Converter

A Greek to Greek Sign Language (GSL) converter has been developed as part of the SYNENNOESE ('mutual understanding' in Greek) project. This project aims to develop an e-learning platform for Deaf Greek primary school pupils [35].

The translation of Greek to GSL starts off by first having an input sentence parsed into chunks. These chunks are small, grammatically correct phrases. However, they also have features associated with them for morphological and structural information.

These parsed chunks are passed to a Greek to GSL mapping module. This module is responsible for translating the chunks to equivalent GSL structures. The translation modules makes use of a hierarchy of rules to accomplish the translation. The first level of this hierarchy consists of structure rules. Structure rules perform actions on chunk sequences. These actions mainly perform re-ordering of the different chunks. The second level of the hierarchy consists of rules that perform actions on individual chunks. These rules are used for conditional addition or deletion of specific chunk entries. The last level consists of rules that perform actions on the features of specific chunks. These feature level rules are mainly used to add GSL features to individual chunks, as well as changing existing features to help with sign synthesis [20]. All the rules used by the Greek to GSL converter are specified in an XML format. This allows for easy modification and addition of rules.

The output of the mapping module is passed on to a GSL synthesis module. This synthesis module uses a GSL lexicon which contains all the necessary sign formations in the HamNoSys notation. The synthesis module also contains rules for generating core GSL sentences. These rules interact with the GSL lexicon and the output of the mapping module to generate the final control parameters for a virtual avatar, which performs the GSL sign [20].

The initial length of this project was 18 months. The developers felt that the Greek to GSL project would only be able to translate simpler examples during this phase [35].

2.3 Different Parsing Strategies

An important part of any transfer based machine translation system is parsing the input into an intermediate form. In this section we look at two of the most general parsing strategies used in various types of parsers. Chapter 3 discusses how these techniques are used in parsing TAGs.

Parsing a sentence aims to derive a structural description of the sentence within the framework of a grammar. One way to implement a parser would have been to simply use the grammar to generate all productions of the same length as the input sentence, and then to check whether the input is one of these productions. This approach however is extremely inefficient. A more efficient way is to enable the parser to check whether a given production is consistent with the input sequence. If it is not, the parser will then go back and choose the next valid production rule. This technique is called *back-tracking*. Two of the most common parsing strategies are the so-called top-down and bottom-up strategies. In the next section we describe these two strategies. We also look at how most parsers combine these two strategies to better parse natural languages. The first strategy that is discussed is top-down parsing.

2.3.1 Top-down Parsing

Top-down parsers attempt to build a parse tree from the root downwards [22]. This means that the parser will start with the start symbol of a grammar and follow all derivation rules from left to right in order to derive the input. In the next example the top-down parsing technique is illustrated. Suppose we have the following grammar:

S	$\rightarrow$	NP VP NP
NP	$\rightarrow$	N | DET N
VP	$\rightarrow$	V
DET	$\rightarrow$	*the*
N	$\rightarrow$	*John* | *goal* | *Mary* | *ball*
V	$\rightarrow$	*kicked* | *scored*

.

The terminal symbols are *the, John, goal, Mary, ball, kicked* and *scored*. Say we wish to parse the sentence *John scored the goal*. We start with the start symbol of the grammar and apply the first leftmost production to it:

1. S $\Rightarrow$ NP VP NP .

We now take the leftmost symbol and apply the leftmost production rule to it until we reach a terminal symbol:

2. NP VP NP ⇒ N VP NP
3. N VP NP ⇒ *John* VP NP .

Each time the derivation contains a terminal symbol we check whether that terminal symbol is consistent with the input sequence. In this case *John* is the correct first word of our input. We now apply the leftmost production to the second non-terminal symbol, and continue to do so, until we again obtain a terminal symbol:

4. *John* VP NP ⇒ *John* V NP
5. *John* V NP ⇒ *John kicked* NP .

Here we can see that *kicked* is not consistent with the input sequence. We now delete the last applied production rule and back-track to the next valid production rule. The last production rule that was applied, was $V \rightarrow$ *kicked*, and the next production rule that will be applied is $V \rightarrow$ *scored*:

6. *John kicked* NP ⇒ *John* V NP
7. *John* V NP ⇒ *John scored* NP
8. *John scored* NP ⇒ *John scored* N .
9. *John scored* N ⇒ *John scored John*

Again we see that the terminal symbol is not consistent with the input sequence, and so we again delete the last production and back-track to the next valid production:

10. *John scored* N ⇒ *John scored goal*
11. *John scored goal* ⇒ *John scored N*
12. *John scored* N ⇒ *John scored Mary* .
13. *John scored Mary* ⇒ *John scored* N
14. *John scored* N ⇒ *John scored ball*

We have now tried all the possible alternatives of this production rule without getting to the required sentence. Hence, we back-track still further. The final few derivations will look like this:

15. *John scored ball* ⇒ *John scored N*
16. *John scored* N ⇒ *John scored NP*
17. *John scored* NP ⇒ *John scored* DET N
18. *John scored* DET N ⇒ *John scored the* N .
19. *John scored the* N ⇒ *John scored the John*
20. *John scored the John* ⇒ *John scored the* N
21. *John scored the* N ⇒ *John scored the goal*

This then is the correct derivation for *John scored the goal*:

S ⇒ NP VP NP
⇒ N VP NP
⇒ *John* VP NP
⇒ *John scored* NP .
⇒ *John scored* DET N
⇒ *John scored the* N
⇒ *John scored the goal*

The final parse tree of this derivation is illustrated in Figure 2.2. The method of applying production rules until a terminal symbol is reached, is called a depth-first parsing strategy. Another method would have been to apply production rules, in left to right order, to each of the non-terminal symbols, until all non-terminal symbols are turned into terminal symbols. This method is called a breadth-first parsing strategy.

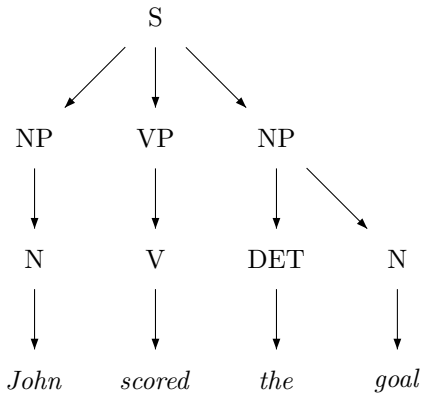


Figure 2.2: Top-down parse tree for *John scored the goal*.

Top-down parsing is 'predictive', in that the grammar tells us which productions should be applied next. The main disadvantages of top-down parsing include an inability to handle left recursion. A second disadvantage is the fact that back-tracking operations can be inefficient in that productions are made that do not lead to the correct parse. Another problem with top-down, back-tracking parsers is that a lot of duplicate work may also be done. In order to keep track of partial parses already matched, the parser may use a data structure called a chart. A chart consists of various items which represent partial parses of the input; thus items keep track of how far into the input productions have been applied. Items are added to the chart according to certain requirements. When parsing is finished, the chart contains items which represent all valid parses of an input sentence. A chart based top-down parser is generally referred to as an Earley type parser.

In the next section, bottom-up parsing is described.

2.3.2 Bottom-Up Parsing

Bottom-up parsers build parse trees from the leaves up to the root [22]. This means that the parser will start with the input and work its way back towards the start symbol. Bottom-up parsers apply production rules in right to left order. In the next example the same grammar and input is used as in the previous section.

Given the input sentence *John scored the goal*, the parser replaces *goal* with the nonterminal in the production rule that contains the terminal symbol *goal* on its right hand side.

1. *John scored the goal* $\Rightarrow$ *John scored the* N .

The parser now checks whether there is any rule with just an N on the right hand side. No such rule exists, so the parser continues on to the next input symbol:

2. *John scored the* N $\Rightarrow$ *John scored* DET N .

Again the parser checks for any production rule with DET N on the right hand side. Such a rule does exist, so the parser, replaces DET N:

3. *John scored* DET N $\Rightarrow$ *John scored* NP .

There is no rule with only NP on its right side, so the parser continues to the next input symbol:

4. *John scored* NP ⇒ *John* V NP .

There is no rule with V NP on its right side. However, there is a rule with V on its right side, and so the parser replaces V:

5. *John* V NP ⇒ *John* VP NP .

No production rule exists with VP NP on its right hand side, so the parser continues on to the next input symbol:

6. *John* VP NP ⇒ N VP NP .

The parser continues to search for valid production rules, and replaces N with NP:

7. N VP NP ⇒ NP VP NP .

When the parser checks for productions with NP VP NP on their right hand side, it will find that the only valid production is S → NP VP NP. This means that the start symbol has been reached and that the input is valid.

The parse tree generated by the bottom-up algorithm is shown in Figure 2.3. This particular

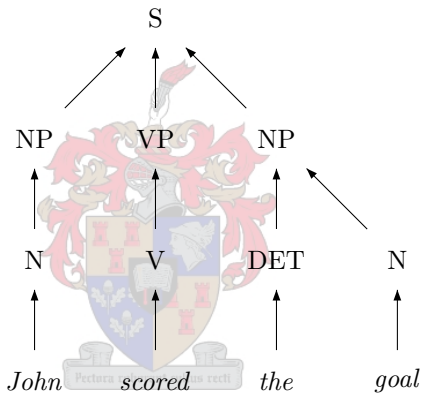


Figure 2.3: Bottom-up parse tree for *John scored the goal*.

derivation generated a right to left derivation, it is possible however to have bottom-up parsers that also generate left to right derivations (for example the well known CKY algorithm for context free grammars).

Bottom-up parsers are data driven. This means that, because bottom-up parsing starts with the input, any partial parses will be consistent with the input. Also, unlike top-down parsing, bottom-up parsing can cope with left recursion. However, bottom-up parsing can also be inefficient in that productions are considered that will not lead to the correct parse.

As mentioned, both top-down and bottom-up parsing have their advantages and disadvantages. The question now arises whether the advantages from both these parsing strategies can be combined to develop better parsers. One way in which this could be done is to use top-down predictive information as a filter on what production rules should be contemplated by a bottom-up component of a parser.

In the next chapter we will an implementation of a parser that uses a chart to keep track of all partial parses created during the parse. This parser uses top-down predictive information to predict which rules in the grammar can be used. It then uses the look ahead ability of bottom-up parsers to ensure that the parses generated are consistent with the input tokens expected.

In the next chapter TAGs are introduced. We consider different methods of parsing TAGs and how the previously described methods are used in those parsers. Finally, the implemented parsing algorithm used in our machine translation system are described in detail.

Chapter 3

Tree Adjoining Grammar Parsing

Tree adjoining grammars (TAGs) form the basis for the intermediate representations of the input and output in our machine translation system.

TAGs are well suited for natural language applications. The fact that TAGs are based on trees, make them a natural candidate for syntax representation of sentences. TAGs also have an extended domain of locality (EDL) over other context-free grammars (CFGs) or CFG based grammars (such as HPSG grammars). The second important property of TAGs is called factoring recursion from the domain of dependencies (FRD). These two properties are discussed in Section 3.1.3.

This chapter starts by giving a formal definition of TAGs. The second section describes the TAG parsing algorithm implemented for our prototype machine translation system. The last section gives an example of a parse using the parsing algorithm.

3.1 Formal Definition of Tree Adjoining Grammars

The following discussion of TAGs and their linguistic relevance is based on [28].

A TAG is a tree rewriting system. This means that a TAG manipulates trees instead of strings. The nodes of the trees used in a TAG are labeled by either terminal or non-terminal symbols. Usually, when using TAGs in a natural language context, it is required that each tree in the TAG has at least one leaf node labeled by a terminal symbol. This terminal symbol is usually an entry in a lexicon. The leaf node which is labeled by the lexicon entry is called the anchor node. All internal nodes in a tree are labeled by non-terminal symbols. The leaf nodes are labeled by either terminal or non-terminal symbols, and may be marked for substitution (denoted by $\downarrow$).

The TAG formalism distinguishes between two types of trees. These two types of trees are called *initial* and *auxiliary* trees. The only difference between the two types of trees, are that auxiliary trees can be used in the adjunction operation (see below). Auxiliary trees have the added restriction that one leaf node must have the same label as the root node of that auxiliary tree. This leaf node is called the foot node (denoted by $\star$).

There are two operations defined in the TAG formalism, namely, substitution and adjunction. Trees on which substitution or adjunction were performed, are called derived trees. Both these operations are explained in the next section.

A tree adjoining grammar can now be formally defined as the quintuple (Σ, NT, I, A, S). Here Σ denotes the set of all terminal symbols used by the trees in the TAG. Next, NT is the set of all non-terminal symbols ($\Sigma \cap NT = \emptyset$). I and A are the sets of initial and auxiliary trees respectively. Lastly, S is a special non-terminal symbol called the start symbol. An example of a small TAG is shown in Figure 3.1.

In the next section we briefly illustrate the two operations defined in the TAG formalism.

3.1.1 Substitution

The substitution operation substitutes a node in a given tree with another tree. Substitution can only be performed on nodes that are marked for substitution. Substitution can also only be performed if the label of the node to be substituted is the same as the label of the root node of the tree which is to replace it. Say we want to substitute tree β into tree α at node η. In order to perform the operation, node η must have the same label as the root node of β.

Figures 3.1, 3.2 and 3.3 illustrate the mechanism of substitution. Figure 3.1 shows an initial tree α with the anchor *met*, and two other initial trees β_1 and β_2. We want to substitute these two trees at the nodes marked for substitution in tree α. The substitution is permissible, because the root nodes of β_1 and β_2 are both labeled by *NP*. Figure 3.2 shows the actual substitution, and Figure 3.3 shows the final result after the substitution was performed on tree α.

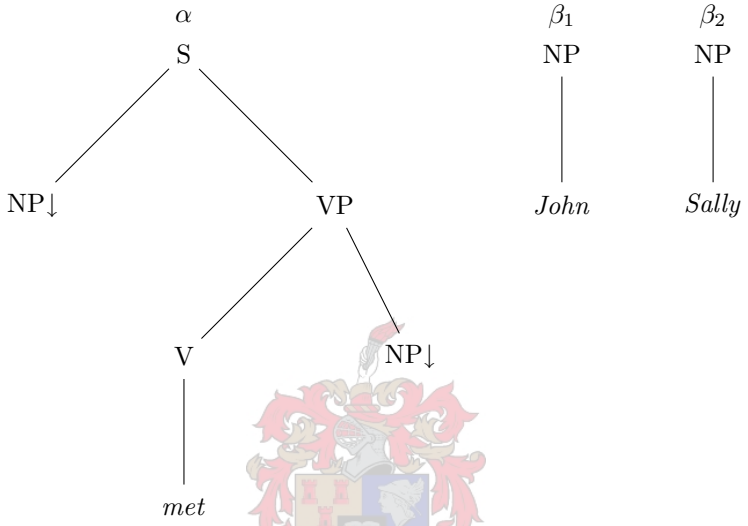


Figure 3.1: Initial tree α, anchored by *met* and trees β_1 and β_2.

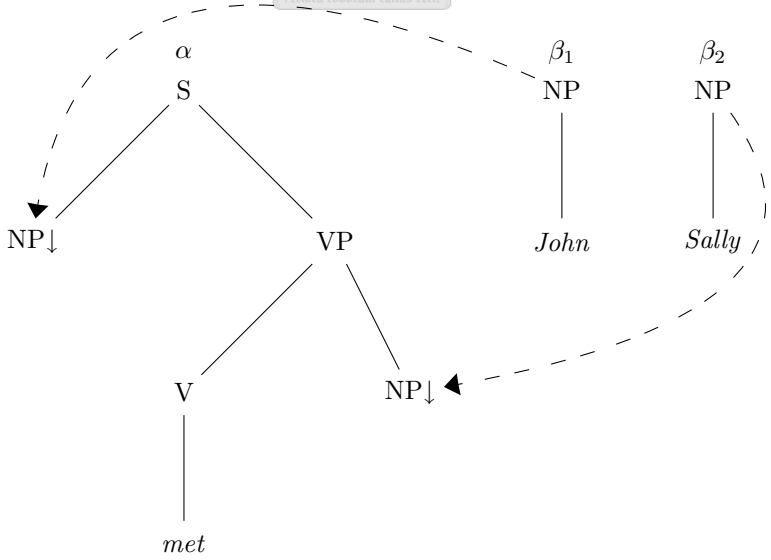


Figure 3.2: β_1 and β_2 are substituted into tree α.

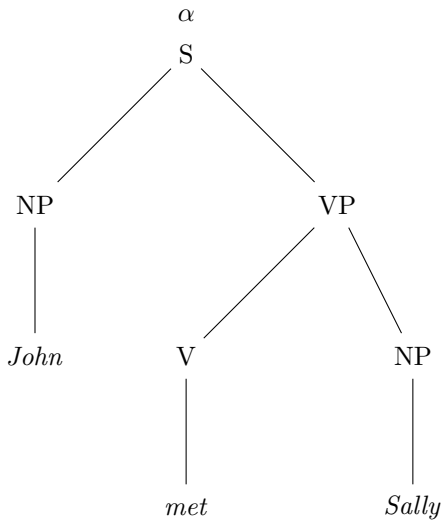


Figure 3.3: Result of substituting β_1 and β_2 into α.

3.1.2 Adjunction

The adjunction operation inserts an auxiliary tree (say β) into another tree (say α). Tree α can be any type of tree (initial, auxiliary or derived). This tree must contain a non-terminal node (call this node η), that has the same label as the root node of β. To perform adjunction we first detach the subtree in α rooted by η. We then attach the auxiliary tree β at the position where node η was previously. We now attach the subtree rooted by η at the foot node of β.

Figures 3.4 to 3.8 illustrate the adjunction operation. Figure 3.4 shows the initial tree α. The interior node labeled by *VP* is marked as node η. This is the node where adjunction will be performed. Figure 3.5 shows the auxiliary tree β. This tree can be adjoined at node η in α, because its root node and node η are both labeled by *VP*. Figure 3.6 shows the subtree (rooted by node η), that is detached from α. Figure 3.7 shows tree α after tree β was attached at node η. Finally, Figure 3.8 shows tree α after the subtree rooted by node η, was attached at the foot node of β to form the completed adjoined tree.

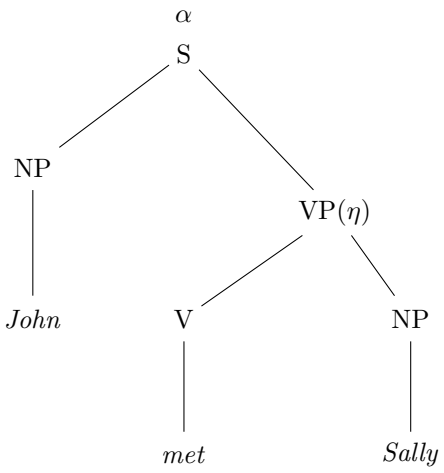


Figure 3.4: Initial tree α, with the VP node marked as node η.

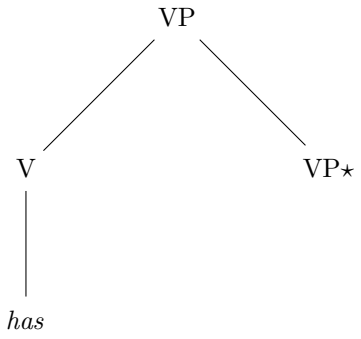


Figure 3.5: Auxiliary tree β.

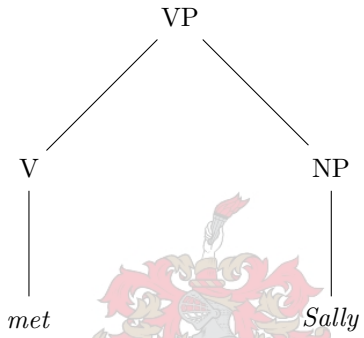


Figure 3.6: Subtree rooted at node η.

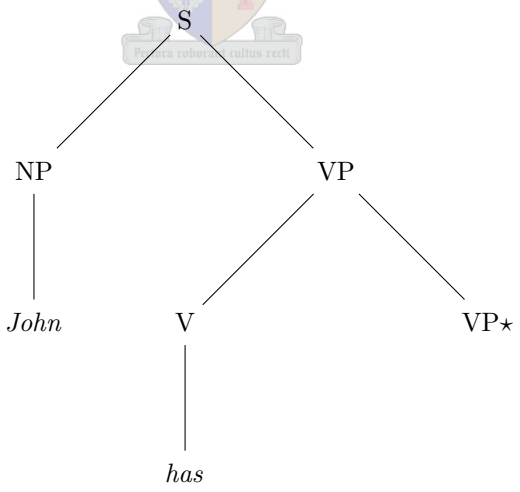


Figure 3.7: β inserted into α.

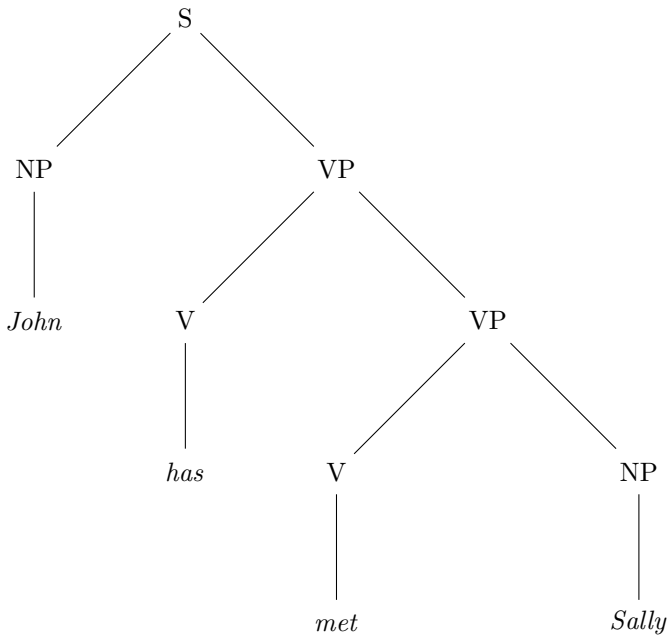


Figure 3.8: The final result obtained from adjoining β into α.

3.1.3 Linguistic Relevance of TAGs

In the previous section a formal definition of a TAG was given. In this section two properties of TAGs are described. These properties make them particularly useful for natural language applications [28]. As mentioned at the start of this chapter, the first property is the so-called extended domain of locality property of TAGs.

Extended Domain of Locality

To illustrate the extended domain of locality of TAGs over CFGs, we look at a small CFG.

S	$\rightarrow$	NP VP
VP	$\rightarrow$	VP ADV
VP	$\rightarrow$	V NP
NP	$\rightarrow$	*John* | *Mary*
V	$\rightarrow$	*kicked*
ADV	$\rightarrow$	*hard*

The verb *kicked* has two arguments (namely a subject and an object argument). The dependencies between the verb and its arguments are expressed through the two grammar rules:

S	$\rightarrow$	NP VP
VP	$\rightarrow$	V NP

.

Say we wish to describe this grammar more succinctly (the grammar must be able to generate exactly the same language, but with fewer grammar rules). One way could be to introduce one grammar rule for the two that are currently describing the arguments of the verb *kicked*. This grammar rule will look as follows:

S $\rightarrow$ NP V NP .

However, if this is done, then the VP node is lost. Note that this is the only way in which the two rules could be reduced. The two rules can not be replaced by a rule such as:

$$S \rightarrow NP\ VP\ NP \quad .$$

If it was, then the language generated by the CFG would also have changed. Therefore, the dependency between the verb and its arguments can not be expressed locally (with one rule).

A TAG, however, with its extended domain of locality, can express this dependency with only one grammar rule. This is illustrated in the TAG grammar in Figure 3.9.

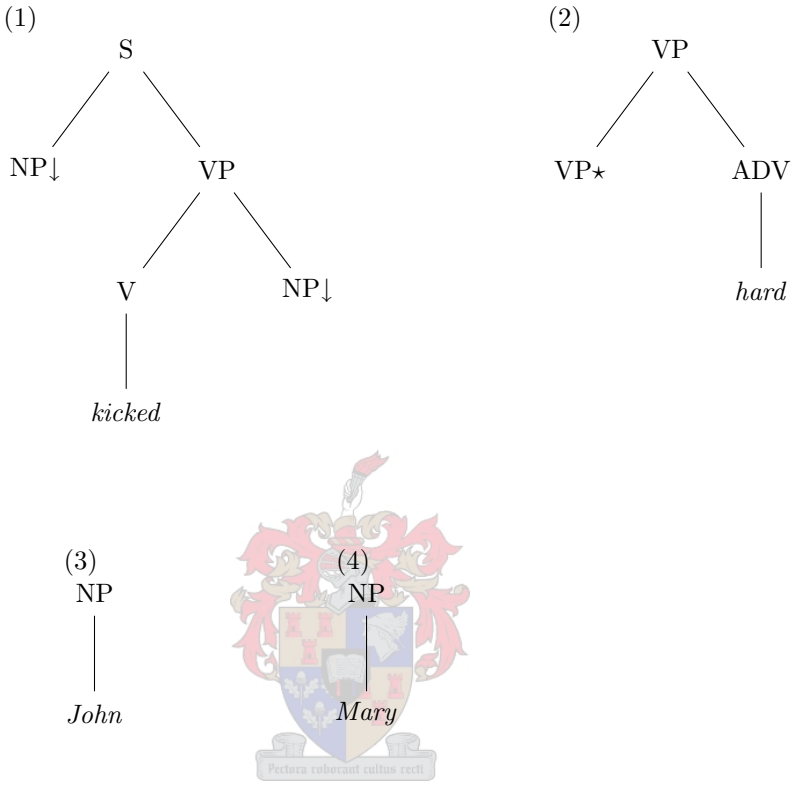


Figure 3.9: Example TAG grammar. The dependency between the verb and its arguments is now locally expressed in tree (1), without the loss of the VP node.

The next property we look at is the so-called factoring recursion from the domain of dependencies property of TAGs.

Factoring Recursion from the Domain of Dependencies

The tree structures of a TAG describes the dependencies between the constituents of a grammar. A constituent in linguistic theory is a unit of grammatical construction (for example a verb, noun phrase or clause). This means that the tree structures describe the relationships between noun phrases and verb phrases, and so on. Thus the elementary trees form the domain of dependency for a TAG grammar. For example, in Figure 3.10, there exists a dependency between the constituent *NP(wh)* and the verb *kick*. The adjunction operation however, allows this dependency to operate over longer distances, as illustrated in Figure 3.11. This is called factoring recursion from the domain of dependencies. Factoring recursion from the domain of dependencies, allow dependencies such as agreement and subcategorization to have long-distance behaviour [28].

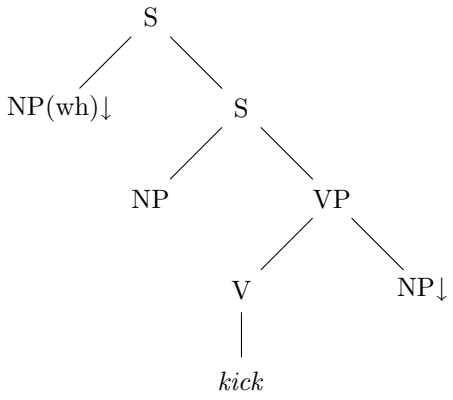


Figure 3.10: A TAG tree with a dependency between the *NP(wh)* constituent and the verb *kicked*.

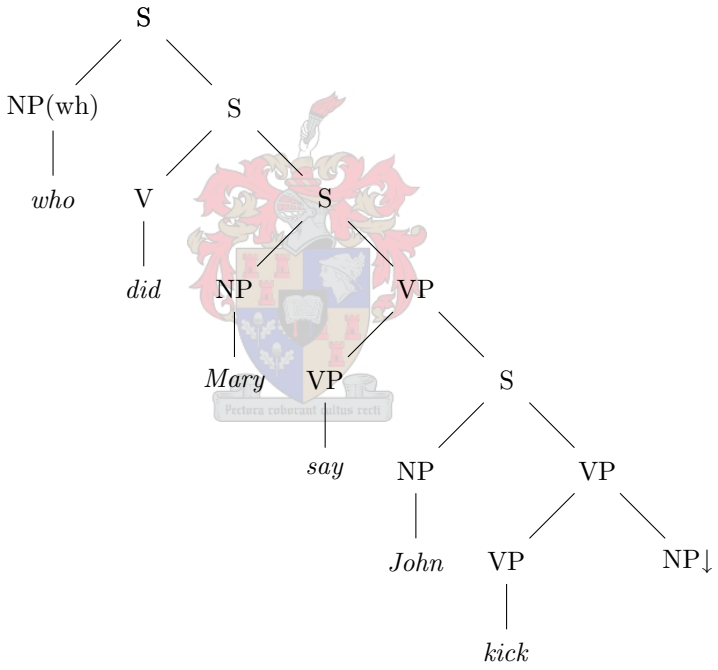


Figure 3.11: Movement of the verb *kick* away from the *NP(wh)* constituent.

These two facts allow TAGs to express all the elements of a particular syntactic or semantic feature of a lexical element within one elementary tree (elementary trees are sometimes called supertags in this context [40]). This is in contrast with grammars such as CFG grammars where we may need multiple rules to cover a single feature (syntactic or semantic). Therefore we can have a compact grammar in which each elementary tree corresponds to a specific feature (a CFG grammar would have had sets of rules corresponding to a feature). The lexicon employed by natural language systems using a TAG grammar will contain entries linking lexical entries with

particular elementary trees (and thus particular features) in the TAG grammar. All this makes the TAG formalism an appealing canditate for use in natural language processing tasks, such as machine translation.

In this section a definition for a TAG was given, and the two operations defined in the TAG formalism was described. Finally, the linguistic relevance of TAGs was briefly considered. In the next section, we turn to the problem of how TAGs can be used for machine translation.

3.1.4 Using Synchronous Tree Adjoining Grammars for Machine Translation

In this section we further explore the theoretical basis for using a TAG formalism in a syntactic transfer approach to machine translation. Shieber and Schabes present an extended TAG formalism with the stated goal of being used for natural language translation [38]. This extended TAG formalism is the so-called synchronous tree adjoining grammars (STAGs).

The definition of a STAG provides for two grammars, both defined in the TAG formalism. Usually, one grammar is called the source grammar, and the other, the target grammar. These two grammars are synchronised. This means that trees in the two grammars are paired up. The substitution and adjunction operations are then applied simultaneously to related nodes in these paired trees. Figure 3.12 shows an example of a tree pair in a STAG. The dotted lines denote the related nodes in the tree pair.

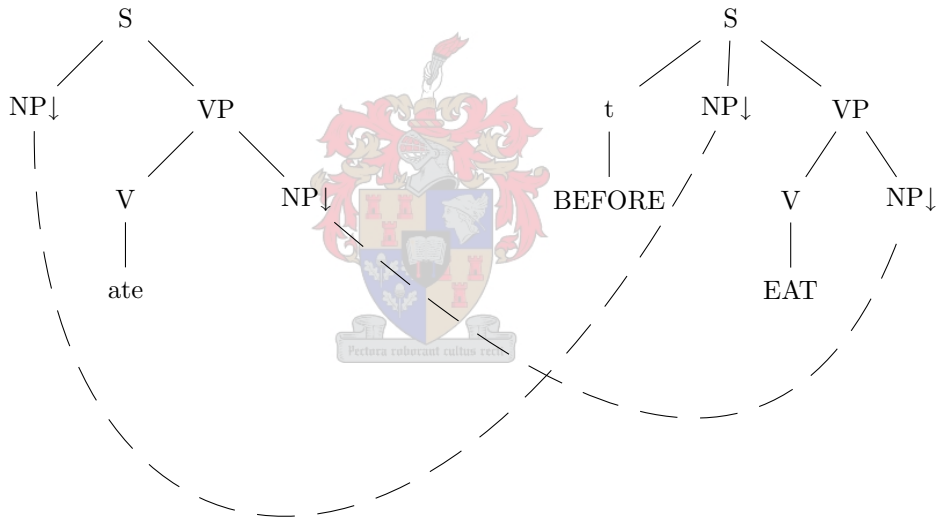


Figure 3.12: An example of a tree pair in a STAG.

In the formal definition of a STAG, given in [38], a synchronous TAG G is given by a set of triples, $\langle S_i, T_i, \frown_i \rangle$. In this definition S_i and T_i are elementary trees from TAG grammars G_S and G_T respectively. The links between the nodes of the two elementary trees are described by the linking relation $\frown_i$.

The derivation process in the STAG formalism begins by choosing an initial tree pair, $\langle I_S, I_T, \frown \rangle$, from the STAG grammar. Next a link $t_S \frown t_T$ between two nodes from the initial tree pair is chosen. The next step is to choose an auxiliary tree pair, $\langle A_S, AT, \frown' \rangle$, so that A_S can be adjoined to I_S at position t_S and A_T adjoined to I_T at position t_T. The adjunction operation is performed at the locations specified by t_S and t_T to create the current derived tree pair, $\langle I_S[A_S/t_S], I_T[A_T/t_T], \frown'' \rangle$. The new linking relation $\frown''$ is defined as all the links described by

$\frown$ (minus the link just chosen, namely $t_S \frown t_T$) along with all the links described by $\frown'$. The process gets repeated as a new link is chosen and adjunction is performed on the current derived pair.

In [36], Shieber described two problems with this original definition for derivation in the STAG formalism. These problems necessitates an alternative definition for derivation. The first problem is one of expressivity. It is possible for STAGs defined as above to recognize non-tree adjoining languages [36].

The second problem is one of implementability. The original definition provides for no practical way in which derivation as defined in the STAG formalism can be implemented. When a STAG is used for transduction [1] the input string w_S is parsed relative to grammar G_S to yield derivation D_S. The STAG derivation D_{STAG} must then somehow be reconstructed from derivation D_S. This new derivation D_{STAG} is then used to yield the output w_T. This transduction process can be illustrated as follows:

$$w_S \rightarrow D_S \rightarrow D_{STAG} \rightarrow w_T \quad .$$

However, the synchronous derivation D_{STAG} is not related to the standard definition for derivations in TAGs. Therefore, if a standard TAG parser is used for the first half of the process (obtaining D_S) it is not clear how to implement the entire process using the standard TAG definition.

An alternative definition proposed by Shieber in [36, 37] calls for a more natural definition of derivation in STAGs. In this definition a derivation is defined as a pair $\langle D_S, D_T \rangle$, where D_S is a derivation tree relative to grammar G_S and D_T is a derivation tree relative to grammar G_T. These two derivation trees are required to be isomorphic. This means that there is a one-to-one mapping of nodes from D_S onto D_T. Furthermore, this mapping (say f) must preserve the dominance of the nodes in the two derivation trees. The dominance of the nodes are preserved when, if $f(\eta_S) = \eta_T$ we have $f(parent(\eta_S)) = parent(\eta_T)$ (here η_S and η_T are nodes from the D_S and D_T respectively, and $parent(\eta)$ denotes the parent of node η).

This more natural definition of derivation within the STAG formalism overcomes the problems highlighted in [36]. Firstly, this new definition ensures that only tree adjoining languages can be expressed through the STAG formalism. Secondly, this definition provides for a more intuitive way of implementing transduction through STAGs. As previously mentioned the steps taken during transduction are as follows:

$$w_S \rightarrow D_S \rightarrow D_{STAG} \rightarrow w_T \quad .$$

With D_{STAG} not well defined. However, in the new alternative definition, D_{STAG} simply becomes $\langle D_S, D_T \rangle$. Thus the entire transduction process will now appear as follows:

$$w_S \rightarrow D_S \rightarrow \langle D_S, D_T \rangle \rightarrow D_T \rightarrow w_T \quad .$$

The step $\langle D_S, D_T \rangle \rightarrow D_T$ is thanks to the closer relationship between the new definition for derivation in STAGs and the original definition for derivation in TAGs.

The STAG formalism employed by our translation system is equivalent to this alternative definition. We further describe our implementation of the STAG formalism in Chapter 4.

In the next section we look at parsing TAGs.

3.2 Parsing TAGs

Various TAG parsing algorithms have been proposed in the literature. The earliest TAG parsing algorithm uses dynamic programming techniques in conjunction with a bottom-up approach to achieve an $O(n^6)$ running time [10, 28], where n is the length of the input. However, this algorithm is not practical for real world applications, as it assumes that TAG trees are only binary branching trees [28].

[1] A process whereby an input string w_S is transformed into an output string w_T

Van Noord [46] describes a head-corner parsing algorithm for TAGs. This algorithm does not parse from left to right, but rather from the head of a phrase outwards in both directions. Other parsing algorithms for TAGs include [16, 31]. These algorithms are important in that they ensure that the valid prefix property for TAGs holds. The valid-prefix property is described in Section 3.2.3.

The parsing algorithm used in our machine translation system is proposed by Schabes in [28]. This is the parsing algorithm that will be discussed in this section. This algorithm was deemed sufficient to be a good starting point for our system. One reason for this, is that the algorithm has already been used successfully in a translation project (this algorithm was implemented as part of the XTAG system, and used in the TEAM project). The second reason is that the algorithm is well described, making it quicker and easier to implement.

However, this algorithm does have some drawbacks, for example it does not incorporate the valid prefix property. This shortcoming does not effect the worst-case running time of this algorithm (both the Schabes and van Noord algorithms have a worst-case running time of $O(n^6)$). It remains unclear however, how the two the average running times of the two algorithms compare. Implementing the van Noord algorithm and comparing it to the Schabes algorithm is a topic of future study.

The Schabes algorithm is an Earley type algorithm, in that it uses both top-down and bottom-up information for parsing input [10]. This algorithm also incorporates dynamic programming techniques in the form of a so-called chart. Recall from Chapter 2, that a chart keeps track of all partial parses already completed. Through this chart, the parsing algorithm avoids duplicating work already done.

Top-down information is used in a phase called the predict phase. The predict phase is responsible for adding new items to the chart according to the rules of the grammar. The bottom-up strategy is used in a phase called the complete phase. The complete phase works with trees already represented in the chart and seeks to work its way up these trees.

The algorithm scans an input string from left to right. While doing this, the algorithm tries to recognize adjunction (on interior nodes) and substitution (on leaf nodes). In order to achieve this an initial tree is traversed and all possible trees that could have been either adjoined or substituted to that tree are considered. If a tree is found that could have been adjoined or substituted, it in turn is traversed. When a node labeled with a terminal symbol is reached and this symbol is not the expected token from the input string, the traversal of that particular tree is stopped. The traversal of the trees are done through the notion of dotted trees.

Dotted trees are used to keep track of the current position in a tree which is being traversed. A dotted tree consists of a tree, along with the position of the current node in the tree. It also has a *dot* which indicates a relative position to the current node. Possible dot positions are *left above (la)*, *left below (lb)*, *right below (rb)* and *right above (ra)*. The dot always starts at *la* of the root node. Traversal of the tree is considered to be complete when the dot is at position *ra* of the root node. The order of traversal through a tree as used in the parsing algorithm can be defined as follows:

- If the position of the dot is *la* of an internal node, move the dot down to position *lb*.
- If the position of the dot is *lb* of an internal node, move the dot to position *la* of the node's leftmost child.
- If the position of the dot is *la* of a leaf node, move the dot to position *ra* of the leaf node.
- If the position of the dot is *rb* of a node, move the dot to position *ra* of the node.
- If the position of the dot is *ra* of a node and the node has a right sibling, then move the dot to position *la* of the right sibling.
- If the position of the dot is *ra* of a node and the node does not have a right sibling, then move the dot to position *rb* of the node's parent.

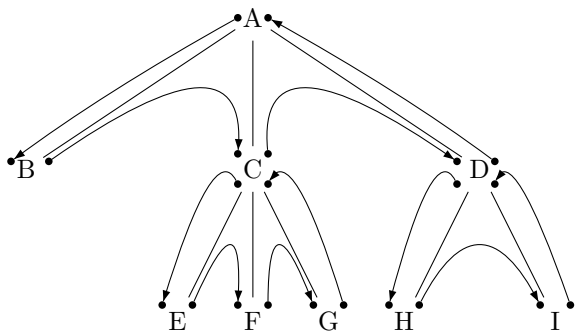


Figure 3.13: An example of tree traversal.

This traversal is illustrated in Figure 3.13.

The algorithm starts with all possible initial trees. It then proceeds to attempt to and traverse all these initial trees. At appropriate nodes, the algorithm tries to adjoin (or substitute) trees. If a tree can be adjoined or substituted in the original initial tree, this new tree will be traversed. If the traversal of this new tree is successful, then the algorithm returns, and continues to traverse the original initial tree. If a complete traversal was successful, then the input was recognized. If the input is recognized, a derivation tree can be built for the successful parse. Once the derivation tree is finished, then parsing is also complete. If the traversal was unsuccessful, the input is not recognized.

The parsing algorithm is a chart based algorithm. This means that the algorithm adds items to a chart, depending on items already in the chart. The chart items used in this parsing algorithm can be defined as being of the form $[\alpha, dot, pos, i, j, k, l, sat?]$. α is a tree from $I \cup A$. The address of the current node in the tree is denoted by *dot*. The relative position of the dot to the current node is given by *pos*. The addresses of the nodes are given by Gorn addresses. Figure 3.14 gives examples of Gorn addresses. Gorn addressing uses a breadth-first numbering of the nodes. For example, if the address of the parent is *21*, then the address of the leftmost child will be *211*. The root node is considered to have address *0*. The *0* at the start of the addresses is not explicitly written except for the root node.

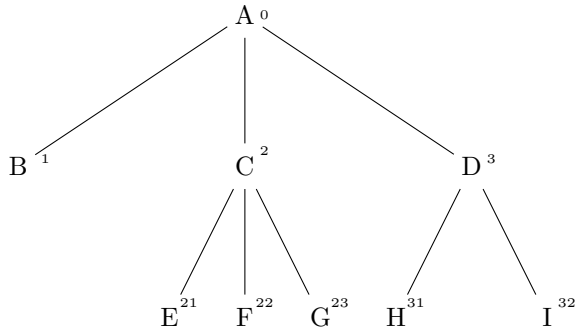


Figure 3.14: Gorn address example (root node = 0).

i, j, k, l are indices used to keep track of positions in the input string. Indices j and k are used in chart items which contain auxiliary trees. They are used to show where adjunction is performed in the input. If no adjunction has yet been attempted in an item, then j and k are unbound (in

which case the value − are assigned to them). The *sat*? value is a boolean value which indicates if an adjunction was recognized on the node at address *dot* in tree α.

The parsing algorithm consists of seven operations. The INITIALIZATION operation is performed once to initialize the chart. The next six operations is applied to all the items in the chart. This in turn adds more items to the chart. The operations continue to be applied to items until no more items are added to the chart.

3.2.1 The INITIALIZATION Operation

Assume the input string is $a_1, a_2, ..., a_n$. Assume also that $G = (\Sigma, NT, I, A, S)$ is a TAG. The initialization operation adds the initial items to the chart. These items are of the form $c = [\alpha, 0, la, 0, -, -, 0, false]$. The trees in these items are all initial trees with S as root label. The items all have the dot position at *la* of the root node, indicating that these are the trees with which the traversals will start. The i and j indices are both 0, this indicates that we are at position 0 in the input string. The j and k indices are both unbounded to indicate the no adjunction has yet been performed on this tree.

The next operation is the SCAN operation.

3.2.2 The SCAN Operation

The SCAN operation attempts to recognize tokens from the input string. The SCAN operation is applied to items in the chart where the dotted node is at position *la* of a node labeled by a terminal symbol. If the label of the node matches the expected token, the l index is increased by one, and the dot moves to position *ra*. If the label of the node is the empty string (ϵ), then the dot automatically moves to position *ra*. In this case, however, the l index does not get increased (because no new token has been matched). Therefore if the current item is $[\alpha, dot, la, i, j, k, l, false]$ and the label is the next expected token, then $[\alpha, dot, ra, i, j, k, l + 1, false]$ is added to the chart. If, however, the label is the empty string, then $[\alpha, dot, ra, i, j, k, l, false]$ will be added.

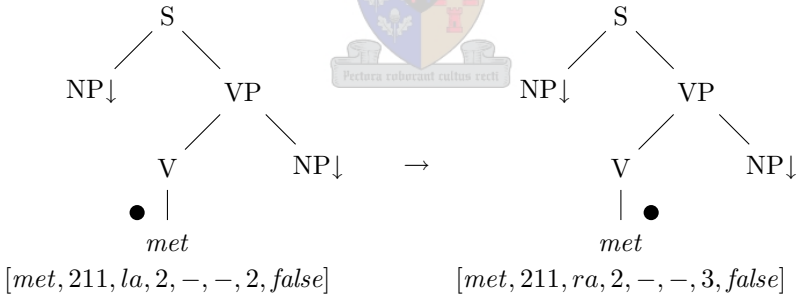


Figure 3.15: Example of the SCAN operation, where the input symbol is *met*. The left hand tree represents an item with the dot at the position shown. To the right of the arrow, the tree with the new item added to the chart is shown.

Figure 3.15 contains an item with the dotted tree to the left and above of the node labeled by *met*. The token obtained from the input is *met*. Since the label of the node is the same as the expected token, the token is recognized. The dot is moved to the right side of the node labeled by *met* to show that the token was recognized.

3.2.3 The PREDICT Operation

The PREDICT operation adds new trees to traverse to the chart. This is done by creating new chart items based upon what trees may be adjoined at the node position of the current chart item.

This operation considers three different cases. The first case is when the dot is at position *la* of a node labeled by a non-terminal symbol. Here the operation will add all trees that are adjoinable at the dotted node. If the current item is $[\alpha, dot, la, i, j, k, l, false]$, then items of the form $[\beta, 0, la, l, -, -, l, false]$ will be added. Here β represents all the auxiliary trees that can be adjoined to α at position *dot*. The *dot* position of 0 in the new item indicates that this is a new tree that needs to be traversed from the root. The fact that both the i and l indices of the new item equals the l index of the previous item, indicates that span of the new item starts at index position l in the input string.

The second case is also when the dot is at position *la* of a node labeled by a non-terminal symbol. In this case we try to recognize the tree without any adjunction by simply moving the dot further down the current tree. Therefore, if the current item is $[\alpha, dot, la, i, j, k, l, false]$, then $[\alpha, dot, lb, l, -, -, l, false]$ will be added to the chart.

The last case which is considered is if the dot is at position *lb* of a foot node of an auxiliary tree. The algorithm considers all chart items with nodes where the tree of the current item could have been adjoined. It then adds items to try and recognize the subtree below each node. Let the current item be $[\alpha, dot, lb, l, -, -, l, false]$ with the dot position corresponding to the foot node of α. Then items of the form $[\delta, dot', la, l, -, -, l, false]$ will be added. This new item indicates that tree α could possibly have been adjoined at position dot' in tree δ.

It is important to note that, when the third case is applied, some of the items added will not be compatible with the left context of the input already seen in the algorithm. This compatibility with the left context is called the valid prefix property [32], and the parser algorithm violates this property. This means that there are some unnecessary items that are added to the chart. Although these items will be culled from the chart during the complete phase, they still have to be considered during subsequent operations, and thus a penalty is incurred for violating the valid prefix property.

In order to avoid this property more complex data structures are needed [28]. There are other TAG parsers that do not violate this property; for example the left corner parser of Daz [16]. It remains a topic for future work to see whether any increase in performance can be obtained from implementing a parser which does not violate the valid prefix property.

The next operation is the COMPLETE operation.

3.2.4 The COMPLETE Operation

As mentioned previously, chart items represent partial parses of the input. The COMPLETE operation combines items in the chart to form new items which represent larger partial parses. The COMPLETE operation also removes unnecessary items that were introduced into the chart due to the violation of the valid prefix property.

Like the PREDICT operation, the COMPLETE operation considers three different cases. The first case considers whether the next token may come from a tree that was adjoined to the current tree. Suppose the current item is $[\alpha, dot, rb, i, j, k, l, false]$ and there exists another item $[\beta, dot', lb, i, -, -, i, false]$ where the dot' indicates the position of the foot node of β. Then, if β is adjoinable at position *dot* in tree α, the item $[\beta, dot', rb, i, i, l, l, false]$ is added to the chart. The j and k indices of the new item equals the i and l indices of the first item, this indicates that the adjunction was performed somewhere between positions i and l of the input.

The second case considers whether the next token comes from the same tree as that of the current item. This case only applies to items where one of the children of the dotted node is the foot node. This is indicated by indices j and k being set. If the current item is $[\alpha, dot, rb, i, j, k, l, \text{sat?}]$ and there is another item of the form $[\alpha, dot, lb, h, -, -, i, false]$, then the item $[\alpha, dot, ra, h, j, k, l, false]$ is added to the chart. The fact the index i of the new item equals h indicates that the new item spans a larger part of the input string.

The third case also considers whether the next token comes from the same tree as that of the current item. However, it only applies when the foot node is not among the children of the dotted node. This is for cases where no adjunction was performed in the subtree rooted by the current node. This is indicated by indices j and k not being set. Therefore, if the current item is $[\alpha, dot, rb, i, -, -, l, \text{sat?}]$ and there is another item of the form $[\alpha, dot, la, h, j, k, i, false]$, then the item $[\alpha, dot, ra, h, j, k, l, false]$ is added to the chart.

The COMPLETE operation adds items to the chart that indicates that the traversal of any subtrees rooted at a particular node is now complete.

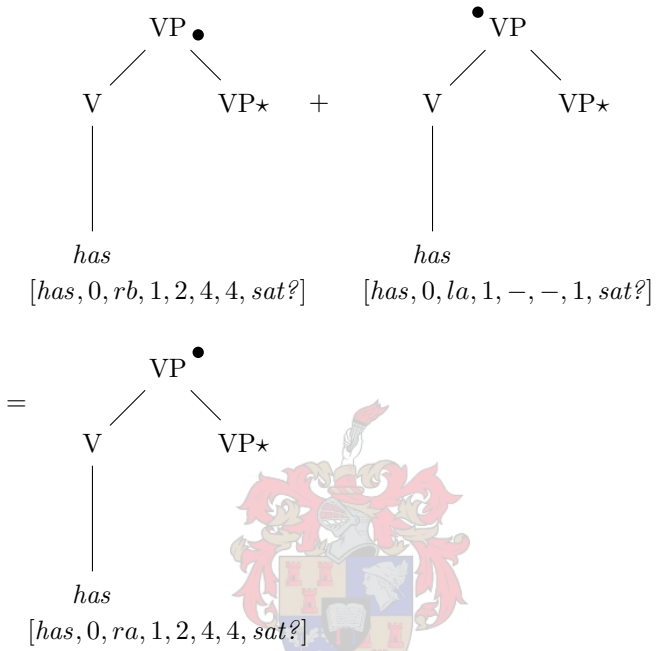


Figure 3.16: Example of the second case considered by the COMPLETE operation. Note that the *sat?* value can be either true or false, since the COMPLETE operation may be performed on items which have their *sat?* value set to true.

Figure 3.16 shows one chart item where the dot is to the left and above the root node of the tree. Another item, with the same tree, has the dot to the right and below the root node. The second item would only have been added to the chart if *has* was recognized during the SCAN operation and if all possible adjoinable trees have been traversed. Therefore a new chart item, with the dot to the right and above the root node of the tree, is added. This shows that this tree, and all possible subtrees, have been fully traversed.

3.2.5 The ADJOIN Operation

The ADJOIN operation recognizes when adjunction was performed on a specific node and adds items which span bigger portions of the input. The new item that is added will have its *sat*? set to *true* to indicate that adjunction was recognized.

The ADJOIN operation takes an item of the form $[\alpha, dot, rb, j, p, q, k, false]$ and an item of the form $[\beta, 0, ra, i, j, k, l, false]$ to form the new item, $[\alpha, dot, rb, i, p, q, l, \text{true}]$. Figure 3.17 shows two chart items with different trees on which the ADJOIN operation is performed. The dot of the first item is to the right and above the root node of an auxiliary tree. This means that any possible subtrees have already been traversed. The dot of the second item is to the right and below a node where adjunction can be performed. This means the input token *met* has already been recognized.

Furthermore, the j index of the first item is the same as the i index of the second item. This means that the auxiliary tree of the first item contains the node with the same label as the token preceding the *met* token. Thus a new item is added to the chart to indicate that adjunction was performed.

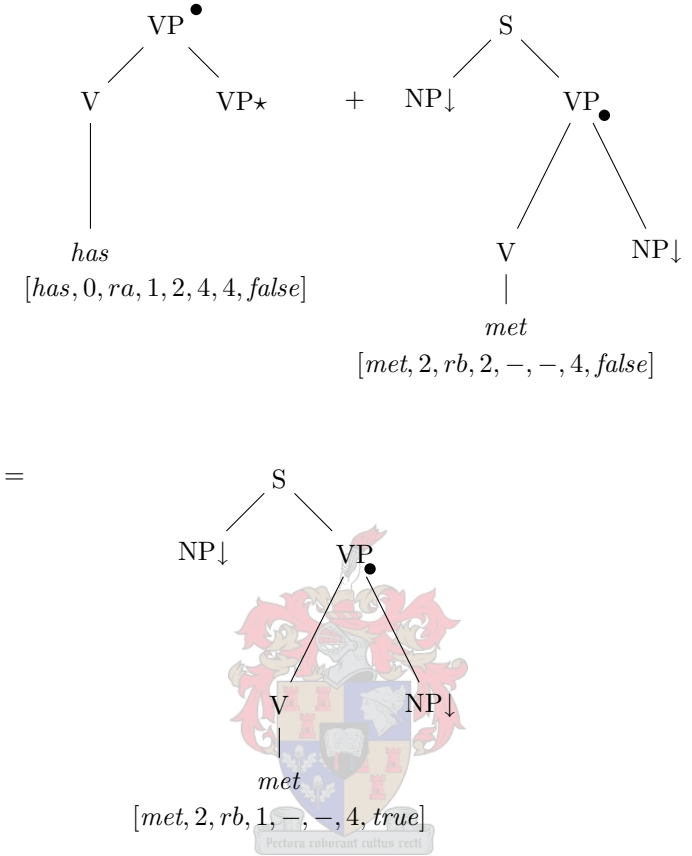


Figure 3.17: Example of the ADJOIN operation.

3.2.6 The PREDICT SUBSTITUTION Operation

The PREDICT SUBSTITUTION operation predicts all possible initial trees that can be substituted at a node marked for substitution.

If the current item is of the form $[\alpha, dot, la, i, j, k, l, \text{sat?}]$ and the node at address *dot* is marked for substitution then items of the form $[\alpha', 0, la, l, -, -, l, false]$ are added to the chart. Here α' is an initial tree that can be substituted at the dotted node in α. This tree will also be traversed from the root node (indicated by the *dot* position being 0), and we are at position l in the input string (indicated by indices i and l of the new item being l).

3.2.7 The COMPLETE SUBSTITUTION Operation

This operation is similar to the ADJOIN operation. It also adds new items to the chart by combining other items. If the current item is of the form $[\alpha, 0, ra, l-, -, m, false]$ and there exists an item of the form $[\alpha', dot', la, i, j, k, l, \text{sat?}]$, where the α' can be substituted into α, we add the item $[\alpha', dot', ra, i, j, k, m, \text{sat?}]$. This new item now spans a larger part of the input (from position i to position m).

The algorithm is finished when no more items can be added to the chart. If there is an item of the form $[\alpha, 0, ra, 0, -, -, n, false]$ in the chart, with α an initial tree and with the label of the root node S (the start symbol), then the input was recognized. This item represents an initial tree which was fully traversed, and the item also spans the entire input (from position 0 to position n). Thus the parse was successful. Otherwise, if no such item exists, the parse fails.

This section described the parsing algorithm; in the next section an example of parsing is given.

3.3 Example of Parsing

In this section an example of parsing according to the algorithm described in Section 3.2 is presented. Figures 3.18 and 3.19 show the elementary trees of the TAG used in the example. The input string that must be recognized is *Mary has met John*.

Figure 3.18: The elementary trees, *John* and *Mary*, of the example TAG.

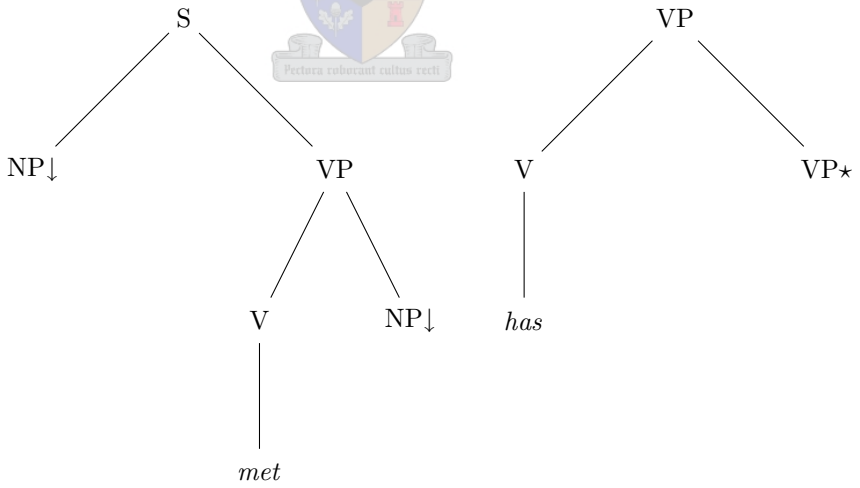


Figure 3.19: The elementary trees, *met* and *has* of the example TAG.

The algorithm starts off by first creating items during the initialization phase. Recall that the

initialization operation adds all initial trees with root nodes labeled by the start symbol S. There is only one such tree in the example grammar, and therefore the first item added to the chart is,

1.	$[met, 0, la, 0, -, -, 0, false]$	(INIT)

.

In the round brackets after the chart item, the reason for the inclusion of that item is given. First the operation that added the item to the chart is given, and if applicable the combination of chart items that led to item being added. In the case of the PREDICT and COMPLETE operations, the specific case of the operation which resulted in the item being added to the chart is also shown.

We now apply all the operations on this item in turn. The only operation that adds a new item to the chart is the PREDICT operation. The chart now appears as follows:

1.	$[met, 0, la, 0, -, -, 0, false]$	(INIT)
2.	$[met, 1, la, 0, -, -, 0, false]$	(PREDICT,1,2nd case)

.

All the operations are now applied to item number 2. Two items are added to the chart during the PREDICT SUBSTITUTION operation, because position 1 is a NP node. These items are:

3.	$[John, 0, la, 0, -, -, 0, false]$	(PREDICT SUBSTITUTION,2)
4.	$[Mary, 0, la, 0, -, -, 0, false]$	(PREDICT SUBSTITUTION,2)

.

All the operations will be applied to these two new items in turn. During the next few phases of the algorithm, the PREDICT operation continues to move down through the respective trees that are represented by the items in the chart. This is reflected by the items added during these few phases:

5.	$[John, 1, la, 0, -, -, 0, false]$	(PREDICT,3,2nd case)
6.	$[Mary, 1, la, 0, -, -, 0, false]$	(PREDICT,4,2nd case)
7.	$[John, 11, la, 0, -, -, 0, false]$	(PREDICT,5,2nd case)
8.	$[Mary, 11, la, 0, -, -, 0, false]$	(PREDICT,6,2nd case)

.

When the algorithm applies the SCAN operation to item number 8, an input symbol is recognized. Thus another item is added to the chart:

9.	$[Mary, 1, rb, 0, -, -, 0, false]$	(SCAN,8)

.

When the COMPLETE operation is performed on item number 9, a new item is added to the chart:

10.	$[Mary, 0, rb, 0, -, -, 1, false]$	(COMPLETE,6+9,3rd case)

.

And again when the COMPLETE operation is performed on item number 10, a new item is added to the chart:

11.	$[Mary, 0, ra, 0, -, -, 0, false]$	(COMPLETE,4+10,3rd case)

.

Item number 11, indicates that the entire tree *Mary* has been traversed. When the PREDICT SUBSTITUTION operation is performed on this item, we recognize that substitution was performed on the *met* tree. This is represented by adding the following item to the chart:

12.	$[met, 1, ra, 0, -, -, 0, false]$	(COMPLETE SUBSTITUTION,2+11)

.

Through the next few phases of the algorithm, the algorithm continues to traverse through the trees represented in the chart. During these few phases the following items are added to the chart:

13.	$[has, 0, la, 1, -, -, 1, false]$	(PREDICT,12,1st case)
14.	$[met, 21, la, 1-, -, 1, false]$	(PREDICT,12,2nd case)
15.	$[has, 1, la, 1, -, -, 1, false]$	(PREDICT,13,2nd case)
16.	$[has, 11, la, 1, -, -, 1, false]$	(PREDICT,15,2nd case)

.

The position of the dot in item number 16 is to the left and above a node labeled *has*. When the SCAN operation is performed on this item, another input token is recognized, and the following item is added to the chart:

17.	$[has, 1, rb, 1, -, -, 2, false]$	(SCAN,16)

.

When the COMPLETE operation is performed on item number 17, the following item is added to the chart:

18.	$[has, 2, la, 1, -, -, 2, false]$	(COMPLETE,15+17,3rd case)

.

Again during the following number of phases the algorithm traverses through all the trees represented in the chart. The items added during these phases are:

19.	$[has, 2, lb, 2, -, -, 2, false]$	(PREDICT,18,2nd case)
20.	$[has, 1, la, 2-, -, 2, false]$	(PREDICT,18,1st case)
21.	$[met, 21, la, 2, -, -, 2, false]$	(PREDICT,18,3rd case)
22.	$[has, 0, la, 2, -, -, 2, false]$	(PREDICT,21,1st case)
23.	$[has, 11, la, 2, -, -, 2, false]$	(PREDICT,21,3rd case)
24.	$[met, 211, la, 2, -, -, 2, false]$	(PREDICT,21,2nd case)
25.	$[has, 11, lb, 2, -, -, 2, false]$	(PREDICT,23,2nd case)

.

The dot in item number 24 is again to the left and above a node labeled by a terminal symbol. Again an input symbol is recognized by adding the following item to the chart:

26.	$[met, 21, rb, 2, -, -, 3, false]$	(SCAN,24)

.

When the COMPLETE operation is performed on item number 26, the following item is added to the chart:

27.	$[met, 22, la, 2, -, -, 3, false]$	(COMPLETE,21+26,2nd case)

.

Because the dotted node in item number 27 is labeled by *NP*, the PREDICT SUBSTITUTION operation adds two more items to the chart:

28.	$[John, 0, la, 3, -, -, 3, false]$	(PREDICT SUBSTITUTION,27)
29.	$[Mary, 0, la, 3, -, -, 3, false]$	(PREDICT SUBSTITUTION,27)

.

During the next two phases of the algorithm, the algorithm will again traverse down the *John* and *Mary* trees, by adding the following items to the chart:

30.	$[John, 1, la, 3, -, -, 3, false]$	(PREDICT,28,2nd case)
31.	$[Mary, 1, la, 3, -, -, 3, false]$	(PREDICT,29,2nd case)

.

The SCAN operation recognizes an input symbol on item number 30. This is represented by the addition of the following item to the chart:

32.	$[John, 0, rb, 3, -, -, 3, false]$	(SCAN,30)

.

Again, when the COMPLETE operation is performed on item number 32, the algorithm traverses back up the tree by adding the following item to the chart:

33.	$[John, 0, ra, 3, -, -, 4, false]$	(COMPLETE,28+32,3rd case)

.

The COMPLETE SUBSTITUTION operation adds a new item to the chart when it is performed on item number 33. The new item looks as follows:

34.	$[met, 22, ra, 2, -, -, 4, false]$	(COMPLETE SUBSTITUTION,27+33)

.

During the next few phases, the algorithm traverses back up the *has* tree by successfully performing the COMPLETE operation on item number 34, and the subsequent items added to the chart. These items are added as follows:

35.	$[has, 2, rb, 2, 2, 4, 4, false]$	(COMPLETE,20+34,2nd case)
36.	$[has, 0, rb, 1, 2, 4, 4, false]$	(COMPLETE,18+35,2nd case)
37.	$[has, 0, ra, 1, 2, 4, 4, false]$	(COMPLETE,13+36,2nd case)

.

When the ADJOIN operation is performed on item number 37, it recognizes that adjunction was performed using the *has* tree. The operation also recognizes that adjunction was performed on the *met* tree. This is represented by the following item:

$$\boxed{38. \quad [met, 2, rb, 1, -, -, 4, true] \quad \text{(ADJOIN,36+37)}} \quad .$$

The algorithm now continues to traverse back up the *met* tree. When the COMPLETE operation is performed on item number 38, the following item is added to the chart:

$$\boxed{39. \quad [met, 0, rb, 0, -, -, 4, true] \quad \text{(COMPLETE,12+38,3rd case)}} \quad .$$

When the COMPLETE operation is performed on item number 39, the final item is added to the chart. This item appears as follows:

$$\boxed{40. \quad [met, 0, ra, 0, -, -, 4, true] \quad \text{(COMPLETE,1+39,3rd case)}} \quad .$$

After this phase, no more items are added to the chart. Item number 40 spans the entire input ($i = 0$, $l = 4$), and the tree represented by this item is an initial tree rooted by the start symbol. Furthermore, the dot is at address 0. This means that the input was successfully recognized.

In the next section it is illustrated how these chart items can be used to build a derivation tree for each successful parse of the input.

3.3.1 Deriving a Parse Tree

In order to obtain a derivation tree from the chart, the chart items are modified to make use of back-pointers. Back-pointers point to the items that has led to the current item being placed in the chart. This allows us to follow a path from the last item to the first item in the chart.

Each item in the chart contains a list of back-pointers. When a new chart item is created the parsing algorithm checks to see whether an item such as the newly created item already exists in the chart. If there is, then the back-pointer list of the item already in the chart is updated. Otherwise, a new item is added to the chart.

The procedure to build a derivation tree from the chart works as follows. Firstly, find all the final chart items indicating a successful parse. These are items of the form $[\alpha, 0, ra, 0, j, k, n, \text{sat?}]$ where α is an initial tree and n is the length of the input. If there are k such items, it means there were at least k successful parses. It should be noted that it can happen that a chart item may have more than one item causing it to be placed in the chart. Therefore, there may be more than k parses. The initial trees in the final items will label the root nodes of each of the derivation trees. For each of the final items the back-pointers are followed along all possible paths until the initial items are reached. The initial items were those items that were added during the INITIALIZATION operation of the algorithm. A new child in a derivation tree is created when an item containing a different tree to the tree of the previous item is reached. The parent of the new child node will be labeled by the tree of the previous item.

Figure 3.20 illustrates the different items within a chart. The arrows indicate which items resulted in a particular item being added to the chart. The item marked as number one is the initial item. The item marked as number ten is the final item indicating a successful parse. Figure 3.20 also shows that a particular item can have more than one item causing it to be in the chart. In this example there are three paths from item number ten to item number one. The first path is *(10,9,4,1)*. The second path is *(10,6,3,1)* and the third path is *(10,6,4,1)*. Therefore, in this chart, three derivation trees will be built.

The next example shows how a derivation tree is built for the parsing example of the sentence *John has met Mary*, as discussed in the previous section. The entire back-pointer path for this example is

(40,39,38,37,36,35,34,33,32,30,28,27,26,24,21,18,17,16,15,13,12,11,10,9,8,6,4,2,1) .

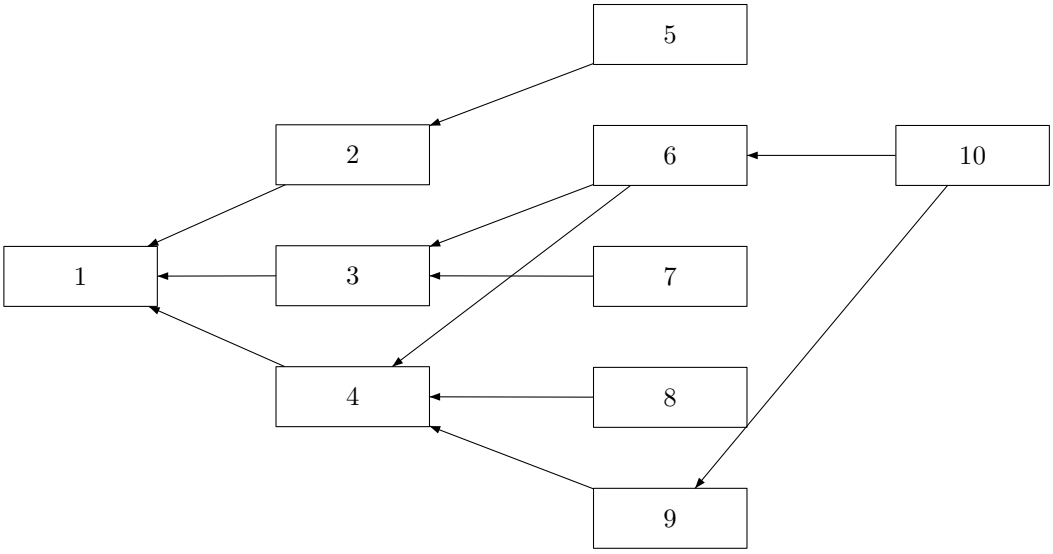


Figure 3.20: Chart representing different paths from the final to the initial item.

The last item in the chart is $[met, 0, ra, 0, -, -, 4, \mathit{false}]$ (number 40). This item indicates that the input was parsed successfully. In this example, each item has only one back-pointer. Therefore, there is only one derivation tree. The root node of this derivation tree is labeled by *met*. We follow the back-pointers until item number 38 is reached. Item number 38 is $[met, 2, rb, 1, -, -, 4, \mathit{true}]$. This item was added to the chart by an ADJOIN operation. This indicates that an adjunction operation was performed at address 2. The tree that was adjoined to *met* is given by the next item in the back-pointer path. Thus, after following the back-pointer path to item number 37, the derivation tree appears as in Figure 3.21.

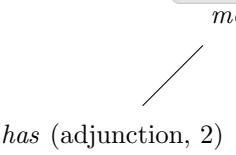


Figure 3.21: Derivation tree after reaching item number 37.

Continuing on the back-pointer path, we reach item number 34. This item was added by a COMPLETE SUBSTITUTION operation. This indicates that a substitution operation was performed on the tree *met*. The address of where the substitution was performed is given by the *dot* value. The tree that was substituted to *met* is again given by the next item in the back-pointer path. After reaching item number 33, the derivation tree appears as in Figure 3.22.

The last substitution is performed on item number 12. If we continue to follow the back-pointer path, we finally reach item number 1. This item was added by the INITIALIZATION operation. Once this initial item is reached, it means that the building of the derivation tree is complete. The final derivation tree is shown in Figure 3.23.

In this section a method that builds a derivation tree from the chart used in the parsing algorithm was discussed. This method was implemented in our prototype machine translation system. In the next section two methods are given for selecting between alternative parses.

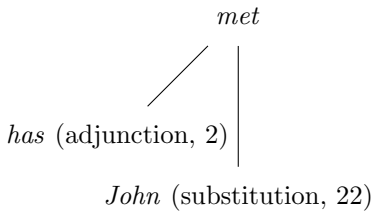


Figure 3.22: Derivation tree after reaching item number 33.

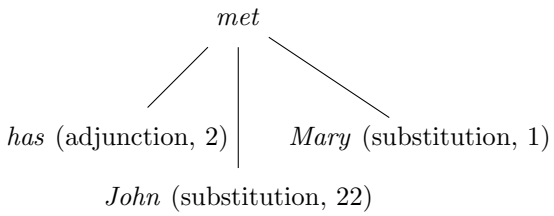


Figure 3.23: The final derivation tree. The last substitution operation was performed on item 12.

3.3.2 Choosing the Correct Parse Tree

In natural language systems, there is usually a large number of successful parses of the input [45]. The most common strategy to deal with these ambiguities is to use a corpus-based method to determine the most appropriate parse tree. The corpus contains human preferences for alternative parses for a large number of sentences. The parser then try and generalize the knowledge from this corpus and apply it to previously unseen sentences. In the absence of a large enough corpus, there are a few methods available with which to try and solve the ambiguity problem. In this section two such methods are described. These methods attempt to select the most acceptable parse from many alternatives. The two methods discussed in this section are described in [45]. The first method is called the method of minimal attachment.

Minimal Attachment

The method of minimal attachment always selects the simpler of any two given parses. Simpler in this situation indicates the parse with the least amount of nodes in the derived tree of that parse. Figures 3.24 and 3.25 are both derived trees generated by our machine translation system. Through the method of minimal attachment, the simpler tree shown in Figure 3.24 will be selected. Note that, in this case the derived tree selected by minimal attachment is also linguistically the most appealing. The reason is that the extra noun phrase in the second derived tree is used in extended wh-question sentences. Extended wh-question sentences are questions starting with *who*, *what*, *when* and *where*, and are usually more complicated than normal questions. An example of such an extended wh-question is *Where would John say Mary works?*. Since *Al ate an apple* is not an extended wh-question, the first tree is the preferred of the two trees.

The method of minimal attachment can, however, also produce incorrect results. Take for example the question, *Where do you work?* Two possible derived trees for this input is shown in Figures 3.26 and 3.27. The derived parse tree in Figure 3.26 has less internal nodes than the derived parse tree in Figure 3.27, and therefore the derivation tree generating the derived parse tree in Figure 3.26 is chosen. However, this is the wrong selection, in that this parse incorrectly assumes *Where* to be a noun, when in fact it is a preposition. Therefore, in this example, the method of minimal attachment fails. In Chapter 5 we look at how a problem such as this can be overcome.

The other method used for disambiguation is the method of right association.

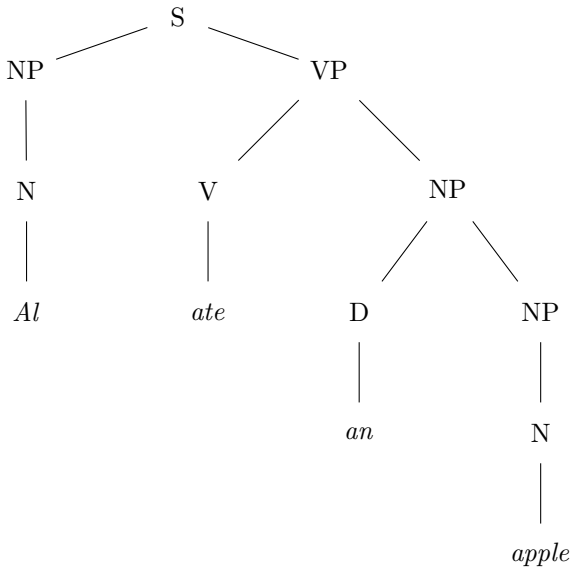


Figure 3.24: First derived tree of minimal attachment example.

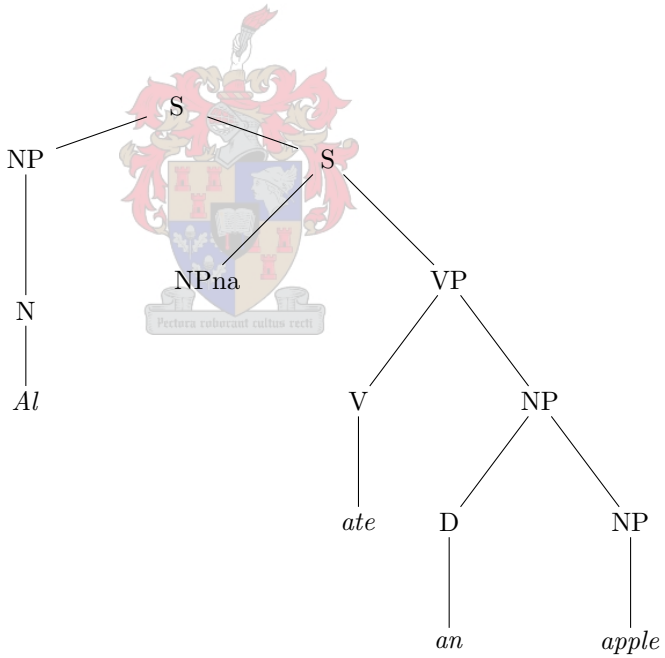


Figure 3.25: Second derived tree of minimal attachment example.

Right Association

The method of right association requires that constituents are attached to the current constituent as opposed to earlier constituents. The principal of right association is illustrated in Figure 3.28 (this example is taken directly from [45]). In this example the AP constituent was attached to the lowest constituent. The parse illustrated in Figure 3.28 would have been chosen over other parses, because *yesterday* (the current constituent) modifies *arrived*, rather than *thought* (an earlier

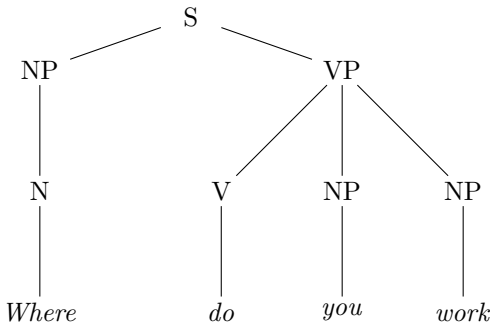


Figure 3.26: First derived tree for *Where do you work?*.

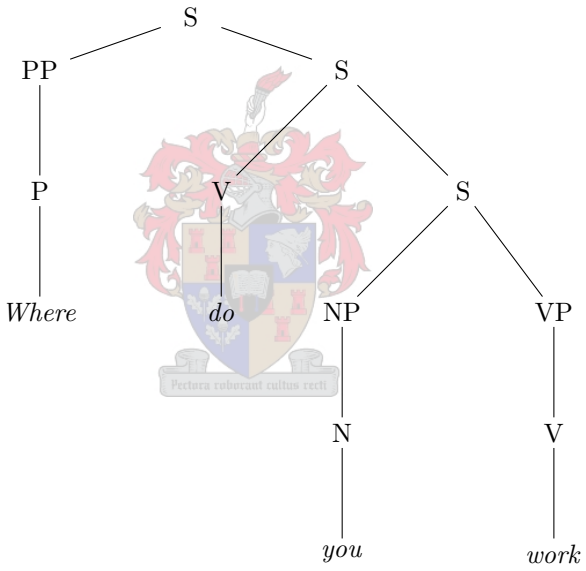


Figure 3.27: Second derived tree for *Where do you work?*.

constituent).

The method of minimal attachment was the only method implemented in our system. A better approach would be to use both the minimal attachment and right association methods for disambiguation purposes [45].

In this chapter a formal grammar called a tree adjoining grammar was introduced. A formal definition for this grammar was given. Different parsers were discussed and one parsing algorithm was discussed in detail. An example of the parsing algorithm was given. It was also shown how to obtain derivation trees from the chart used during parsing. Finally, two methods were discussed to select between alternative parses.

In the next chapter the implementation of our prototype machine translation system is described. The input data used by the system and the methods used for the procurement of the

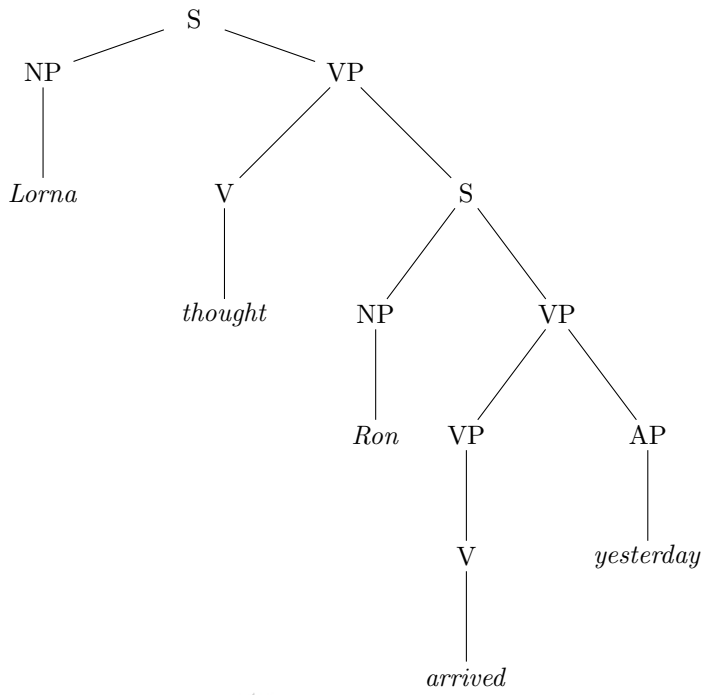


Figure 3.28: Example illustrating right association.

data are also described. Lastly, examples of translation rules used by the system are given.

Chapter 4

System Overview, Data Capture and Rule Derivation

In this chapter the implementation of our prototype machine translation system is discussed. The first section gives a general overview of the system and its different components. In the second section the methodology used in the capture of the system data is presented. The last section describes different translation rules that were derived, based on the captured data.

4.1 System Overview

The main ideas behind the implementation of our system were simplicity and ease of implementation. This is reflected in the relative simplistic approaches taken towards the various components within our machine translation system. These different components are shown in figure 4.1. Our machine translation system uses the XTAG grammar [49] for our English grammar. The morphological and syntactic databases used in our system also forms part of the XTAG grammar. Both these databases used their own file formats (as Lisp datafiles) to store the data. However, since XML [4] is widely used for data interchange, and in fact has become the standard for data representation in the natural language processing communities, it was decided to convert both the morphological and syntactic databases to an XML based format. For this conversion, we used the XTAG grammar datafiles as they were used by the XTAG parser (where they already have been converted from Lisp datafiles into a flat file structure).

The model that was used to manipulate the XML databases, was the document object model (or DOM) [5]. When a program uses the DOM model on XML files, a DOM object is created when an XML file is accessed for the first time. This DOM object is a tree-like structure which provides access to all elements within the XML file. A DOM object therefore can be said to provide random access to the data contained within the XML file.

There is, however, some processing time involved in first building the DOM object. This processing time depends on the size of the particular XML file being accessed. The original XML conversions of both databases resulted in large (19MB in the case of the syntactic database, and 31MB in the case of the morphological database) files. This led to exceedingly long processing times. The database files were therefore divided into smaller files on an alphabetical basis. Although this led to significant gains in the processing time, it still could not compete with the original database implementations used in the XTAG system.

The other model that could have been used for manipulating XML files, is the SAX model [6]. The SAX model provides serialized access to the data in XML files, and not random access like DOM. This means that a program will not have a tree structure from which to work. The program itself must keep track of XML tokens and elements as they are read from the XML file. However, the SAX model does not suffer from the processing time needed to build the DOM objects. The DOM objects, however, only have to be built once for each XML file. This is important because

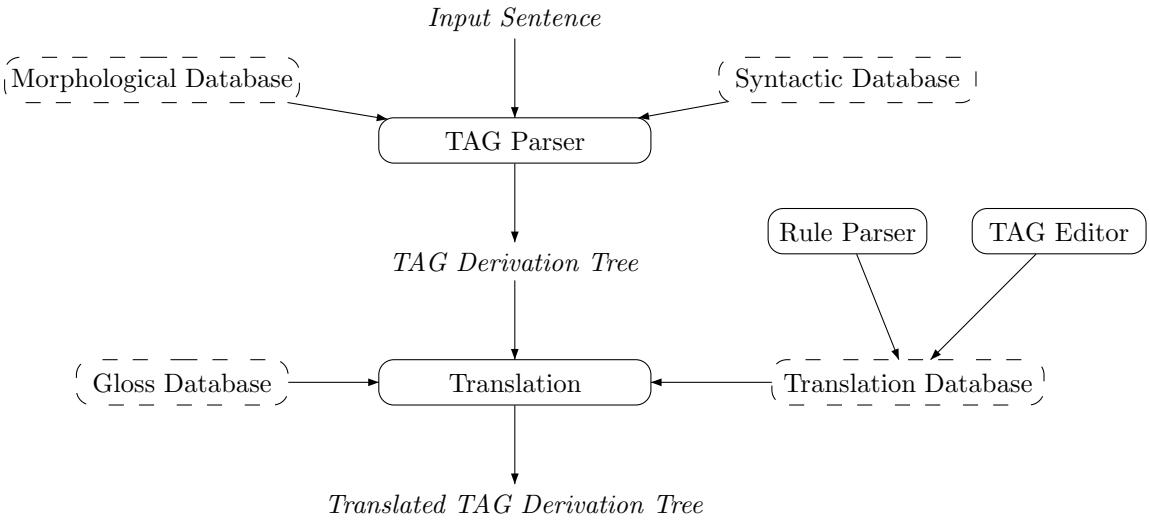


Figure 4.1: Illustration of the different components of our prototype machine translation system.

words can have multiple entries in both databases. This means that if we used the SAX model, we would have to perform multiple searches in the XML files for the entries. With the DOM model no searching is needed, and the entries are obtained quickly from the tree structure. It remains a goal for future work to see whether the SAX model could do better than the DOM model currently used by the translation system.

The different components of the system are briefly discussed in the next few sections.

4.1.1 Morphological Database

The first step of the translation process is to prepare the input sentence for parsing. This involves obtaining the stem of each of the words in the sentence. Our system follows the same technique used by the XTAG system [49]. In our system, as in the XTAG system, a morphological database [15] is used to obtain the stems. This morphological database contains over 317,000 entries. Each entry consists of an inflected form of a word, along with the stem of that inflected word. The entry also contains additional morphological information such as the part of speech of that particular inflected word, whether the inflected word is singular or plural, and so on. For example, when the database is searched for the entry *handle*, there will be separate entries for both the noun *handle* and the verb *handle*.

The user has the option to specify the part of speech for each word in the input sentence. Only those entries with the specified parts of speech will then be returned from the morphological and syntactic databases. Specifying the parts of speech decreases the number of elementary trees used by the parser, which results in faster parsing times. The following sentence shows how the user can specify the part of speech of each word in a sentence:

Al[PropN] ate[V] an[Det] apple[N].

In this example, the *[PropN]* stands for proper noun. In this case it means that *Al* is the name of a person. *[V]* after *ate* means that *ate* is a verb. The *[Det]* shows that *an* is a determiner. Lastly, *[N]* means that *apple* is a noun. The abbreviations used for each of the parts of speech are the same as those used by the XTAG system. These abbreviations are listed in Appendix A.

The morphological items obtained from the database are used in the next stage of the parsing process to build up the elementary trees that are used for parsing. These elementary trees are built using the syntactic database described in the next section. The syntactic database, as well as the mechanisms used to build the elementary trees, form part of the XTAG system.

The next section describes the mechanism employed by the syntactic database to create elementary trees.

4.1.2 Syntactic Database

The second step of the translation process is to build up elementary trees that will be used to parse the input sentence. Our machine translation system uses the XTAG grammar to build the elementary trees. The process of building the trees starts off by searching for entries in a syntactic database. While the morphological database provides morphological information about a word, the syntactic database provides syntactic information. In a TAG formalism this means that the syntactic database provides the information necessary to build the elementary trees associated with each word. The syntactic database consists of entries for each word. There can be multiple entries for each word. For example, there are two entries for the word *handle* in the syntactic database: one for the noun *handle* and the other for the verb *handle* (the same as for the morphological database).

In the syntactic database each entry can have either single elementary trees or a whole family of elementary trees associated with it. The entry will contain the name of either a single elementary tree, or the name of a family of elementary trees. The translation system now uses these names to build the elementary trees for each word in the input.

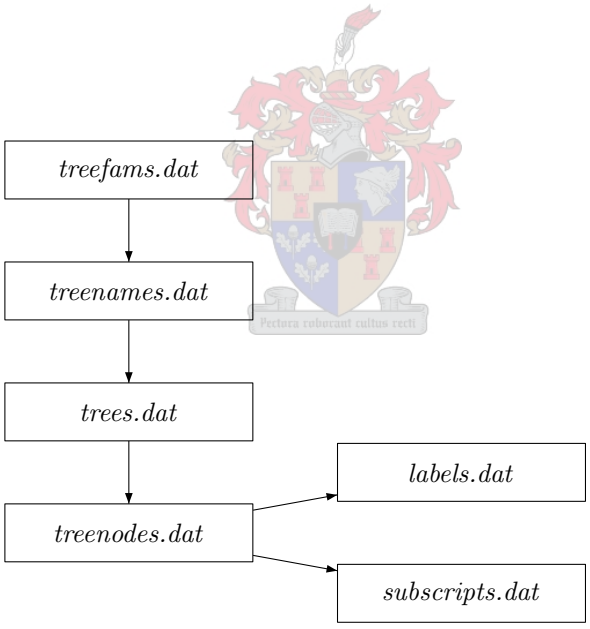


Figure 4.2: The different files used to construct elementary trees in the XTAG grammar.

The XTAG grammar consists of several files in which the information, needed to build the different types of elementary trees, are stored. Figure 4.2 illustrates the dependencies between the different files used in the XTAG grammar. The first file is called *treefams.dat*. This file contain the names of the different families of elementary trees. Following each family name comes the number of elementary trees in that family. After this number, the names of each of the elementary trees in the family are listed. A typical entry in this file looks as follows:

Ts0Vnx1 4 alphaW0s0Vnx1 alphas0Vnx1 betaNpxs0Vnx1 betaNcs0Vnx1 .

The *Ts0Vnx1* (the *T* indicates a family) gives the name of the family. The number *4* indicates that this family consists of four elementary trees. The names of each are then listed. Each name of an elementary tree starts with either *alpha* or *beta*. If the name starts with *alpha*, then this tree is an initial tree. Otherwise, if the name starts with *beta*, the tree is an auxiliary tree. The rest of the name describes the leaf nodes of the elementary tree from the leftmost to the rightmost leaf node. The letters following *alpha* or *beta* refer to the labels of the leaf nodes. Any numbers following a letter indicates the subscript for a node. For example, in the name *alphas0Vnx1* the *alpha* indicates that this tree is an initial tree. The leftmost leaf node is labeled by *S*. This node also has the subscript *0*. The next leaf node is labeled by *V*. Lastly, the letters *nx* refers to a node labeled by *NP*. The *1* indicates that this node has the subscript *1*. The convention for naming the elementary trees is fully described in [49].

The second file is called *treenames.dat*. This file contains the names of all the elementary trees found in the XTAG grammar. Each name is found on a new line in the file. This file is then searched until the name of a specified elementary tree is found. The line number on which the elementary tree is found, is then used to search the third file in the XTAG grammar. Suppose for example we wish to build the elementary tree *alphas0Vnx1*. If we search *treenames.dat*, we find the name *alphas0Vnx1* on line 770.

The third file is called *trees.dat*. Each line in this file gives a description of an elementary tree found in the XTAG grammar. The *trees.dat* file consists of entries with the following structure:

ROOT FOOT #SUBST #ANCHOR #NA SUBST_L ANCHOR_L NA_L .

Here *ROOT* and *FOOT* are numbers. These numbers correspond to line numbers in another file called *treenodes.dat*. Each line of the *treenodes.dat* file gives the description of a specific node found in an elementary tree in the XTAG grammar. So, the *ROOT* entry corresponds to the root node of the elementary tree, and the *FOOT* entry to the foot node of the elementary tree. The *#SUBST* entry is a number which specifies the number of substitution nodes for the elementary tree described in this line. The *#ANCHOR* specifies the number of anchor nodes in this elementary tree. The *#NA* entry specifies the number of nodes which are marked as null-adjunction nodes. Null-adjunction nodes are nodes on which the adjunction operation may not be performed. The *SUBST_L* entry is the list of nodes marked for substitution. As with the *ROOT* and *FOOT* entries, these nodes consists of line numbers. The *ANCHOR_L* entry is the list of anchor nodes. Lastly, the *NA_L* entry is the list of null-adjunction nodes.

If we go to line number 770 in the file *trees.dat* (which is the line number for the tree *alphas0Vnx1*), the following line is found:

6414 -1 2 1 0 6410 6412 6411 .

The *6414* is the line number in the file *treenodes.dat* which corresponds to the description of the root node of *alphas0Vnx1*. The *-1* entry indicates that this particular tree does not have a foot node, which implies that *alphas0Vnx1* is an initial tree. The *2* entry indicates that there are two substitution nodes in *alphas0Vnx1*. The *1* entry indicates that there is only one anchor node for this tree. The *0* indicates that there are no null-adjunction nodes in *alphas0Vnx1*. After the *0* comes the list of substitution nodes. The number of substitution nodes is two, and therefore *6410* and *6412* correspond to the line numbers of the two substitution nodes. Finally, the *6411* is the line number for the anchor node of *alphas0Vnx1*.

The lines of the file *treenodes.dat* have the following structure:

LEFTCHILD RIGHT FLAG LABEL SUBSCRIPT .

Here the *LEFTCHILD* entry is the line number for the leftmost child of the node described in the current line. If the *LEFTCHILD* entry has the value *-1*, it means that this particular node does not have any children. The *RIGHT* entry is the line number for the right sibling of the node described in the current line. If the *FLAG* entry has the value *1*, then it means that the *RIGHT*

node is not the right sibling of the current node, but the parent. The *LABEL* entry gives the label of the current node. Lastly, the *SUBSCRIPT* entry gives the subscript of the current node, if it has one. Both the *LABEL* and *SUBSCRIPT* entries are line numbers. These numbers are used as indices in the files *labels.dat* and *subscripts.dat* to obtain the label and subscript values respectively. If the *SUBSCRIPT* entry has a value of *-1*, it means that the current node does not have a subscript.

To continue the example of the elementary tree *alphas0Vnx1*, we go to line number *6414* in *treenodes.dat*. The entry for the root node of *alphas0Vnx1* is then given by:

6410 6414 1 6 0 .

Therefore, the line number of the node corresponding to the leftmost child of the root node of *alphas0Vnx1* is *6410*. The *RIGHT* entry is *6414*. This is the same as the root node. However, because the *FLAG* value is *1*, we know that this only indicates that the current node is the root node, and therefore it has no parent. The entry on line number *6* of the file *labels.dat* is *S*. Therefore, the root node of *alphas0Vnx1* is labeled by *S*. The *0* entry in *subscripts.dat* is *r*. This indicates that the current node is the root node. At the moment, the *alphas0Vnx1* tree only consists of the root node, and is shown in Figure 4.3. The next step is to go to line number *6410*

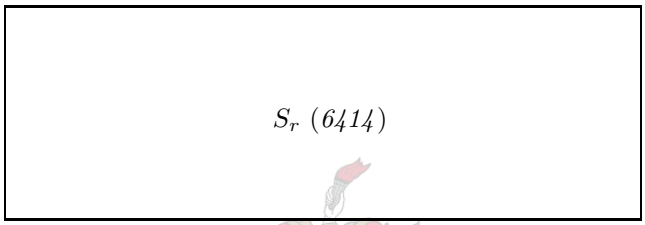


Figure 4.3: The root node of *alphas0Vnx1*. The line number of the node has also been added.

in the *treenodes.dat* file. This line contains the following entry:

-1 6413 0 6 5 .

The *-1* entry tells us that the current node does not have any children. The right sibling of the current node can be found on line number *6413*. The entry on line number *6* in the file *labels.dat* is *S*, and the entry on line number *5* in the file *subscripts.dat* is *0*. The tree *alphas0Vnx1* will now appear as in Figure 4.4. To obtain the right sibling of this node, we go to line number *6413*. This

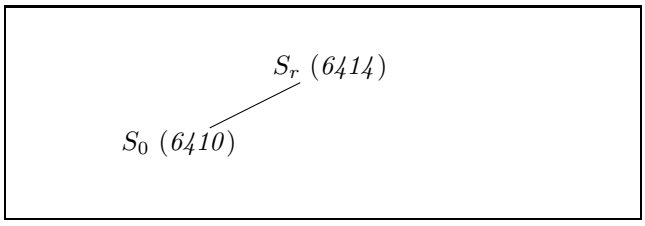


Figure 4.4: The tree *alphas0Vnx1* after the second node has been added.

entry is:

6411 6414 1 12 -1 .

The leftmost child of this node is described on line *6411*. The *FLAG* value is *1*, which means that the *6414* refers to the parent of the current node. The entry on line number *12* of the file *labels.dat* is *VP*. The tree *alphas0Vnx1* will now look as in Figure 4.5. We now continue on to line number *6411*. The entry on this line is:

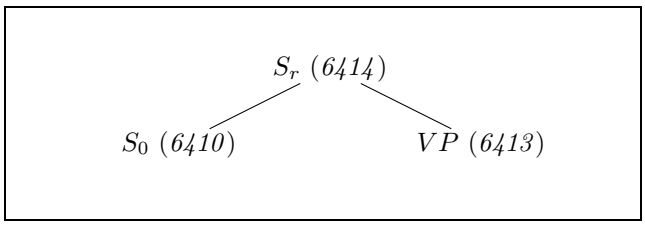


Figure 4.5: The tree *alphas0Vnx1* after the third node has been added.

-1 6412 0 11 -1 .

This node has no children. The right sibling of this node can be found on line number *6412*. The label of this node is found on line number *11* in the file *labels.dat*. This label is *V*. Therefore, after adding the fourth node, *alphas0Vnx1* appears as shown in Figure 4.6. We now have to add the

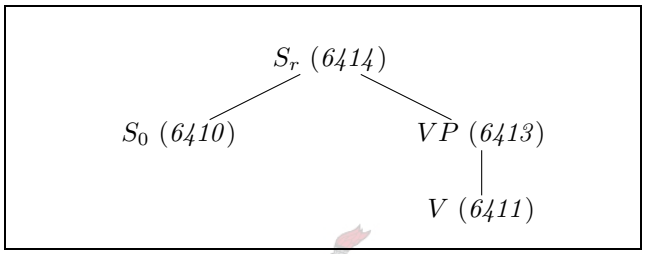


Figure 4.6: The tree *alphas0Vnx1* after the fourth node has been added.

right sibling of this node to the tree. The line number for the right sibling is *6412*. The entry on this line is:

-1 6413 1 2 2 .

Again this node does not have any children. The *6413* refers to the parent of this node. The label of this node is found on line number *2* in the file *labels.dat*. The entry on this line is *NP*. The subscript of this node can be found on line number *2* in the file *subscripts.dat*. The entry for the subscript is *1*. There are no more nodes which we must look up, and therefore all the nodes of the tree *alphas0Vnx1* have been added. All that is left is to mark the substitution and anchor nodes. These nodes were found in the entry in the *trees.dat* file. According to that entry, nodes *6410* and *6412* were substitution nodes. The anchor node was node *6411*. After marking the necessary nodes, the resulting tree is shown in Figure 4.7.

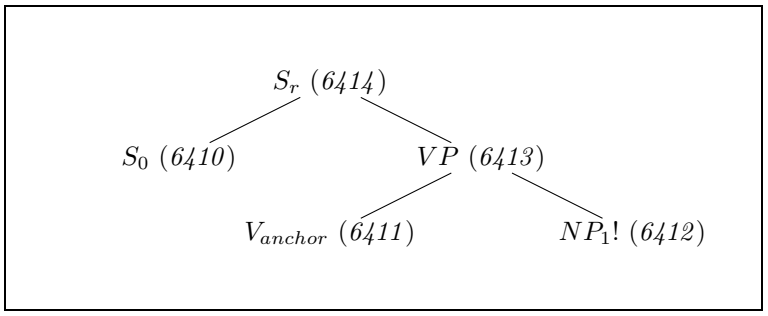


Figure 4.7: The tree *alphas0Vnx1* after the fifth node has been added and the substitution and anchor nodes have been marked.

The last thing to do before the tree is finalized is to take the information obtained from the morphological database and insert it at the anchor node of the tree. For example, say we had the verb *handles*. Then the following entry would be found in the morphological database:

<handles>
<STEM>handle</STEM>
<POS>V</POS>
<Features>3sg</Features>
</handles> .

This information would then be inserted at the anchor node of the *alphas0Vnx1* tree. The final tree that will be used by the parser to parse the input is shown in Figure 4.8. This procedure for

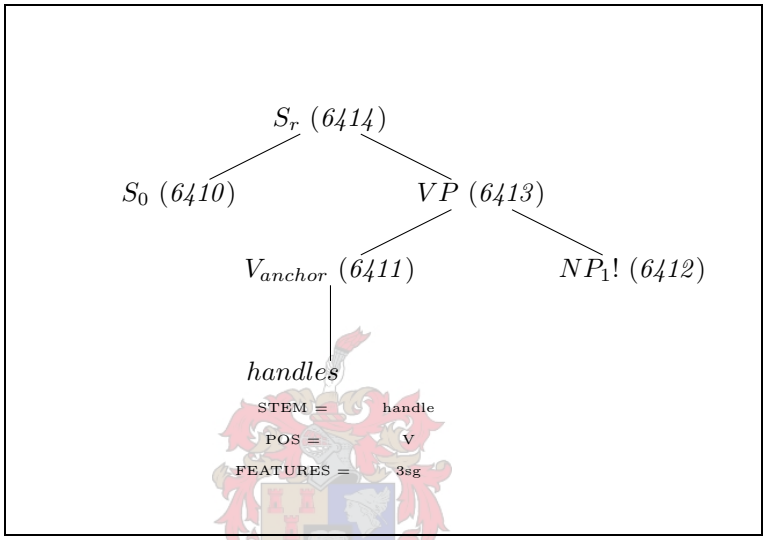


Figure 4.8: The final tree *alphas0Vnx1* after the morphological information has been added.

building elementary trees is repeated for all the trees in the tree family and for all the words in the input sentence.

The next section gives an overview of the TAG parser that was implemented for our translation system.

4.1.3 TAG Parser

The TAG parser is a Java implementation of the parsing algorithm discussed in Chapter 3. A list of derivation trees is generated by the parser as output. Each derivation tree corresponds to a parse of the input. The user can select any derivation tree for translation. Alternatively, the system can select a parse using the method of minimal attachment. Figure 4.9 shows the graphical user interface of the parser.

An example of a derivation tree generated by the parser for the sentence *Eat your carrots* is shown in Figure 4.10. The derivation tree in Figure 4.10 consists of three nodes. Each node corresponds to an elementary tree. The derivation tree tells us that in order to obtain the sentence *Eat your carrots* the tree *betaDnx* (anchored by the word *your*) should be adjoined to the tree *alphaNXN* (anchored by the word *carrots*). The adjunction should be performed at the root node of the *alphaNXN* tree. The resulting tree should then be substituted at node *22* in the *alphaInx0Vnx1* (anchored by the word *eat*) tree.

Once a derivation tree is selected the translation process can begin. The next section describes the rule mechanism employed during the translation process.

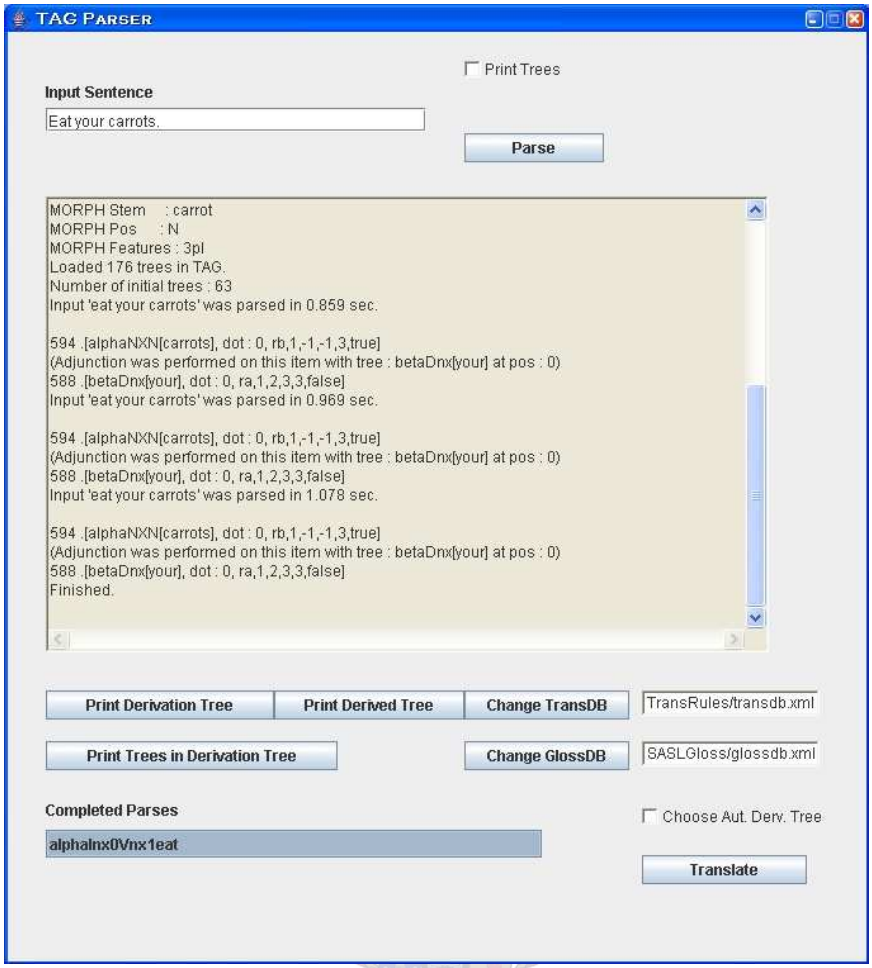


Figure 4.9: Parser GUI with a successful parse highlighted.

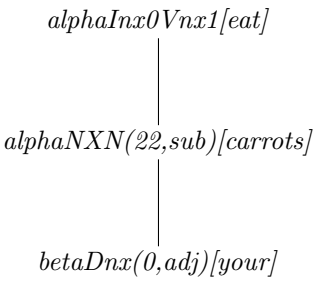


Figure 4.10: An example of a derivation tree.

4.1.4 Rule Mechanism and Translation Database

Any translation system that uses the synchronous TAG (STAG) formalism as basis for translation, requires two TAG grammars: one for the source language, and the other for the target language. In our system these grammars are for English and SASL respectively. A mechanism was needed with which the two grammars could be synchronized. The simplest way in which this can be accomplished, is to explicitly state how the trees and nodes from the different grammars are associated with each other. Again, an XML database was used for this task.

Explicitly stating the associations between trees was also done in the STAG [7] implementation of the XTAG system. The STAG implementation of the XTAG system also allowed multiple associations between trees. Selecting the correct association for translation was done through the use of keywords [8].

Our implementation also allows for multiple associations between trees. With an XML database, it is easy for the elementary trees from the source grammar to be associated with more than one elementary tree from the target grammar. This allows flexible translation rules where different associated target grammar trees can be used depending on a particular translation rule.

The next example illustrates how a new translation database entry is created. In this example, the source grammar tree *alphanx0Vnx1* is associated with the target grammar tree *alphaBnx0Vnx1*. If the translation database does not already have an entry for the source grammar tree *alphanx0Vnx1*, a new one is created as follows:

<alphanx0Vnx1> </alphanx0Vnx1> .

The next step is to add a *RULE* entry for the source grammar tree. The *RULE* entry contains information as to when a particular tree association should be applied. In this example the *alphanx0Vnx1* tree should only be associated with the *alphaBnx0Vnx1* tree if the verb which anchors the *alphanx0Vnx1* tree is in the past tense.

<alphanx0Vnx1>
<RULE number="12" features="PAST" POS="V">
</RULE>
</alphanx0Vnx1>

Here the *POS="V"* indicates that the word anchoring the tree *alphanx0Vnx1* should be a verb. Furthermore, the *features="PAST"* indicates that this verb should be in the past tense. If both these conditions are met, then this translation rule entry can be applied. The next step is to indicate the target grammar tree with which the tree *alphanx0Vnx1* should be associated.

<alphanx0Vnx1>
<RULE number="12" features="PAST"" POS="V">
<TRANSTREENAME>alphaBnx0Vnx1</TRANSTREENAME>
</RULE>
</alphanx0Vnx1>

Here the *TRANSTREENAME* entry indicates that the *alphanx0Vnx1* tree is associated with the *alphaBnx0Vnx1* tree from the target grammar. The next step is to link individual nodes between the two trees. This linking is illustrated in the final translation database entry:

<alphanx0Vnx1>
<RULE number="12" features="PAST" POS="V">
<TRANSTREENAME>alphaBnx0Vnx1</TRANSTREENAME>
<SNODE>
<NODENAME>1</NODENAME>
<TRANSNODENAME>2</TRANSNODENAME>
</SNODE>
<SNODE>
<NODENAME>22</NODENAME>

<TRANSNODENAME>32</TRANSNODENAME>
</SNODE>
</RULE>
</alphanx0Vnx1> .

The nodes at address *1* and address *22* in *alphanx0Vnx1* are associated with nodes at addresses *2* and *32* in *alphaBnx0vnx1*. This means that any trees that are to be substituted or adjoined at address *1* in *alphanx0Vnx1*, will be substituted or adjoined at address *2* in *alphaBnx0vnx1*. The same applies for the node at address *22*. These associations between trees and nodes in the source and target grammars constitute the translation rules for machine translation systems based upon the STAG formalism [24].

Although the user is free to edit the translation database by hand, we added a small utility to help create translation rules. The rule parser utility was added as a more convenient way to add translation rules to the system. It takes a rule entry and converts it into an XML entry in the translation database. A graphical interface will be provided in future versions of the system. The following format is accepted by the rule parser:

$$Rule\_Num;\ \underbrace{Tree\_Name(Node\_Address)[Features,Pos]}_{Source\ Grammar}\ =\ \underbrace{Tree\_Name(Node\_Address)[Tfeatures]}_{Target\ Grammar}\ .$$

The rule parser generates a *RULE* element from this line. This *RULE* element is added to the translation database entry specified by the *Tree_Name* entry from the left hand side of the equality sign. The tree from the target grammar that is associated with this entry is given by *Tree_Name* from the right hand side of the equality sign. The *Node_Address* on the left hand side of the equality sign indicates a node in the source grammar tree. This node is associated with the node in the target grammar tree at the address specified by the *Node_Address* on the right hand side of the equality sign. The *Features* and *Pos* entries are used to add morphological information to the *RULE* element. This morphological information can be used as a constraint as to when a particular association should be applied. The *Tfeatures* element specifies additional information that is to be added to the gloss anchor in the target grammar.

An example of a rule that could be given to the rule parser is,

11; alphanx0Vnx1(22)[FEAT-Inf,POS-V] = alphannv(2)[] .

The rule number for this entry is *11*. The tree from the source grammar is *alphanx0Vnx1*. The tree from the target grammar is *alphannv*. The node at address *22* in the tree *alphanx0Vnx1* is associated with the node at address *2* in the tree *alphannv*. The tree on the left in Figure 4.11 is the tree *alphanx0Vnx1* from the source grammar. The tree on the right is the tree *alphannv* from the target grammar. The dashed line shows how two nodes are associated in the trees. The following is the final entry generated by the rule parser that will be added to the translation database.

<alphanxOVnx1>
<RULE number="11" features="Inf" POS="V">
<TRANSTREENAME>alphannv</TRANSTREENAME>
<SNODE>
<NODENAME>22</NODENAME>
<TRANSNODENAME>2</TRANSNODENAME>
</SNODE>
</RULE>
</alphanxOVnx1>

In Appendix B an example of the contents of the translation database is shown. Here it can be seen that many entries in the translation database have multiple *RULE* elements. The derivation tree that is selected for translation consists of a number of elementary trees from the

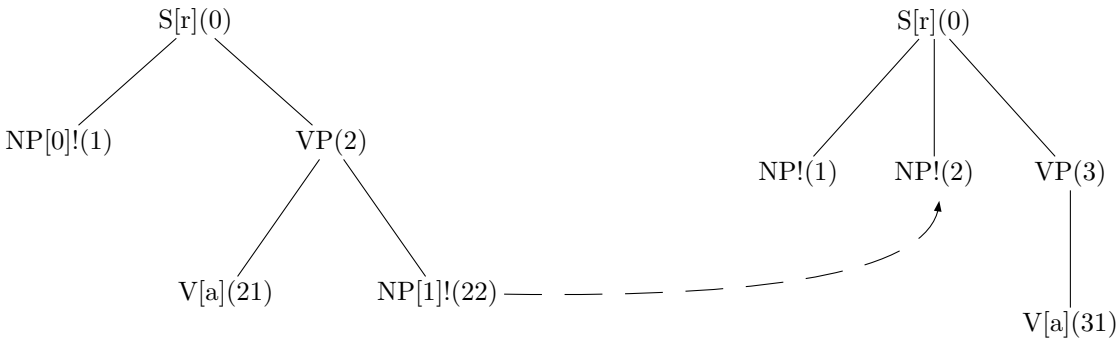


Figure 4.11: The trees *alphanx0Vnx1* and *alphannv* from the source and target grammars respectively. The node addresses have also been added.

source grammar. Most of these elementary trees have entries in the translation database. Note however, that it is not required for an elementary tree to have an entry in the translation database. Each of the entries in the translation database can have multiple *RULE* elements. Each of the *RULE* elements has one rule number. Thus for each entry of the translation database there is a set of rule numbers. Therefore every elementary tree in the derivation tree that has an entry in the translation database will have a set of rule numbers. An AND operation is performed on these sets of rule numbers. The result of this AND operation determines which rule number to use for this particular derivation tree. This rule number will then be used to select the appropriate *RULE* elements for each of the translation database entries.

Assume the input *Eat your carrots.* is given to the parser. Then a possible derivation tree generated by the parser is shown in Figure 4.12.

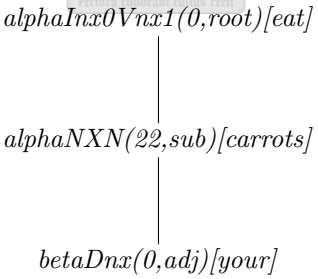


Figure 4.12: Derivation tree for *Eat your carrots.*

The translation system will now generate a set of rule numbers for each of the elementary trees in the derivation tree. Since the tree *alphaInx0Vnx1* contains two *RULE* elements the set generated for the *alphaInx0Vnx1* tree will be $\{1, 4\}$. The translation database entry for the elementary tree *alphaNXN* has five different numbered *RULE* elements. The set generated for the tree *alphaNXN* will be $\{1, 2, 5, 6, 12\}$. The *betaDnx* tree does not have an entry in the translation database. This means that the tree *betaDnx* is not used in the rest of the translation process. When an AND operation is performed on the sets $\{1, 4\}$ and $\{1, 2, 5, 6, 12\}$ the result is $\{1\}$. Therefore the system will use the *RULE* elements with the rule number 1 in the translation process.

In this section the translation database was described. The rule mechanism used by the translation system was also discussed. In the next section another utility used to facilitate the creation of a TAG grammar is discussed.

4.1.5 TAG Editor

A utility for creating and manipulating TAG trees was added to the translation system. This was necessary in order to be able to create new target grammars used by the translation system. The trees created by the TAG editor are stored in an XML format. Figure 4.13 shows the TAG editor in use.

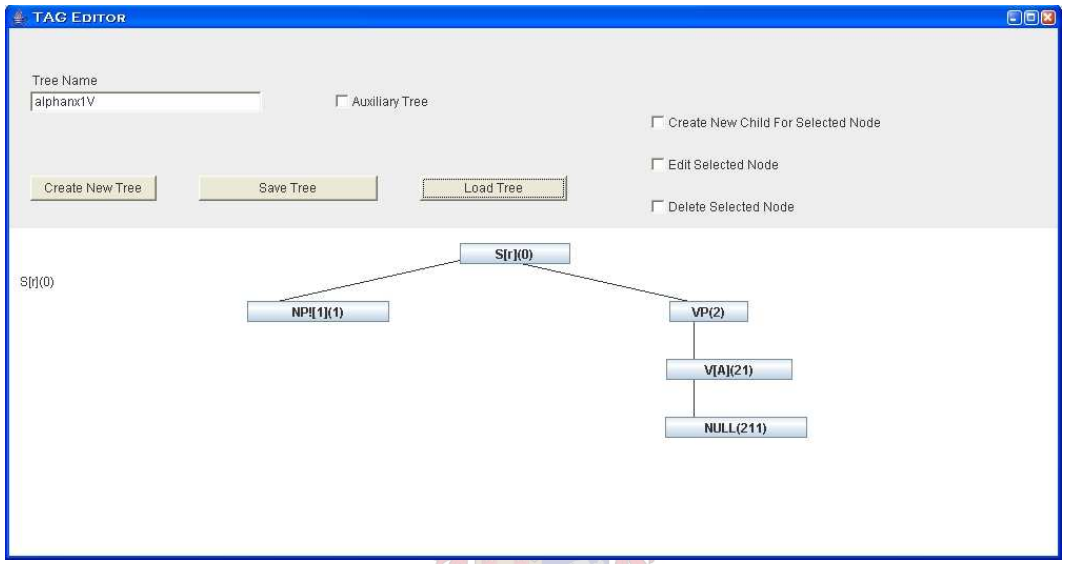


Figure 4.13: Example of the TAG editor.

The last part of the translation system is the gloss database. This is discussed in the next section.

4.1.6 Gloss Database

Glosses are used to represent individual signs in SASL. They are usually written in capital letters and called by the English name best describing the individual signs [33]. For example, the sign for *see* will be written in gloss notation as *SEE*.

The gloss database is an XML database that contains a lexicon of English-gloss pairs. An English word is looked up in the database in order to obtain the gloss for that word. A default gloss is assigned to an English word if that word is not found in the gloss database. For example, if the word *eat* is not found in the gloss database, then a default gloss *EAT* is assigned to it.

This finishes the overview of our machine translation system. In this section the different components of the system, as well as some implementation details were discussed. The next section describes the methodology used during data capture and the different translation rules derived from that data.

4.2 Data Capture Methodology and Translation Rules

South African Sign Language does not have a large corpus of already translated material. It was therefore decided to use a small phrase book as a basis for translation. This phrase book

was developed at the University of Stellenbosch [9]. The phrase book consists of a set of English phrases and their gloss equivalents. For example, the following phrase is found in the phrase book:

Please call the police.
CALL POLICE PLEASE .

The top sentence is the English phrase. The bottom phrase represents the gloss translation. The gloss translation shows in what order the signs represented by each gloss should be signed. In this example the *CALL* gloss would be signed first, then the *POLICE* gloss and finally the *PLEASE* gloss. The complete SASL phrasebook is included in Appendix C.

There are various aspects that are important to consider when obtaining sign language linguistic data. Neidle describes many of these aspects in [33]. These aspects are essential in ensuring the integrity and overall value of the linguistic data.

The first important aspect is the need for appropriate Deaf signers. The Deaf signer should have SASL as their first language. The Deaf signer should also have the correct cultural and language background. This is needed so that the signer can make informed linguistic judgments about what he or she is signing.

The second important aspect is the way in which data is elicited. The main goal of data elicitation is to get as natural a response from the signer as possible. It is therefore important to minimize the chance of any interference with the signing process. A dangerous method of data elicitation highlighted in [33] is presenting the signer with a written English phrase to sign. This is dangerous because the Deaf signer may be influenced by the English grammar of the phrase. This can result in the native signer not signing the true sign language translation of the sentence. Instead, the signer may end up signing a form of signed English.

The last important aspect is to record all signing data on video. This allows careful study of each of the signs used by the signer.

These different aspects were taken into consideration during the development of the phrase book. The signer used for the phrase book was a native SASL signer, although he was not a second generation native signer. Although written English was used to elicit data, an interpreter was used to convey the context and meaning of each phrase. The native signer was encouraged to first sign phrases for himself and to confer with a second native signer about the correct signing for each phrase. These steps allowed for the most natural responses from the native signer. A video recording was made for each phrase of the phrase book. The video recording of each translated phrase was then analyzed by a trained linguist who is also a SASL interpreter to identify individual glosses. The entire phrase book video recording was transcribed in this manner.

This phrase book formed the basis from which the translation rules used by our machine translation system were derived. The next section describes some rules that were derived for the system. These rules were derived in order to demonstrate the capabilities of the machine translation system, and we foresee their continued evolvement as further development of the system progresses. The reason why these specific rules were used, is because they are the most easily identifiable and the most representative of the linguistic phenomena occurring within thephrase book.

We describe the translation rules below, and illustrate each of the rules with an example. One important fact that should be mentioned is that the rules given below and their associated STAG derivations are all isomorphic. Recall that isomorphism is a requirement for the definition for STAG derivation given in Chapter 3.

The first translation rule illustrated is a rule describing the dropping of pronouns in SASL.

4.2.1 Dropping of Pronouns

The first translation rule that we describe is the dropping of pronouns. This can be seen in the phrase *See you tomorrow*, which is translated as *SEE TOMORROW*. The verb *SEE* is a directional[1] verb. This needs to be indicated somehow in the final derived tree. To illustrate how

[1]This means that the sign for this verb is performed in a certain direction to indicate the subject and object of the verb. For example, the verb *SEE* would be performed in such a manner as to indicate that the person who will

this translation rule can be implemented, it is necessary to first look at the derivation tree of *See you tomorrow*. This is illustrated in Figure 4.14.

To drop the pronoun, one simply leaves out the entry for the tree *alphaNXN[you]* from the translation database. In order to do this, the following rules are given to the rule parser:

```
4; alphaInx0Vnx1[FEAT-Inf,POS-V] = alphaInx0Vnx1[DIRECTIONAL]
4; betavxN[FEAT-3sg,POS-N] = betavxN[]           .
```

In the first line the tree *alphaInx0Vnx1* from the source grammar is associated with the *alphaInx0Vnx1* tree from the target grammar. Furthermore, the *DIRECTIONAL* entry indicates that the verb which anchors the *alphaInx0Vnx1* tree in the target grammar, is a directional verb. This information will be added to the anchor of the *alphaInx0Vnx1* tree in the final derived tree. The second line states the same for the *betavxN* tree. When the above rules are applied to the derivation tree of *See you tomorrow*, the resulting translated derivation tree will appear as in Figure 4.15. The final derived translation is shown in Figure 4.16. The two lines above allow all sentences with the same derivation tree structure as in Figure 4.14 to be translated to a derivation tree similar to the one in Figure 4.15. For example, these rules will translate the sentence *Call you tomorrow* to *CALL TOMORROW*.

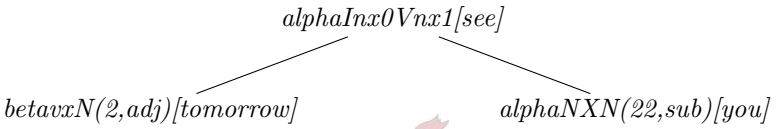


Figure 4.14: Derivation tree for *See you tomorrow*.

The previous example works well for a sentence such as *See you tomorrow*. However, it will not work for the sentence *See you soon*. This is because *soon* is an adverb, and not a noun. This can be seen in the derivation tree for *See you soon*, which is shown in Figure 4.17. In order to address this situation, another line is added to the rules given to the rule parser:

```
4; alphaInx0Vnx1[FEAT-Inf,POS-V] = alphaInx0Vnx1[]
4; betavxN[FEAT-3sg,POS-N] = betavxN[]
4; betavxARB[FEAT-,POS-Adv] = betavxARB[]        .
```

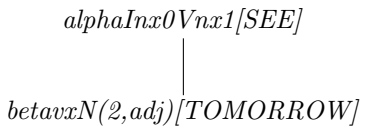


Figure 4.15: Translated derivation tree for *See you tomorrow*.

be seen, is the person spoken to.

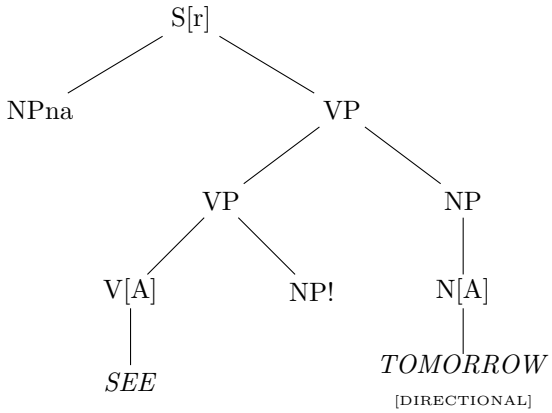


Figure 4.16: Translated gloss based translation for *See you tomorrow.*

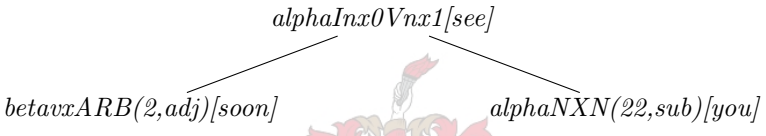


Figure 4.17: Derivation tree for *See you soon.*

The last line associates the *betavxARB* tree from the source grammar with the *betavxARB* tree from the target grammar. This allows sentences such *See you soon* to be translated as *SEE SOON*. The translated derivation tree for *See you soon* is shown in Figure 4.18.

In this section examples were given of how to implement a translation rule for the dropping of pronouns in our machine translation system. The next section describes a similar rule to the one in this section, namely, the dropping of determiners.

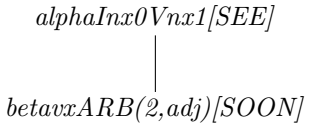


Figure 4.18: Translated derivation tree for *See you soon.*

4.2.2 Dropping of Determiners

This translation rule is similar to the rule for the dropping of pronouns. It is implemented in the same way by leaving out the elementary tree anchored by a determiner in a sentence. This translation rule is illustrated in the following example. The translation for the sentence *Eat your carrots* is *EAT CARROT*. Note that the noun *carrot* and its plural *carrots* are both represented by only one SASL gloss, namely *CARROT*. The derivation tree for *Eat your carrots* is shown in Figure 4.19. The following lines are given to the rule parser in order to correctly translate *Eat your carrots* to *EAT CARROT*:

```
1; alphaInx0Vnx1[FEAT-Inf,POS-V] = alphaInx0Vnx1[]
1; alphaNXN[FEAT-3pl,POS-N] = alphaNXN[]          .
```

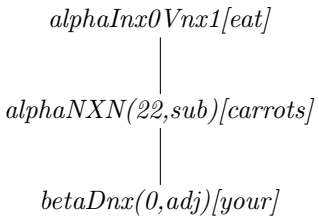


Figure 4.19: Derivation tree for *Eat your carrots*.

The first line states that the *alphaInx0Vnx1* tree from source grammar should be associated with *alphaInx0Vnx1* tree from the target grammar. The second line says that the *alphaNXN* tree from the source grammar should be associated with the *alphaNXN* tree from the target grammar. Note however, that the *FEAT-3pl* entry states that this only applies if the noun that anchors the *alphaNXN* is a plural noun. If we wanted the rule to also apply to singular nouns then another line would have to be added:

```
1; alphaInx0Vnx1[FEAT-Inf,POS-V] = alphaInx0Vnx1[]
1; alphaNXN[FEAT-3pl,POS-N] = alphaNXN[]
1; alphaNXN[FEAT-3sg,POS-N] = alphaNXN[]          .
```

When these rules are applied to the derivation tree for *Eat your carrots*, the translated derivation tree will appear as in Figure 4.20. The final derived tree is shown in Figure 4.21.

This section gave an example of how a translation rule for the dropping of determiners can be implemented in our machine translation system. In the next section, a translation rule for the movement of adverbs in a sentence is described.

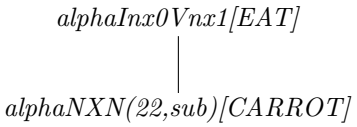


Figure 4.20: Translated derivation tree for *Eat your carrots*.

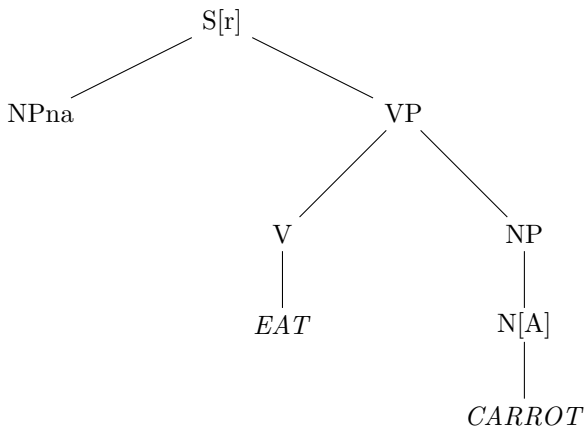


Figure 4.21: The final derived gloss based translation for *Eat your carrots*.

4.2.3 Movement of Adverbs

The translation rule described in this section is applicable to situations where adverbs such as *please* move from the front of an English sentence to the end of a SASL sentence. For example, the translation of the sentence *Please call an ambulance* is *CALL AMBULANCE PLEASE*. The derivation tree for *Please call an ambulance* is shown in Figure 4.22.

The tree *betaARBs* is anchored by the adverb *please*. This tree is shown in Figure 4.23. The adjunction of this tree to the rest of the trees in the derivation tree results in the adverb *please* appearing at the front of the sentence.

In order to change the sentence so that *PLEASE* comes at the end of the SASL sentence, the *betaARBs* tree from the source grammar must be associated with the *betasARB* tree from the target grammar. The *betasARB* tree is illustrated in Figure 4.24. When this tree is adjoined to the rest of the trees in the derivation tree it will result in the *PLEASE* gloss appearing at the end of the sentence. The rules given to the rule parser to implement this translation rule for sentences such as *Please call an ambulance* are as follows:

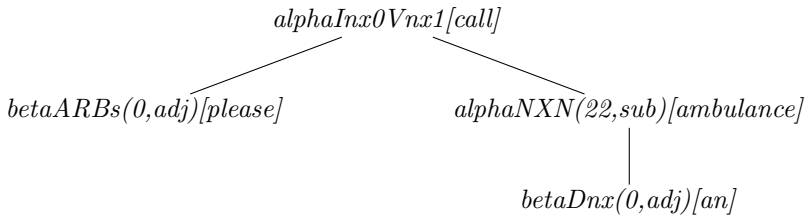


Figure 4.22: Derivation tree for *Please call an ambulance*.

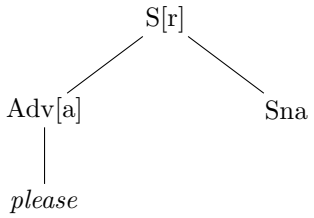


Figure 4.23: The *betaARBs* tree, anchored by *please*.

```
1; alphaInx0Vnx1[FEAT-Inf,POS-V] = alphaInx0Vnx1[]
1; alphaNXN[FEAT-3sg,POS-N] = alphaNXN[]
1; betaARBs[FEAT-,POS-Adv] = betasARB[]          .
```

The translated derivation tree for the sentence *Please call an ambulance* is shown in Figure 4.25. The final derived translation is shown in Figure 4.26. The above rules applies to all sentences with the same derivation tree structure as in Figure 4.22. Such sentences include *Please call the police*, *Please eat your vegetables*, and so on.

In this section we described how to implement a translation rule for the movement of adverbs. In the next section we look at a translation rule for questions in SASL.

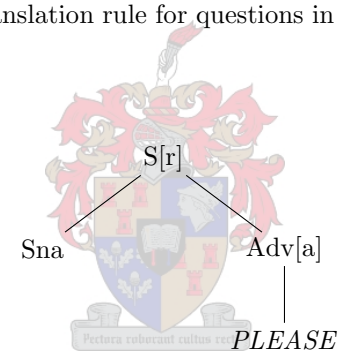


Figure 4.24: The *betasARB* tree from the target grammar, anchored by the *PLEASE* gloss.

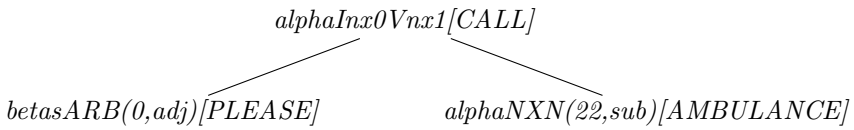


Figure 4.25: Translated derivation tree for *Please call an ambulance*.

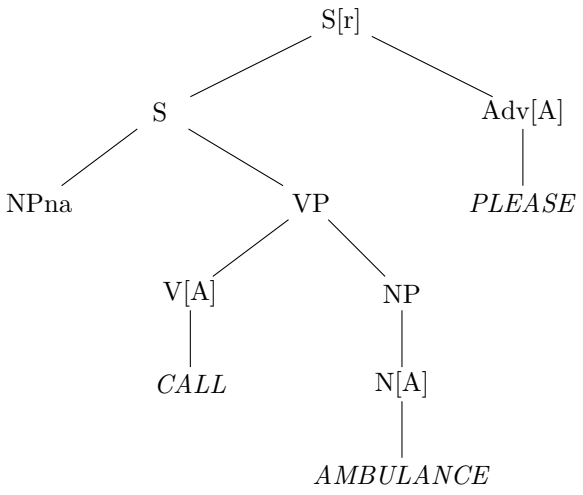


Figure 4.26: Final derived translation for *Please call an ambulance.*

4.2.4 Wh-Questions

This section describes a translation rule where the wh-word in a question moves to the end of a sentence. This can be seen in a question such as *Where do you work?*. The gloss translation for this sentence is *WORK WHERE*. The derivation tree for *Where do you work?* is shown in Figure 4.27.

Figure 4.27: Derivation tree for *Where do you work?*.

The tree *betaPs* which is anchored by *where* is the reason why *where* appears at the start of the question. This tree needs to be replaced with the tree *betasP* from the target grammar. The rules that are given to the rule parser to accomplish the translation are:

```
8; alphanx1V[FEAT-Inf,POS-V] = alphanx1V[]
8; betaPs[FEAT-,POS-Prep] = betasP[]
8; betaPs[FEAT-,POS-Prep] adjoin in alphanx1V(0)[] .
```

This rule differs from previous rules mentioned so far. The second line states that the *betaPs* tree from the source grammar should be associated with the *betasP* tree from the target grammar. The third line says that the tree *betaPs* should be adjoined to the *alphanx1V* tree at address *0*. This is necessary, because when the tree *betaIVs* is left out of the translated derivation tree, the tree

below it (*betasP*) will also be left out. Applying these translation rules will result in the translated derivation tree in Figure 4.28. The final translation is shown in Figure 4.29.

In the first section of this chapter the different components of our machine translation system were described. Mechanisms by which TAG trees are created and manipulated were also explained. The mechanism which allows tree from different grammars to be associated with each other was also explained. In the second section of this chapter the data capturing methodology was described. Lastly, various translation rules were given. These translation rules serve to illustrate the capabilities of the prototype system. In the next chapter we discuss future work to be done to the system, as well as our conclusions.

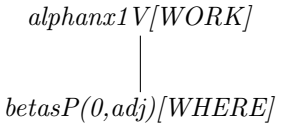


Figure 4.28: Translated derivation tree for *Where do you work?*

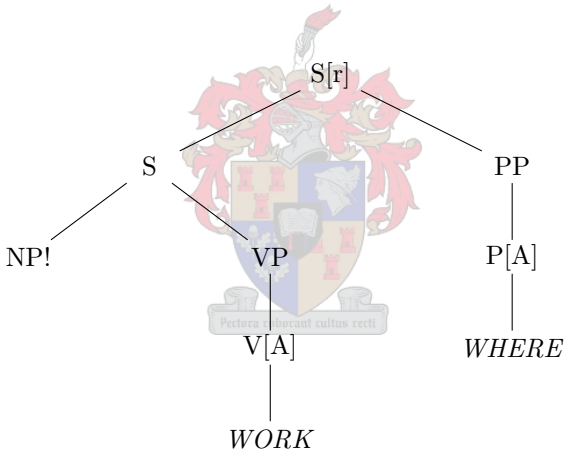


Figure 4.29: Final translation for *Where do you work?*

Chapter 5

Results and Conclusions

In this chapter we present the results from our system. Suggestions for future work are made, and in conclusion we look at the time and space complexity of the implemented TAG parser.

5.1 Goal of Work

The original goal of this thesis was to implement a machine translation system in the minimum amount of time using available linguistic resources. The implemented translation system was based upon a transfer based approach. Because of the limited linguistic resources available for SASL, a transfer based approach was taken over other translation mechanisms, such as statistical translation, which require a large corpus of already translated sentences.

In a transfer based approach a source language sentence is parsed into an intermediary form which is then transferred to an equivalent intermediary form for the target language output. For the intermediary form a synchronous tree adjoining grammar was used. The main reason for this was the availability of the XTAG grammar, which provided a ready to use English TAG grammar. This grammar was used by the implemented TAG parser to parse English input sentences. Finally, various translation rules were created to demonstrate successful translations using the system. In the second section of this chapter, a complete example of a translation is given to demonstrate the system.

Another part of the system described in Chapter 4 are the various language tools that were added to the system. These tools were created in order to extend and help test the linguistic capabilities of the system. The first tool, the rule parser, was added to facilitate the creation of new translation rules. The second tool, the TAG editor, was added in order to manipulate individual TAG trees. Lastly, the gloss database was added so that a user can assign individual glosses to particular words.

The combination of these tools and the TAG parser forms the basis of our prototype machine translation system. This system was successful in being able to translate rudimentary English sentences for which we created translation rules.

In the next section we present an example of the usage of the system.

5.2 Example of Usage

In this section we illustrate an example of a translation using our system. To use the TAG parser, the user has to provide the English input to be translated (see Figure 5.1).

Suppose the sentence to be translated is *Where do you work?* The user can specify the various word categories by adding the information in square brackets, for example:

Where[Prep] do[V] you[Pron] work[V]? .

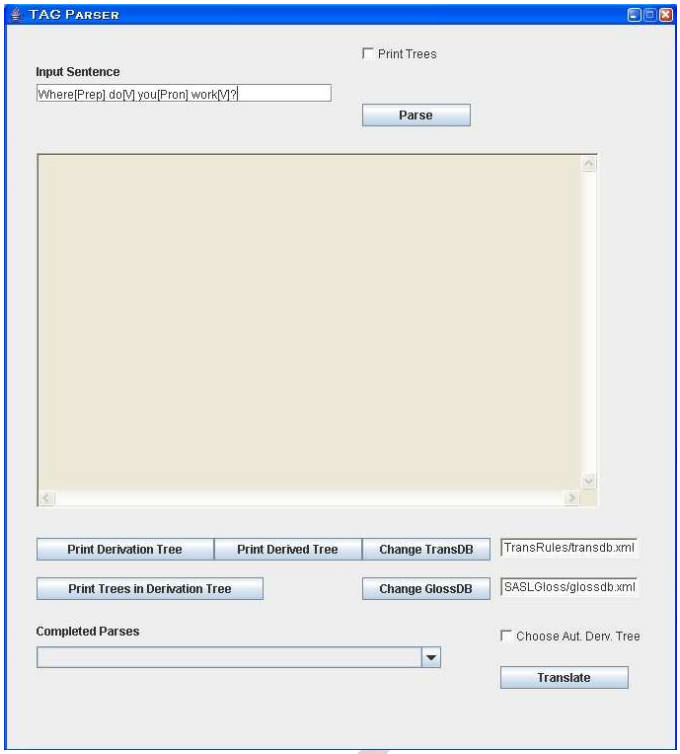


Figure 5.1: The TAG parser.

In this sentence, the *[Prep]* indicates that the word *Where* is a preposition. The *[V]* indicates that the words *do* and *work* are verbs. Lastly, *[Pron]* indicates that *you* is a pronoun. When the user wishes to proceed, the parse button is pressed.

In the initial phase of the parsing process, the parser builds all the trees from the TAG to be used in this particular parse. The next phase is the running of the actual parsing algorithm. The last phase is the creation of a list of derivation trees, with one derivation tree for each successful parse.

Once this last phase is complete, the user can either select a derivation tree, or let the parser automatically choose a derivation tree. In this example, we chose to let the parser select a derivation tree. This is illustrated in Figure 5.2. In this example, the parser chooses the *alphanx1V[work]* derivation tree. It then proceeds to apply the translation rules and displays the final translated output. The translated derivation tree is shown in Figure 5.3, and the final derived tree is shown in Figure 5.4.

In this section an example of translation using the system was illustrated. In the next section we discuss some potential problems faced by our system.

5.3 Potential Problems

This section discusses three issues which can result in potential problems for the translation system. We take a look at each of these in the next few sections. The first potential problem is the selection of the correct derivation tree.

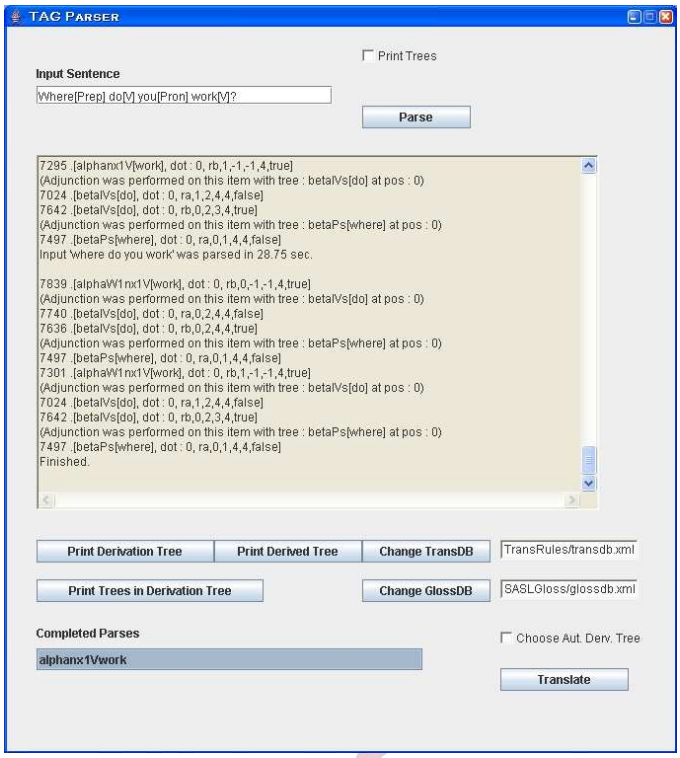


Figure 5.2: The TAG parser with input.

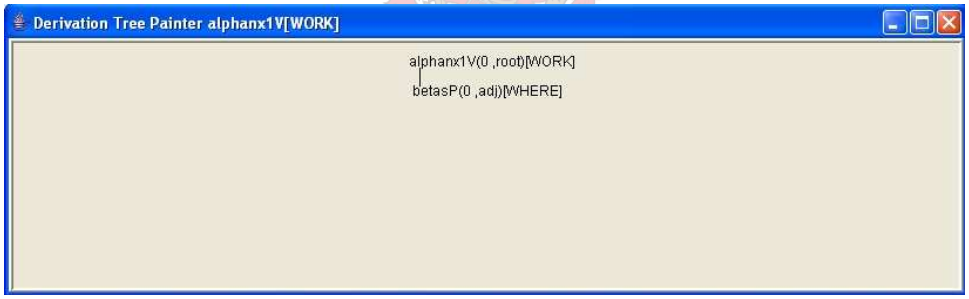


Figure 5.3: The translated derivation tree for *Where do you work?*

5.3.1 Automatic Derivation Tree Selection

The first potential problem is choosing the most appropriate derivation tree. As mentioned in Chapter 3, the method used by the system is based on minimal attachment. To illustrate what can go wrong, we again look at the example given in Chapter 3.3.2. Say for instance, that we take the input *Where do you work?*, but this time no word categories are given. The parser now generates more parses than in the previous example, although some of them will now be grammatically incorrect.

In the previous example, using the method of minimal attachment, the parser correctly chose the *alphanx1v[work]* derivation tree. However, the parser now selects the *alphanx0Vnx2nx1[do]* derivation tree. This derivation tree is shown in Figure 5.5.

This tree is incorrect, in that it incorrectly assumes that the word *work* is a noun. Therefore, the method of minimal attachment fails for this example. It is, however, easy to correct this problem for this particular example by simply providing the correct word categories for each of the words in the sentence. This reduces the number of parses generated by the parser, and also

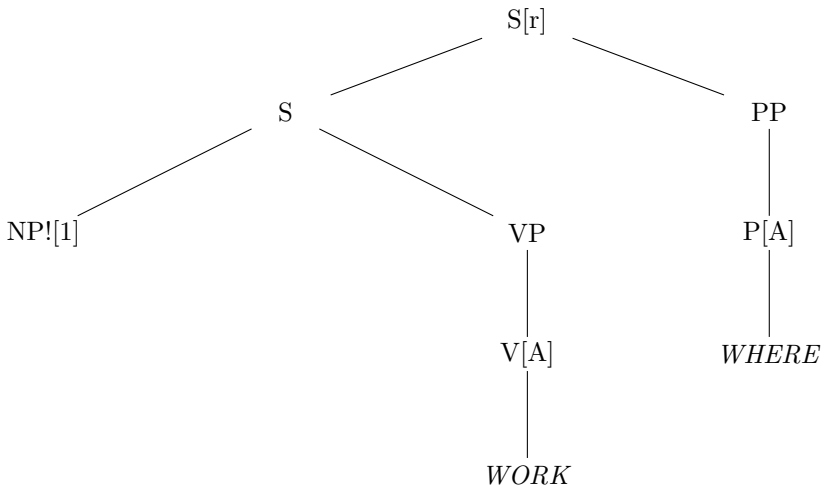


Figure 5.4: Final translated output for *Where do you work?*

reduces the likelihood of any ambiguities arising in the parses.

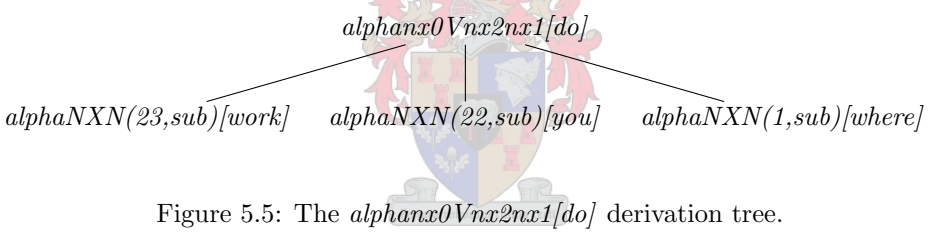


Figure 5.5: The *alphanx0Vnx2nx1[do]* derivation tree.

Just as for the XTAG parser, our system already allows the user to select a particular derivation tree from all the generated derivation trees. However, it would be interesting to see whether the method of right association discussed in Chapter 3 would provide better derivation tree selection. Another improvement would be to incorporate machine learning techniques in the parser. These techniques apply corpus based knowledge to derivation tree selection, as mentioned in Chapter 3. These improvements, however, remains a topic for future work.

The next potential problem we look at is rule ambiguities.

5.3.2 Rule Ambiguities - a Linguistic Problem

This problem arises from the data used to derive rules for the translation system. In particular, it arises because of seemingly conflicting translations in the SASL phrase book. For example, the sentence *I need medicine* is translated as *ILL MEDICINE NEED*. However, the sentence *I need a doctor* is translated as *NEED DOCTOR*. The derivation trees for both these sentences are shown in Figures 5.6 and 5.7.

The translations for both these derivation trees are governed by almost the same translation rules. These rules are:

5; alphanx0Vnx1[FEAT-Inf,POS-V] = alphaInx0Vnx1[]
5; alphaNXN[FEAT-3sg,POS-N] = alphaNXN[] .

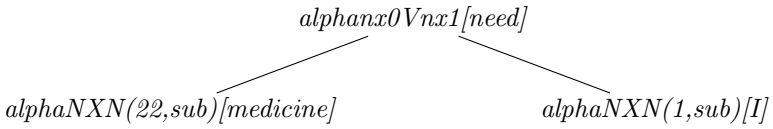


Figure 5.6: Derivation tree for *I need medicine*.

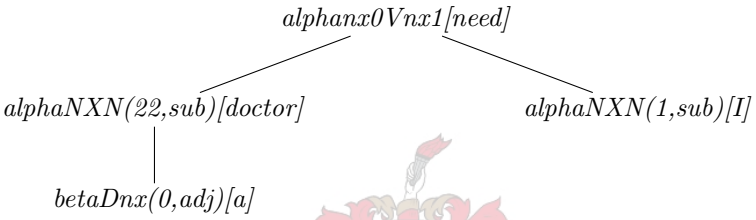


Figure 5.7: Derivation tree for *I need a doctor*.

The only difference between the two translations is that the tree *betaDnx[a]* falls away in the translation of *I need a doctor*. If we apply the translation rules to the derivation tree for *I need a doctor*, we get *NEED DOCTOR*. If the same rules are applied to *I need medicine*, then the translation is *NEED MEDICINE*. However, according to the phrase book the translation should be *ILL MEDICINE NEED*. This is not as big a problem as it may appear at first. Word order in SASL is not as important as in English [23], and therefore the translation *NEED MEDICINE* can just as easily be understood as the translation *ILL MEDICINE NEED*. The reason why there is an explicit *ILL* gloss at the start of the translation is to provide context.

Differing word order in the SASL translations also appear in question sentences such as, *What is your name?* translated as *YOUR NAME WHAT* and *What do you want to eat?* translated as *WHAT EAT X*. Again, whether the *WHAT* gloss came at the end or the beginning of the sentence would not in general prevent the translation from being understood correctly.

The last potential problem we look at is the problem of using glosses.

5.3.3 Glosses

Although glosses provide for a convenient way of dealing with signs in SASL, they are limited in the amount of information they carry. Glosses in themselves do not provide adequate non-manual information. This type of information needs to be represented in some way. Usually, non-manual information is represented by drawing a line over the affected signs. For example, when a sentence is negated, the signer usually makes a facial expression expressing displeasure. This is illustrated in the translation of the sentence, *I do not understand*.

$$\overline{\text{UNDERSTAND NOTHING}}^{neg}$$

Future expansion of this system includes the addition of a pre-processing component which will add prosodic information to the TAG translations [14].

Apart from non-manual signs, the signer makes use of the space around them to convey additional information. This additional information can either be grammatical, descriptive or discourse related [25]. Glosses by themselves do not provide this extra information. There are various ways to handle this problem; [25] proposes a "spatial interlingua", where a 3D model of a scene is used to solve this problem of additional information faced by glosses. Future expansion of our prototype system includes the addition of a scene graph component [14]. This scene graph will represent discourse related information, such as classifier predicates.

In this section we discussed three possible problems of our prototype machine translation system. In the next section we discuss the time and space complexity of the implemented parsing algorithm used in the system.

5.4 Time and Space Complexity

The parser implemented in the prototype system consists of three separate phases. The first pre-processing phase involves the building of the elementary trees needed by the parser to execute the parsing algorithm. Some of the issues facing this phase was discussed in Chapter 4. The second phase is the parsing algorithm described in Chapter 3. The time and space complexity for this algorithm is discussed in [28]. The worst case time complexity for the second phase is:

$$O(|A||A \cup I|Nn^6) \quad .$$

Here $|A|$ is the number of auxiliary trees, and $A \cup I$ the number of elementary trees used by the parsing algorithm. N is the largest number of nodes in an elementary tree, and lastly n is the length of the input. The worst case space complexity for the second phase is:

$$O(|A||A \cup I|Nn^6) \quad .$$

The reason why the space complexity is the same as the time complexity, is because of back-pointers. Each item has a set of back-pointers that keep track of the items causing that particular item to be added to the chart. In the worst case scenario there is an item added during each step of the algorithm, resulting in the worst case space complexity being the same as the worst case time complexity.

When parsing is complete, all the successful parses are represented by a graph formed by all the back-pointers. The size of this graph is in the worst case:

$$O(|G|^2 Nn^6) \quad .$$

Here $|G|$ is the size of the TAG grammar. The third phase now follows the back-pointers through this graph to build the possible derivation trees for each of the successful parses. This enumeration of possible derivation trees can take exponential time depending on the ambiguity inherent in the TAG grammar [28].

The parsing algorithm employed by the system can be improved in a number of ways. One way is to extend the parsing algorithm to make use of unification-based feature structures [28]. Feature structures can be used as restrictions on adjunctions, with an adjunction allowed if unification succeeds. Feature structures are used in the XTAG grammar to model various grammatical restrictions, such as number agreement [49]. Not implementing feature structures did not affect the accuracy of the parses generated by the parser. However, by implementing them, many ungrammatical parses would be eliminated out early on in the parsing process, resulting in overall faster parsing times.

The second way in which the parser can be improved is to implement a TAG parser which guarantee the valid-prefix property [31]. The valid-prefix property ensures that errors in the input

is observed as soon as possible. This is accomplished by ensuring that all partial parses generated by the parser are valid prefixes of the language defined by the TAG grammar [28].

Thirdly, it is possible to improve the space complexity of the parser. In [48] the authors show that it is possible to improve the worst case space complexity to $O(n^4)$. They do this by using a context-free grammar (or alternatively a linear indexing grammar) to represent the positions of where adjunction was performed during the parse. The implementation of this improvement is also a topic for future study.

Lastly, a system by which our machine translation system can be evaluated needs to be implemented. This evaluation systems should consist of two parts. The first part should evaluate the overall accuracy of the parses created by the TAG parser. This can be done in line with the general approach taken to evaluate the accuracy as the XTAG grammar as discussed in [41]. The second part should evaluate the overall accuracy of the translations generated by our system. However, in order to obtain useful results from this second part of the evaluation system, we first need to greatly increase the number of translation rules that cover a wider set of linguistic phenomena.

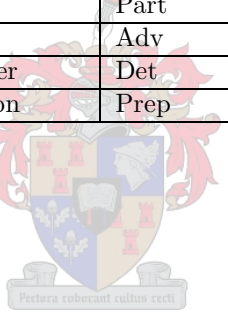
This section discussed the time and space complexity of the parsing algorithm. Some comments were made on improving the parsing algorithm and evaluating the machine translation system.

In this final chapter we again looked at the original goals of this thesis. Various potential problems were highlighted and possible future work was discussed. Although only in its infancy, we believe that this prototype machine translation system forms a good basis for future development. And with proper improvement and expansions, this system will result in a tool that will be of practical value to the South African Deaf community.

Appendix A

List of Parts of Speech Abbreviations

Part of Speech	**Abbreviation**
Pronoun	Pron
Proper Noun	PropN
Noun	N
Adjective	A
Verb	V
Participle	Part
Adverb	Adv
Determiner	Det
Preposition	Prep



Appendix B

Contents of Translation Database

```
<TRANS>
  <alphaNXN>
    <RULE number="12" features="3sg" TransFeature="" POS="PropN">
      <TRANSTREENAME>alphaNXN</TRANSTREENAME>
    </RULE>
    <RULE number="12" features="3sg" TransFeature="" POS="N">
      <TRANSTREENAME>alphaNXN</TRANSTREENAME>
    </RULE>
    <RULE number="1" features="3sg" TransFeature="" POS="N">
      <TRANSTREENAME>alphaNXN</TRANSTREENAME>
    </RULE>
    <RULE number="1" features="3pl" TransFeature="" POS="N">
      <TRANSTREENAME>alphaNXN</TRANSTREENAME>
    </RULE>
    <RULE number="1" features="1sg" TransFeature="" POS="Pron">
      <TRANSTREENAME>alphaNXN</TRANSTREENAME>
    </RULE>
    <RULE number="2" features="3sg" TransFeature="" POS="N">
      <TRANSTREENAME>alphaNXN</TRANSTREENAME>
    </RULE>
    <RULE number="6" features="3sg" POS="PropN">
      <SPECRULE>
        <TREE>alphaNXN</TREE>
        <OPERATION>sub</OPERATION>
        <TARGET>alphannv</TARGET>
        <NODE>1</NODE>
      </SPECRULE>
    </RULE>
    <RULE number="5" features="3sg" TransFeature="" POS="N">
      <TRANSTREENAME>alphaNXN</TRANSTREENAME>
    </RULE>
    <RULE number="5" features="3pl" TransFeature="" POS="N">
      <TRANSTREENAME>alphaNXN</TRANSTREENAME>
    </RULE>
  </alphaNXN>
  <alphanx0Vnx1>
    <RULE number="12" features="PAST" TransFeature="" POS="V">
      <TRANSTREENAME>BEFOREalphanx0Vnx1</TRANSTREENAME>
      <SNODE>
```

```
      <NODENAME>1</NODENAME>
      <TRANSNODENAME>2</TRANSNODENAME>
    </SNODE>
    <SNODE>
      <NODENAME>22</NODENAME>
      <TRANSNODENAME>32</TRANSNODENAME>
    </SNODE>
  </RULE>
  <RULE number="12" features="PAST" TransFeature="" POS="V">
    <TRANSTREENAME>BEFOREalphanx0Vnx1</TRANSTREENAME>
  </RULE>
  <RULE number="11" features="Inf" TransFeature="" POS="V">
    <TRANSTREENAME>alphannv</TRANSTREENAME>
    <SNODE>
      <NODENAME>22</NODENAME>
      <TRANSNODENAME>2</TRANSNODENAME>
    </SNODE>
  </RULE>
  <RULE number="5" features="Inf" TransFeature="" POS="V">
    <TRANSTREENAME>alphaInx0Vnx1</TRANSTREENAME>
  </RULE>
</alphanx0Vnx1>
<alphaInx0Vnx1>
  <RULE number="1" features="Inf" TransFeature="" POS="V">
    <TRANSTREENAME>alphaInx0Vnx1</TRANSTREENAME>
  </RULE>
  <RULE number="4" features="Inf" TransFeature="Direct" POS="V">
    <TRANSTREENAME>alphaInx0Vnx1</TRANSTREENAME>
  </RULE>
</alphaInx0Vnx1>
<betasARB>
  <RULE number="1" features="" TransFeature="" POS="Adv">
    <TRANSTREENAME>betasARB</TRANSTREENAME>
  </RULE>
</betasARB>
<betaARBs>
  <RULE number="1" features="" TransFeature="" POS="Adv">
    <TRANSTREENAME>betasARB</TRANSTREENAME>
  </RULE>
</betaARBs>
<betavxN>
  <RULE number="4" features="3sg" TransFeature="" POS="N">
    <TRANSTREENAME>betavxN</TRANSTREENAME>
  </RULE>
  <RULE number="3" features="3sg" TransFeature="" POS="N">
    <TRANSTREENAME>alphaNXN</TRANSTREENAME>
  </RULE>
</betavxN>
<betavxARB>
  <RULE number="4" features="" TransFeature="" POS="Adv">
    <TRANSTREENAME>betavxARB</TRANSTREENAME>
  </RULE>
</betavxARB>
<alphanx1V>
```

```
    <RULE number="8" features="Inf" TransFeature="" POS="V">
      <TRANSTREENAME>alphanx1V</TRANSTREENAME>
    </RULE>
    <RULE number="2" features="Past" TransFeature="" POS="V">
      <TRANSTREENAME>alphanx1V</TRANSTREENAME>
    </RULE>
    <RULE number="3" features="Inf" TransFeature="" POS="V">
      <TRANSTREENAME>alphaVnx1</TRANSTREENAME>
      <SNODE>
        <NODENAME>2</NODENAME>
        <TRANSNODENAME>2</TRANSNODENAME>
        <NODEOP>adj</NODEOP>
        <TRANSNODEOP>sub</TRANSNODEOP>
      </SNODE>
    </RULE>
  </alphanx1V>
  <betaPs>
    <RULE number="8" features="" TransFeature="" POS="Prep">
      <TRANSTREENAME>betasP</TRANSTREENAME>
      <SPECRULE>
        <TREE>betaPs</TREE>
        <OPERATION>adj</OPERATION>
        <TARGET>alphanx1V</TARGET>
        <NODE>0</NODE>
      </SPECRULE>
    </RULE>
  </betaPs>
  <alphaW1nx1V>
    <RULE number="6" features="Past" TransFeature="" POS="V">
      <TRANSTREENAME>alphannv</TRANSTREENAME>
      <SNODE>
        <NODENAME>1</NODENAME>
        <TRANSNODENAME>2</TRANSNODENAME>
      </SNODE>
    </RULE>
  </alphaW1nx1V>
  <alphanxOVnx1>
    <RULE number="11" features="Inf" TransFeature="" POS="V">
      <TRANSTREENAME>alphannv</TRANSTREENAME>
      <SNODE>
        <NODENAME>1</NODENAME>
        <TRANSNODENAME>1</TRANSNODENAME>
      </SNODE>
    </RULE>
  </alphanxOVnx1>
  <alphanxON1>
    <RULE number="13" features="3sg" TransFeature="" POS="N">
      <TRANSTREENAME>alphanxON1</TRANSTREENAME>
    </RULE>
  </alphanxON1>
</TRANS>
```

Appendix C

SASL Phrase Book

Phrase Book Glosses
Interpreter : H. Fourie
Signer : B. Botha
The X occurring at the end of some of the phrases indicates a discourse marker. This seems to indicate that the other person must now respond.
The SJOE gloss is a sign usually indicating effort or exacerbation.

Greetings	
Good day.	GOOD DAY
Good afternoon.	GOOD AFTERNOON
Good morning.	GOOD MORNING
Good evening.	GOOD EVENING
Sleep well.	ENJOY SLEEP
Hello.	HELLO
Goodbye.	BYE
See you soon.	SEE SOON (verb inflected to indicate 2nd person)
See you tomorrow.	SEE TOMORROW
See you next week.	SEE NEXT WEEK
I work very hard.	WORK HARD SJOE
How are you?	HEALTHY
I haven't seen (inflected) you for a long time.	SEE LONG AGO (SEE inflected to include 2nd person)
How is your mother?	YOUR MOTHER GOOD
I am very busy	BUSY VERY
Come, we must go now.	COME DEPART

Introductions / Small talk	
Nice to meet you.	NICE MEET (inflected to include 2nd person)
How old are you?	OLD YOU
My name is Bennie.	MY NAME WHAT B-E-N-N-I-E (fingerspell) BENNIE (name sign)
Where do you go to school?	SCHOOL GO WHERE
This is my wife Sara.	MY WIFE NAME S-A-R-A (fingerspell) SARA (name sign)
Do you like animals?	LIKE ANIMALS X
Is your husband deaf?	YOUR HUSBAND DEAF HE
Is your wife deaf?	YOUR WIFE DEAF
Do you have children?	CHILDREN HAVE YOU
Let's go drink coffee.	WE DRINK COFFEE
Let's go watch a movie.	WE GO MOVIE
Are you married?	MARRIED DONE
We must go have lunch one day.	TOGETHER GO EAT
Sara, this is Bennie.	THAT PERSON NAME WHAT B-E-N-N-I-E
Where do you work?	WORK WHERE
Where do you live?	SLEEP STAY WHERE
What is your name?	YOUR NAME WHAT

Communication	
I can speak a little sign language.	SIGN LANGUAGE SO-SO
I can't speak sign language very well.	SIGN LANGUAGE GOOD NO
I can't fingerspell very well.	FINGERSPELL GOOD NO
I am learning sign language.	LEARN SIGN LANGUAGE
I do not understand.	UNDERSTAND NOTHING
I do not understand that word.	(point in space) WORD UNDERSTAND NOTHING
Please repeat.	AGAIN PLEASE
You are signing too fast.	SIGN LANGUAGE FAST
Please talk slower.	SIGN LANGUAGE SLOW PLEASE
Fingerspell slower please.	FINGERSPELL SLOW PLEASE
Please excuse me.	EXCUSE ME PLEASE

In the home: Parents and children	
Please clear the table.	PLATES CUPS AWAY PLEASE
It is bath time.	BATH (verb) TIME
It is bed time.	BED TIME
Eat your carrots.	EAT CARROT
Have you done your homework?	WRITING DONE
Have you brushed your teeth?	TEETH BRUSH DONE
Do you have sport practice tomorrow afternoon?	TOMORROW AFTERNOON SPORT PRACTICE
Careful!	CARE
Are you hungry?	YOU HUNGER X
Are you thirsty?	YOU THIRSTY X
Yes, you may go.	LIKE YOU GO
Yes, you may go but you must be back by 2 o'clock.	GO TIME TWO HERE
Please close the door.	DOOR SHUT PLEASE
Please close the window.	WINDOW SHUT PLEASE
Please clean your room.	YOUR ROOM CLEAN PLEASE
Hurry up, we are late for school.	SCHOOL LATE HURRY
Please do not jump on your bed.	SLEEP BED JUMP DON'T
Watch out! (warning)	WATCH OUT
What do you want to drink?	WHAT DRINK X
What do you want to eat?	WHAT EAT X
Do you like chocolate?	LIKE CHOCOLATE WHAT X
Please turn off the television.	TV (TURN OFF) PLEASE
Please put your clothes in the wash bin.	YOUR CLOTHES WASH BIN THROW PLEASE
Please turn the TV's volume down.	TV VOLUME LESS PLEASE
Oops!	OOPS BAD
Please take off your school clothes.	SCHOOL CLOTHES (TAKE OFF) PLEASE
Put on your sweater, it is cold outside.	SWEATER (PUT ON) OUTSIDE COLD
Where is your book?	BOOK WHERE

Travel	
The flight to Johannesburg is cancelled.	JOHANNESBURG FLIGHT SCRAP
The flight to Johannesburg was delayed by two hours.	JOHANNESBURG FLIGHT TWO HOURS LATE
Can you please explain to me how to get to the airport?	GO AIRPORT HOW EXPLAIN PLEASE
Excuse me, is this the train to Stellenbosch?	EXCUSE ME THIS TRAIN GO STELLENBOSCH (fingerspell) STELLENBOSCH (name sign) RIGHT
Can you please explain to me how to get to the station?	I GO TRAIN STATION HOW EXPLAIN PLEASE
Where is the hotel?	I GO HOTEL HOW EXPLAIN PLEASE
Where is Pleinstreet?	PLEIN (fingerspell) STREET WHERE
Is this bus going to Stellenbosch?	THIS BUS GO PLACE STELLENBOSCH RIGHT

Health	
I feel sick.	SICK
My stomach hurts.	STOMACH TURN TURN
I need medicine.	SICK MEDICINE NEED
I need a doctor.	NEED DOCTOR
To what medical aid scheme do you belong?	YOU DOCTOR SCHEME WHAT
Please call an ambulance.	CALL AMBULANCE PLEASE
I feel ill.	FEEL ILL
I don't feel good.	FEEL BAD
Where does it hurt?	BODY HURT SJOE
Are you pregnant?	BABY BIG STOMACH YOU
Are you hurt?	BODY PAIN HAVE
I have a cold.	COLD HAVE

Law	
Please call the police.	CALL POLICE PLEASE
The thief ran away.	THIEF AWAY
This is an emergency.	BAD BAD BAD
I was raped.	RAPE
A thief stole my wallet.	THIEF MY WALLET STEAL
He stole my cellphone.	CELLPHONE STEAL
I want to report a burglary.	(I) GO POLICE FORM COMPLETE
I got a speeding fine.	FLASH FINE GOT
Is everything all right?	EVERYTHING GOOD
I was hi-jacked.	HI-JACK
My car has been stolen.	MY CAR STEAL

Bibliography

[1] WordNet - http://wordnet.princeton.edu/.

[2] Definite Clause Grammars - http://www.cs.sunysb.edu/~sbprolog/manual1/node87.html.

[3] HamNoSys, Sign Language Notation System - http://www.sign-lang.uni-hamburg.de/Projects/HamNoSys.html.

[4] Extensible Markup Language (XML) 1.0 - http://www.w3.org/TR/1998/REC-xml-19980210.

[5] W3C Document Object Model - http://www.w3.org/DOM/.

[6] SAX - http://www.saxproject.org/.

[7] Synchronous Tree Adjoining Grammars - http://www.cis.upenn.edu/~xtag/stag/index.html.

[8] Synchronous Tree Adjoining Grammars, Algorithms and Implementation - http://www.cis.upenn.edu/~xtag/stag/algorithms.html.

[9] SASL Phrase Book - http://www.cs.sun.ac.za/~lynette/SASL.

[10] Alonso M.A., Cabrero D., de la Clergerie E. and Vilares M. (1999). Tabular Algorithms for TAG Parsing. In *Proceedings of the 9th Conference of the European Chapter of the Association for Computational Linguistics*, pages 150–157, Bergen, Norway. http://citeseer.ist.psu.edu/article/alonso99tabular.html.

[11] Bangham J.A., Cox S.J., Elliot R., Glauert J.R.W. and Marshall I. (2000). *Virtual Signing: Capture, Animation, Storage and Transmission - an Overview of the ViSiCAST Project.* IEE Seminar on "Speech and language processing for disabled and elderly people", London. www.visicast.sys.uea.ac.uk/Papers/iee2000-04PaperBFinal.pdf.

[12] Brown P.F., Della Pietra S.A., Della Pietra V.J. and Mercer R.L. (1993). The Mathematics of Statistical Machine Translation: Parameter Estimation. *Computational Linguistics*, 19(2):263–311.

[13] Bungeroth J. and Ney H. (2004). Statistical Sign Language Translation. In *Proceedings of the Workshop on Representation and Processing of Sign Languages, 4th Internation Conference on Language Resources and Evaluation (LREC 2004)*, pages 105–108.

[14] Combrink A. (2005). Adaptive Animation: a Preprocessor for an English-to-Sign Language Machine Translation System. MSc thesis, University of Stellenbosch. http://www.cs.sun.ac.za/~lynette/publications/acombrink.pdf.

[15] Daniel K., Schabes Y., Zaidel M. and Egedi D. (1992). A Freely Available Wide Coverage Morphological Analyzer for English. In *Proceedings of the 14th International Conference on Computational Linguistics (COLING-92).* http://citeseer.ist.psu.edu/daniel92freely.html.

[16] Daz V.J, Carillo V. and Alonso M.A. (2002). A Left Corner Parser for Tree Adjoining Grammars. In *Proceedings of the Sixth International Workshop on Tree Adjoining Grammars and Related Frameworks (TAG+6)*, pages 90–95.

[17] Dorr B., Jordan P. and Benoit J. (1998). A Survey of Current Paradigms in Machine Translation. Technical Report LAMP-TR-027, Language and Media Processing Lab, University of Maryland. `http://citeseer.csail.mit.edu/555445.html`.

[18] Egedi D. and Martin P. (1994). A Freely Available Syntactic Lexicon for English. In *Proceedings of the International Workshop on Shareable Natural Language Resources*. `http://citeseer.ist.psu.edu/egedi94freely.html`.

[19] Fabian P. and Francik J. (2001). Synthesis and Presentation of the Polish Sign Language Gestures. In *Proceedings of the 1st International Conference on Applied Mathematics at Universities*, pages 190–197. `http://sun.iinf.pols1.gliwice.pl/sign/pub/pe06.pdf`.

[20] Fontinea S., Efthimiou E. and Kouremenos D. (2005). Generating Linguistic Content for Greek to GSL Conversion. In *Proceedings of the 7th Hellenic European Conference on Computer Mathematics & its Applications*. `http://www.aueb.gr/pympe/hercma/proceedings2005/H05-FULL-PAPERS-1/FOTINEA-EFTHYMIOU-KOUREMENOS-1.pdf`.

[21] Grinberg D., Lafferty J. and Sleator D. (1995). A Robust Parsing Algorithm for Link Grammars. Technical Report CMU-CS-95-125, Computer Science Department, Carnegie Mellon University. `http://citeseer.csail.mit.edu/grinberg95robust.html`.

[22] Grishman R. (1994). *Computational Linguistics: An Introduction*, pages 27–33. Cambridge University Press, New York, USA.

[23] Hough R.P. (1998). *Beginsels vir 'n Bybelvertaling vir Afrikaanse Dowes*. PhD thesis, University of Stellenbosch.

[24] Huenerfauth M. (2003). A Survey and Critique of American Sign Language Natural Language Generation and Machine Translation Systems. Technical report, Computer and Information Sciences, University of Pennsylvania. `http://www.seas.upenn.edu/~matthewh/publications/huenerfauth-2003-ms-cis-03-32-asl-nlg-mt-survey.pdf`.

[25] Huenerfauth M. (2004). A Multi-Path Architecture for Machine Translation of English Text into American Sign Language Animation. In *Proceedings of the Student Workshop at Human Language Technologies Conference HLT-NAACL*. `http://www.seas.upenn.edu/~matthewh/publications/huenerfauth-2004-hlt-naacl-multipath-architecture-english-asl.pdf`.

[26] Hutchins J. (1986). *Machine Translation: Past, Present, Future*. Ellis Horwood, Chichester. `http://ourworld.compuserve.com/homePAGES/WJHutchins/PPF-TOC.htm`.

[27] Hutchins J. (1999). The Development and Use of Machine Translation Systems and Computer-based Translation Tools. In *Proceedings of the International Symposium on Machine Translation and Computer Language Information Processing*. `http://ourworld.compuserve.com/homepages/wjhutchins/Beijing.pdf`.

[28] Joshi A.K. and Schabes Y. (1997). *Handbook of Formal Languages*, volume 3, pages 69–120. Springer-Verlag, Berlin-Heidelberg.

[29] Marshall I. and Safar E. (2002). Sign Language Generation using HPSG. In *Proceedings of the 9th International Conference on Theoretical and Methodological Issues in Machine Translation*. `http://citeseer.ist.psu.edu/565062.html`.

[30] Morrissey S. and Way A. (2005). An Example-Based Approach to Translating Sign Language. In *Proceedings of Second Workshop on Example-Based Machine Translation*, pages 109–116. `http://www.mt-archive.info/MTS-2005-Morrissey.pdf`.

[31] Nederhof M-J. (1997). Solving the Correct-Prefix Property for TAGs. In *Proceedings of the Fifth Meeting on Mathematics of Language (MOL 5)*, pages 124–130. `http://citeseer.ist.psu.edu/72252.html`.

[32] Nederhof M-J (1999). The Computational Complexity of the Correct-Prefix Property for TAGs. *Computational Linguitics*, 25(3):345–360. `http://acl.ldc.upenn.edu/J/J99/J99-3002.pdf`.

[33] Neidle C., Lee R.G., Kegl J., MacLuaghlin D., Bahan B. (1999). *The Syntax of American Sign Language*, pages 7–25. MIT Press, Cambridge, MA.

[34] Safar E. and Marshall I. (2001). The Architecture of an English-Text-to-Sign-Languages Translation System. *Recent Advances in Natural Language Processing*, pages 223–228.

[35] Sapountzaki G., Efthimiou E., Karpouzis K. and Kourbetis V. (2004). Developing an e-Learning Platform for Greek Sign Language. In *Proceedings of the 4th International Conference on Language Resources and Evaluation (LREC)*. `http://www.image.ece.ntua.gr/publications.php`.

[36] Shieber S.M. (1994). Restricting the Weak-Generative Capacity of Synchronous Tree-Adjoining Grammars. 10(4):371–385. `http://www.eecs.harvard.edu/~shieber/Biblio/Papers/synch-deriv.pdf`.

[37] Shieber S.M. (2006). Unifying Synchronous Tree-Adjoining Grammars and Tree Transducers via Bimorphisms. In *Proceedings of the 11th Conference of the European Chapter of the Association for Computational Linguistics (EACL-06)*, Trento, Italy. `http://www.eecs.harvard.edu/shieber/Biblio/Papers/stag-bimorphism.pdf`.

[38] Shieber S.M. and Schabes Y. (1990). Synchronous Tree-Adjoining Grammars. In *Proceedings of the 13th International Conference on Computational Linguistics*, pages 253–258. `http://citeseer.ist.psu.edu/shieber90synchronous.html`.

[39] Sleator D. and Temperley D. (1991). Parsing English with a Link Grammar. Technical Report CMU-CS-91-196, Computer Science Department, Carnegie Mellon University. `http://citeseer.csail.mit.edu/ARTICLE/sleator91parsing.html`.

[40] Srinivas B. and Joshi A.K. (1994). Disambiguation of Super Parts of Speech (or Supertags): Almost Parsing. In *Proceedings of the International Conference on Computational Linguistics (COLING 94)*, Kyoto University, Japan. `http://citeseer.ist.psu.edu/article/joshi95disambiguation.html`.

[41] Srinivas B., Sarkar A., Hockey B.A. and Doran C. (1998). Grammar and Parser Evaluation in the XTAG Project. In *Proceedings of the International Conference on Language Resourcers and Evaluation (LREC) Workshop on the Evaluation of Parsing Systems*, Grenada, Spain. `http://www.cs.sfu.ca/~anoop/papers/pdf/eval-final.pdf`.

[42] Stein D. (2005). Morpho-Syntax Based Statistical Methods for Sign Language. Diploma thesis, RWTH Aachen University. `http://www-i6.informatik.rwth-aachen.de/~stein/diplomarbeit.pdf`.

[43] Suszczanska N., Szmal P. and Francik J. (2002). Translating Polish Texts into Sign Language in the TGT system. In *Proceedings of the 20th IASTED International Multi-Conference, Applied Inofrmatics AI 2002*, pages 282–287. `http://sun.iinf.polsl.gliwice.pl/sign/pub/pe01.pdf`.

[44] Szmal P. and Suszczanska N. (2001). Selected Problems of Translation from the Polish Written Language to the Sign Language. *Archiwum Informatyki Teoretycznej i Stosowanej*, 13(1):37–51. `http://sun.iinf.polsl.gliwice.pl/sign/pe03.pdf`.

[45] Trujillo A. (1999). *Translation Engines: Techniques for Machine Translation*. Springer-Verlag, London.

[46] Van Noord G. (1994). Head-Corner Parsing for TAG. *Computational Intelligence*, 10(4):525–534. `http://citeseer.ist.psu.edu/vannoord94head.html`.

[47] Veale T., Conway A. and Collins B. (1998). The Challenges of Cross-Modal Translation: English to Sign Language Translation in the ZARDOZ System. *Machine Translation*, 13(1):81–106.

[48] Vijay-Shanker K. and Weir D.J. (1993). The Use of Shared Forests in TAG Parsing. In *Proceedings of the 6th Meeting on the European Chapter of the Association for Computational Linguistics*, pages 384–393, Utrecht, The Netherlands. `http://acl.ldc.upenn.edu/E/E93/E93-1045.pdf`.

[49] XTAG Research Group (2001). A Lexicalized Tree Adjoining Grammar for English. Technical Report IRCS-01-03, IRCS, University of Pennsylvania. `http://www.cis.upenn.edu/~xtag/gramrelease.html`.

[50] Zhao L., Kipper K., Schuler W., Vogler C., Badler N. and Palmer M. (2000). A Machine Translation System from English to American Sign Language. In *Proceedings of the 4th Conference of the Association for Machine Translation in the Americas*. `http://www.cis.upenn.edu/~lwzhao/papers/amta00.pdf`.