The Understanding Computer

- Natural Language Understanding in Practice

Preliminary Version

Tore Amble

September 9, 2004

Abstract

TUC—The Understanding Computer—is the title of a natural language processing (NLP) system that is developed at the Norwegian University of Science and Technology (**NTNU**). Its primary goal is to be a generic flexible NLP system in which NLP applications can be implemented with moderate ease. The purpose of this text is twofold; to make a documentation of the principles and techniques that are behind TUC, and to serve as lecture notes for a course on Natural Language Interfaces that is given at NTNU.

Acknowledgements

I wish to thank the following persons, former students at our department, for help during the evolution of TUC, in chronological order:

John Teigen, Vidar Vetland, Katrine Wøien, Espen Wøien, Jon Bratseth, Jone Gudmestad, Anders Mathisen, Thomas Oldervold, Erik Overrein, Øystein Fledsberg, Kim Bjerkevoll, Einar Føystad Dørum, Truong Van Le, Johan Braanen and Anders Andenæs, Rune Sætre and Martin Thorsen Ranang.

Especially, Anders Andenæs has kindly lent some of his written material from his Masters thesis [**?**] for use in these lecture notes, esp. the chapter on Linguistics for non linguists, and GeneTUC. Finally, I would like to thank Martin Thorsen Ranang for skillful help with the editing process using LaTeX 2_ε.

Preface

The purpose of this text is to make a documentation of the principles and techniques used for natural language understanding systems. It is made for a lecture note in a course on computational linguistics held at NTNU.

The intended course will be a part of the computer science curriculum, whereby the students are assumed to be aquainted with the following topics:
First order predicate logic, lambda calculus, relational databases, regular and context free grammars, the programming language Prolog, and the elements of Artificial Intelligence (AI).

On the other hand, no prerequisites are required from the topics of linguistics and natural languages.

The course in general, and this book in particular will be focused on the interesting topic of natural language understanding, with a perspective to AI, more than linguistic theories. The work and competence behind this report is an ongoing project called The Understanding Computer, which aims, and in part has succeeded in building useful NLP applications based on a proper understanding of the texts. Whatever the material lacks in theory, it has proved its worth by providing workable systems.

This is a preliminary version, that will be improved during the course of time.

Tore Amble,
August 2004

Contents

1 Introduction **1**

1.1 Computational Linguistics 1
1.2 Knowledge-Based Natural Language Systems 1
1.3 TUC—The Understanding Computer 2
1.3.1 BusTUC 2
1.3.2 GeneTUC 3
1.3.3 General Methodology 4

2 Linguistics for Non-Linguists **5**

2.1 Word Classes 5
2.1.1 Verbs 5
2.1.2 Nouns 6
2.1.3 Adjectives 6
2.1.4 Adverbs 7
2.1.5 Pronouns 7
2.1.6 Prepositions 8
2.1.7 Conjunctions 8
2.2 Sentence Categories 8
2.2.1 Declarative 8
2.2.2 Imperative 9
2.2.3 Interrogative 9
2.3 Extragrammatical Phenomena 9
2.3.1 Ellipsis 9
2.3.2 Anaphora 9
2.3.3 Garden-Path Sentences 10
2.3.4 Metaphor 10
2.4 Lexical Semantics 10
2.4.1 Synonymy 10
2.4.2 Homonymy 11
2.4.3 Polysemy 11
2.4.4 Hyponymy 11
2.5 Punctuation 11
2.5.1 Comma 11
2.5.2 Semicolon 12
2.5.3 Colon 13
2.6 Grammatical Relations 13
2.7 Ambiguities 13

3 Grammar for a Fragment of English **17**
3.1 Introduction . . . 17
3.1.1 Context Free Grammar . . . 17
3.2 The Elements of Grammar (Parts of Speech) . . . 19
3.3 Comments to Word Categories . . . 19
3.4 Syntax of Statement . . . 21
3.5 Context Sensitive Grammars . . . 22
3.5.1 Modified Categorial Grammar . . . 23
3.6 Simplified Grammar for Statement . . . 26
3.7 Simplified Grammar for Commands and Questions . . . 28
3.7.1 Comments on the syntax description . . . 30
3.7.2 Advanced Topics in Grammar . . . 31
3.8 EXERCISES . . . 33

4 Logic and Knowledge Bases **35**
4.1 Relational Databases . . . 35
4.1.1 A Relational Example . . . 35
4.1.2 Binary Relations . . . 37
4.1.3 Composite Keys . . . 37
4.1.4 Data Base Retrieval . . . 37
4.1.5 Virtual Tables . . . 38
4.1.6 Symbolic Naming . . . 39
4.2 Data Modelling . . . 40
4.2.1 Normal Forms . . . 40
4.2.2 Relational Normal Forms . . . 41
4.3 Beyond the Relational Model . . . 41
4.4 Semantic Nets . . . 42
4.4.1 The Class Concept . . . 42
4.4.2 Another Example: A Course Knowledge Base . . . 47
4.4.3 The Course Model . . . 47
4.4.4 Coupling Semantic Nets to Tables . . . 50
4.4.5 Example of Questions Asked . . . 50
4.5 Issues on Semantic Nets . . . 51
4.5.1 Flexible Attribute Classes . . . 51
4.5.2 Inheritance in a Hierarchy . . . 52
4.5.3 Inheritance in a Heterarchy . . . 52
4.6 EXERCISES . . . 55

5 Natural Language Processing in Prolog **57**
5.1 Natural Language Systems in Prolog . . . 57
5.2 Natural Language Query Processing Overview . . . 58
5.2.1 Verb-Free Language . . . 58
5.2.2 The Dialog Context . . . 59
5.2.3 The Reference Model . . . 60
5.3 Lexical Analysis . . . 61
5.4 Syntax Analysis . . . 63
5.4.1 Definitie Clause Grammars (DCG) . . . 64
5.4.2 Attributes . . . 65
5.5 Generating Syntax Trees . . . 66
5.6 Attribute Grammars . . . 67
5.7 Selectional Restrictions . . . 69
5.8 Text Scanner in Prolog . . . 72
5.9 EXERCISES . . . 74

6 Advanced NLP in Prolog **77**
6.1 Metaparser in Prolog 77
6.2 Parsing with Ambiguities 79
6.3 Parsing with Probabilistic Grammars 81
6.4 Feature Structures 82
6.4.1 Feature Structures in the Grammar 85
6.5 Parsing Strategies 88
6.6 Consensical Grammar 89
6.6.1 Language Hierarchy 89
6.6.2 Consensical Grammar Metaparser 96
6.7 EXERCISES 98

7 Computational Semantics **99**
7.1 Proper Treatment of Quantifiers 102
7.2 Scoping Problems 106
7.3 From Formula to Knowledge Base 107
7.4 Towards a Micro Implementation of SHRDLU 110
7.5 CHAT-80 Revisited 111
7.6 EXERCISES 113

8 Discourse **115**
8.1 What Is Discourse? 115
8.2 Extrasentential References 117
8.3 Discourse and TUC 117
8.4 Using TUC to Understand Biomedical Texts 119
8.5 Coherence 122
8.6 EXERCISES 122

9 Logic, Events and Natural Language Understanding **125**
9.1 Ontology and Representation 125
9.2 Events 125
9.2.1 Event Logic 126
9.3 Second Order Logic 126
9.3.1 FOL in SOL 127
9.4 Combinatory Logic 127
9.5 Proper Treatment of Quantifiers 128
9.6 The Event Complement Construction 129
9.6.1 Compositional Semantics 132
9.6.2 Subordinate Sentences 133
9.6.3 Collective and Distributive Interpretations 133
9.7 Conclusions 134

10 BusTUC - A natural language bus route oracle **135**
10.1 Introduction 135
10.2 Previous Efforts, CHAT-80, PRAT-89 and HSQL 136
10.2.1 Making a Norwegian CHAT-80, PRAT-89 136
10.2.2 HSQL - Help System for SQL 136
10.2.3 The Understanding Computer 137
10.2.4 The TABOR Project 138
10.3 Anatomy of the bus route oracle 138
10.3.1 Features and figures of BusTUC 138
10.3.2 The Parser System 139
10.3.3 The semantic knowledge base 140
10.3.4 The Query Processor 141

10.4 Conclusions 142

11 GeneTUC **143**
11.1 History 143
11.2 Goals 143
11.2.1 TUC-related 144
11.2.2 Genetics-related 144
11.3 Adapting TUC 145
11.3.1 Key Relationships 145
11.3.2 Agents 146
11.3.3 Unfamiliar Words 147
11.4 Other Changes 147
11.4.1 Vocabulary 147
11.4.2 Standard Complements 148
11.5 Conclusions 148
11.5.1 TUC 148
11.5.2 GeneTUC 149

Scanner for Prolog grammars **153**

Metaparser for phrase structured languages **155**

Semantic Agreement Check **157**

Early parsing in Prolog **159**

Probabilistic Context Free Parsing **161**

Feature Unification in Prolog **167**

A micro version of SHRDLU **175**

CHAT-80 revisited **185**

A combinatory kernel for consensical grammar **191**

Sample questions to BusTUC **193**

Examples from Abstracts about Molecular Genetics **199**

List of Figures

3.1 A nice parse tree . . . 18
3.2 A simplified example of a parse tree in a categorial grammar . . . 23

4.1 A simple semantic net . . . 42
4.2 Semantic net legend. . . . 44
4.3 A semantic net. . . . 45
4.4 A course knowledge base. . . . 48
4.5 A simple heterarchy. . . . 52
4.6 A more complicated heterarchy. . . . 53

5.1 The principal model of the translation. . . . 60
5.2 Predicates for semantic agreement check . . . 71
5.3 Predicates for semantic compliance . . . 72
5.4 A grammar with semantic agreement check . . . 73

6.1 A Probabilistic grammar . . . 82

7.1 Annotated Compositional Parse Tree . . . 102
7.2 Grammar fragment for compositional analysis . . . 103

11.1 Key relationships in the GeneTUC system. . . . 146
11.2 TUC's agent subclasses. . . . 146
11.3 GeneTUC's agent subclasses . . . 147

Chapter 1

Introduction

The industrial societies are now entering the so-called Information society, where the majority of the employees are handling information of some kind. The proliferation of computers in almost all aspects of professional work makes communication with humans and computers in a natural-like language a challenging possibility.

1.1 Computational Linguistics

The FAQ[1] for the comp.ai.nat-lang [?] newsgroup gives the following definition of Natural Language Processing (NLP), or *computational linguistics*:

> Computational linguistics (CL) is a discipline between linguistics and computer science which is concerned with the computational aspects of the human language faculty. It belongs to the cognitive sciences and overlaps with the field of artificial intelligence (AI), a branch of computer science that is aiming at computational models of human cognition. There are two components of CL: applied and theoretical. [...] The applied component of CL is more interested in the practical outcome of modelling human language use. The goal is to create software products that have some knowledge of human language. Such products are urgently needed for improving human-machine interaction since the main obstacle in the interaction between human and computer is one of communication. [...] Natural language interfaces enable the user to communicate with the computer in German, English or another human language. Some applications of such interfaces are database queries, information retrieval from texts and so-called expert systems.

1.2 Knowledge-Based Natural Language Systems

One of the aims of Artificial Intelligence (AI) is to make it possible to describe a problem and have the machine solve it with general reasoning techniques. Typically, a general-purpose reasoning program operates on a formal description of the particular problem. Like a capable human being, the program may need to use background knowledge of the subject area along with general common sense knowledge about the world. Obviously, this knowledge must be represented in the machine.

Knowledge Systems are a subfield of Artificial Intelligence (AI). According to [?], the phrase *knowledge systems*, or, more accurately, *knowledge-based systems*, is used to describe programs that reason over extensive knowledge bases, containing facts and *rules*.

[1] Frequently Asked Questions

Natural language has evolved to meet the needs for communication of common sense information, and is certainly sufficiently expressive for that purpose.

However, when it comes to understand the information, a need for formal knowledge representation languages arises. A good knowledge representation language should combine the advantages of natural languages and formal languages.

1.3 TUC—The Understanding Computer

TUC (The Understanding Computer) is a name of a project where the aim is to build intelligent knowledge based system with natural language interfaces.

TUC is governed by a pragmatic view of intelligence as an intelligent agent that communicates in a restricted modality, i.e. natural language text, doing a useful task that requires expertise.

The TUC project was initiated at NTH[2] in the early 1990's. It was based on a number of previous efforts in creating a natural language interface for querying data bases, among them CHAT-80 [**?**], PRAT-89 and HSQL. The research goals for the project could be summarised as follows:

- Give computers an operational understanding of natural language
- Build intelligent systems with natural language capabilities
- Study common sense reasoning in natural language

The TUC project seeks to define a language denoted by NRL[3]. This language is as readable as plain English, but has well-defined syntax and semantics. In TUC, NRL serves as both a declarative knowledge definition language, and as a query language [**?**].

TUC relies on grammatical analysis for marking sentence elements. A sentence not being grammatically correct (according to TUC's internal grammar), will be rejected without further treatment. Enhancing TUC is thus both a question of adding to its vocabulary and semantics, *and* defining new grammatical constructs.

The knowledge-based approach, using natural language processing is not without problems. It is important to realise that the knowledge based-approach relies on semantical, rather than syntactical, analysis of the text.

The preparation of the system, building semantic nets and defining a sensible grammar, is both tedious and time-consuming. Work on the TUC project was started in the early 1990's, but the grammar and semantics are still far from complete.

1.3.1 BusTUC

BusTUC [**?**, **?**] is an application built upon the TUC framework. This is an application of TUC for answering questions about bus routes in Trondheim.

BusTUC consists of a parser system, a grammar, a semantic net, an expert system for route planning, and a route database.

The system was developed in 1996 , and has been in daily operation since 1998. It will play an important role in this presentation as an example of an intelligent natural language expert system. It is a notable fact that BusTUC is bilingual (English and Norwegian), where most of system is common to the two languages. This is an indication that the techniques used to understand the queries are not just surface level pattern matching, but relies on a deeper understanding.

[2] Norwegian Institute of Technology, now the Norwegian University of Science and Technology
[3] Naturally Readable Logic

Using NLP for Text Analysis in Biomedical Texts

A significant portion of the information required for biology research is currently recorded as free-text such as MEDLINE abstracts and comment fields of relevant reports like GenBank feature table annotations. Such information is important for many types of analysis including

- classification of proteins into functional groups,
- extraction of protein-protein interaction facts,
- discovery of new functional relationships,
- maintaining information of materials and methods,
- and increasing the the precision and relevance of hits returned from information retrieval systems.

Information Retrieval/Information Extraction

Text Mining

refers to retrieving knowledge from a piece of text. This can be performed manually by reading through the text, or automatic by having a computer process the text and extract factual assertions.

Information Retrieval (IR)

is what is often referred to as computerised searching, and can provide an aid in manual text mining. In IR, we seek to find the sources of the knowledge; extracting the knowledge from these sources is left to the reader.

Information Extraction (IE)

is an application of natural language processing which takes a piece of free text and produces a structured representation of the points of interest in it. One way for this to work is to perform syntactical and semantical analysis of the input in order to produce sound output.

1.3.2 GeneTUC

GeneTUC is another application built upon the TUC framework. Here, scientific texts on a certain domain are read, and partially understood by the computer, so that an intelligent summary, cross reference or summarisation is made possible.

The original framework was augmented with a moderate number of words from the biomedical domain, regarding gene - protein interactions. All concepts were entered manually, using Medline abstracts as a "training set", trying to incrementally expand GeneTUC's capabilities on a per-sentence basis.

The GeneTUC application utilises much of the experience made developing the BusTUC bus route oracle, but is primarily aimed at extracting knowledge from research articles pertaining to molecular biology and genetics.

Still, basing retrieval on searching a knowledge base, holds potentially great advantages. The user friendliness of a well-constructed natural language interface, in lieu of a conventional interface languages needs not be stressed.

It is a notable fact that GeneTUC utilities exactly the same grammar as BusTUC.

1.3.3 General Methodology

The grammar must often be updated, reflecting how sound sentences describing the domain may be formed. Then the knowledge base will have to be fed with information. In the case of a natural language competent system, this is most likely a straightforward task. The system will readily accept texts written in a normalised readable language ("NRL"). Structured texts, i.e. texts using some kind of field-formatting, presumably not in natural language, must be reformatted upon entry, or accessed by special interfaces.

Inner workings

The language analysis in TUC is a six step process

- **Lexical analysis**
  The individual words of the input string is looked up in TUC's internal dictionary. If a word is not found in the dictionary, the lexical analyser tries to find it in a domain specific data base containing words mentioned in earlier sentences. The lexical analyser also performs some spelling correction. The set of words are output as tokens in their inflective root forms, together with their possible word classes.

- **Syntactic and semantic analysis**
  The list of tokens is parsed using a grammar which can be described as a context sensitive categorial attribute logic grammar (CONSENSICAL grammar). The parser builds a TFOL[4] [**?**] formula representing the semantics of the sentence. It will output the first TFOL representation it finds that is syntactically and semantically satisfying.

- **Anaphora resolution**
  Anaphora[5] are replaced with the internal object they represent.

- **Optimising**
  The TFOL formula is (Skolemised and) simplified into a TQL[6] formula.

- **Reasoning**
  TUC uses the TQL formula and a domain specific knowledge base to translate TQL into database query language.

- **Query Processing**
  This involves processing the queries, where the answers are transformed to natural language answers by a natural language generation process.

[4]Temporal First Order Logic
[5]Anaphora are words or phrases taking its reference from another word or phrase, e.g. she, it, then, see 2.3.2.
[6]TUC Query Language

Chapter 2

Linguistics for Non-Linguists

In order to appreciate a grammatical NLP system like TUC, knowledge of the fundamental principles of grammar is required. In this chapter some of the key concepts of natural language are introduced, for which an understanding is crucial when dealing with TUC, or NLP systems in general. Much of what is written in this chapter is based on the excellent book, [**?**].

The chapter concludes with a section on Naturally Readable Logic. This is a subset of natural language, with certain properties making it ideal as a basis for NLP systems. TUC, and consequently GeneTUC, is based on analysis and processing of NRL, rather than natural language.

This chapter is a re-worked version of the linguistics chapter in [**?**].

2.1 Word Classes

The different word classes are the fundamental building blocks of the language. This section describes the most important word classes, their functions and use, according to [**?**] and [**?**]. Although not exhaustive, in the sense that some classes are left out, this provides a foundation for the discussions later in the text.

2.1.1 Verbs

Verbs is the class of words used for denoting actions. These can be categorised further according to

- **Regularity**
  Verbs can be placed in the subclasses regular and irregular depending on how they are inflected in the past and past participle form. The regular verbs are all inflected according to a general schema, whereas the irregular ones have individual patterns of inflection.

- **Transitivity**
  All verbs can be put into at least one of the these transitivity classes:

  - Intransitive - not taking object, e.g., "John *laughs*"
  - Copular - taking a subject predicative, e.g., "John *became angry*"
  - Transitive - taking one object, e.g., "John *saw Mary*"
  - Ditransitive - taking two objects, e.g., "John *gave Mary a rose*"
  - Complex transitive - taking a direct object and an object predicative, e.g., "John *found Mary titillating*"

  Note that transitive verbs may require an adverbial:

Mary put the book *on the desk*.

Mandatory adverbials are also called "verb complements" as distinguished from voluntary "verb modifiers", but the distinction is not always precise.

2.1.2 Nouns

Nouns give names to persons, places, things and concepts in general.

- *Common nouns* denote any member of a set of concepts, e.g., a car, thoughts, a girlfriend.
- *Proper nouns* give names to a specific member of the set, e.g., John, the Theory of Relativity, Oslo Airport Gardermoen.
- *Mass nouns* give names to unspecified uncountable quanta of stuff like coffee, wood, and space.

Nouns can be derived from verbs and vice versa. Thus, the English are said to "verb their nouns":

The noun *progression* is derived from *to progress*.
The verb *to house* is formed from *house*.

The *Gerund* is a type of word which can act as both a verb and a noun. It is formed as the present participle of the verb. Examples of usage:

As a noun: John likes *programming*.
As a verb: John is *programming* his VCR.

Nouns can also function as adjectives, modifying other nouns, as in

John likes *action* movies.
"...and a partridge in a *pear* tree."

A noun phrase can be expanded by *apposition*, that is two usually adjacent nouns or noun phrases having the same referent standing in the same syntactical relation to the rest of the sentence. Most commonly, a proper noun and a noun phrase further describing the noun is used, like

Bill Clinton, the president of the USA, committed perjury.
They shot *my cousin Vinny*.
There is a *rumour that petrol prices will drop* after the next EU summit.

2.1.3 Adjectives

Adjectives are words used to modify the noun, either as a part of a noun phrase or following a copular verb. The adjective can be complemented, forming adjective phrases. These phrases are formed in four ways:

- **Adverb and adjective**
  John is *rarely late*.
  This report is not *good enough*.
- **Adjective and prepositional phrase**
  Mary is *fond of boxing*.
  John is *sitting on the chair*.
- **Compared adjectives**
  I am *taller than* you.
  Mary thought *as hard as she could*.

- **Adjective and subordinate clause, participle or infinitive clause**
  I'm *afraid John died*.
  Mary is *good at doing nothing*.
  This key is *supposed to fit*.

2.1.4 Adverbs

The adverb is a word class that modify verbs, adjectives, other adverbs or complete sentences. Adverbs can be combined into adverbial phrases, with the same function as adverbs. The adverbs are grouped into three subclasses:

- **Simple**
  The first subclass is a simple modifier, e.g., "I am leaving *tomorrow*", "I'll eat my dessert *first*"

- **Interrogative**
  Interrogative adverbs are used for asking questions, e.g., "*Where* is my other sock?", "*When* was that?"

- **Conjunctive**
  The conjunctive adverbs connect independent clauses, e.g., "It was raining; *consequently*, John stayed at home", "I think; *therefore*, I am"

Some adverbs are called *particles* ; they are single words that redefine the meaning of words, especially verbs.

Example : " He gave *up* and came up " .

Here , the first up is a particle that redefines the meaning of give, while the second up is an ordinary adverb.

2.1.5 Pronouns

Pronouns are used in place of nouns. There are five principle groups of pronouns:

- **Personal**
  Personal pronouns point directly to a person or an object, e.g., "*He* is a good teacher", "Mary saw a film. *It* was scary"

- **Possessive**
  Possessive pronouns are pronouns showing ownership or possession, e.g., "Get off *my* cloud!", "The dog tried to bite *its* tail"

- **Demonstrative**
  Demonstrative pronouns focus the attention on the object pointed out, e.g., "*These* boots are made for walking", "Who's *that* girl?"

- **Reflexive**
  The reflexive pronouns point back at the noun or pronoun that has just been named, e.g., "Mary looked at *herself* in the mirror", "They've bought *themselves* a new car"

- **Relative**
  The relative pronoun joins a subordinate clause to a main clause, e.g., "John saw the girl with *whom* he was in love", "The parrot *that* I bought not half an hour ago, is dead"

2.1.6 Prepositions

Prepositions are words used to show a relationship between its object (noun or pronoun following the preposition) and another word in the sentence

In a galaxy, far, far away.
First *among* equals.

A prepositional phrase includes a preposition, the object of the preposition and a number of modifiers on the object. The prepositional phrase may have an adjectival or adverbial function

Adjectival function: The car *outside the house* is nice (the phrase gives more information on the subject).
Adverbial function: Mary looked *at the man*.

2.1.7 Conjunctions

Conjunctions are employed to connect words, phrases or clauses, possibly indicating the relationship between the elements they connect in the sentence. There are three types of conjunctions:

- **Coordinating**
  Coordinating conjunctions connect elements having the same grammatical function, e.g., “Sticks *and* stones may break my bones”, “Many are called, *but* few are chosen”
- **Correlative**
  Correlative conjunctions act as coordinating conjunctions, but work in pairs to connect elements in a sentence, e.g., “*Neither* rain *nor* snow will stop him”, “I like *both* vanilla *and* chocolate”
- **Subordinating**
  Subordinating conjunctions (subjunctions) connect two elements with different grammatical function, most commonly an independent and a dependent clause, e.g., “It looks *as though* it’s going to rain”, “*Since* you’ve been gone, I’ve been missing you”

2.2 Sentence Categories

Sentences may be categorised according to function and structure. In terms of structure, the most important property is whether the verb is placed in front of or behind the subject. Also, sentences belonging to a certain category may have a function similar to a sentence from one of the other categories.

2.2.1 Declarative

The declarative sentence, in which the verb is placed behind the subject, is the most common of the major sentence groups. It is usually less marked in form and less restricted in function than the other groups. As implied by the name, declarative sentences state facts, as in

John likes to play guitar.
It was the best of times, it was the worst of times.
Mary has not eaten her peas yet.

Declarative sentences can be positive, i.e., affirm a fact, or negative, denying a fact. The first two examples above are thus positive declarative; the last one is negative.

2.2.2 Imperative

Imperative sentences are most often employed to issue a command. The imperative sentence often lacks an explicit subject and use the verb in its base form:

> Shut the door!
> Please keep your luggage with you at all times.

The subject of the sentence is, if not the addressee, often given from the context.

2.2.3 Interrogative

Interrogative sentences are used to query the addressee for information. In contrast to the declarative sentences, the verb often precedes the subject in interrogative sentences:

> Have you ever loved a woman?
> Where did all this mail come from?

Sometimes interrogative sentences have a non-interrogative function. Such *rhetorical questions* act as statements or commands, while avoiding having to use declarative sentences, which may seem blunt or obvious:

> Who cares?
> Do you mind closing the door when you leave?

2.3 Extragrammatical Phenomena

2.3.1 Ellipsis

Ellipsis is a phenomenon often encountered in dialogue. Ellipsis is the omission of a phrase mentioned earlier in the discourse but which can be inferred from the context.

Elliptic sentences are categorised as sentence fragments; words or phrases not included in a phrase structure but still carry a communicative function:

> Mary showed up late, but then again, she always used to [show up late].
> Do you know how to get there? Yes, I do [know how to get there].

2.3.2 Anaphora

Using personal pronouns for referring to persons or objects mentioned earlier in the discourse, is called anaphora or anaphoric references. Anaphoric references require the speaker and addressee to share enough common knowledge to resolve the anaphora. *Internal anaphora* are anaphora referencing persons or objects cited in the same sentence, *external anaphora* reference earlier sentences:

> Internal: Mary had given up on her fear of flying, and started to like *it*.
> External: John had no idea what *she* was talking about.
> He thought *the woman* was silly.

The pronoun *it* is extremely general, often making the resolution of the anaphora hard, or introducing ambiguities.
Cataphoric references is a phenomenon closely related to anaphora. A cataphoric reference is a reference dependent on something following the references, i.e., a forward pointer:

> *She* didn't know what to do with *it*, but Mary thanked politely for the gift, and placed it swiftly in the back of the closet.

2.3.3 Garden-Path Sentences

Garden-path sentences is a a class of seemingly ambiguous or incoherent sentences. The sentences may seem grammatically incorrect at first glance, but are in fact not. The term "garden-path" relates to the fact that the human reader is "lead down the garden-path" into an interpretation which is ultimately incorrect, and is left confused when the initial parse fails:

> The horse raced past the barn fell.
> The complex houses married and single students and their families.
> The computer screens all the entrants.

Often, the confusion is created by incorrectly determining the class of certain words, or erring when parsing parts-of-speech. Garden-path sentences display three properties, according to [**?**]:

- The are *temporarily ambiguous*, i.e., the initial portion of the sentence is seemingly ambiguous, but the whole sentence is not
- The human parsing mechanism will somehow prefer one of the multiple interpretations of the initial portion
- One of the dispreferred parses is the correct one

Some interesting observations on the psychology of Garden-path sentences is found in [**?**].

2.3.4 Metaphor

Metaphors are figures of speech, in which words and phrases literally denoting one concept is used to described another, unrelated concept, suggesting likeness on some level.

> He was so angry he was about to explode.
> Act quickly, time is about to run out!
> Elliot Ness brought the Mafia to its knees.

Arguably, many of our common-day metaphors are motivated by a relatively small number of *conventional metaphor schemas*, such as organisation-as-person, anger-as-heat and time-as-resource.

2.4 Lexical Semantics

Lexical semantics focuses on the meaning of single words, rather than the meaning of whole sentences. The "atom" of lexical semantics is the *lexeme*, which can be thought of as a pairing of a orthographic and phonological form with some sort of symbolic meaning. Lexemes are compiled in a *lexicon*, a finite list of lexemes.

Four of the key concepts of lexical semantics are described in the following sections. Examples are provided in the Table.

2.4.1 Synonymy

The definition of a synonym is apparently simple. Synonyms are two or more different lexemes with the same meaning. A criteria for this is *substitutability*, i.e., two lexemes may be substituted for one another in a sentence without changing the meaning or loss of acceptability. Examples of synonymic words are 'bus' and 'coach' .

Generally, the concept of substitutability crosses all domains, thus two synonyms may be interchanged regardless of context. This opens for the weaker notion of *restricted synonyms*, which are lexemes that are substitutable in some context, but not generally.

Table 2.1: Some examples of different phenomena in lexical semantics.

Phenomenon	Lexeme	Comments
Synonymy	car - automobile	
Restricted synonymy	great - big	Synonymous when describing size
Homonymy	ball	Spherical object and formal gathering for social dancing
Homophony	great - grate	
Homography	bow	Archer's bow /'bO/ and forward part of ship /'bau/
Polysemy	fly	move in the air and move fast
Hyponymy	bus - vehicle	A bus is a hyponym of vehicle

2.4.2 Homonymy

Homonyms are lexemes with the same form, but with different, and unrelated, meanings. The distinct senses of a homonym often have very different etymological[1] origins.

Related to homonyms are *homophones*, distinct lexemes sharing pronunciation, and *homographs*, which have identical form but distinct pronunciation.

2.4.3 Polysemy

Polysemes are almost like homonyms, in the sense that they are identical lexemes. They differ from the latter by having *related* meanings. Distinguishing polysemy from homonymy is, as one might expect, not necessarily straightforward.

How will one know if two senses of a word are related? Often, etymology and a native conception of the word can give a pointer to distinguish between polysemes and homonyms.

2.4.4 Hyponymy

Hyponymy is an asymmetrical pairing of lexemes where one noun lexeme denotes a subclass of the other. The more specific lexeme is denoted a *hyponym* of the more general one. Conversely, the more general lexeme is denoted the *hypernym* of the more specific one.

Hyponymy is closely related to, and a prerequisite for, creating a *taxonomy*. A taxonomy is a particular ordering of the elements of an ontology[2] into a tree structure, where hyponymy is the ordering constraint.

Object hierarchies presented in section (4.4) Semantic nets are examples of taxonomies.

2.5 Punctuation

Punctuation is used to clarify meaning and separate structural units in the written word. The following is a quick run-through of some rules and recommendations for using punctuation.

2.5.1 Comma

Commas often cause some distress among inexperienced writers. Below is a brief discussion of the comma's usage:

- **Compound sentences**
  Generally, when concatenating two or more clauses using a coordinating conjunction, chiefly "but". However, when the second clause is very short, and the subject is ellipted, use of commas are less common.

[1] Etymology is the history of a linguistic form since its earliest occurrence in the language where it is found, and by decomposing, tracing and analysing its transmissions and influences from forms in other languages.

[2] An ontology is a set of objects existing in a domain or micro-world.

He wanted to go, but he could not make up his mind.
The secretary was in her late twenties and built like a goddess.

- **Initial adverbials**
  When starting sentences with adverbial clauses, a comma should be used.

  Generally, her cooking doesn't taste that good.
  Keeping in mind last Saturday's effort, our national site's chances of qualifying are slim.

- **Trailing adverbials**
  On the other hand, trailing adverbials are not separated by a comma, unless they belong to a separate information unit.

  She came by at lunch while I was taking a nap in my office chair.
  She came by at lunch, as mothers tend to do.

- **Enumerations**
  Commas should be placed between parallel phrases or clauses. There is usually no comma before the "and" at the end of an enumeration.

  My brother likes bananas, apples, chocolate and ice cream.
  The car went through the road block, raced past the officers and crashed into an off-license.

- **Parenthetical elements**
  Parenthetical elements or final elements representing separate information units, such as non-restrictive relative clauses, comments and appositions, should be set off by commas.

  Jimmy Hoffa, the notorious Teamsters leader, was formally declared dead in 1982.
  I wanna know, have you ever seen the rain?

- **Disjuncts and conjuncts**
  Disjuncts[3] are often set off by commas. There are some exceptions, mainly single-word disjuncts placed mid-sentence. Conjuncts[4] should always be marked off by commas.

  Luckily, this wasn't the case.
  The plan couldn't possibly go wrong.
  Mary could, nevertheless, never stop thinking she had done something wrong.

2.5.2 Semicolon

Semicolons are used between two independent main clauses which are not connected by a coordinating conjunction. Use of the semicolon is at the writer's discretion, and may in most cases be replaced by a period.

Peter was late; Monday was never a good day for him.
People were starting to wonder about the PM; the Governments last initiative on public transportation was clearly ludicrous.

[3]Disjuncts are adverbials which are loosely connected to a sentence and convey the utterer's assessment of the sentence's content.
[4]Conjuncts are adverbials which indicate the utterer's assessment of the connection between two clauses.

2.5.3 Colon

A colon is used to signal to the user that what follows is an explication of what has already been said. The text following the colon is often not complete clauses.

> And the Lord said: Let there be light.
> Be sure to bring the following things: sleeping bag, thermos, a sharp knife and food to last for three days.

2.6 Grammatical Relations

In traditional grammatical analysis, we not only classify the words, but also determine their role or relations in the whole sentence. Grammatical relations can be defined by their position in the phrase structure. For example, an object is an NP immediately following a Verb. The common relations are summarised in Table 2.2.

Table 2.2: Table of some grammatical relations

Grammatical	Explanation	Sub-class	Example
Subject	The main agent of a an event	—	TWO MEN broke into the house.
Object	Other entities involved in the events		
		Direct	He killed FRED
		Indirect	Fred gave MARY a kiss
		Prepositional	to THE CITY
Predicate	Any description of entities		
		Verb Phrase	John WENT TO SCHOOL YESTERDAY
Complement	Any description of a verb other than objects		
		Adjective	Mary was ANGRY
		Prepositional	She lived IN A HOUSE
Modifier	Any phrase that describes another		
		Adjective	HAPPY children
		Adverb	sing LOUDLY
		Prepositional	The house IN THE GARDEN was yellow
Subject Clause	The clause is the subject	—	THAT THE BUS IS LATE is a shame
Object Clause	The clause is the object	—	She said “ GET OUT”
Complement Clause	The clause is a complement	—	She knew that THE BUS WAS LATE

2.7 Ambiguities

Natural language is inherently ambiguous on all levels. We have already touched the various levels of ambiguities concerning word interpretations. However, there are other ambiguities

hiding:

Structural Ambiguity

Structural ambiguities is concerned with what belongs to what in a sentence, how the parts of speech are grouped.

PP attachment ambiguity

A famous example is

```
The man  saw the dog  in the park with a telescope.
```

There are many possible interpretations here:

The man can see with a telescope, and he saw a dog that was in the garden, or

The man saw a dog, the dog was in a garden and the garden had a telescope, or

The man used a telescope to see a dog while he was in the garden

etc. (A complete list of all possibilities is left as an exercise).

The ambiguities of theses sentence is related to the syntax analysis. The example is an example of prepositional phrase (PP) attachment ambiguities, because the PP's

'in the park' and 'with a telescope'

can be connected to a variety of nouns or verbs.

Relative clause attachment ambiguity

John visited a country in Asia that was big.

It probably means that the country was big, for the other interpretation says that Asia was big, but that is quite redundant information.

Conjunctive ambiguity

The basket had many red apples and oranges.

It could be that both the apples and oranges were red, but that is contrary to common wisdom.

Distributive/Collective ambiguity

John and Mary had two children (together) and two parents (each !).

Scope ambiguity

Every student that read a book ran a program.

Did they read the same book , did they run the same program?

There are at least 6 different interpretations of this. See how many you can find ! (Exercise).

An final amusing example:

The two sentences below mean something different. See if you can find the difference ! (Exercise)

```
List the only Frenchman among the programmers who
      understands English.

List the only Frenchman among the programmers who
       understand English.
```

Exercises

```
1. Find as many significantly different interpretations of
   the sentence with respect to scoping.

     Every student that read a book ran a program.

   (There should be at least 6)


2. The following sentences have different meanings.
   Analyse them with respect to structural ambiguity.

  A: List the only Frenchman among the programmers who
      understands English.

  B: List the only Frenchman among the programmers who
       understand English.
```

Chapter 3

Grammar for a Fragment of English

3.1 Introduction

3.1.1 Context Free Grammar

Grammars have existed for languages since ancient times (Paniani in ancient Persia is noted). The most prominent thing about grammars is that they describe the legal phrases in terms of a composition of its subphrases. We call these grammars for *Phrase Structure Grammars*. The simplest form of Phrase structure grammars is called *Context Free Grammars* (CFG).

A context free grammar consists of a set of rules or productions. A CFG is usually thought of in two ways:

- as a device for generating sentences
- as a device for assigning structure to a given sentence.

The symbol that we use, `-->` , derives its legend from the former as a rewriting symbol, while we for our purposes use the grammar as a tool for analysis.

```
Statement --> Noun_Phrase Verb_Phrase
```

which can be paraphrased as

```
a text in the format   Statement

   is composed of

a text in the format   Noun_Phrase

   followed by

a text in the format   Verb_Phrase
```

It is significant aspect of phrase structure grammars that the text parts are contiguous, and appear in the same order as specified in the grammar.

To show that a phrase is in accordance with a grammar means to fill in the elements of the text into this phrase structure, often in a form of a parse tree.
We shall briefly describe the outlines of a simple grammar for English. The grammar will in the first place be built on a context free grammar description, with some extensions. The grammar

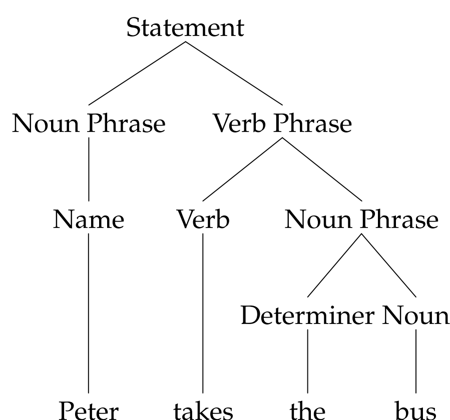

Figure 3.1: A nice parse tree

will sometimes deviate from standard approaches, but this is done so for the purpose of conformance with the computational grammars that is developed later.

The grammar will of obvious necessity be an abstraction of a complete grammar for English (if that is a meaningful concept).

The subset is not a small subset however, but will cover the language that is used in a natural language applications, e.g. the bus route information system. That is of course a very limited area of natural language application, but in principle the same grammar can be applicable to other domains (as has been shown in practice, e.g. molecular biology).

It is of course hoped that the grammar will be useful in other applications, but work on grammars for practical purposes will always be bound to hard work. [1]

What to abstract away is guided by the view that behind each utterance is an intended well formed sentence (in a language we have called NRL) that expresses the essential information content of what is said.

An abstraction will not capture the set of legal sentences exactly. In the first place, only correct and complete sentences will be covered (despite that humans understand a lot of out-of-grammar texts). On the other hand, it will also lead to an over-generation (over-acceptance) of possible sentences. However, as long as the purpose is to understand the sentences more than correcting them, liberal interpretations are acceptable.

The structure of a text is usually composed of sentences, and the full meaning is seldom captured by analysing isolated sentences. The precessing of several sentences together is called *Discourse* and is mentioned in later chapter. We shall thus start with a description of single sentences. There are three main categories of sentences:

```
Statement    The bus goes to NTNU on sunday .

Question     Which bus goes to NTNU on sunday ?

Command      Give me a list of buses !
```

The main constituency of a text is a set of sentences, separated by tokens. A sentence is made up of words, eventually separated by tokens (e.g. ',', ';', ':'). We shall start with the

[1]Natural language processing is known to be

10 % inspiration,
90 % transpiration, and finally
10 % desperation

syntax of Statement, and derive the syntax of questions and commands by certain transformation principles.

3.2 The Elements of Grammar (Parts of Speech)

The starting point is the lexicon, the words and classes of words. The basic constituents of a grammar are the words which can be put into word classes.

Normally, a word in one class can be substituted by another word in the class without changing the syntactic correctness of the sentence, but there are severe exceptions to that rule.

The grammatical categories that denote single words are called Parts of speech (POS).

```
Nouns           bus, train, person, city, appointment, stop,
                time, noon, departure, date

Poper nouns     Oslo, Dragvoll, Nidarvoll, NTNU, Tore

Numbers         42

Verbs           take, go, stop, reach,  will, take, chose,
                appear, arrive, depart, live, reach, do

Articles        a, the

Auxiliaries     shall, will, do, be, can, must

Adjectives      first, last, next, previous, earlier

Prepositions    to, from, before, after, at

Determiners     the, all, a, an

Adverbs         now, early, tonight

Grade Adverbs   too, very

Conjunctions    and, or, not, else, to,

Exclamations    thanks, hello

Pronouns        I | it | we | its | his | she | they

Quantifiers     some | every | 43

Miscellaneous   than, to
```

3.3 Comments to Word Categories

These are some of the complications that must be taken into account.

- The same phrase can have different interpretations. E.g.

”thanks” (noun and exclamation)

”stop” is both a verb and a noun

”to” is both an infinitive and a preposition

”that” may be four different things

  - a conjunction (he said that the bus was blue)
  - a relative pronoun (a bus that is black)
  - a (demonstrative) pronoun (that is true)
  - a determiner (that bus is black)

- Words can be grouped together to form phrases that acts as the word.

  For example, ”in order to” is a phrase that is used as a standard phrase (a subordinate conjunction), and where the literal analysis of the actual expression is irrelevant. Such combined phrases are called *phrasal*, e.g.

  as fast as possible (phrasal adverb ”soon”)

  directly on [the nose] (phrasal preposition)

  show up (phrasal verb “appear”) (phrasal verb)

- Many words appear as *idiomatic language*, i.e. they mean something different from their literal interpretation.

  Example: to hang around

  means to be somewhere in the neighbourhood, not actually hanging. The treatment of idiomatic language belongs to the field of pragmatics, but inevitably, a grammar is often bent to capture the meaning at an early stage.

- Many words are created by suffixes and inherits its meaning from the root form of another word class.

  [The bus is] ”delayed” comes from the verb delay

  ”fielded” is used as an adjective, but is the past participle of the verb ”to field” which stems from the noun ”field”

- Numbers are both quantifiers and noun phrases (and in some cases, e.g. bus routes, also names)

- Adverbs contain words that may be used in widely different ways:

  - to modify a sentence, (*Certainly*, I died)
  - to mofify a verb, (I died *suddenly*)
  - to modify an adjective (the *very* high girl)
  - to modify an adverb (he died *very* quickly)
  - to modify an noun (a beer *now* would be nice)

In the syntax description, this means that only the right usage is applicable. Adverbs that modifies verbs are someties called *(verb) particles*. However, particles usually change the meaning of the verb (he gave *(up)*), while adverbs are acts as a modifier (he came *(up)*).

- Determiners may be of a complex kind. Instead of simply

  `the house`, we can say `John's house` or even `John's father's house`. We see that the determiner can be replaced by a complex row of noun phrases connected by a genitive "s".

3.4 Syntax of Statement

We shall discuss the syntax of single sentences (that are meaningful disregarding tokens). We shall call these *complete (single) sentences.*

The grammar formalism will be an extension of context free grammars (CFG). The extensions will be threefolds:

- First we introduce a simple extension for options and repetitions.
- Secondly, we introduce grammar elements for constructions that violate the rules of phrase structure grammars.
- Finally, in a later chapter, we shall introduce attributes to parametrise the context, and to handle the semantic definitions.

To make the grammar small and comprehensible, we use the following conventions:

- `P?` means that P is optional
- `P*` means zero or more occurrences of P, possibly connected by (coordinating) conjunctions
- `P+` means one or more occurrences of P, possibly connected by (coordinating) conjunctions.
- `P | Q` that P and Q are alternative right sides

Parentheses ( ) may be used for grouping (to avoid a proliferation of names for syntactic categories). [2]

We have made a simplification of conjoined expressions (using 'and', 'or', 'not') by treating them uniformly as sequences.

For better readability, a syntactic category appearing to the right is supposed to be non-empty unless suffixed by '?' or '*' .

An example of the result of using * and + is the noun phrase

"Two or three more or less green or yellow apples, pears and bananas"

The phrase is highly ambiguous, but to take just one possible analysis, this can be shown using brackets, and comparing the syntax definition of `Noun phrase`.

```
(Two or three)              Number +
```

[2]Some authors use parentheses for denoting optionality.

```
(more or less)              Grade_Adverb+

(green  or yellow)          Adjective+

(apples, pears and bananas) Noun+
```

3.5 Context Sensitive Grammars

So far, the context free grammars that we started with follows the rules of phrase structure grammars. However, there are syntactic constructions that fall outside context free grammars. One such phenomenon is *movement*.

In all the sentences below, each relative clauses consists of a statement that misses a noun phrase, i.e. it contains a *(noun phrase) gap*. We denote these gaps with an empty parenthesis.

```
A boy kisses a girl that () loves Fred.

A boy kisses  a girl that  Fred loves ().

A boy kisses a girl that a boy that John hates () loves ().
```

Note that the distance between a noun phrase gap and its corresponding noun phrase can be arbitrarily long, but nevertheless, they have to match each other. This amounts to the concept *Long Distance Dependencies*.

There are numerous variations of phrases that would demand a separate grammar rule for each of the statements, depending on where the gap occurs. However the principle is very simple if we allow certain phrases to be *moved* into the gaps where they are missing.

```
A mouse(1)  that a cat  catches (1) squeaks
```

Here, we imagine a gap (1) is where the filler (mouse(1)) should logically fit in, i.e.

```
A mouse(1) squeaks and
a cat  cathes (1)
```

A grammar using this principle extends the principles of contex free phrase structure grammars, and we need to extend the grammar formalism to a grammar that we will call *Context Sensitive Categorial grammar*.

Categorial Grammar

The origin of the extended grammar formalism derives from a grammar type called *categorial grammar*. We shall describe classical definitions, but add into it non standard elements that will be of immediate use.

A categorial grammar (CG) specifies a language by describing the combinatorial possibilities of its lexical items. A category is a kind of type so that phrases of different category combine to form new categories. The set of categories is defined inductively as follows:

- a primitive catagory is a category. Specifically, S and NP are primitive categories
- If A and B are categories, then A/B and A\B are categories.

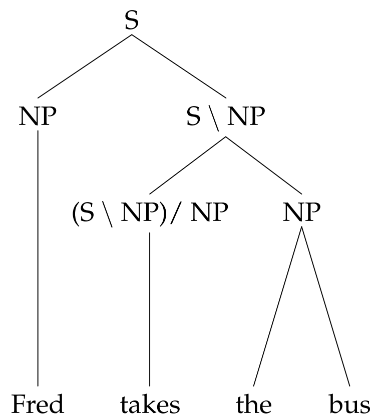

Figure 3.2: A simplified example of a parse tree in a categorial grammar

Combination of phrases is sanctioned by their categories according to these rules:

- **Forward application** (FA): A phrase P of category A/B concatenated with a phrase Q of category B forms a phrase PQ of category A.
- **Backward application** (BA): A phrase P of category B concatenated with a phrase Q of category A\B forms a phrase PQ of category A.

As an example of a categorial grammar, we might associate lexical items with categories as follows

```
Type of word        Example       Category
------------        -------       --------

proper nouns        Tore          NP

intransitive verb   smile         S\NP

transitive verbs    kiss          (S\NP)/NP

verb phrase         take bus      S\NP

relative clause     Mary saw      S/NP

relative clause     saw fred      S\NP
```

Note that the direction of the operators determines which side of the functor phrase its argument will be found on (right or left).

3.5.1 Modified Categorial Grammar

We shall use the ideas of Categorial grammar in our syntax descriptions, but with two important modifications.

- **Inwards application**

The first modification has to do with the fact that we often need to fill in a missing part (gap) elsewhere than exactly at the end.

We therefore generalise the operator / to *Inwards application:*

A phrase P of category A//B

can be combined with a phrase Q of category B
to form a phrase R (of category A)
if the phrase P can be split into P1 P2, and R = P1 Q P2.

- **Outwards application**

  In addition to the operators above, we will need a third operator – which we can call *Outwards application*. It is similar to the Inwards application, but the phrase Q doesn't need to be used by the phrase P, but can actually be used by later productions.

A few examples can be sketched:

"The man Mary saw in the bus smiled"

The subphrase

`P = Mary saw in a bus`
is of category S//NP because the sentence

`Mary saw the man in the bus`
is of category S,

`the man`
is of category NP and

`S` can be split into

`Mary saw + the man + in the bus`
and

`P = Mary saw + in the bus.`

Movement

The rules for movement can now be formulated by the extension of the CF -grammar.

```
Statement   --> Noun_Phrase Verb_Phrase

Noun_Phrase --> Determiner Noun Rel_Clause

Rel_Clause  --> that  (Statement // Noun_Phrase)

Rel_Clause  -->

Determiner  --> a

Noun --> mouse | cat | dog
```

```
Verb_Phrase --> Intrans_Verb
Verb_Phrase --> Trans_Verb  Noun_Phrase
```

The meaning of the phrase

```
Statement // Noun_Phrase (1)
```

is any Statement where a noun phrase is missing.

Example:

```
John hates a woman that loves Fred

John hates a woman that Fred loves.
```

Here, both

```
loves Fred
Fred loves
```

are examples of Statement where a noun phrase is missing. Here the expression `Statement // Noun Phrase` covers both

```
() loves Fred
Fred loves ()
```

where `()` denotes the gap of the missing noun phrase.

Since

```
(A)  Statement --> NP Verb_Phrase
```

Noun_Phrase(1) (NP1) may be missing in one of the right sides of Statement. That means that (A) implicitly defines two new grammar rules:

```
(A1)  Statement//NP1 --> (NP // NP1) Verb_Phrase

(A2)  Statement//NP1 --> NP  (Verb_Phrase // NP1)
```

The same principle applies recursively to the newly introduced rules. Note that NP//NP1 may be matched by an empty word.

Note also that the gapping feature of the grammar saves a lot of explicit rules in a context free grammar.

More gaps

In some cases, the missing subphrase is just missing in the *beginning* of a sentence. In that case, we use the operator \ to denote this fact. A small fragment is meant to illustrate the principles.

```
Whose dog barked ?
```

can be defined by the rephrased query

```
Who has a dog that barked ?
```

which can be rephrased as

```
Which person  has a dog that barked ?
```

Grammar Fragment

```
WhoseQ -->  whose Noun
            WhoQ \(who,has,a,Noun,that)

WhoQ   -->  who  WhichQ\(which person)

WhichQ -->  which Noun Verb_Phrase


Sentence --> Statement .|
             Question ? |
             Command !

Question --> WhoseQ |
             WhoQ |
             WhichQ
```

3.6 Simplified Grammar for Statement

```
Statement           --> Complement* Noun_Phrase Verb_Phrase

Noun_Phrase         --> Determiner? Nominal_Clause | Pronoun+

Nominal_Clause      --> Noun_Prefix?  Nominal Noun_Modifier* |
                        Clausal_Phrase

Clausal_Phrase      --> Complementiser?  Statement |
                        Subordinate_Conjunction Statement |
                        Infinitive Verbal_Clause

Nominal             --> Noun +

Noun                --> Proper_Noun | Common_Noun

Noun_Prefix         --> Adjective_Phrase

Verb_Phrase         --> Verb_Prefix? Verbal_Clause

Verbal_Clause       --> Verb_Head  Verb_Complements |
                        Be_Head    Complement*

Verb_Head           --> Verb+

Be_Head             --> be Negation? Be_Complement

Be_Complement       --> Adjectival_Clause+ |
                        Verb-ED+ |
                        Verb-ING

Verb_Prefix         --> Auxiliary_Verb+ Negation?

Noun_Modifier       --> Complement |
```

```
                      Relative_Clause

Relative_Clause   --> Relative_Pronoun (Statement // Noun_Phrase) |
                      Reduced_Relative |
                      whose  Nominal_Clause Verb_Phrase

Reduced_Relative  --> Be_Complement


Verb_Complements  --> Verb_Argument*  Complement*

Verb_Argument     --> Noun_Phrase

Complement        --> Prepositional_Phrase |
                      Adverb

Conjunction       --> Coordinating_Conjunction |
                      Subordinating_Conjunction

Preposition_Phrase--> Preposition Noun_Phrase

Adjective_Phrase  --> Adjective_Prefix* Adjective+

Adjective_Prefix  --> Grade_Adverb

Adjectival_Clause --> Noun_Phrase |
                      Adjective_Phrase |
                      Comparison

Comparison        --> Comparator than Noun_Phrase

Comparator        --> than | like

Determiner        --> Quantifier | Article | Possessive_Pronoun |
                      Noun_Phrase (genitive)s
```

Parts Of Speech

The categories that corresponds to lexical words are called Part Of Speech (POS). In these examples, the words are listed in their root form.

```
Common_Noun --> bus | centre |  city | train | appointment | stop

Proper_Noun --> NTNU | Tore |

Verb --> appear |arrive | depart | live | reach | go | take

Auxiliary_Verb -->   shall | will | do | shall | will | can

Pronoun --> I | it | we
```

```
Possessive_Pronoun --> its | his

Article --> a | the

Quantifier --> Number | every | no

Relative_Pronoun --> that | which | who

Preposition --> by | of | at | before | between | from | on |
      near | through

Coordinating_Conjunction --> and | or

Subordinating_Conjunction --> while | when | in_order_to

Adverb --> tonight | on | downtown | tomorrow | here | how

Grade_Adverb -->  very | too

Number --> 5, 6

Interjection --> hi

Adjective   --> next | early

Negation    --> not

Infinitive  --> to

Complementizer --> that

Verb_ING  --> Present Participle  of Verb

Verb_ED   -->  Past Participle of Verb

Relative_Pronoun --> who | which | that

Quote --> `` any text enclosed in ``
```

3.7 Simplified Grammar for Commands and Questions

Command Syntax

Commands are easy to express in the same manner.

```
    Shut the door !
```

is just a statement with an implicit noun phrase , i.e. a verb phrase.

```
  Command --> Verb_Phrase
```

Question Syntax

The list of questions is a bit more complicated. The main types of questions can be listed as follows:

```
Question  --> WhenQ        When does the train arrive ?
Question  --> WhichQ       Which station does the bus pass ?
Question  --> WhoQ         Who has eaten my porridge ?
Question  --> WhereQ       Where is my hat ?
Question  --> WhatQ        What time is it ?
Question  --> HowQ         How can I thank John ?
Question  --> HowadjQ      How old are you ?
Question  --> WhyQ         Why must I go to bed ?
Question  --> PPQ          In which bus did you see a telescope ?
Question  --> WhoseQ       Whose dog barked ?
Question  --> HaveQ        Have you any idea ?
Question  --> BeQ          Is the moon out ?
Question  --> YesNoQ       Did you find the sock ?
```

A further analysis can divide these according to the main characteristic, what we are asking for.

```
Question -->   AdverbialQ |  % we are asking for an adverbial
               NominalQ |    % we are asking for an NP
               FrontedQ      % we are asking for confirmation


AdverbialQ --> WhenQ |
               WhereQA |
               HowQ |
               HowadjQ |
               WhyQ |
               PPQ

NominalQ  -->  WhichQ
               WhoQ |
               WhatQ |
               WhoseQ

FrontedQ -->   BeQ |
               HaveQ |
               YesNoQ
```

The following elaboration is only seen as a sketch of the principles, of typical examples. A crude grammar for these types of questions can be elaborated. Just to take one example to demonstrate an important principal method that we reduce the grammar for a question to the grammar for Statement by performing some move operations.

```
YesNoQ  --> Aux Statement

WhenQ   --> when YesNoQ
```

```
WhereQA --> where  (YesNoQ // Complement)   % where does the bus go

WhereQN --> where  (YesNoQ // Noun_Phrase)  % where does the bus go to

HowQ -->    how  YesNoQ

WhyQ -->    why  YesNoQ

PPQ -->     WhArg ( YesNoQ // PP).

WhichQ -->  which  Nominal_Phrase   Qverb_Phrase

WhoQ   -->  who    WhichQ // (which agent).

WhatQ  -->  what   WhichQ // (which thing).

WhoseQ -->  whose (Noun_Phrase \ Determiner)
               WhoQ \ (who has a Noun_Phrase that)

BeQ -->     is NP Be_Complement

HaveQ -->   has NP  Statement \ (Noun_Phrase  has)


Qverb_Phrase -->  Verb_Phrase |
                  YesNoQ  Statement // Noun_Phrase

WhArg --> Preposition  which  (NP \ Determiner).
```

Examples

```
Which train passes Trondheim ?

Which buildings did you see in a telescope ?

Which bus did you see a telescope in ?
```

3.7.1 Comments on the syntax description

The dual purpose of a grammar between generation and analysis is also reflected in the fact that a grammar for reading and understanding a language can be more lenient than than a grammar for writing the language.

The grammar listed above both *undergenerate* and *overgenerate* the language to use the production terminology. We could also say that they *underaccept* or *overaccept*, respectively.

A grammar overgenerates when it accepts sentences that are not grammatical. A system needs not be too strict concerning exact grammar, but it should not accept meaningless input without a warning, and sometimes it is necessary with an accurate analysis in order to find the right interpretation among several possibilities.

Ellipsis

A major source of undergeneration is a phenomenon called *Ellipsis*. An ellipsis means an omitment in the text because it is assumed to be understood from the context and by convention. As

such, Ellipsis violates the syntax. The simple forms of Ellipsis may be

```
"John had two red apples and a green (apple)"

 "The man worked in 1960 and later (than 1960)"
```

In a dialog, an ellipsis may be just a part of speech that is understood to be put into an earlier stated phrase.

```
" I want to take a train ."

( response: " Where do you want to go ?)

" (I want to take a train ) to Oslo "


(response:  " Do you want the 1200 train ?'')

" No, (I want a train) later (than 1200) "
```

Anaphora

A related phenomenon called anaphora where the uses of certain constructions like pronouns and definite forms refer to objects elsewhere in the text is treated elsewhere. Note that, unlike ellipsis, this phenomenon does not violate the syntax; the syntax of anaphora is straightforward. The problem is to decide what they are referring to, which is a kind of semantic and pragmatic problem.

Agreement

Another source of overgeneration is the lack of agreement checks.

```
Number agreement    The boys takes the chocolate *

Gender agreement    (None in English)
                    Norwegian:  Gutten  stjal en eple*

Case agreement      He killed she *

Tense               He will takes his ball  *

Subcategorization   He  ran  the mocking bird *

Semantic agreement  The apple ate the boy *
```

3.7.2 Advanced Topics in Grammar

There are a number of new grammatical constructions that we must have. These are listed below, and demands a further analysis. Especially, when the combine together, the analysis can sometimes be rather tricky.

Example:

> "The professor said there were no possibility for John to know the grade before the committee held the meeting"

We shall briefly discuss some of the issues.

Sentential clauses

Whole statement phrases can also act as noun phrases in a composite sentence. These are usually complements to so called reporting verbs, of which there are to kinds:

- with statement clauses: `Olaf said that he sings terribly`
- with infinitive clauses: `Tore wanted to sing`

Examples:

```
John said a statement
John said that his mother had died
John said "My  mother has died"
```

Here, "his mother has died" is a statement in its own right, but acting as a noun phrase.

Subordinate Clauses

We can combine sentences using conjunctions. If the sentences are of equal status, we use the wording conjunctions 'and', 'or', 'but'. However, we can also have main sentences combined with subordinate sentences.

```
John took the car before Mary got home.
```

Here, 'before' is a subordinating conjunction combining two statements.

```
John took the car.
Mary got home.
```

Antecipatory Subjects

In many sentences, we use the words "there" or "it" for referring to something that appears later:

```
It is a fact that there are no oranges in the fridge.
```

means

> "No oranges are in the fridge" is a fact.

Question Clause Nominalisation

This construction allows questions to be turned into noun phrases

```
I know how much it costs.
I dont know what the price is.
I paid him what he wanted
```

3.8 EXERCISES

```
1. Analyse the following complex sentences according to the
   description of the grammar:

   The following sentences are both a part of the task,
   and examples of complex sentences.

   You may need to consult othere sources of grammar in order to
   manage this task.

   Without being rigorous try to make an account for how the parsing
   would proceed.

   How much does it cost to take the train before the holiday starts.

   I want to know if the bus goes on monday now.


2. The prose language used in this chapter is of course an
   example of English text, and as such it should follow the rules of
   English that it prescribes.

   Find examples of the text in this chapter that is not
   covered by the grammar.
```

Chapter 4

Logic and Knowledge Bases

Natural language can be used to query information systems of various kinds. The prototypical application is natural language access to databases. It is therefore important to study how these systems can be structured at all, and how they are accessed, using formal query languages. Later, we shall learn how to translate natural language queries into the equivalent formal language queries.

We shall in this chapter discuss the content and structure of so called knowledge bases. In doing so, we shall use Prolog as our common representation language.

4.1 Relational Databases

A data base (DB) is a shared collection of interrelated data stored independently of the programs using it, allowing the data to be retrieved, inserted, deleted and modified in a controlled way.

The amount of data is typically large, and the contents change over time, while the database as such may have a longer life-span than application programs written to manipulate it.

In Prolog, we define the database to be the *set of facts* , even though this definition does not fulfil all the characteristics above. However, there is nothing in the language Prolog that should prohibit it from working directly on large, permanently stored relational databases. In any case, Prolog is well adapted to interface a relational database.

One of the landmarks in the evolution of the database field is the introduction of the relational model (Codd 1970). Here, the data are considered as being defined by *relations* over domains, and the individual facts are represented as tuples of values from these domains. A relation with a set of tuples is also called a *table*.

The relational model is conceptually clean, with a solid mathematical foundation, and lends itself to data modelling.

In contrast to ordinary databases, the relational model has no pointer concept so the associations between different tables are via explicit identity of the values of attributes. This principle puts more strain on the implementors to achieve a high speed, while the gain is increase in the flexibility and closeness to an easily understandable model.

The relational model led at its introduction to enormous research activity, but is now an established method. The readers should consult the many good text books on the topics, e.g. Date (1986). Relational models are important for us, both because tables are a natural way of storing interrelated facts in Prolog, and because relational databases are an important application area for natural language.

4.1.1 A Relational Example

As a little example of relational databases, we will model two relations

```
PERSON -  containing name, sex  and parents,

and

CAR - containing   car number, make, owner, colour, type and nationality


PERSON

name        | sex  | father |  mother
------------------------------------
halvard     | m    | tore   |  catherine
tore        | m    | ole    |  ragnhild
catherine   | f    | harold |  helga
anne        | f    | tore   |  catherine
robin       | m    | harold |  helga
ragnhild    | f    | olaf   |  alma

CAR

number  |  make     |  owner     | colour | type   | nationality
---------------------------------------------------------------
123     |  fiat     |  ole       | brown  | sedan  | italy
321     |  volvo    |  ragnhild  | green  | sedan  | sweden
314     |  citroen  |  tore      | blue   | combi  | france
111     |  ferrari  |  catherine | yellow | sport  | italy
```

A static database in Prolog of the above example would be:

```
%% table person(name,sex,father,mother).

     person(halvard,m,tore,catherine).
     person(tore,m,ole,ragnhild).
     person(catherine,f,harold,helga).
     person(anne,f,tore,catherine).
     person(robin,m,harold,helga).
     person(ragnhild,f,olaf,alma).

%% table car(number,make,owner,colour,type,nationality)

     car(123,fiat,ole,brown,sedan,italy).
     car(321,volvo,ragnhild,green,sedan,sweden).
     car(314,citroen,tore,blue,combi,france).
     car(111,ferrari,catherine,yellow,sport,italy).
```

One or more of the attributes have a special status of being unique within the table. Such an attribute is called a *key,* and identifies the objects that we store information about. We use underscore to tell which attributes are keys. Example:

```
person

name  sex father mother
----
```

4.1.2 Binary Relations

There exists an even simpler form of relations called binary relations, with only one extra attribute for each key. The car relation would then represented in an equivalent way, but split into 5 relations

```
number-make     number-owner    number-colour number-type  number-nat
123    fiat     123    ole      123    brown  123    sedan 123    italy
...........     ............    .............
```

However, for convenience we usually collect related information into one relation as often as we can. A minor problem occurs when we want to handle missing data in relations. In a binary representation, we just leave out the tuple, e.g. a car without an owner. But if we make a combined tuple, with all the attributes, then we need to represent the missing value by a special symbol, e.g. nil.

```
Example:

      car(222,mercedes,nil,purple,bus,germany).
```

4.1.3 Composite Keys

In a simple implementation strategy, we may assume that there is only one key, which is the first argument. For composite keys, we make an ad hoc convention to represent the composite key as a list of simple attributes,

```
    (k1,k2,k3)
```

but with all the individual attributes k1,k2,k3 appearing as separate attributes.

4.1.4 Data Base Retrieval

Data base retrieval means combining and presenting the content of the data base in ways that serve our needs. We want to use databases to retrieve information. In ordinary database applications, this is done by a combination of programs and databases. In Prolog, it is done by specifying the conditions that the tuples must fulfil.

A few defined general Prolog operators are useful. (They are defined in Prolog somewhere else).

```
list X: Y.   % For all solutions of Y, output X

listall Y.   % List all solutions of Y


Examples:

"Who has  a VOLVO ? "


:- list P: car(N,volvo,P,C,_,_).

 ragnhild

" What make of cars do the women have ? "
```

```
:-list MAKE:
   (person(P,f,_,_),
    car(N,MAKE,_,_,_,_)).


  ferrari
  volvo

" List the car  relation ! "


:-listall car(_,_,_,_,_,_).

     car(123,fiat,ole,brown,sedan,italy).
     car(321,volvo,ragnhild,green,sedan,sweden).
     car(314,citroen,tore,blue,combi,france).
     car(111,ferrari,catherine,yellow,sport,italy).
```

4.1.5 Virtual Tables

The relational model is a subset of predicate logic, and it is not necessary to stay within its borders. One of the facilities of Prolog is the possibility of defining new tables without creating them, i.e. by logical implication. We call such tables *virtual tables* (*views* in DB terminology).

For example, I can define a table carcolour containing only the number and the colur:

```
carcolour(X,Y):-car(X,_,_,Y,_,_).

?-listall carcolour(X,Y).


    carcolur(123,brown)
    carcolour(321,green)
    carcolour(314,blue)
    carcolour(111,yellow)
```

The concept of virtual tables is just an adaption of the subset of Prolog without function or list symbols (sometimes called Datalog). In addition to the Prolog database, we define new concepts and rules to extend the level of information. We then no longer have a pure database, the more appropriate name is a Knowledge base, although we shall define that concept later. Consider the query:

```
" Who  has a grandmother with a green VOLVO ? "
```

In Prolog, we define the rules of grandmother, ownership and car colour if that suites our purposes:

```
grandmother(X,Z):-mother(X,Y),parent(Y,Z).
parent(X,Y):- father(X,Y).
parent(X,Y):- mother(X,Y).
father(F,X):- person(X,_,F,_).
```

```
mother(M,X):- person(X,_,_,M).
owns(X,N)  :- car(N,_,X,_,_,_).
colour(N,C):- car(N,_,_,C,_,_).
make(N,M)  :- car(N,M,_,_,_,_).

?-list P:
  (grandmother(X,P),owns(X,C),make(C,volvo),colour(C,green)).

halvard
anne
```

4.1.6 Symbolic Naming

As we see, when we access Prolog tables, we are using the position in the table of the argument for accessing the column. This becomes awkward when the number of arguments gets big. Moreover, it ties the program to a concrete realisation of the relations. What we need is a more abstract representation, which allows programs to survive changes in the data model. A solution is to make virtual binary tables, containing the attribute name as an explicit argument. For larger tables, this technique may become a virtue of necessity.

We can make a general access predicate "hasvalue" defined for all the attribute names.

```
 hasvalue(car,number,N,N):-      car(N,_,_,_,_,_).
 hasvalue(car,make,N,M):-        car(N,M,_,_,_,_).
 hasvalue(car,owner,N,O):-       car(N,_,O,_,_,_).
 hasvalue(car,colour,N,C):-      car(N,_,_,C,_,_).
 hasvalue(car,type,N,T):-        car(N,_,_,_,T,_).
 hasvalue(car,nationality,N,A):- car(N,_,_,_,_,A).

Example: (same as above)

"Who has a VOLVO ?"


?-list P: (hasvalue(car,make,N,volvo),
           hasvalue(car,owner,N,P)).

 ragnhild
```

The relational model imposes a restriction that certain fields are key fields, whose values are unique in the table. Thus, NAME is the key in the PERSON relation, while number is the key in the CAR relation. We can define a predicate

```
key(car,Key):-hasvalue(car,number,Key,_).
```

so that

```
?-key(car,Number)
```

becomes equivalent to

```
:-car(Number,_,_,_,_,_).
```

In Prolog, a system for creating and manipulating relational databases may be implemented. We can make a Prolog operator `createtable`, so that the call

```
?- createtable   car(number,make,owner,coulour,type,nationality)
```

will create the access predicates "hasvalue" and "key".

A system for manipulating relational databases may be implemented very naturally. The basic operations are:

```
- forget(Predication)  % remove tuples that match Predication
- remember(tuple)      % insert a tuple if it is not there
```

```
Example:

?- forget(car(_,_,tore,_,_,_)).  % Remove whatever cars Tore has

?- remember(car(555,rolls_royce,tore,pink,sedan,england)). % New car
```

4.2 Data Modelling

A data base is not only a collection of data items, but also the associations or relationships between these items. The associations between data is called a *data model*.

When we build knowledge based systems, we must not forget the fundamental fact that knowledge based systems also are Data based systems.

Data base technology has given us methods and tools for solving complex and large data management problems. The construction of stable logical data models is a very important task for any application.

We want to store information about things and their properties. Pure predicate logic is a very strong formalism, capable of representing almost everything, so we often have to restrict the language used for modelling, but revert to predicate logic to explain the semantics and to cover non-standard features.

4.2.1 Normal Forms

As with all modelling, the only important things to model are the fundamental invariants of a problem domain.

The most important invariant property is that objects belong to *classes* that can be stored uniformly in tables.

The evidence that data are *functionally dependent* on other data is another such principle:

A set of data B is functionally dependent on a set of data A if

for each data element a in A, there exists a unique element b in B such that b is related to a.

We commonly use the notations

```
A -> B    B is functionally dependent on A

A,B -> C  C is dependent on the combination of A,B
```

Example

```
employee -> employer
```

From the uniqueness of the key, it follows automatically that all the attributes are functionally dependent on the key.

An important principle behind any good design is to avoid redundancy. The same piece of information is only stored once. Thus, for any changes of a value, the data base needs only be changed in one place.

For relational databases, these principles are taken care of by a process called normalisation (Date, 1986). When designing relational databases in Prolog, the merits of normalisation are equally valid, when we have a database that is going to be updated.

4.2.2 Relational Normal Forms

Some principles of good design are embedded in the Third Normal Form (3NF) which is a refinement of Second Normal Form (2NF) and First Normal Form (1NF). 3NF is memorised by the motto:

> "Every item in a tuple is functionally dependent on the key, the whole key and nothing but the key."

These principles also apply to Prolog databases, even if they are smaller and not as dynamic as conventional databases.

The principles of normalisations can be applied manually for small models. However, for larger models, the normalisation can be automated by the help of programs.

Just as a reminder, the essence of the normalisations is

First normal form- (... dependent on the key ...)

Avoid repeating groups.

Second Normal Form (... the whole key)

Don't let any attribute depend on an attribute that is only a part of a composite key.

Third Normal Form (... and nothing but the key)

Don't let an attribute depend on a non key attribute.

4.3 Beyond the Relational Model

The pure relational model is not always powerful enough for advanced modelling, because it lacks semantic expressiveness. For one thing, every possible set of tuples in a normalised relational database is allowed. For example, the relational model does not require that for each employee, the employer attribute corresponds to an existing employer tuple in the database.

In realistic models, there are two kinds of rules that relate the tables to each others.

- generic rules that define new (virtual) tables that are not explicitly stored
- constraint rules that set restrictions on what is allowed to be in the data base

An example of constraint rules is functional dependencies which specify that key attributes are and must be unique. Another example might be a rule that

```
" All elephants are grey "
```

which allow us to deduce the colour of an elephant in a database, provided we have imposed a constraint on every update that no elephants may be stored with any other colour. Such databases are called *deductive databases*. Gallaire (1978) is a classic source of ideas for treating the database field from a logical point of view.

4.4 Semantic Nets

Questions of meaning are more important for the design of a knowledge base than methods of encoding data. When database designers add more semantic information their models begin to look like the knowledge representation systems developed by the AI-communities. One of these knowledge representation systemes are the *semantic nets*.

Semantic nets is a formalism for representing facts and relations between facts with binary relations.

Example:

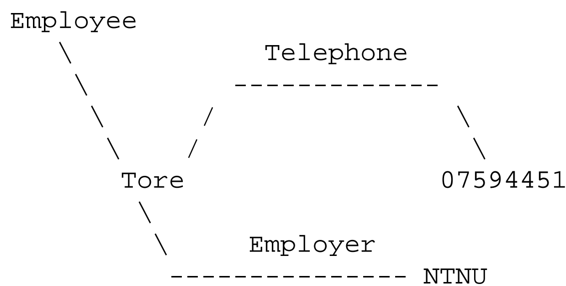

Figure 4.1: A simple semantic net

- Here ”Tore”, ”07594451” and” NTNU” represent objects
- ”Telephone” represents a relation between the objects ”Tore” and ”07594451”
- ”Employer” represents a relation between Tore and NTNU
- “Employee” represents a class of objects that Tore belongs to.

The individual relationships are connected into a net by letting the objects (e.g. Tore) be presented only once. For binary relations, Semantic nets is an excellent formalism with a perspicuous graphical notation. When moving from binary relations to n-ary relations, the semantic net loses some of its appeal, as it forces artificial constructions.

It is believed that human reasoning for a large part is based on associations along associative links. So the semantic net model is also an interesting model of human thinking.

In Prolog, we implement the binary relations individually, duplicating the object names.

```
Example:

    telephone(tore,07593016).
    employer(tore,ntnu).
```

4.4.1 The Class Concept

As soon as we have classified an object, we already know a lot of that object.

A *class* is a description of a set of similar objects, specifying attributes and properties common to all its members. For example, Tore is an object belonging to the class of employees.
An *attribute* may be something that can hold a value: For example, telephone is an attribute of employees (whether they have it or not). A *property* is an attribute together with a *value*.

Example:

```
A rose has the property of having colour = red

Tore has the property of having telephone = 07594451
```

A property may also be just an adjective when an attribute, assuming a truth valued attribute, e.g. for red roses, we have

```
A rose has red = true
```

A class may very well be void of elements (unicorns), and two classes with the same elements (incidentally) may in theory be quite distinct, e.g. the class of research managers and the class of Labrador retriever owners.
Example of classes:

```
animal
mammal
sperm_whale
elephant
shark
```

Example of attributes:

```
colour
inhalant
texture
food
blood_temp
habitat
```

A class may be a subclass of another class. If S is a subclass of C, and x is a member of S, then also x is a member of C.

```
mammal is a subclass of animal.

elephant is a subclass of mammal.
```

If Clyde is an elephant (is a member of class elephants), then Clyde is at the same time a member of 'mammal' and thereby a member of 'animal'.

If the class has an attribute or a property, it is shared by all its subclasses. A class may also have an attribute even if there are (currently) no members of that class.

Asking for values of non existing attributes, like the telephone of an elephant needs to be rejected as meaningless, and not only by the answer "none". If an entity has an attribute which is functionally dependent on it, (for instance "every person has a name"), and this attribute value is missing, the appropriate answer is "unknown", and not "none". If on the other hand, an attribute is not functionally dependent, like the children of a person, then missing occurrences of children should correspond to the answer "none", and not "unknown".

Example:

> All animals have a colour, (varies of course). Therefore, all mammals have a colour, and elephants have a colour, and so has Clyde.

If a class has a property, it is *inherited* by all the subclasses and members of the class.
Example:

```
All elephants  have colour = grey

   implies  that

Clyde has colour = grey.
```

In some traditions, an “attribute” is a class attribute that defines an adjective, i.e.

“red” for colour = red

“African” for origin = Africa

We can also store adjectival properties as attributes that have the value true (or false). Example:

```
quakers are pacifists
republicans are not pacifists
```

can be represented as

```
quakers     have pacifist = true.
republicans have pacifist = false.
```

To store the information of classes in a net together with the information about objects requires a few general relationships, as shown in Figure 4.2.

```
    ako
C -------> D     Class C is a kind of D
                   (C is a subclass of D)
                   (All C's are D's)

   have A
C -------> V     Class C has an attribute A with default value V


   has_a
C -------> A     Class C  has an attribute A


    apo
C -------> D     Class C may be a part of D


   is_a
E -------> C     Entity E is a C


   has A
E -------> V     Entity has attribute A with actual value V
```

Figure 4.2: Semantic net legend.

With this notation, we can draw a semantic net, see Figure 4.3 on the facing page.

The information in this net may adequately be represented as a set of Prolog clauses. A few infix operator declarations make it more readable:

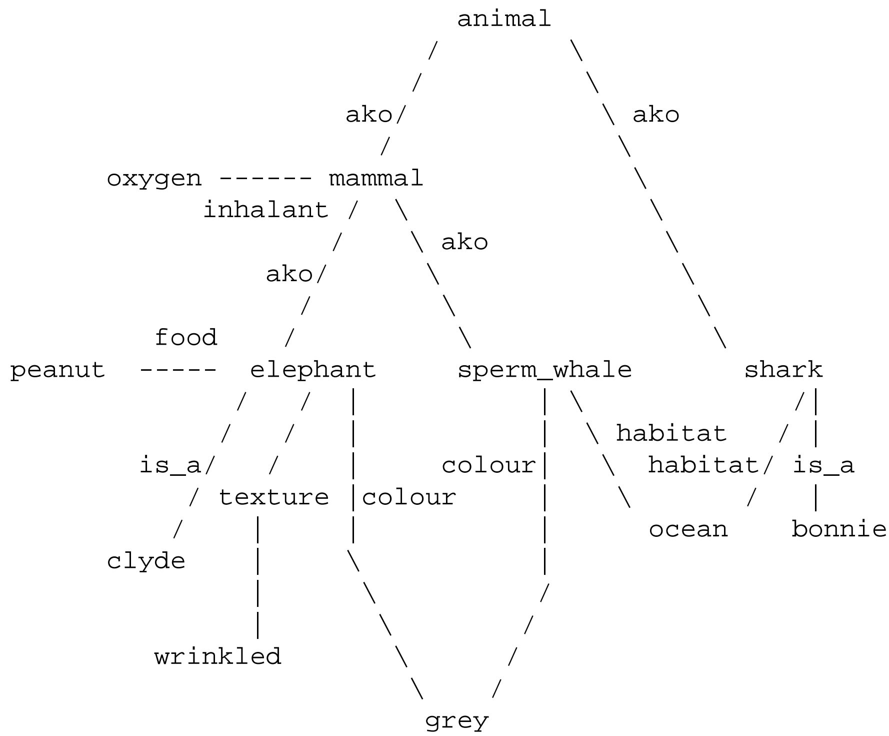

Figure 4.3: A semantic net.

```
:-op(900,xfx,[ako,is_a,has,have,has_a,apo]).


% animal      have inhalant=nil.
% animal      have blood_temp=nil.
% animal      have food=nil.

mammal      have inhalant=oxygen.
mammal      have blood_temp=warm.

elephant    have texture=wrinkled.
elephant    have food=peanuts.
elephant    have colour=grey.

sperm_whale have habitat=ocean.
sperm_whale have colour=grey.

shark have  habitat=ocean.
shark have  colour =grey.



mammal      ako animal.
shark       ako animal.
sperm_whale ako mammal.
elephant    ako mammal.

clyde  is_a elephant.
bonnie is_a shark.

bonnie has colour=white.
```

General axioms defining the class structure

```
 X is_a Z2:-
     Z1 ako  Z2,
     X is_a Z1.

 X has ATT=VAL:-
     X is_a C,
     C have ATT=VAL.
```

We are now able to answer general questions like

"Which properties does Clyde have ?"

```
?-list (Attr=Value):
   (clyde has Attr=Value).


texture=wrinkled
food=peanuts
colour=grey
inhalant=oxygen
```

```
blood_temp=warm


" What is  grey and wrinkled ?"

?- list OBJ:
     (OBJ has X=grey,
      OBJ has Y=wrinkled).

clyde



" Which properties do sharks have ? "

?- list PROP:(shark have PROP).

habitat=ocean
colour=grey



" List all the class properties ! "

?-listall (X have Y=Z).

mammal have  inhalant=oxygen
mammal have  blood_temp=warm
elephant have texture=wrinkled
elephant have food=peanuts
......
```

4.4.2 Another Example: A Course Knowledge Base

Below is an example of a small database containing information about a university course. Our programme is here to build up a knowledge base, which together with this information shall be able to answer questions pertinent to its content. The knowledge base will not be intelligent in any sense, and have no expertise except to have enough additional information to "understand" the information in the paper.
The language used to query this knowledge base will be that of Prolog. In a later chapter, we will discuss how to access its contents in in a very restricted, but nevertheless quite useful natural language subset.

4.4.3 The Course Model

CLASS DESCRIPTION

"thing " is the universal class

```
course_series ako  thing.
person ako  thing.
place ako  thing.
building ako  thing.
```

```
COURSE : LOGIC PROGRAMMING                  DATE  01-01-15

  PLACE OF EDUCATION:   ROOM R4
                        REALFAGBYGGET,
                        NTNU

  CONTACT PERSON (PHONE)  Florence Nightingale   (5266)


COURSE LEADER    Tore Amble

LAST_NAME,FIRST_NAMES         EMPLOYER
   OCCUPATION                 ADDRESS
   (AREA) PHONE               ZIP  PLACE


Bond, James                    Centre for Intelligence
   Research Scientist          Box 461
   (02) 563562                 0314 Oslo


Columbus, Christoffer          Technological Institute
   Consultant                  Box 9227
   (02) 5770991                0032 Oslo


Einstein, Albert               Television  Inc.
    Engineer                   Box 70, Horsemarket
    (02) 638800                0500 Oslo


Freud, Jan Sigmund             University of Trondheim
    Psychologist               Dragvoll
    (07) 031522                7000 Trondheim

........

West, Mae                      Institute of Science
   Scientific Assistant        Kings  Street 29
   (05) 323709                 5000  Bergen
```

Figure 4.4: A course knowledge base.

```
floor ako  thing.
time ako  thing.
name ako  thing.

person have full_name.
person have last_name.
person have first_name.
person have occupation.
person have phone.
person have employer.
person have address.
person have zip_code.
person have post_address.

leader ako  person.
secretary ako  person.
participant ako  person.


date ako  time.

course_series have name.

course have secretary.
course ako  course_series.
course have start_date.
course have exam_date.
course have leader.
course ako  course_series.
course have place.

room have building.
room have floor.

building have location.

lp ako  course_series.
lp have name='Logic Programming'.

researcher ako  participant.
researcher has occupation='Researcher'.



ENTITY DESCRIPTION

lp1 is_a course  % inheriting course attributes
lp1 is_a lp.     % inheriting the name 'Logic Programming'.

lp2 is_a course.
lp2 is_a lp.

lp2 has place='Room 229'.
lp2 has leader='Tore Amble'.
```

```
lp2 has secretary='Florence Nightingale'.

'Room 229' is_a room.
  .
  .
```

4.4.4 Coupling Semantic Nets to Tables

When facts are stored in tables, we have to combine the semantic net with the relational table access at the bottom level. We can then let the semantic relations be the attributes, prefixed with the table name to make the relations unambiguous.

```
    X is_a Y :-key(Y,X).
    X  has Tab:Att = Val :-
        hasvalue(Tab,X,Att,Val),
        Val \== nil.
```

Participant database

```
participant('Mae West','Mae','West',
     'Scientific Assistant','(05)323709',
     'Institute  of Science',
     'Kings Street 29',5000,'Bergen').

......
```

The above example simulates the following definitions:

```
   'Mae West' is_a participant.

   'Mae West' has participant:full_name='Mae West'.
   'Mae West' has participant:first_name='Mae'.
   'Mae West' has participant:last_name='West'.
    .....
   'Mae West' has participant:post_address='Bergen'.
```

4.4.5 Example of Questions Asked

The following questions asked are generic examples that by slight generalisations indicate what the knowledge base must be able to answer. The examples are given, with a natural language formulation as initial comment.

```
1.   " Who is the leader of the  course ?"

 ?-  list X: (L is_a course, L has leader=X).

2.   " How many participants are there ?"

 ?-  list N: countall(Y,Y is_a participant,N).
```

```
3.   " Which of the researchers are from Centre  of Intelligence ?"

 ?-  list X: (X is_a researcher,
             X has employer='Center of Intelligence').

4.   " Which of the participants have got no phone ?"

 ?-  list X: (X is_a participant, not X has phone=Y).

5.   " What is the name and phone  of the secretary of the course ?"

 ?-  list (Name,Phone):(L is_a course,L has secretary=S,
        S has name=Name, S has phone=Phone).

6.   " Who are not participants ?"

 ?-  list X: (X is_a person, not  X is_a participant).

7.   " Where is the course location ?"

 ?-  list X: (L is_a course, L has place=X).


8.   " Have I got any companies with more than one
        participant "

 ?-  list C : (C is_a company,
         countall(X,X has employer = C,N), N > 1).
```

4.5 Issues on Semantic Nets

Semantic nets is a way of organising the categories and their properties in a structured and uniform way. It is important to acknowledge that a class is more than a set. A set of properties is defined for a class. However, if a property is applied outside the class, it is not only false, but undefined (illegal).
In addition to the above, there are the following complications that also need to be handled.

4.5.1 Flexible Attribute Classes

An attribute is normally a class, and the elements are usually a member of that class

Every person has (can have) an address, (each of which belongs to the class of addresses).

However, there are exceptions:

- A bird has a parent which is a bird (not only parent)
- A dog has a father which is a dog

This can be formalised in various ways, e.g.

```
class_attribute(Class,Attribute,AttributeClass)
```

e.g.

```
class_attribute(horse,mother,horse)
```

4.5.2 Inheritance in a Hierarchy

The simplest form of semantic nets are made as a *hierarchy*, i.e. each class has a unique parent class. A semantic net can be used to store assumed or default values and at the same time contain definitions that by inheritance would contradict these. For instance

```
bus  have  weight = 4.

minibus ako bus.

minibus have weight = 2.

X is_a minibus.
```

If X is a minibus, it could derive two values for its weight, one value from the class definition of minibus, and another indirectly from the class definition of Bus, since X by definition also is a bus. To solve this dilemma, a principle of scoping is understood, if a member or a class can inherit an attribute value for several of its superclasses, only the closest enclosing class will define its attribute value.

4.5.3 Inheritance in a Heterarchy

In some applications, there is a need for multiple superclasses. We then call the semantic net for a *Heterarchy*. A good example is 'girl', which is both 'child' and 'female'. Wherever there is a conflict, care must be taken, because the simple rule for inheritance in hierarchies is not sufficient. Let us take a contrived example of average body weight as shown in Figure 4.5.

```
                person  weight=50

                  /   |
                 /    |
                /     |
               /      |
    male   female   child weight=30   adult weight=70
              \       |
               \      |
                \     |
                 \    |

                  girl
```

Figure 4.5: A simple heterarchy.

A girl both inherits a weight directly from the class child, and indirectly from the class person via the female class. The rule for inheritance in a hierarchy does not apply, because there are two different enclosing superclasses (female or child). In that case, it is a convention that the child superclass is defined as closer to girl than person is, because the class person is outside the class child. We can paraphrase the inheritance rule by saying that

- An object inherits a value from a superclass if there are no *intervening* inheritance.
- If there are multiple values so inherited, the value shall be undefined.

In the case above, the girl could have inherited an attribute weight from person, but the inheritance from child is an *intervening* inheritance that excludes the former from consideration. But,

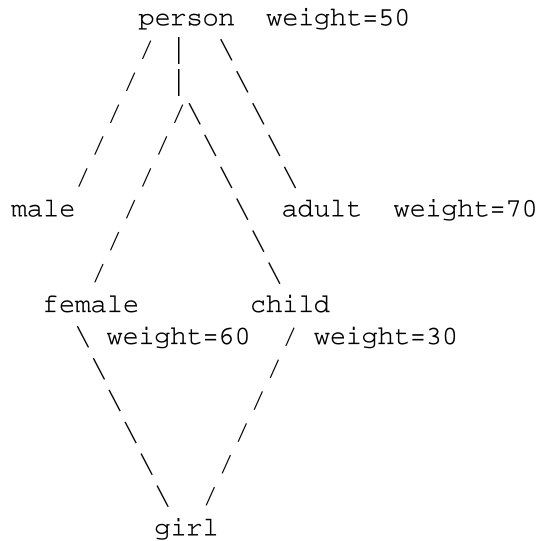

Figure 4.6: A more complicated heterarchy.

if we had the picture in Figure 4.6, where also female has got an attribute 'weight', there would be no rule to help us prefer one for the other.

The rules can be summarised by predicates that defines inheritance in a general semantic network. A summary of the fuller picture of inheritance can be done by studying the following Prolog code.

```
%%%% Representation axioms

ako(mammal,animal).
  have(mammal,weight,50).

ako(elephant,mammal).
   have(elephant,texture,wrinkled).

have(elephant,weight,5000).

is_a(clyde,elephant).
    has(clyde,weight,3000).

ako(shark,animal).
   is_a(bonnie,shark).

has(bonnie,weight,1000).

have(person,pacifist,nil).

ako(quaker,person).
ako(republican,person).
```

```
have(quaker,pacifist,true).
have(republican,pacifist,false).

is_a(nixon,quaker).
is_a(nixon,republican).

ako(person,thing).
    have(person,weight,50).

ako(female,person).
    have(female,weight,60).

ako(child,person).
    have(child,weight,30).

ako(girl,female).
ako(girl,child).


is_a(alice,girl).

 % Access relations

subclass(C,D):- %% exclusive
       ako(C,C1),
       subclass0(C1,D).

subclass0(C,C). %% inclusive
subclass0(C,E):-ako(C,D),
                subclass0(D,E).

instant(X,D) :- is_a(X,C),
                subclass0(C,D).

hasval(X,A,V) :- has(X,A,V).
hasval(X,A,V) :- instant(X,C),
                 have(C,A,V),
            \+   interveningatt(X,C,A).


interveningatt(X,C2,A) :-
    interveningclass(X,C1,C2),
    have(C1,A,_).


interveningclass(X,C1,C2):-
     instant(X,C1),
     subclass(C1,C2).
```

4.6 EXERCISES

1. Define two relations

```
emp(Name,Dept,Salary)

dept(Department,Manager).
```

Make a Prolog query answering

" Which employees have a bigger salary than their manager ".

2. For classes.

Do the course example with your class as a case. Discuss the semantic model, with the teacher as a moderator. Supply your own information about yourself into a common knowledge base. Try it out, and let it grow.

3. Define the relations

```
country(X)          X is a country
sea(X)              X is a sea
population(X,Y)     X has population Y
border(X,Y)         X and Y are bordering
```

Define in Prolog a query to answer the question:

Which country bordering the Mediterranean borders a country that is bordered by a country whose population exceeds the population of India ?

4. Make a query database of the results of the Olympic Winter games, from the start in 1924 up to now. One kind of competion is the ski relay race over various distances. Include the items

Year,
Place,
Competition,
Men or Women,
The first 6 places in each competition with result.
For team competions, (relay race, games),
the name of the 6 best nations, and the members
of the teams.

Answer the following query:

" Who were team members of the medal winners (1-3) in the ski race relays (any distance) at least 3 times ? "

Chapter 5

Natural Language Processing in Prolog

5.1 Natural Language Systems in Prolog

Natural language processing in Prolog started with Prolog itself, when Colmerauer in the early 1970's developed a Prolog-based natural language system called *metamorphosis grammar* (Colmerauer,1978). The ideas developed here have since been applied in various logic grammar formalisms and systems, e.g. by Dahl (1977), who made a natural language system for Spanish.

The most influential formalism, Definite Clause Grammar, was developed by Warren and Pereira and applied to make a knowledge based system CHAT-80 (Pereira,1980) for geography with a natural language interface. CHAT-80 can answer the following question *in extensio*:

```
" Which country  bordering the Mediterranean borders
  a country that is  bordered by a country whose population
  exceeds  the population  of India "
```

The solution, Turkey, was found within 405 milliseconds (1980).

Also the ORBI expert system for environmental resources (Pereira,L.,1982) is a Prolog based system with very advanced language capabilities:

```
 > Which descriptors of the aptitude intensive agriculture
   at point 56,78 that are greater than 2, are equal to the
   factor F1 of point 12,95?
   .......

 > and at point 65,78 ?
   ........
```

The system will understand that this is an ellipsis, whose meaning is found by replacing ”point 56,78” in the question above with ”point 65,78” .

Prolog is superbly adapted to the task of natural language processing in all its main aspects:

- - as a grammar formalism
- - as a formalism for meaning representation
- - as a knowledge base query language

5.2 Natural Language Query Processing Overview

Now, we shall return to the to the Knowledge Base of the Logic Programming course description. For that case, we have presented a Prolog-based query language, and gave several examples. Not incidentally, we presented the queries with natural language phrases as informal descriptions of the meaning of the questions.

After the semantic analysis of a syntax tree, and a transformation of the tree to a suitable form, where all semantic ambiguities and discrepancies have been removed, a query processor is called. If there is a larger data base at the bottom, a query optimisation is needed.

We shall just indicate how a query can be processed in an interpretive way, using the semantic net as a Knowledge base.

The result of the question

”Who is the leader of the dull Prolog course”

can be as follows.

```
list Z:
      (X is_a course,
      X has Someattribute=prolog,
      X has dull=true,
      X has leader=Z).
```

5.2.1 Verb-Free Language

An interesting simplification for a query language is to orient it around what is called ”verb-free language”, under the motto

```
"To be or to have, that is the question"
```

i.e. a language where only the verbs ”be” and ”have” were taken as semantically primitive, while all other verbs had to be defined in terms of these. The motto also implies implicitly that we only analyse *questions* for their meaning, while the knowledge is given as a set of exact, unambiguous facts and rules. For example, the statement

```
" Most people have a car "
```

is not represented. What may be represented is a database of cars and people, so that the question

```
" Do  most people have a car ?"
```

means counting and comparing people with or without cars.

The rationale behind these ideas is that databases store information as factual data, organised by letting the attributes of the database be nouns.

```
table PERSON

      Date of Birth | Department | Address
     ---------------------------------------
```

Thus, questions may be reformulated as:

```
not  "When were you born"
but  "What is your date of birth"
```

```
not   "Where do you live ?"
but   "What is your address ?"


not   "I walked to the course today "
but    the explicit  formulation
       of the conclusion of the state of matter
       you want the listener to draw:

      "I am tired"
or    "My house is close to the lecture room"
or    "Today my car is broken"
or    "My leg is healed".
```

In this section, we will try to describe how to process the natural language queries directly. It is left as a challenging project exercise for the advance students to elaborate it.

All the questions must be answered from the information stored in the semantic net of the course example. However, there may be additional information necessary to capture all the variants of the same questions. Below follows a list of questions that will be asked by collecting the natural language programs herein.

```
" Who is the leader  of the Prolog course ?"

" How many participants are there ? "

" Which of the researchers are from   RUNIT?"

" Which of the participants have got no phone ?"

" What is the name and phone number of the secretary
  of the course  ?"

" Who are not participants. "

" Where is the course location ?"

" Have I got any companies  with  more than one participant ?"
```

5.2.2 The Dialog Context

The meanings of all the phrases here are to be understood in the framework of one person communicating by natural language text to a computer. The answers to the questions are presented back to the user. The answer sets are supposed to be a sequence of factual answers. There are two kinds of questions:

- Context free questions, having no reference to previous answers.

  " What researchers are there ?"

- Context sensitive questions referring to previous answers

  " Which of these are from Bergen ?"

- General questions referring to the systems capabilities.

  ” What can you answer ?”

Each question/answer in the dialog represents a ”theme” which is the class qualifier (e.g. researcher) representing the type of answer. In theory, all the answer sets could be remembered by the computer. However, as it is cumbersome to refer to all of them in natural language, a simplification is understood:

```
If there is more than one answer set for a theme,
the system remembers the last answer (set) -
any other  answers are forgotten.
```

It is a part of an intelligent dialogue handler to find out which answer sets are referred to. We will discuss this lightly in a later chapter.

5.2.3 The Reference Model

In TUC Systems, there are 4 distinct phases. They were chosen so, because it is considered good engineering practice to separate subproblems into submodules in order to get each module manageable. Figure 5.1 shows the principal model of the translation.

The architecture is reflected in the main Prolog program:

```
tuc_system:-
    lexical(User,Wordlist),
    syntax(Wordlist,Syntaxtree),
    semantics(Syntaxtree,Queryexpression),
    query(Queryexpression).
```

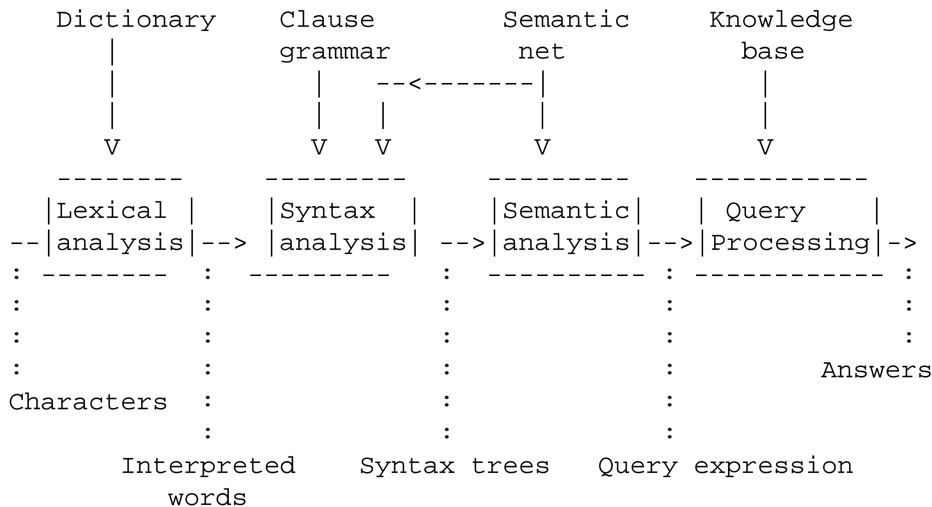

Figure 5.1: The principal model of the translation.

The information in the semantic net is available as global information.

5.3 Lexical Analysis

The task of the lexical preprocessor for natural language is to scan the character of the input file, assemble strings into words, analyse each word for morphological inflections, and to perform spelling corrections.

Many words, especially names are in practice gathered from the semantic net of the application domain and the associated database.

There are many strategies for spelling corrections. From an implementation point of view, as seen from the rest of the system, it is not so important which strategy is chosen, the important thing is to live with it and let spelling correction be adapted to new environments when necessary.

An important design criteria is the limit of allowable errors. By using expert system techniques, it is possible to calculate probabilities for typing errors, so that with a combination of statistical calculation involving both lexical, syntactical and even semantical considerations of surrounding fragments, a meaning can be extracted from a very fuzzy text. But, while such techniques are a must for speech understanding, a quite acceptable human reaction to nonsense or complexity is to reject it and ask for a clarification. For TUC, a modest approach is adopted. It represents a *plausible* sloppy typist.
A priori, all words could be interpreted as a name, (not relying on any capital letter convention, as in the example: " The book on the Wall by green and wall is Green " .), so all words were represented to the syntax analyser in two forms:

- the exact representation, (in case that it is a name).
- the most probable correction

The most probable correction was chosen according to the following criteria.

If the word could be transformed from a correct known word by

- - one additional *charracter*
- - one missing *caracter*
- - one wrong *cjaracter*
- - two (adjacent) *charatcers* interchanged

then the known word was plausible.

If there was one plausible word, it was taken. However, a more sophisticated analyser should be able to carry the whole list of plausible candidates for a misspelled word.

Another shortcut in TUC Systems is that only the root form of the known word is used.

In query systems, it usually suffices to use only the root form in order to understand the plausible interpretation of a sentence, while the information thereby lost either could only be used for correction of the user, or was redundant information which the system could make no use of. This policy derives from TUC's *attitude model*:

There are at least three attitudes towards handling erroneous input:

- The post office detective attitude to find the correct address at all costs
- The teacher's attitude to reject and correct all errors (as teachers are supposed to)
- The servants attitude to understand an order when there is only one plausible interpretation.

Output from the lexical analyser is a list of triples.

```
w(Word,Tag,Spell)
```

where

```
Word     is the actual word, though in small letters only
Tag      is the cassification of the word
Spell    is the degree of spell correction
              (0 perfect match,1 spell correction)
```

The classification is one of several terms:

```
noun(Root,Num,Def,Gen)   Noun
verb(Root,Time,Fin)      Verb
adj(Root)                Adjective
prep(Prep)               Preposition
[Word]                   Any other word
```

```
Legend

   Root - root form
   Num  - grammatical number (sin,plu)
   Def  - definite/indeefinite ( Norwegian only)
   Time - grammatical time (pres,past,inf)
   Fin  - finite/participle (fin/part)
```

As an example, the not quite correct sentence

```
" Has I got any  copanies  with more than   1 parcitipants ? "

is represented as  a list of triples

  [   w(has,verb(have,pres,fin),0),
      w(i,[i],0),
      w(got,verb(get,past,part),0),
      w(any,[any],0),
      w(copanies,noun(company,plu,indef,nogen),1),
      w(with,prep(with),0),
      w(more,[more],0),
      w(than,[than],0),
      w(1,num(1),0),
      w(parcitipants,noun(participant,plu,indef,nogen),1)
  ]
```

Actually, the lexical processor of a TUC systems is in fact a **Multitagger**, where each word is assigned a list of possible interpretations.

```
w(drink, [verb(drink,pres,fin), noun(drink,sin,indef,nogen)] ).
```

It is then the task of the syntax analysis to find the interpretations that make sense, and perform the analysis with these.

TUC also translates the lists of words into a word graph with explicit numbering of the points in the text, and a word item that connects them.

5.4 Syntax Analysis

We shall briefly introduce the reader to Definite Clause Grammars.

The sentence to analyse is

```
"John loves Mary"
```

We can let the text "John loves Mary" be represented as Prolog facts concerning numbered points in the text and which words that connect the points.

```
txt(john,0,1).
txt(loves,1,2).
txt(mary,2,3).
```

A context free grammar in DCG format could look like

```
s --> np,vp.

np --> proper_noun.

vp --> verb,np.

verb --> txt(loves).

proper_noun --> txt(john).
proper_noun --> txt(mary).
```

This text is automatically translated into a set of Prolog clauses

```
s(X0,X2)  :- np(X0,X1),vp(X1,X2).
np(X0,X1) :- proper_noun(X0,X1).
vp(X0,X2) :- verb(X0,X1),np(X1,X2)

proper_noun(X0,X1) :-txt(john,X0,X1).
proper_noun(X0,X1) :-txt(mary,X0,X1).
verb(X0,X1)  :-txt(loves,X0,X1).


Now, the analysis is left to Prolog as a combinatorial  puzzle:

    ?-s(0,3).   Yes, because

        np(0,1) because
           proper_noun(0,1) because
               txt(john,0,1)
        vp(1,3) because
           verb(1,2) because
              txt(loves,1,2)
           np(2,3) because
              proper_noun(2,3) because
                txt(mary,2,3)
```

5.4.1 Definitie Clause Grammars (DCG)

The real Definite Clause Grammar of Prolog uses pair of lists in stead of pair of points, and one generic txt predicate.

```
txt(Word,[Word|Rest],Rest).
```

The parsing above would then be evoked by

```
?-s([john,loves,mary],[])
```

Instead of `txt(Word)`, we can use `[Word]`

The empty phrase is accepted by `[]`, which corresponds to `txt(_,X,X)`

A proper understanding of the DCG formalism involves list handling in Prolog, but that is not necessary for the understanding at the moment.

DCG grammars are translated into executable Prolog code, and are as such executed by a top down, left to right manner with unification and backtracking. Therefore, left recursive rules usually cause endless recursion and must be avoided.

Definite Clause Grammar facilities

Prolog has the capacity to load definite clause grammar rules (DCG rules) and automatically convert them to Prolog parsing rules. As an illustration, consider the following little grammar. Interspaced between the right side goals, we can have Prolog conditions surrounded by curly brackets {}. These predications are not expanded into Prolog by augmentation of twin text variables, but are just copied as they are. These can be used to perform tests and other operations on the attributes that are not already implicit by the unification.

We can have variables inside literal expressions in square brackets, which may be subsequently bound by Prolog predications in curly brackets.

```
pn --> [PN], {pn(PN)}.        /* proper noun */
```

Furthermore, grammar rules can be mixed with Prolog clauses quite ad lib.

A complete example of a DCG grammar

```
% Rules

    s --> np, vp.                 /* sentence */

    np --> pn.                    /* noun phrase */
    np --> d, n, rel.

    vp --> tv, np.                /* verb phrase */
    vp --> iv.

    rel --> [].                   /* relative clause */
    rel --> rpn, vp.

    pn --> [PN], {pn(PN)}.        /* proper noun */
```

```
    rpn --> [RPN], {rpn(RPN)}.   /* relative pronoun */

    iv --> [IV], {iv(IV)}.       /* intransitive verb */

    d --> [DET], {d(DET)}.       /* determiner */

    n --> [N], {n(N)}.           /* noun */

    tv --> [TV], {tv(TV)}.       /* transitive verb */


% Facts

    pn(mary).
    pn(henry).

    rpn(that).
    rpn(which).
    rpn(who).

    iv(runs).
    iv(sits).

    d(a).
    d(the).
    n(book).

    n(girl).
    n(boy).

    tv(gives).
    tv(reads).
```

5.4.2 Attributes

The mechanism for expanding nonterminals in the grammar into Prolog goals extends to terms with parameters in general. Thus, if the DCG grammar was changed to

```
s(k(U,V)) --> np(U),vp(V).

np(pling) --> proper_noun.

vp(plong) --> verb,proper_noun.

proper_noun --> txt(john).
proper_noun --> txt(mary).

verb --> txt(loves).
```

this would be expanded to

```
s(k(U,V),X0,X2)    :- np(U,X0,X1),vp(V,X1,X2).
np(pling,X0,X1)    :- proper_noun(X0,X1).
vp(plong,X0,X2)    :- verb(X0,X1),proper_noun(X1,X2)
```

```
    proper_noun(X0,X1)  :-txt(john,X0,X1).
    proper_noun(X0,X1)  :-txt(mary,X0,X1).
    verb(X0,X1)  :-txt(loves,X0,X1).
```

(Note that the variables X0,X1,X2 are not attribute variables).

Attribute unification with DCG

One last example of DCG, with attribute unification in curly brackets is listed below to show the principle, but it is stressed that this example is strongly equivalent to an example below (in Generating Syntax trees).

```
s(SyntS) --> np(NP),vp(VP),{SyntS=s(NP,VP)}.

np(SyntN) --> proper_noun(N),{SyntN=np(N)}.

vp(SyntVP) --> verb(V),np(NP),{SyntVP=vp(V,NP)}.

proper_noun(SyntN) --> [john],{SyntN=proper_noun(john)}.
proper_noun(SyntN) --> [mary],{SyntN=proper_noun(mary)}.
```

5.5 Generating Syntax Trees

The attributes of clauses can be made to make syntax trees. Let us suppose we want to make a syntax tree of the sentence

John loves Mary

```
s(s(NP,VP)) --> np(NP),vp(VP).

np(np(N)) --> proper_noun(N).

vp(vp(V,NP)) --> verb(V),np(NP).

proper_noun(proper_noun(john)) --> [john].
proper_noun(proper_noun(mary)) --> [mary].

verb(loves) --> [loves].

?- s(SyntS,[john,loves,mary],[]).


SyntS = s(np(proper_noun(john),vp(verb(loves),np(proper_noun(mary)))))
```

which can be rendered as a proper parse tree.

```
  s
    np
      proper_noun
        john
```

```
vp
  verb
    loves
  np
    proper_noun
      mary
```

5.6 Attribute Grammars

The arguments in the nonterminals of the DCG grammar follows the translation to Prolog. The unification mechanism of Prolog applies also for the codes that are expanded by Prolog.

Such arguments can then be used in a manner that resembles and supercedes conventional translation schemes and attribute grammars as known in compiler techniques. In natural language analysis, attributes are used for at least 3 different purposes:

- to parametrize the syntax analysis for agreement checks
- to create a parse tre or a syntaxtree
- to create the result in terms of a semantic structures

Let us make an example, where we implement an agreement check.

Agreement in English is concerned with at least 4 features:

- Case:

  Nominative: he,she
  Accusative: him,her
  Genitive: his,her

- Number:

  Singular: A man
  Plural: They,Two men

- Subcategorization of verbs
  - Transitive verbs take an object, like kiss,kill

    John kissed Mary.

  - Intransitive verbs like sing and die, take no objects.

    John sings, Mary dies.

  - Ditransitive verbs take two objects

    John gave Mary an apple

  - Reporting verbs take sentential clauses

    John said that the bus was late

- Reporting modal verbs (?) take infinitive complements

  John wanted to take the bus.

- Semantic agreement

  Sentences like

  ```
  The tail wagged the dog
  ```

  is considered illegal even if the sentence is grammatically correct. It is not possible for any member of class `tail` to wag any member of class `dog`. We need a check on the semantics of the noun phrases, and how they relate to the allowed arguments (and subjects) of the verb.

These features have to agree in correct English. For example, the starred (*) sentences are not correct

```
John sings.
They die.
They dies.      *
They kill Mary.
He met.         *
They kiss they. *
They die them.  *
Her loves John  *
Yellow nations pass midnight *
```

Some verb forms must agree in number with the subject, (but never with the object). The subject of a phrase is usually in nominative case, while the object is in accusative case.

Furthermore, in English, only pronouns are modified in the accusative case:

he - him , I - me.

In order to keep a record of these features, we let there be parameters to the categories for case, number and subcategory. (We shall leave semantic agreement check for a later treatment).

```
s --> np(Num, nom ),vp(Num).   % nom nominative

np(sin,_)     --> proper_noun.
np(Num,Case) --> pronoun(Num,Case).

vp(Num) --> verb(intrans,Num).
vp(Num) --> verb(trans,Num),np(_,acc). % acc accusative

proper_noun --> [john].
proper_noun --> [mary].

pronoun(sin,nom) --> [he].
pronoun(sin,acc) --> [him].

pronoun(plu,nom) --> [they].
pronoun(plu,acc) --> [them].
```

```
pronoun(sin,nom) --> [she].
pronoun(sin,acc) --> [her].

verb(intrans,sin) --> [dies].
verb(intrans,plu) --> [die].

verb(trans,sin) --> [kisses].
verb(trans,plu) --> [love].
```

Note especially that the Num in the clause

```
vp(Num) --> verb(trans,Num),np(_,acc).
```

only concerns the number of the subject, and not of the object.

Agreement Is Not Always Critical

Much effort can be put into making a correct analysis of agreements in the text, but when it comes to natural language understanding with a servants attitude, the best way of handling agreement discrepancies is to ignore them, as long as they don't cause ambiguities. Therefore, agreement checks do not play a critical role in natural language understanding.

In some rare occasions, number agreement is necessary to avoid ambiguities, for example

```
List the only Frenchman among the programmers who
understand English.

List the only Frenchman among the programmers who
understands English.
```

A correct analysis of this is left as an exercise.

Also concerning verb subcategorisation, many transitive verbs also allow a usage that lacks the required object. In these cases, the object is assumed to be "something".
Examples:

```
he served a meal / he served

she sang a song / she sang
```

5.7 Selectional Restrictions

When we introduce verbs into the language, we also need to check that the verbs fit together with the nouns. This is called "selectional restrictions" or "semantic restrictions" .
Semantic restrictions are important for two reasons:

- We don't want to allow meaningless questions to be answered, even if they are grammatical Example:

  ```
  Which yellow nations pass midnight on the city.
  ```

  The empty answer `None` would logically be a correct answer, but that is not satisfactory.

- We need to find the right interpretation among several. This is called *disambiguation*. Examples:

  1 The man saw the dog in the park with a telescope

  2 The dog saw the man in the park with a telescope.

It is obvious to us that the phrase "with a telescope" is a verb modifier in (1) but a "noun modifier" in (2). We can say this, because dogs do not have telescopes, and dogs can not see with a telescope. Such information is to be stored in a semantic net together with the class and attribute hierarchy. In TUC, these declarations are stored in a tabular form as shown by some examples. We need the following new predicates:

```
ako(Class,SuperClass)

iv_templ(Intranstive Verb,Subject Class)

tv_templ(Transtive Verb,Subject Class, Object Class)

v_compl(Verb, Subject Class,Preposition,Prepositional Object Class)
```

It is important that if a verb or complement is allowed in connection with a class C, then it is also allowed for all (inclusive) subclasses S of C. We therefore need to generalise the templates listed above. Some of the semantic net access relations are defined in a chapter 4.4 (Semantic Nets).

A small example of a grammar with semantic agreement check can now be devised 5.7 on page 73. The crux of the agreement check is to add a class attribute C to the noun phrases, and use this in tests with other parameters.

```
%% Class defintions

    ako(object,thing).
    ako(animate,thing).
    ako(place,thing).
    ako(person,animate).
    ako(animal,animate).
    ako(park,place).
    ako(man,person).
    ako(dog,animal).
    ako(telescope,object).

%% Semantic Verb templates

         % an animate can sit
    iv_templ(sit,animate).

         % both persons and dogs see things
    tv_templ(see,animate,thing).

         % but only persons can see with a telescope
    v_compl(see,person,with,telescope).

         % animates can see in a place
    v_compl(see,animate,in,place).

         % animates can sit in a place
    v_compl(sit,animate,in,place).

         %  only persons can have telescope
    v_compl(be,person,with,telescope).

         % things are in places
    v_compl(be,thing,in,place).
```

Figure 5.2: Predicates for semantic agreement check

The following examples show that a parser now rejects unsemantic sentences, and selects the correct of several possible parses. The parser metaparse is explained in the next chapter, and in the appendix Semantic Agreement Check.

Semantic Agreement Examples

```
?-metaparse(sentence,
     [The,dog, saw, the, dog, in ,the, park, with, a ,telescope]).

  ==> no

% This has no semantically meaningful parse.
%  because we have not allowed a park to be with a telescope
```

```
%% Template axioms

   iv_template(Verb,S):-
        iv_templ(Verb,C),
         subclass0(S,C).

   tv_template(Verb,S1,S2):-
         tv_templ(Verb,C1,C2),
         subclass0(S1,C1),
         subclass0(S2,C2).

   v_complement(Verb,S1,Prep,S2):-
        v_compl(Verb,C1,Prep,C2),
        subclass0(S1,C1),
        subclass0(S2,C2).
```

Figure 5.3: Predicates for semantic compliance

```
%----------------

?-metaparse(sentence,
     [The,man, saw, the, dog, in ,the, park, with, a ,telescope]).

%% There are actually two interpretations:

%% The dog is in the park

%% The man's seeing was in the park

  ==> yes

%--------------------------------

?-metaparse(sentence,
    [The,dog, saw, the, man, in ,the, park, with, a ,telescope]).

%% There is one interpretation, park and telescope belongs to the man

  ==> yes
```

5.8 Text Scanner in Prolog

In the examples above, we have represented the texts as explicit lists of atoms. This is of course unpractical in use. For that purpose, we have included a program called 'readin' that contains a scanner. The purpose of the scanner is to convert lines of texts to lists of atomic symbols. The listing of the scanner is listed in the appendix, and is available at the file repository. The details of the listing is not important however.

Workings of the scanner:

```
sentence         --> noun_phrase(C),
                      verb_phrase(C).

noun_phrase(C)  --> det,noun(C),
                      noun_complements(C).

verb_phrase(C)  --> intransverb(V),
                      {iv_template(V,C)},
                      verb_complements(V,C).

verb_phrase(C1) --> transverb(V),
                      noun_phrase(C2),
                      {tv_template(V,C1,C2)},
                      verb_complements(V,C1).

verb_complements(V,C) --> complements(V,C).
noun_complements(C)   --> complements(be,C).

complements(V,C) --> pp(C,V),
                       complements(V,C).
complements(_,_) --> [].

pp(C1,V)          --> prep(P),
                       noun_phrase(C2),
                       {v_complement(V,C1,P,C2)}.

transverb(see)   --> [saw].
intransverb(sit) --> [sits].

noun(animate)   --> [animate].
noun(person)    --> [person].
noun(man)       --> [man].
noun(dog)       --> [dog].
noun(telescope) --> [telescope].
noun(park)      --> [park].

det --> [a].
det --> [the].

prep(in) --> [in].
prep(with) -->  [with].
```

Figure 5.4: A grammar with semantic agreement check

The scanner has two entry points: scanner1 and scanner2.

scanner1(Atomlist) will give a prompt (E:) and then accept one line of text. At the end of line, the scanner will be activated and return an atom list.

scanner2(Atomlist) is similar, but may accept text over several lines. First when a line is ended with a terminator character ('.', '!' or '?'), the scanner will return the atom list.

Prolog programs using this scanner will work as follows. In stead of of the predicate s(SyntS), we could have a modified predicate

```
    parseS(SyntS) :-
        scanner1(List),
        s(SyntS,List,[]).
```

This is invoked as

```
  ?- parseS(SyntS).

  E: john loves mary
```

```
SyntS = s(np(proper_noun(john),vp(verb(loves),np(proper_noun(mary)))))
```

as before.

5.9 EXERCISES

```
1.   Make a Prolog  program to understand the following type of
     dialogue:

           All boys are something.
           Some boys are good.
           Some boys are bad.
           Magnus is a boy.
           Magnus is good.
           Is Magnus good ?
               yes
           Who is good ?
               Magnus



2. Make a Prolog program that will recognize
   bus station names from a list of names, with the same competence
   as yourself regarding background information,
   misspelling, truncation and other distortions.


3. Use the agreement check attribute parser to
   make  correct analyses of the following

      List the only Frenchman among the programmer(s) who
      understand(s) English !


4. Make an attribute grammar to capture the sentence:

     Which country  bordering  the Mediterranean
     borders a country that is bordered by a country
     whose population exceeds the population of India.
```

5. Make a natural language system to parse rules stated in Naturally Readable Logic, and translate them to Prolog.

Example:

```
a man is the grandfather of a person if
    this man is the father of another person and
    this other person is the parent of the person.

a man is the parent of a person if
     this man is the father of the person.

a woman is the parent of a person if
     this woman is the mother of the person.

Olav is the father of Harald.
Harald is the father of Haakon
Harald is the father of Martha
```

Chapter 6

Advanced NLP in Prolog

6.1 Metaparser in Prolog

It is sometimes convenient to make a parser that handles the parse trees and other information as a side effect, i.e. it is not necessary to explicitly have the syntax tree in the grammar. This motivates the concept of a *Metaparser*, where the grammar is represented as a set of Prolog facts, and where a separate metaparser program is interpreting this grammar dynamically. (Not as in DCG, where Prolog translates it into executable Prolog). The price is a layer of inefficiency, but for demonstrative purposes, it is an ideal tool. (A compiler that translates the extended grammar into Prolog similar to DCG is of course possible).

We shall describe the principles using only a simple DCG Metainterpreter, but shall later use the same principle to implement more general language interpreters:

- Parsers for extended language formalisms (Categorial Grammars)
- Parsers that uses other strategies (Bottom up parsers, Chart Parsers)

A grammar that is represented in this way is called an M-grammar (M for meta, but metagrammar means something different).

To distinguish these grammars from DCG grammars, we use another production symbol

`''--->''` which has to be declared by

```
:-op(1100,xfy,--->)
```

The grammar above is then stated as an M-grammar as

```
s ---> np,vp.

np ---> proper_noun.

vp ---> verb,np.

verb ---> txt(loves).

proper_noun ---> [john].
proper_noun ---> [mary].
```

The metaparser can now be formulated

```
:-op(1100,xfy,--->).


metaparse(S,List):-
    parse(S,PSTree,List,[]),
    prettyprint(PSTree).


parse(S,prod(S,PSTree),X0,X1):-
    (S ---> PS),
    parse(PS,PSTree,X0,X1).

parse((First,Rest),(FirstTree,RestTree),X0,X2):-
   parse(First,FirstTree,X0,X1),
   parse(Rest,RestTree,X1,X2).

parse([Word],[Word],[Word|Rest],Rest).
```

(The prettyprint will be shown later.)
We get a formatted parse tree by the call

```
?-metaparse(s,[john,loves,mary]).

  ==>

s
  np
    proper_noun
      [john]
  vp
    verb
      [loves]
    np
      proper_noun
        [mary]
```

Metagrammar in DCG

It is an amusing consequence that the above parser can be formulated as a DCG grammar. In a way, we have implemented DCG in DCG, because Prolog translates this metagrammar exactly to the parser program above. A grammar to describe a grammar formalism is called a *metagrammar*.

```
parse((First,Rest),(FirstTree,RestTree)) -->
   parse(First,FirstTree),
   parse(Rest,RestTree).

parse(S,prod(S,PSTree)) -->
    {S ---> PS},
    parse(PS,PSTree).

parse([Word],[Word]) --> [Word].
```

6.2 Parsing with Ambiguities

As another amusing example, consider the sentence

```
Time flies like an arrow .
```

This innocuous little example is sometimes used to show the futility of trying to analyse natural language sentences, because it has so many interpretations:

- Time passes very quickly
- The time is flying with the same speed as an arrow.
- The time is flying in a way similar to arrows.
- Some time-related flies are fond of arrows. (as fruit flies like a banana)
- Some flies called "Time" flies are fond of arrows.
- Take the time of flies quickly !
- Take the time of flies in a way similar to how an arrow would do it!
- Take the time of flies with arrow form !
- Take the time of flies in the same way as you would take the time of an arrow !
- etc.

A grammar for this may not necessarily be unambiguous:
Consider the grammar, which we represent as a M-grammar.

```
sentence       ---> noun_phrase,verb_phrase.
sentence       ---> verb,noun_phrase,comparison.
noun_phrase    ---> noun.
noun_phrase    ---> proper_noun.
noun_phrase    ---> article,noun.
noun_phrase    ---> composite_noun.
verb_phrase    ---> verb,object.
verb_phrase    ---> verb,comparison.
object         ---> noun_phrase.
comparison     ---> comparator,noun_phrase.
composite_noun ---> proper_noun,noun.

verb ---> [time].
verb ---> [flies].
verb ---> [like].
comparator ---> [like].
noun ---> [time].
noun ---> [flies].
noun ---> [arrow].
proper_noun ---> [time].
article ---> [an].
```

```
?-metaparse(sentence,[time,flies,like,an,arrow]),fail.
```

```
sentence
  noun_phrase
    noun
      [time]
  verb_phrase
    verb
      [flies]
    comparison
      comparator
        [like]
      noun_phrase
        article
          [an]
        noun
          [arrow]
```

```
sentence
  noun_phrase
    proper_noun
      [time]
  verb_phrase
    verb
      [flies]
    comparison
      comparator
        [like]
      noun_phrase
        article
          [an]
        noun
          [arrow]
```

```
sentence
  noun_phrase
    composite_noun
      proper_noun
        [time]
      noun
        [flies]
  verb_phrase
    verb
      [like]
    object
      noun_phrase
        article
          [an]
        noun
          [arrow]
```

```
sentence
  verb
    [time]
  noun_phrase
    noun
      [flies]
  comparison
    comparator
      [like]
    noun_phrase
      article
        [an]
      noun
        [arrow]
```

While ambiguity in programming languages is considered an unacceptable fault of the language itself, we have to live with it in natural language, because it is inherently ambiguous at all levels. A later semantic analysis will know the ambiguity of ”like”, and can check in a semantic net what attributes are meaningful, i.e

```
Time can fly   (in a certain sense)

A fly has no attribute which can  have a value Time.

It is not relevant to take the time of flies.

etc.
```

It is important to find the right sense of the word. This is often possible by taking into account the grammar, and also the semantic classes of the nouns involved. In general, finding the right sense also gives the right semantic code, i.e. we can safely forward the problem to the pragmatic processing.

6.3 Parsing with Probabilistic Grammars

One approach to handling ambiguities is to calculate the various probabilities for a given parse tree. Some parse trees can be eliminated if they can be proven to be less probable that others. For instance, the interpretation of Time as a proper noun in Time flies (viz. 'Time' magazine) is less probable that the interpretation as a noun.

Probablistic parsing has its proponents, but we could also cite N. Chomsky to the contrary. [1]

PCFG can be implemented easily in a M-grammar approach, extending the metagrammar rules with one extra parameter prob.

```
sentence ---> noun_phrase,verb_phrase     prob 0.8 .
sentence ---> verb,noun_phrase,comparison prob 0.2 .
```

This says that given that the phrase structure is a sentence, there are the following probabilities for the various phrase structures:

```
P(noun_phrase,verb_phrase|sentence)     =  0.8
P(verb,noun_phrase,comparison|sentence) =  0.2
```

It can be proved that when comparing the various complete parse trees, an unnormalized probability of a parse is achieved by multiplying all the applied probabilities of the (leaf) nodes in the parse trees.

In the grammar below, we have added some fictitious probabilites. We note that the sum of probabilities for each left side is 1. (In order to stress the point that "Time" as a proper noun is improbable, we have added a proper noun John with relative higher probability).

In practice, the probabilities must be collected by statistical analyses of annotated text databanks ("Treebanks"), where verified correct parses have been added.

In order to achieve this, the metaparser has been extended, and run on the examples above. We show the first run, which gives a probability of 1.15E-03 . The following parses have probabilities 3.64E-04 , 6.72E-05 , 9.60E-05 respectively. In the syntax tree, the probability of a node represent the product of the probabilities of its daughter nodes.

```
sentence    PROB 1.15E-03
  noun_phrase    PROB 1.00E-01
    noun    PROB 5.00E-01
      [time]
  verb_phrase    PROB 1.44E-02
    verb    PROB 4.00E-01
      [flies]
    comparison    PROB 1.20E-01
      comparator    PROB 1.00E+00
        [like]
      noun_phrase    PROB 1.20E-01
        article    PROB 1.00E+00
          [an]
        noun    PROB 3.00E-01
          [arrow]
```

A listing of the probabilistic metaparses can be found in the appendix `Parsing with Probablistic Context Free Grammars.`

Needless to say, in many cases, disambiguation without a semantic analysis notoriously gives spurious results. For example, for the sentence `John saw the dog with binoculars`, it is irrelevant that noun modifiers (dog with binoculars) are in general more frequent than verb

[1]But is must be recognized that the notion "probability of a sentence" is an entirely useless one, under any known interpretation of this term. (Noam Chomsky ,1969).

```
sentence ---> noun_phrase,verb_phrase     prob 0.8 .
sentence ---> verb,noun_phrase,comparison prob 0.2 .

noun_phrase    ---> noun                 prob 0.2 .
noun_phrase    ---> proper_noun          prob 0.3 .
noun_phrase    ---> article,noun         prob 0.4 .
noun_phrase    ---> composite_noun       prob 0.1 .

verb_phrase    ---> verb,object          prob 0.7 .
verb_phrase    ---> verb,comparison      prob 0.3 .

object         ---> noun_phrase          prob 1.0 .

comparison  ---> comparator,noun_phrase prob 1.0 .

composite_noun ---> proper_noun,noun    prob 1.0 .

verb ---> [time]   prob 0.1 .
verb ---> [flies]  prob 0.4 .
verb ---> [like]   prob 0.5 .

comparator ---> [like]    prob 1.0 .

noun ---> [time]          prob 0.5 .
noun ---> [flies]         prob 0.2 .
noun ---> [arrow]         prob 0.3 .

proper_noun ---> [time]   prob 0.1 .
proper_noun ---> [john]   prob 0.9 .

article ---> [an]         prob 1.0 .
```

Figure 6.1: A Probabilistic grammar

complements (saw with binoculars). On the other hand, together with a full (syntactic and) semantic analysis, the use of statistical analysis may give parses that selects the most probable among several alternatives.

6.4 Feature Structures

We have seen examples of attributes in the grammars that allow for a number of purposes:

- agreement checks for number, gender, case.
- agreement check for verb subcategorisation
- semantic agreement checks

We used unifiaction of attributes, and this was formulated as explicit unification .

It is convenient for many purposes to assemble these attributes or features into structures called **Feature structures**. A feature structure is a structure that can hold a number of attributes at a time, and make groups of features that belong together.

One such scheme, which is conformant with Prolog list structure is the following, using a relevant example. (We have added a spurious feature `case:X` for demonstration purposes).

```
[ orth:sing,
  cat:verb,
  head: [subcat:intrans,
         agreement:
              [number:pl,
               case:X]    % spurious
        ]
]


[ orth:they,
  cat:pronoun
  head: [agreement:
              [number:pl,
               case:nominative]
        ]
]
```

Here we say that the word (with ORTHography) 'sing' is a a word of CATegory 'verb'. It has several attributes that we group under a composite feature HEAD, saying that the verb is of SUBCATegory intransitive verb, and it has a composite feature with number = PLural. We have placed a feature for case under agreement which we have left with a variable X. The other feature gives a similar description about the pronoun 'they'.

We could then perform an agreement check on the sentences

```
they sing
him sing
```

by comparing the subfeature structures of 'they' ,'him' and 'sing' respectively

```
they    agreement:[number:pl,
                   case:nominative]

him     agreement:[number:sg,
                   case:accusative]

sing    agreement:[number:pl, case:X]
```

We see that we can perform an agreement check by merely performing a unification test

```
agreement:[number:pl, case:nominative]  % they
  =
agreement:[number:pl, case:X]           % sing
```

The resulting most general common instance is

```
agreement:[number:pl, case:nominative]
```

where `X = nominative`.

This works as long as we have exactly the same positions of each feature in the list, and using free variables wherever the test is not critical. However, this is impractical for bigger grammars, so we introduce a concept called *Feature unification* which allows us to relax on this strict requirement. We then no longer need to have an irrelevant attribute case in the feature structure of `sing`.

With feature unification, we can allow the structures to be unified (we use a === operator for this kind of unification):

```
agreement:[number:pl, case:nominative]

===

agreement:[number:pl]
```

The resulting most general common instance is

```
agreement:[number:pl, case:nominative]
```

In general, feature unification allows an attribute value pair to be added to a list if the attribute is not already occuring there.

An attempt to parse

'he sing'

would fail because

```
agreement:[number:sg, case:nominative]  % he
```

cannot be unified with

```
agreement:[number:pl] % sing
```

because the subfeature `number` have different values.

Another example concerning subcategorisation:

```
They  serve breakfast
```

In order that the sentence shall be checked, we must have that the case of the subject `they` must be nominative, the subcat of the verb `serve` must be transitive, and that the agreement of the subject and the verb must be unifiable. Furthermore, the object must be in the accusative case, wherever relevant.

For that matter, breakfast has the same spelling in nominative and accusative, however it becomes important for pronouns:

```
They serve they          NO
They serve him           OK
```

Feature agreement that fails

```
[ orth:serve,
  cat:verb,
  head: [subcat:trans,
         agreement:
              [number:pl]
        ]
]

[orth:breakfast,
 cat:noun,
 head:[agreement:
        [number:sg]
      ]
].
```

If FSubj and FVerb and FObj are the feature structures of the subject, the verb and the Object respectively, the above constraint can be expressed as follows.

```
FSubj :: head:agreement:case   === nominative,

FVerb :: head:agreement:subcat === trans,

FSubj :: head:agreement:number === FVerb :: agreement:number

FObj :: head:agreement:case === accusative.
```

This scheme would then disallow all the wrong sentences like

```
they serve  *

him serve breakfast  *

they serve they  *
```

6.4.1 Feature Structures in the Grammar

It is relatively easy to incorporate feature structures and unification in Definite Clause Grammar as an extension. The main grammatical structure remains the same, but we add feature unification as explicit extra conditions. We use the following conventions:

```
Generic grammar rule

     g0(G0) --> g1(G1),... gn(Gn),
     {Gi :: f1:f2:...:fk === Fval}, ...


  gi       Syntactic category as in DCG

  Gi       Feature structure of g

  fi       feature

  Gi :: f1:f2:...:fk     Feature expression

  f1:f2:...:fk     Feature path

  ===      Feature unification symbol

  Fval     Feature value     Atom or Feature expression
```

We shall show a small fragment of a grammar using this notation. A more complete grammar will be found in the appendix Feature and Unification. Here, also some technical details of implementations will be explained.

```
% Does this flight serve breakfast
```

```
s(S) --> aux(Aux),np(NP),vp(VP),
    {S :: cat  === s},
    {S :: head === VP ::head},
    {Aux :: head:agreement === NP :: head:agreement},
    {VP :: head:agreement:number === pl}.

% This flight serves breakfast

s(S) --> np(NP),vp(VP),
    {S :: cat  === s},
    {S :: head === VP :: head},
    {NP :: head:agreement === VP :: head:agreement}.

% This flight

np(NP) --> det(Det),nominal(Nominal),
    {NP ::  cat  === np},
    {NP ::  head === Nominal :: head},
    {Det :: head:agreement === Nominal :: head:agreement}.

% breakfast

np(NP) --> nominal(Nominal),
    {NP :: head === Nominal :: head}.


aux(Aux) --> [do],
    {Aux :: head:agreement:number === pl},
    {Aux :: head:agreement:person === 3}.

aux(Aux) --> [does],
    {Aux :: head:agreement:number === sg},
    {Aux :: head:agreement:person === 3}.

det(Det) --> [this],
    {Det :: head:agreement:number === sg}.

det(Det) --> [these],
    {Det :: head:agreement:number === pl}.

vp(VP) --> verb(Verb),
    {VP :: head === Verb :: head},
    {VP :: head:subcat === intrans}.

vp(VP) --> verb(Verb),np(_NP),
    {VP :: head === Verb :: head},
    {VP :: head:subcat === trans}.


%% Lexicon part

verb(Verb) --> [serve],
    {Verb :: head:agreement:number === pl},
    {Verb :: head:subcat           === trans},
```

```
    {Verb :: head:subcat:first:cat === np},
    {Verb :: head:subcat:second    === end}.

verb(Verb) --> [serves],
    {Verb :: head:agreement:number === sg},
    {Verb :: head:subcat           === trans},
    {Verb :: head:subcat:first:cat === np},
    {Verb :: head:subcat:second    === end}.

verb(Verb) --> [exists],
    {Verb :: head:agreement:number === sg},
    {Verb :: head:subcat           === intrans},
    {Verb :: head:subcat:first     === end}.

noun(Noun) --> [flight],
    {Noun :: head:agreement:number === sg}.

noun(Noun) --> [flights],
    {Noun :: head:agreement:number === pl}.

noun(Noun) --> [breakfast],
    {Noun :: head:agreement:number === sg}.

nominal(Nominal) --> noun(Noun),
    {Nominal :: head === Noun :: head}.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

One run example:

(The feature structure S is prettyprinted by the parser program after parsing.)

```
E: Does this flight serve breakfast


[
   cat:
      s
   head:
      [
         agreement:
            number:
               pl
         subcat:
            [
               trans
               first:
                  cat:
                     np
               second:
                  end
```

```
                ]
          ]
    ]
```

The topic of features and unifiaction deserves its own chapter (or even book), but we are not going into unification as our main method of implementation.

We refer to the the Appendix **Feature unifiaction in Prolog**.

Other Unification Based Formalisms

Unification is the central theme also in other grammar mechanisms (LFG, FUG, PATR-II HPSG etc). We shall not go into these formalisms in any detail, because we will be content with the unification and attribute mechanisms in Prolog for several reasons:

Firstly because they are translated to Prolog, and run directly as Prolog code,and extremely efficient Prolog compilers exist that analyse several million grammar reductions pr second.

Secondly because of the tight relationship between DCG and Prolog, DCG grammars can be easily integrated with programs written in Prolog.

Consequently, integrating a DCG natural-language system with other programs, say a database or expert system can be straightforward. Finally, the human-engineering problems associated with the linear order and arity requirements of term unification is not terribly burdensome for small to medium-sized grammars. Thus for the rapid development of simple and efficient natural language systems, DCG can be the formalism of choice.

Often, a criticism of DCG is made that because the standard form of compilation into Prolog, DCG cannot handle left recursive rules. This is only partly true. The formalism itself can of course state left-recursive rules with no difficulty. The same formalism can actually be interpreted by a bottom up parser, or be automatically translated to Prolog where the left recursive rules are eliminated.

On the other hand, there are standard crafting techniques to engineer left recursive grammar rules into right recursive ones. The majority of the rules are often naturally stated as right recursive rules, and the remaining cases of left recursion is usually solved without problems.

6.5 Parsing Strategies

TUC's parser is implemented in Prolog, and relies on backtracking, which means that ambiguities of any kinds are not really harmful, because the parser will try all alternatives after another.

The first parse that is accepted both as syntactically correct with all agreement checks including semantic agreement checks will be accepted. This is called a greedy heuristics, and relies on a very important principle of *longest first preference*.

This is a heuristic principle that the parser, in principle should try the longest phrase first. That is because longer phrases means more context, and therefore gives more information before determining the meaning.

This principle puts much responsibility on the grammar writer, to write it according to these guidelines. For instance, many verbs are both transitive and intransitive (e.g. sing). Therefore, the rules for transitive verbs should appear before intranstive, so that `sing a song` is tried before `sing in the church`.

However, it must be admitted that this is not always practical to achieve, because one cannot always determine how long a phrase will be.

This is a rough method, and it sometimes leads to misunderstandings. In defence, it can be said that a good grammar based on these guidelines makes *plausible misunderstandings*, i.e. misunderstandings that humans also are likely to make. There is however possibilities for a good dialogue handler to indicate in the answers how the questions were understood, so misunderstandings are avoided.

As a final admittance, the overwhelming number of alternative interpretations sometimes makes it necessary to perform committed choices during the parsing. (Technically, this is done by cuts in Prolog). This means that some sentences, typically Garden path sentences, stay the chance of not being understood at all. However, the strategy of longest first preference makes the committed choices less hazardous than otherwise.

6.6 Consensical Grammar

TUC's grammar formalism is called *Consensical Grammar* which means

```
Context Sensitive Categorial Attribute Logic Grammar
```

and is a reminder of all the extensions of context free grammars that are needed to cope with natural language parsing. They are

- Context sensitive

  The grammar formalism is in effect an extension of context free grammars, and implements at least a proper subset of context sensitive grammars, as will be shown below

- Categorial

  Categorial grammar is a name of a grammar type that uses operators to manipulate the combination of subphrases into larger phrases. Consensical grammar borrows some of its notation from classical categorial grammar.

- Attribute

  The classical context sensitive grammars use names of syntactic categories without arguments. However, this grammar allows attributes and attribute variables to be embedded in the grammar besides translation schemes in the form of extra condtions on the attributes.

- Logic

  The grammar is based on Definite Clause Grammars, and inherits all the logic machinery that is imbedded in Prolog. However, the grammar formalism as such is not dependent on a top down parsing method.

Consensical grammar relies heavily on the principles of the Extraposition grammars [**?**] which was used to implement the CHAT-80 systems.

The examples below show that this categorial grammar has the generating capacity that goes beyond context free grammars.

6.6.1 Language Hierarchy

The languages, both formal and natural, can be classified into a hierarchy that was devised by Chomsky. The hierarchy sets up the relation between the languages defined as sets of strings, the grammar formalism and the kind of automata that are able to accept these strings, and no others.

Regular Languages And Grammars

The simplest languages are regular languages. Regular languages form the model for low level morphological or lexical analysis, both in programming and natural languages.

(Regular language are accepted by Finite State Automata). Regular grammars can be made into a special case of context free languages in that all productions are of the form

```
S1  --> t
S1  --> t S2
```

where `t` denotes a terminal symbol, S1,S2 are nonterminals, and all the nonterminals appear at the end of the right sides.

An example of a regular language is the set of strings `a* b*` which means means none or more repetions of a and b. We could say that this repesents the set of strings `a^m  b^n` for various unrestricted integers m and n.

```
S --> A B
A --> a A
A -->
B --> b B
B -->
```

There is a limitation in regular languages, that is due the lack of memory in FSA. For example the following language is not regular:

The set of strings `a^n b^n` for any n,
which means a repetion of n a's followed by an equal number of b's.

(We can say that regular grammars cannot count).

Context Free Languages and Grammars

Context free languages have been introduced already. They are defined by productions that contains single left side nonterminal symbols, and unrestricted right sides.
(They are accepted by (one-stack) Pushdown atomata (PDA)). Context free languages are larger than regular languages. For example, the non regular language above is easily defined by the grammar

```
S --> a S b
S -->
```

(We can say that context free languages can count, but only two things.)

Context Sensitive Languages and Grammars

Context sensitive grammars allow the left hand sides to contain more that one symbol, so that a production is only allowed when all the left side symbols match with the text.
(Context senstive languages can be accepted by a two-stack Pushdown automaton).

```
X A --> X a
X   -->
```

In the classical definition of context sensitive grammar, the surrounding pharses are not allowed to change in the production.

'A' can produce 'a', but only allowed in the context of X, and X shall be unchanged.

We shall be a somewhat liberal with this restriction in our treatise.

An example of a language that is not context free is

```
a^n b^n c^n
```

where all the n's denote the same number of repetitions However, this language is context sensitive. (We can say that context sensitive languages can count more than two things.)

```
S   --> S1

S1   -->   X S1 S2
S1   -->

S2   -->   A b S2 c
S2   -->

X A   -->    a
```

The latter grammar can be reformulated into categorial grammar as follows. We note that the purpose of X is to push an A on a stack (waiting list). This is expessed as follows, using the extraposition operator.
Instead of

```
X A --> a
```

we write

```
X --> []-A a
```

In Prolog form, the grammar is

```
s  ---> s1.

s1 ---> x,s1,s2.
s1 ---> [].

s2 ---> a1,[b],s2,[c].
s2 ---> [].

x  ---> []-a1,[a].
```

The run examples indicate that grammar copes with the counting.

```
E: a a b b c c

s
   s1
      x
         []-a1
         [a]
      s1
         x
            []-a1
```

```
                [a]
             s1
                []
             s2
                a1-a1
                   []
                [b]
                s2
                   a1-a1
                      []
                   [b]
                   s2
                      []
                   [c]
                [c]
          s2
             []

==> yes


E: a a b b c .

==> no
```

Consensical Grammar Applied to NL

In the following is an example of a grammar, a sentence and a parse tree in Consensical grammar.

```
sentence  ---> statement,['.'].
sentence  ---> question,['?'].

statement ---> np,vp.

question ---> [which], statement \ det.


np ---> det,noun,relclause.
np ---> pnoun.


vp ---> tv,np.
vp ---> iv.


relclause ---> [that],
                statement // np. %% NB

relclause ---> [].
```

```
begin ---> []-[lock].
end  --->  [lock].

% Lexicals

det ---> [a].
det ---> [the].
det ---> [every].

noun ---> [man].
noun ---> [woman].
noun ---> [dog].
noun ---> [cat].
noun ---> [mouse].
noun ---> [fish].

iv ---> [dies].
iv ---> [lives].
iv ---> [squeaks].

tv ---> [catches].
tv ---> [chases].
tv ---> [kills].
tv ---> [kisses].
tv ---> [likes].
tv ---> [loves].


pnoun ---> [fred].
pnoun ---> [john].
pnoun ---> [mary].
```

Sample sentences

```
  The cat that catches a mouse likes fish.

  The mouse that the cat chases dies.

  The mouse that the cat that likes a fish chases squeaks.

  The mouse that the cat that chases likes a fish squeaks. *

  Which mouse that a cat chases dies ?
```

```
E: the mouse that the cat that likes a fish chases  squeaks.


sentence
   statement
      np
```

```
         det
            [the]
         noun
            [mouse]
         relclause
            [that]
            statement-np
               np
                  det
                     [the]
                  noun
                     [cat]
                  relclause
                     [that]
                     statement-np
                        np-np        % the cat (gap)
                           []
                        vp
                           tv
                              [likes]
                           np
                              det
                                 [a]
                              noun
                                 [fish]
                              relclause
                                 []
               vp
                  tv
                     [chases]
                  np-np              % the mouse (gap)
                     []
      vp
         iv
            [squeaks]
   [.]

==> yes
```

The next example shows an example where a relative clause cannot be formed.

```
E: the mouse that the cat that chases likes a fish squeaks.

==> no
```

Restriction on Movements

Inward Application Move Restriction

There is a restriction on relative sentences that the nominal gap should be "used" inside the relative clause. If we relaxed this condition, e.g. by using the outwards applictation –

```
relclause ---> [that],
               statement - np.
relclause ---> [].
```

we would have to accept the following sentence:

```
The mouse that the cat that chases likes a fish squeaks.
```

(although with the same reasonable analysis as for the original sentence.). In order to avoid this, we used the Inwards application (//) instead.

```
relclause ---> [that],
               statement // np.
relclause ---> [].
```

The difference here is that the inward application // will not succeed unless the np is used by the same statement, while the – would allow the np to be stacked and used later by some productions.

Non Conjunction Move Restriction

We can easily extend the grammar above to allow more than one noun phrase instead of a simple one. We can do that by extending the grammar:

```
np ---> np1, andnps.

andnps ---> [and],np.
andnps ---> [].

np1 ---> det,noun,relclause.
np1 ---> pn.
```

This will allow the sentences

```
John and Fred lives.
```

From earlier, we can recognise a sentences

```
Fred loves Mary.

A man  that  lives loves Mary.
```

This is grammatical. It means that there is woman X, a man Y, Y lives and Y loves X.
Now consider the sentence.

```
A man that John and () lives loves Mary.
```

Here, the gap is placed inside the conjunction

```
John and ()
```

The logical meaning is that there is an X (e.g. Fred) so that John and X live, and X loves Mary. However, this construction is strictly forbidden in English.

Again, the grammar can be remedied, but we shall not give the explicit details since this over-acceptence seems to be rather harmless for language understanding.

6.6.2 Consensical Grammar Metaparser

The following program implements the elements of syntactic analysis of context sensitive categorial grammars. It is listed as an example of a *Metagrammar parser* which dynamically interprets the grammar rules, and perform a parse accordingly. The parser needs a lot of explanation, but even if you don't understand it now, it is an executable program that can be used for experimentation. In principle, it is the same parser as is used by TUC, although we have not incorporated agreement checks and code generation.
The program also appears in the appendix "Metaparser for phrase structured languages" together with a program 'readin' that scans input from user.

```
%% Categorial Grammar Meta Interpreter


:-op(1100,xfy,'--->'). %% Metagrammar Production
:-op(500,xfy,  \ ).    %% Backwards application
:-op(500,xfy,  / ).    %% Forwards application
:-op(400,yfx,  // ).   %% Inwards application
:-op(500,yfx,  - ).    %% Outwards application


% parse(PhraseStructure,Syntaxtree,GapStack1,GapStack2).


parse([X],[X],GS,GS)   --> [X], %% read word from list
    {\+ frontgap(GS)}.          %% unless front gap

parse([],[],GS,GS)     --> [].

parse((X,Y),(TX,TY),GS1,GS3) -->
    !,
    parse(X,TX,GS1,GS2),
    parse(Y,TY,GS2,GS3).


parse(X \ U,T-U,GS1,GS2)   -->
    parse(X,T,[gap(U,first)|GS1],GS2).

parse(X - U,T-U,GS1,GS2)   -->
    parse(X,T,[gap(U,free)|GS1],GS2).

parse(X // U,T-U,GS1,GS4)   -->
    {lock(GS1,GS2)},
    parse(X,T,[gap(U,free)|GS2],GS3),
    {unlock(GS3,GS4)}.

parse(X,prod(X,YT),GS1,GS2)  -->
    {X ---> Y},
    parse(Y,YT,GS1,GS2).

parse(X,prod(X-X,[]),[gap(X,_)|Y],Y)   --> [].
```

```
%% Runtime utility

lock(GS,[gap(lock,last)|GS]).

unlock([gap(lock,last)|GS],GS).

frontgap( [gap(_,first)|_]).
```

```
run :-
    scanner(Wordlist),
    syntax(Wordlist,Syntaxtree),
    prettyprint(Syntaxtree).


syntax(List,ST) :-
    parse(sentence,ST,[],[],List,[]).
    %% 2 extra arguments for compatibility with DCG expansion


prettyprint(ST):-
    nl,
    pretty(0,ST),
    nl.

pretty(N,prod(X,Y)-U):- !,
    tab(N),write(X-U),nl,N2 is N+3,
    pretty(N2,Y).

pretty(N,prod(X,Y)):- !,
    tab(N),write(X),nl,N2 is N+3,
    pretty(N2,Y).

pretty(N,(X,Y)):- !,
    pretty(N,X),
    pretty(N,Y).


pretty(N,ST):-tab(N),write(ST),nl.

:-compile(readin).
```

6.7 EXERCISES

The exercises here are advanced.

```
1. Make an analysis of the classical Context Sensitive Grammar
   with Consensical Grammar, and find out if they are equivalent.

2. Try to combine feature unification with consensical grammar,
   i.e. using feature unification as the main argument carrying
   mechanism.

3. Try to combine chart parsing with consensical grammar formalism.

4. Try to combine consensical grammar and feature unification and
   chart parsing.
```

Chapter 7

Computational Semantics

We have up to now discussed the grammar, syntax analysis, and we have studied a knowledge based system that can be accessed by a logical query language. We shall now start to make the transformation from the content in the parse, to the application. What we seek is the meaning or *semantics* of the sentences, expressed in a logical form. We can start with examples from the blocks world.

```
 ---
| A |
 ---
 ---
| B |
 ---
 ---
| C |
 ---
```

Let A, B and C be 3 blocks, of which it is known that

```
A is blue.

C is not blue

B is blue.
```

The above scenario can be described in logic as follows:

```
Block(A).
Block(B).
Block(C).

Blue(A).
Blue(B).
not  Blue(C).

On(A,B).
On(B,C).
```

Then it may be proved by logic that the following proposition holds:

Proposition:

Table 7.1: Elementary semantics

Proper nouns John	$\Longrightarrow$	constants 'John'
Common nouns John is a man	$\Longrightarrow$	unary predicates Man('John')
Adjective phrases dead	$\Longrightarrow$	unary predicates Dead('John')
Copular phrases Fido is a dog	$\Longrightarrow$	unary predicates Dog(Fido)
Intransitive verbs sing	$\Longrightarrow$	unary predicates Sing('John')
Transitive verbs John loves Mary	$\Longrightarrow$	binary predicates Loves('John','Mary')
Prepositional phrases B is on C	$\Longrightarrow$	binary predicates On(B,C)

```
There is a blue block that is on a block that is not blue
```

(Actually, the proposition holds regardless of B's colour, but we shall not delve into the logic puzzle, but concentrate on the natural language semantics of the matter.) A reasonable first order formulation of the proposition is then

```
Exists(x:y:  Block(x) and Block(y) and
             Blue(x) and not Blue(y) and
             On(x,y)).
```

The elementary semantics

We shall assign logical formulas to various phrases. In a simplified setting, we can choose the following mapping as in table 7.1.

Lambda abstraction

In formal logic, Dead(Socrates) means that Socrates is dead, and Man(Plato) means that Plato is a man, but how do we express simply "Dead" ? According to Church [**?**], the property of being dead can be expressed by the lambda calculus notation.

$$\lambda x.Dead(x)$$

which defines exactly what it is (A truth-valued function), and how it is applied (the argument is replacing x in Dead(x)) as in

$$\lambda x.Dead(x)(John) \Longrightarrow Dead(John)$$

Compositional Semantics

An important concept is *compositional semantics* whereby we regard the semantics of a sentence as the result of a combination of the semantics of the components of the sentence, very similar to how we define the semantics of programing language.

In order to make a semantic logical representation of a sentence, the grammar productions must be extended with attributes in the form of parameters that represent the semantic interpretetations of each component. For this purpose, we use lambda calculus for representing the

intermediate results, while the final result will be an expression in FOPL. We show a small example:

```
B is on C.
```

The final logical formula to express the meaning of the predication is

```
On(B,C)
```

so we therefore say that the semantics of “`B is on C`” is “`On(B,C)`”.

The NL phrase “`B is on C`” has a syntax composition

```
Sentence(SemS) --> Noun_Phrase(SemNP),Verb_Phrase(SemVP)
```

where SemS is the semantics of the sentence, and is supposed to be a combination of the semantics of the subphrases SemNP and SemVP. We can assign the semantics of the phrase `'B'`,

```
SemNP  =  'B'
```

Then we say that the semantics of SemVP is a predication, i.e. a Boolean function which is true when applied to the argument `'B'`. In other words, `SemS = SemVP(SemNP)`.

Here, we exceed the bounds of FOPL, and we therefore introduce an appropriate notation from the Lambda Calculus.

We let SemVP = $\lambda x.On(x,C)$

Then we get the semantics of SemS by

$SemVP(SemNP)$ i.e.

$\lambda x.On(x,C)(B) = On(B,C)$

according to the rules of the lambda calculus.
For various good reasons, we shall use a slightly different notation for the lambda expressions. We shall let $\lambda x.P(x)$ be denoted by the expression

$x : P(x)$

where the ‘:’ is an infix lambda-abstraction operator instead of the prefix operator λ .

(Other authors use ‘^’ and ‘\’ respectively for the same purpose).

‘`:`’ is a **right** associative opererator, i.e.

`x:y:P(x,y)` means `x:(y:P(x,y))`.

Now it remains to assign the semantics `x:On(x,C)` to the phrase “is on C” .

```
verb_phrase(SemVP) -->
   [is],
   preposition(SemP),
   noun_phrase(SemObj),
   {SemVP= < ... SemP(SemObj) ... >}
```

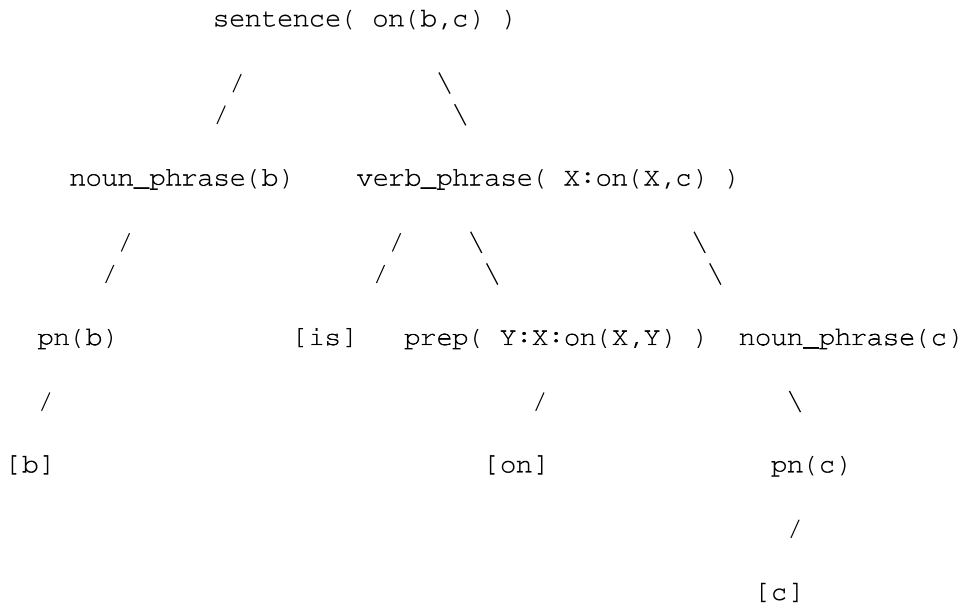

Figure 7.1: Annotated Compositional Parse Tree

We let the semantics of "on" be a two place predicate 'on', i.e. `Y:X:on(X,Y)` (the reversal of X,Y will become appearent)

```
preposition(SemP} --> [on], {SemP = Y:X:on(X,Y)}.
```

Then we can have

```
SemVP  = SemP(SemObj)

       = Y:X:on(X,Y)(c)

       = X:on(X,c)
```

So, the full parse tree with compostional semantic annotation is shown in the Figure 7.1, with a grammar fragment in Figure 7.2 on the facing page.

7.1 Proper Treatment of Quantifiers

The above treatment went all right, but there is more to noun phrases than just names of objects. If we look at the sentence

```
B is on C
```

and naively replace B with "a block" which has the "semantics"

```
Exists(x:Block(x))
```

we get

```
On( Exists(x:Block(x)),C)
```

which is not FOPL, since a quantified expression occurs inside literal. (It is sometimes called a "complex term" and is denoted

```
sentence(P) --> noun_phrase(X),verb_phrase(X:P).

verb_phrase(X:P) --> [is],complement(X:P).

complement(X:P) --> adjective(X:P).

complement(X:O) --> preposition(Y:X:O),object(Y)

adjective(X:blue(X)) --> [blue].

preposition(V:U:on(U,V)) --> [on].

noun_phrase(P) --> pn(P)

pn(b) --> [b].

pn(c) -->  [c].
```

Figure 7.2: Grammar fragment for compositional analysis

```
On( <Exists(x:Block(x))>,C)
```

What is correct FOPL is

```
Exists(x: Block(x) and On(x,C))
```

A solution to this problem was presented by R.Montague, [**?**], in his work " Proper treatment of quantifiers", (in short PTQ). It is to regard noun phrases as primary subjects that have verb phrases as their properties. In stead of saying that

```
Block B is Blue,
```

we say that "Blue is in the set of properties that B has". The set of properties that B has can be stated as a set {`P:P(B)`} so we say that in conventional set notation

$$\texttt{Blue} \in \{\texttt{P:P(B)}\}$$

That Y is a member of a set defined by a predicate R, is the same as "R(Y) holds",
so we are just saying that `Blue(B)` is true.

Similarily, the proposition Q = "being on C" can be denoted

```
Q =  X:On(X,C)
```

This means that if B is on C, this is the same as saying that Q is a member of the property set of B

$$Q \in \{P : P(B)\}$$

in other words

$$(X : On(X, C))(B) \in \{P : P(B)\}$$

i.e.

$(X : On(X, C))(B) = On(B, C)$

All this reformulation gets to fruition when we say “Some block is blue”. The set of properties that some block has can be written

{`P2: Exists(x: Block(x) and P2(x))`}

so we have

$$\text{Blue} \in \{P2 : Exists(x : Block(x) and P2(x))\}$$

and resp. for “ Some block is on C”

$$(x : On(x, C)) \in \{P2 : Exists(x : Block(x) and P2(x))\}$$

All this gets nicely together when we define the semantics of “some block” as

the set of properties that some block has, i.e.

$\{P2 : Exists(x : Block(x) and P2(x)))\}$

This again can be composed of the semantics of ’some’ applied to the semantics of ’block’.

```
Sem( 'some' )   = P1:P2:  Exists(x: P1(x)  and P2(x))

Sem( 'block' ) =  x:Block(x)

Sem( 'some block') =
     Sem( 'some' ) ( Sem( 'block') ) =

     (P1:P2:  Exists(x: P1(x)  and P2(x)) (x:Block(x) ) =

      P2: Exists(x: Block(x)  and P2(x))
```

Partial evaluation of lambda expression

Prolog is made for FOPL, but the lambda calculus extends the notions of first order predicate logic. We can implement this in Prolog. However, we get a small inconvenience problem when we shall evaluate an application of a lambda expression on an argument. There is nothing in Prolog that will automatically evaluate `(X:P(X))(Y)` for us. Instead, we make an extra condition `apply` that performs the application, (i.e. a β reduction). The definition is

```
apply(X:PX,X,PX).
```

In order for this to work, we do a slight change in the representation of a predicate, we say that `X:PX` represents the predicate `PX` where `PX` is a term presumably containing X.
We show the principle on a simplified grammar subset:

```
noun(SemNoun) ---> [block],
   {SemNoun = X:block(X)}.

determiner(SemDet) ---> [some],
   {semDet = (X:P1):(X:P2):  exists(X: P1  and P2)}.
```

```
determiner(SemDet) ---> [every].
    {SemDet = (X:P1):(X:P2):forall(X:P1 => P2)}.

noun_phrase(SemNP) --->
    determiner(SemDet),
    noun(SemNoun),
    {apply(SemDet,SemNoun,SemNP)}.

verb_phrase(X:blue(X)) ---> [is],[blue].

sentence(SemS) --->
    noun_phrase(SemNP),
    verb_phrase(SemVP),
    {apply(SemNP,SemVP,SemS)}.
```

This example gives

```
exists(X: block(X) and blue(X))
```

from the input

```
[some,block,is,blue]
```

Not all combinations are identical to 'apply'. There are some other that are more specialised. A more complete grammar is summarised below. (with some shortcuts from the explanation above).

```
sentence(P2)       --> noun_phrase((X:P1):P2),verb_phrase(X:P1).

verb_phrase(X:P)   --> [is],complement(X:P).

verb_phrase(X:P)   --> intrans_verb(X:P).

verb_phrase(X:Q)   --> trans_verb(X:Y:P),noun_phrase((Y:P):Q).

complement(X:P)    --> adjective(X:P).

complement(X:Q)    --> preposition(X:Y:P),noun_phrase((Y:P):Q).

noun_phrase((X:P):P) --> name(X).

noun_phrase((X:P2):P) --> determiner((X:Q1):(X:P2):P),noun(X:P1),
                          rel_clause((X:P1):Q1).

determiner((X:P1):(X:P2): exists(X: P1 and P2)) -->
           [some] | [a] | [an].

determiner((X:P1):(X:P2): forall(X: P1 => P2)) -->
           [every] | [all] | [each] .

adjective(X:blue(X)) --> [blue].

preposition(U:V:on(U,V)) --> [on].
```

```
  intrans_verb(X:lives(X)) --> [lives].

  trans_verb(X:Y:loves(X,Y)) --> [loves].

  rel_clause((X:P): P and Q) --> [that],verb_phrase(X:Q).
  rel_clause((X:P):P) --> [].

  noun(X:box(X)) --> [box].
  noun(X:man(X)) --> [man].
  noun(X:woman(X)) --> [woman].


  name(john) --> [john].
  name(mary) --> [mary].


  name(b) --> [b].
  name(c) -->  [c].

| ?- run.
|: [every,man,loves,mary].

forall(X:man(X)=>loves(X,mary))

|: [every,box,that,is,on,c,is,blue].

forall(X:(box(X)and on(X,c))=>blue(X))
|:
```

7.2 Scoping Problems

Whenever we combine universal and existential quantifiers, we get into a scoping problem. It is called so, because the implicit variables may have a different scope in the interpretations.

Consider the sentence:

"Every man loves some woman"

For all we now, there could be one woman for each man, or all the men could love the same woman. The two formulations would be

```
forall(X : man(X) => exists(Y: woman(Y) and loves(X,Y))).

exists(Y: woman(Y) and forall(X: man(X) => loves(X,Y)))
```

One school of implementation strategies is to use complex terms as mentioned above. This results in a form that is "not PC" and is therefore calle *Quasi Logical Form (QLF)*.

```
Loves(< Forall(x: Man(x) >, <Exists(y: Woman(y)>)
```

From this form, the proper PC formula are derived through a form of extraction to yield one of the formulas above.

The full meaning of such scoping seems to be that every occurrence of a quantified noun ("every man") can be replaced by a variable ("x"), and a corresponding quantified expression (" $Forall(x : Man(x)$ ") can be put in front of any subphrase that contains the variable.

We shall return to the problems of scoping, but in a wider setting, where we have to consider also the events that the verbs occur in.

Similar scoping decisions also occur with prepositional phrases.

```
Every box is above a floor.

forall(X:box(X)=>exists(Y:floor(Y)and above(X,Y)))
```

This translation can be paraphrased as:

for every box, we can find a floor such that the box is above the floor.

What we actually mean, given that there is only one floor, is the following:

```
exists(X:floor(X)and forall(Y:box(Y)=>above(Y,X)))
```

but with the fixed translation scheme, this can only be achieved in an indirect manner.

```
A floor is below every box.

exists(X:floor(X)and forall(Y:box(Y)=>below(X,Y)))
```

It is customary to apply the following default scoping rules to sentences with prepositional phrases, starting with the wider scope

- the subject
- the prepositional phrases, from back to front in
- the object

As an example following the principle (I would say)

"Every hunter shot a moose in the evening on every sunday."

```
For every hunter,
    for every sunday,
         there is a evening
            there is a moose
               so that bang
```

But this can of course give dubious scopings when applied strictly, e.g.

"Every hunter shot a moose on every sunday in the evening"

7.3 From Formula to Knowledge Base

We have discussed how to translate sentences, statements and questions into logic. We have also assumed that we had a knowledge base available, with a logical database at the bottom.

In this section we shall se how we can convert natural language sentences into this kind of knowledge bases, and how this can be queried afterwards using questions from the same language.

Naturally Readable Logic

Using natural language as a knowledge representation language and Knowledge Query Language, by relying on a translation to First Order Predicate Logic (FOPL).

```
John loves Mary.

FOPL:  Loves(John,Mary).

Does John love Mary ?

FOPL:  Loves(John,Mary) ?

   ==> Yes

Who loves Mary ?

FOPL:  Which(x: Loves(x,Mary)) ?

   ==> John
```

We use the results from the theory of Resolution, which will be assumed as known to the reader.

> Any set of formulas (A) in FOPC can be put into clausal form.
>
> Any question (B) to this account can be formulated in FOPC, negated and then put into clausal form .
>
> A Resolution theorem prover may be invoked to derive a contradiction.

In that case, it follows that the set of formulas logically implies the existence of a positive answer to the question. In other words,
if

```
A is assumed to be true

A and not B leads to a contradiction

( by a sound proof method )
```

then

```
A implies B
```

Furthermore, we can introduce variables into our questions, and extract the answers from the substitutions made in the proof process.

```
Who loves Mary ?
```

is stated as

```
not Exists(x:Loves(x,Mary))
```

which can be shown to lead to a contradiction by setting

```
x = John
```

We let the quantifier `Which` capture this mode of question (provoking the answer by denying its existence) in the following way:

```
Who loves Mary ?

 FOPL:  Which(x: Loves(x,Mary)) ?

    ==> John
```

Now we come to the knowledge formulated in more general terms.

```
Some man loves Mary
```

This is traditionally represented by inventing a fictitious unique identifier of which we state that this identifier represents an entity which is a man and loves Mary. Such a unique identifier is called a Skolem constant. Let us use the notation `Sk<n>` for a Skolem constant, where $< n >$ is a number. Then we get

```
Man(Sk1).

Loves(Sk1,Mary).
```

A more general example is

```
Every man  loves a woman.
```

Here, we say that for every man, we can find a woman such that the man loves the woman. Another way of saying this is that there is a (Skolem) function Sk2 such that for each man X, Sk2(X) is a woman that X loves. In logic

```
Forall(x: Man(x) => Woman(Sk2(x)) and Loves(x,Sk2(x))).
```

The process of bringing a FOPC formula into clausal form is described elsewhere [**?**]. A readable form of this is *Kowalski Clausal form* , where we

- eliminate existential quantifiers by Skolemisation (as above),
- eliminate universal quantifiers as redundant information
- bring all conjunctions into separate atomic formulae
- write all non atomic formulas on the form
  `B <= A1 and ...  An.`

The resulting form will for all practical purposes be equivalent to Prolog programs. With these conventions, we can transform the following sentences in NRL:

```
Every man that lives loves Mary

John is a man that lives.

Mary is a woman

Which man  loves a woman ?

==> John
```

This can be translated to FOPC as follows

```
Loves(x,Mary) <= Man(x) and Lives(x).

Man(John).

Lives(John).

Woman(Mary).

Answer(x) <= Man(x) and Woman(y) and Loves(x,y).
```

A resolution proof of this (as if executed by Prolog) will derive the expected answer `John`.

7.4 Towards a Micro Implementation of SHRDLU

The famous program SHRDLU was made as early as 1971 by T.Winograd. The program could answer questions like

```
Put the red pyramid on the blue box !

What is the red pyramid supported by ?
```

The present implementation will answer questions as shown in the sample dialog:

```
E:  box a is blue and box a is on box b.

==>  (box(a)and blue(a))and box(a)and box(b)and on(a,b)


E:  box b is blue and box b is on box c.

==> (box(b)and blue(b))and box(b)and box(c)and on(b,c)


E: which box is blue.

==> which(X:box(X)and blue(X))

==> a
==> b



E: which box that is blue is on a box that is not blue.

==> which(X:(box(X)and blue(X))and
    exists(Y:(box(Y)and not blue(Y))and on(X,Y)))

==> b
```

```
E:q.
yes
| ?-
```

In the appendix “A micro version of SHRDLU”, we describe the program further, with listings.

7.5 CHAT-80 Revisited

As a final example of the capabilities of the theories and programs that have been demonstrated, we shall make one final visit to the program CHAT-80 that has been mentioned earlier.

In this version, we shall use the Consensical grammar meta-interpreter in stead of the pure DCG version, and we shall mimic the quite impressive language capabilities of CHAT-80 in a smaller scale.

CHAT-80 is a query system about countries, borders, cities, rivers and populations. In addition to the language system, the original CHAT-80 contained quite a complete database of these matters. It also contained a planner that produces a very efficient database query language.

In this version, only a few database items are shown for the demonstration. The really interesting thing is that the NL system not only understands the questions, but proves this by evaluating the queries and returns correct answers !

The Microworld of CHAT-80

An excerpt of the database speaks for itself.

```
% Entity classes

ako(measure,thing).
ako(place,thing).
ako(country,place).
ako(city,place).
ako(capital,city).
ako(sea,place).
ako(population,measure).

% Entities

isa(population=_,population). % ad hoc for attributes

isa(turkey,country).
isa(china,country).
isa(india,country).
isa(soviet,country).
isa(mediterranean,sea).

isa(trondheim,city).

border1(china,india).
border1(china,soviet).
border1(turkey,mediterranean).
border1(turkey,soviet). % in 1980

have(india,(population=900)).
```

```
have(china,(population=1100)).


border(X,Y):- border1(X,Y).
border(X,Y):- border1(Y,X).
```

Some Examples

The examples are given so that the reader can verify them toward the micro world database, and to the answers to earlier questions.

```
E: What is the population of India ?

which(X:exists(X:(isa(X,population)and
   of(X,india)and true)and true))

  ==> population=900


E: which country has a population that exceeds
   the population of India ?


which(X:(isa(X,country)and true)and
  exists(Y:(isa(Y,population)and
  exists(Z:(isa(Z,population)and
  of(Z,india)and true)and exceed(Y,Z))and
  true)and have(X,Y)))


  ==> china



E: which country borders china ?

which(X:(isa(X,country)and true)and border(X,china))


  ==> india
  ==> soviet

E: does turkey border soviet ?

test(border(turkey,soviet))


  ==> YES

E: does turkey border the mediterranean ?

test(border(turkey,mediterranean))
```

```
  ==> YES

E:  Which country bordering the mediterranean
    borders a country that is
    bordered by a country whose population exceeds
    the population of India ?

....

  ==> turkey
```

Outline of CHAT-80 Grammar

The grammar viewed as a CF grammar is meant to be a comparable subset of the grammar of Chapter 3 Grammar for a fragment of English. However, in this grammar, we have added the semantic code, much in the spirit of the SHRDLU example. There is however no dynamic updating of the database, this is a fixed microworld.

A listing of the grammar can be found in the appendix "CHAT-80 Revisited"

7.6 EXERCISES

```
1. Try to formulate the description  of the blocks world
   scenario listed above.  Modify the grammar so that they are
   translated into first order logic.


2. Extend  and modify (debug :-) the grammar to accomodate the
   theorem listed in the beginning of this chapter.

3. Consult an expert on Resolution theorem proving (maybe  yourself), and
   make a program that converts the first order logical formula
   into clausal form.

4. Consult an expert on theorem proving (maybe the teacher), to prove
   the theorem automatically.

5. Find examples and counterintuitive examples of the default scoping rules
   that are listed above.
```

Chapter 8

Discourse

8.1 What Is Discourse?

We have discussed in great detail the meaning of sentences as they appear one by one. However, in any real text situation, the meanings are seldom given in isolation, but in a context of several sentences that must be comprehended together. We call this a *discourse* . For one thing, it is not practical to pack all the information into one sentence like in:

```
Which are the last 2 buses that go from the University
of Dragvoll so that I am on the  Gløshaugen campus
before the lecture on Monday at 10 am.
```

It is often more convenient to split the request over several informative statements and a final question.

```
I study at the University of  Dragvoll.
On Monday morning I  have lecture at 10 at Gløshaugen
campus. What are the 2 last buses  I can take ?
```

This discourse is an example of a *monologue*, with one speaker and one hearer. Another form of discourse is *dialogue*, where the service agent (the hearer) take an active part in the conversation.
There are several types of dialogue:
The simplest (for the hearer) is a *System Directed Dialogue*, whereby the agent asks specific questions, and the client answers in essentially one word phrases. This dialogue form is often the strategy of choice for speech based information systems, where the recognition rate is low.

```
A: Where do you want to go from ?

C: Dragvoll.

D: Where do you want to go to ?

C: Gløshaugen

A: The following bus go from Dragvoll to Gløshaugen:
   Bus 8. Do you want information about the time ?
```

```
C: Yes

A: What day do you want to travel ?

C: on Monday

A: What time do you want to travel ?

C: 10 am

A: Is that time arrival or departure time

C: Arrival

A: On Monday, Bus 8 goes from Dragvoll 9.44 and arrives at NTH 9.57

C: Do you want to know the previous or next bus ?

A: Previous

C: On Monday, Bus 8 goes from Dragvoll 9.34 and arrives at NTH 9.47

....
```

At the other extreme, the system can be made as a *Mixed initiative dialogue*, where the client queries and responds, while the agent responds and queries (with that priority).

```
A: Hello, this is an understanding system speaking:
   How can I help you ?

C: I want to go to Gløshaugen on Monday .

A: What time do you want to travel ?

C: I must be there before 10 am.

A: Where do you travel from ?

C: Dragvoll.

A: On Monday, Bus 8 goes from Dragvoll 9.44 and arrives at NTH 9.57
...
```

It is implicit in this dialogue that the agent has an agenda to fill in sufficient information to make a meaningful answer, and that the system by convention (conversational maxims) shall use a maximum of the information given by the client.

In this text,we shall leave the very interesting topic of dialogue systems, and return the monologue mode as is also the mode for text understanding systems.

(Just mention that a dialogue version of BusTUC, called BUSTER has been made on an experimental basis ([**?**]).

8.2 Extrasentential References

Closely connected to discourse is the two important phenomena: Ellipsis and Anaphora. They both occur in texts, and refers to other parts of the text, but in two different ways.

Ellipsis is Greek, and refers to the omitment of text that is assumed to be fetched from earlier text.

```
1. I took the first bus to the city centre.
2. John took the next.
```

Here, "the next" obviously refers to the next bus to the city centre.
The treatment of ellipsis is difficult but interesting, and we shall not treat in any detail, except occasionly as a pragmatic interpretation.

Anaphora: means literally "outside phrase".
We use special phrases like pronouns and definite expressions to refer to entities that have been mentioned elsewhere in the text (usually earlier).

External anaphora:

```
1. John saw a woman.
2. The sun was shining.
3. John liked her.
```

In (3) "her" refers to the woman mentioned in a previous sentence (1)

Internal anaphora:

```
John saw a woman  that he liked
```

" he" refers to John earlier in the same sentence.

Cataphora:

```
When she came home, Mary made dinner.
```

"she" referes to Mary later in the sentence.

8.3 Discourse and TUC

The analysis of anaphoric expression is called Anaphoric Resolution. (not related to Resolution in predicate logic). We don't have to tell you that anaphoric resolution is full of ambiguities, and it is extremely difficult to capture the correct reference as well as a human would do.

TUC, as a simple minded system has implemented some parts of anaphoric references. There have been proposed various complicated algorithms, many of them based on syntax, to capture the most probable interpretation. The system that TUC applies is heavy reliant on semantic agreement. The strategy can be summarised as

The referent is the last mentioned thing that
agrees on intrasyntactical restrictions and on semantics.

(By intrasyntactical restriction is meant that "himself" to the subject, "him" doesn't, a.s.o).
In a discourse, TUC contains a list of referents, one for each semantic class.

Every time a noun phrase of a certain class is mentioned by the speaker, it evokes a new referent to that class, and the previous referent is forgotten. (This is also true when a proper noun appears as an answer in a dialogue). Thus, TUC maintains a list of the last mentioned referent of each class. When an anaphoric expression arises, (like a pronoun or a definite expresion), the reference is replaced by the last mentioned referent of that class fetched from this list. This also applies when the referring expression is more general than the referent, e.g. "this person" may refer to a man or a woman, since 'man' is a sublass of 'person'. We take an example from [**?**].

1. John saw a beautiful Acura Integra at the dealership.
2. He showed it to Bob.
3. He bought it.

After (1), the list of last mentioned referents are:

```
John      is_the man
Acura     is_the car
Sk1       is_the dealership
```

(`X is_the C` means that X is the last mentioned referent of class C).

In (2), since "He" refers to a man and John "is the" man., so "He" becomes John.

"He" might also have been "Bob" (cataphor), but since "He" never is reflexive (intrasyntactical restriction), "Bob" is excluded.

"it" may refer to either the car Acura or the dealership (Sk1).

However, as a semantic restrictions, "it" may never refer to a an (adult) person. Furthermore, the reference Sk1 (dealership) is also excluded because the following template is *not* allowed semantically:

```
An agent shows a dealership
```

Therefore, sentence (2) is paraphrased as

```
(2') John showed Acura to Bob
```

After (2), since Bob is the last mentioned male, the list of referents is

```
Bob       is_the man
Acura     is_the car
Sk1       is_the dealership
```

Consequently, sentence (3) is paraphrased as

```
(3') Bob bought Acura
```

Here one could argue that it is more natural to interpret (3) as

```
(3') John bought Acura
```

since John was the subject of the previous sentence, and therefore likely to be subject in the next, all else being equal. This is hard to say. It might be that Bob is John's rich father.

1. John saw a beautiful Acura Integra at the dealership.
2. He showed it to (his rich father) Bob.
3. He bought it. (i.e. Bob).

Or it could be opposite:

1. John saw a beautiful Acura Integra at the dealership.
2. He showed it to (his poor son) Bob.
3. He bought it. (i.e. John).

The task of making a correct analaysis of anaphora is compared to the task of understanding a story as well as a human would do, i.e. it would require all the mental faculties that a human possess. We call such problems AI-complete in analogy to measures of complexity of algorithms. This is a far fetched goal, an we have to live with more restricted interpreters for the while. The challenge is to make the systems good enough. Another example that TUC would manage is a classical one (Jespersen), although a bit macabre.

The mother gave her baby som porridge.
The baby did not like it.
The mother boiled it.

The last mentioned thing that could be referred to as "it" is a baby, but it is meaningless to boil that. Instead, the porridge, which is the previous referent before that, is chosen.

8.4 Using TUC to Understand Biomedical Texts

In order to understand text, and extract information from it, a discourse analysis is obviously necessary. The methods described above have also been used in experiments for extracting information from biomedical texts in the system called GeneTUC, as mentioned in the introduction. We shall give a few examples of such analysis here, but refer to the appendix to show more examples.

Example of Sentences Covered by Grammar

A sentence from a text may be

An active phorbol ester must therefore presumeably by activation of protein kinase C, cause dissociation of a cytoplasmic complex of NF-kappa B and I kappa B by modifying I kappa B.

which is then analysed according to a dictionary, grammar and a semantic net adapted for the domain.

```
NP            An active phorbol ester
```

```
VerbPrefix
   Auxiliary  must
   AdvPhrase  therefore  presumeably by
              activation of protein kinase C,

Verb          cause

NP            dissociation of a cytoplasmic
              complex of NF-kappa B and
              I kappa B by modifying I kappa B.
```

represents a sequence of three biological reactions, i.e.

- an active phorbol ester activates protein kinase C
- the active phorbol ester modifies I kappa B
- the active phorbol ester dissociates a cytoplasmic complex of NF-kappa B and I kappa B.

Another example from BioMedicine

```
  phorbol esters                    NP

  may induce                        V

  posttranslational modifications   NP

  of                                Prep

  cellular transcription factors    NP
  that alters their DNA-binding
  characteristics
```

Another example of internal representations. Note that this also contains a PP attachment ambiguity.

Alcohol ingestion stimulates glucocorticoid secretion in animals'

```
statement
  noun_phrase
                      Alcohol ingestion

  verb_phrase
    verb
                      stimulates

     noun_phrase
        nominal
                      glucocorticoid secretion

        noun modifiers  none
```

```
verb complement

                        in animals
```

From this is extracted an event frame representation that is stored in the database as event feature structures

```
Event:

   Action:      'activate'

   Agent:       'Alcohol ingestion',

   Patient:     'glucocorticol secretion',

   Complement:  'in animals'
```

Biomedical Texts as Naturally Readable Logic

We can demonstrate how natural language can be used as a Knowledge Representation Language and Knowledge Query Language, by relying on a translation to First Order Predicate Logic (FOPL). To take one example from Biomedicine:

```
E: Modest elevations of circulating homocysteine are common
   in patients with vascular disease.
```

Internal KB representation is in TQL format which is the actual code produced by the TUC system. It is a reified and Skolemised version of FOPL. The slash (/) is the application operator. `sk(177)` etc. are anonymous (Skolem) constants.

```
sk(177)      isa  elevation.
homocysteine isa  amino_acid
sk(178)      isa  patient
sk(180)      isa  disease

adj/modest/sk(177)/event(0)
nrel/of/elevation/amino_acid/sk(177)/homocysteine
adj/circulating/homocysteine/event(0)
adj/vascular/sk(180)/event(0)
has/patient/disease/sk(178)/sk(180)
adj/common/sk(177)/event(181)
srel/in/agent/sk(178)/event(181)
```

Question to KB:

```
E: which elevations of homocysteine are common
   in patients with vascular disease ?

==> sk(177)
```

8.5 Coherence

The final topic that has to do with discourse is *Coherence*. Coherence is the interrelationship between the sentences. It is not only to understand each sentence, and to resolve the right references. It is also important for us understand why it was written, especially what one sentence has to do with the previous sentences. Without that, the text appears incomprehensible, even if all it says is true. Example:

```
Mary fell on the floor. John studies Mathematics.
```

Since these sentences has little in common, we react with bewilderment. There is no mental model with coherent events that can bring these sentences in harmony. Without an interrelated model, there is something missing. Take some more examples:

```
(1) Mary fell on the floor.

(2) John pushed her.

(3) Fred helped her.

(4) She hated him.
```

- Sentence (2) and (1) are related, and we find that (2) is the cause of (1).
- Sentence (3) is in a way caused by (1), but only indirectly.
- In order to understand that (4) means that Mary hated John (and not Fred), it may be necessary to have a model of what happened.

The analysis of coherence is a very interesting topic that we unfortunately must leave untreated. Neither is anything of coherence analysis implemented in TUC. However, it is also a question to what degree a coherence analysis is really necessary in a natural language processing systems of practical use.

8.6 EXERCISES

```
Consider the following sentences

0) Mary fell on the floor.

1) John pushed her.

2) Fred helped her.

3) She hated him.


a)

Make an analysis of the anaphoric coreferences of these sentences
```

according to the following strategy:

The referent is the last mentioned thing that agrees on intrasentential syntax and or

b)

Make an analysis of the coherence of these sentences using the
method of Hobbs (See Jurafski& Martin, section 18.2).

Chapter 9

Logic, Events and Natural Language Understanding

9.1 Ontology and Representation

The meaning of the word Ontology is a particular theory of the nature of being or existence. The ontology must be matched by the language so that any sentence can be interpreted as a statement about the things in the world and be given a meaning in terms of this world description.

We will regard the ”universe” as a set of connected worlds. Each world has a linear time, a set of *entities*, a set of discrete *events* and a set of *situations*. In each world, the entities' names and types exist invariably, while their properties vary as a situation dependent predicate. Events happen in time and cause changes, while situations are the state of the world between changes.

Each statement connects two worlds, i.e. the world in which the utterance was made, and a new world that we talk about. The starting point will always be the *real* world (REAL), while there may be one or more *imaginary* worlds.

9.2 Events

Events are a bit more volatile than entities, they seldom occur explicitly in natural language, while their implicit existence is of paramount importance.

An event is something that happens in a time, where the time scale itself may be implicit. (Ref. Relativity Theory). Usually, an event will imply a change in the state of affairs, so that things true in the past of the event may not hold after the event.

Quite informally, we shall regard an event as a kind of strange object represented on par with the representation of entities. As such, an event is a thing that has a number of properties and attributes. There may also be subclasses of events with additional attributes. Often the verb we use to denote the event defines its main classification.

The main class of events will have the following attributes

```
Event  ako thing

Event  have action
Event  have agent.
Event  have patient.
Event  have location.
Event  have time.
```

Example:

John takes bus 5 to station NTNU at 5 o'clock in the afternoon.

This can be paraphrased as:
There was an event with

```
action      take
agent      John
patient    bus 5
location   station NTNU
time       1700 hours
```

the action which defines the type of event is usually denoted by a verb

the agent is the subject of the sentence

the patient (recipient, benefactor) is the object of a transitive verb

time and and location are typically produced by prepositional phrases.

9.2.1 Event Logic

A standard assumption in computationally oriented semantics is the knowledge of its truth conditions: that is, knowledge of what the world would be like if the sentences were true. The truth conditions are naturally expressed in logic.

It can be stated that a sentence is hardly understandable unless it can be reformulated in first order logic (FOL). ([**?**]). This does not mean that we are bound to use FOL as our only means of expression, just that every meaningful sentence is translatable to FOL. Actually, we shall assume that an expression has been properly defined when it is expressed in FOL. A corollary of knowledge of the truth conditions of a sentence is knowledge of what inferences can be legitimately be drawn from it. These ideas lie behind the concept of naturally Readable Logic (NRL).

A world will be considered consisting of entities and events, and thus the sets that we are allowed to quantify over. In each event, the entities plays certain roles.

John kissed Mary

is an event E, in which there was a kiss action, John was the agent, Mary was the patient (recipient), and the event occurred in the past time in the real world time. In pure first order logic, this event can be described as:

$$\exists E\ Event(E) \wedge Action(kiss, E) \wedge Agent(John, E) \wedge \\ Patient(Mary, E) \wedge Past(REAL, E).$$

9.3 Second Order Logic

In order to be able to talk about concepts that are above first order, it is necessary to use the expressibility of second order logic. Such a language is SOLON, which is short for "Second Order Logic for Natural Language ". It is based on a formal language SOL, which is a representation of the notions of the Lambda Calculus ([**?**]). The lambda calculus has been recast in a new form, in order to combine the expressibility of the lambda calculus and first order predicate logic in one notation. The lambda calculus may be well known to the reader, so we shall only rephrase it in the new notation. (Otherwise, the reader should consider texts in logic).

Function application is denoted by a left associative binary infix operator, chosen to be the concatenation `' '`.

`(F G X)` means F(G)(X) (i.e. F(G,X) in conventional notation)

Lambda abstraction is denoted by an infix right associative binary operator
`':'`.

`X: F (G X)` means $\lambda x.F(G(x))$ in conventional notation

Important concepts in lambda calculus are the

Alpha conversion `(X: (P X))` $\Longleftrightarrow$ `(Y: (P Y))`

Beta conversion `(X: (P X))(Y)` $\Longleftrightarrow$ `(P Y)`

Eta conversion `P` $\Longleftrightarrow$ `(X: (P X))`

Currying `(X:Y: (P X Y))` $\Longleftrightarrow$ `(X:(Y: (P X Y)))`

9.3.1 FOL in SOL

SOL is a superset of First order logic. Logical operators are represented as function constants, i.e. prefix operators.

`(and P Q), (or P Q), (implies P Q), (not P)`

besides the logical constants `true, false`.

Quantifiers are represented as second order predicates acting on functions ([?]). Thus, the quantified variable is connected to the function rather than to the quantifier.

`exists P` $\Longrightarrow$ `exists ( X:(P X))`

which corresponds to $\exists X\ P(X)$ in standard FOL notation. Similarly,

`forall P` $\Longrightarrow \forall X\ P(X)$

The FOL formula $\forall X\ Woman(X) \Rightarrow Loves(John, X)$ can then be expressed in this notation as

`forall (X: implies (woman X) (loves john X))`

The examples are meant to show that SOL is a natural superset of first order logic. The above treatise has been simplified by ignoring the events. By introducing events, we extend the First order logic by new concepts, and we are likely to call the new logic First Order Event Logic (FOEL). The kiss event example stated in the beginning, *John kissed Mary* can be given a complete FOEL formulation follows:

```
(exists(E: and (event E)
          (and (action kiss E)
          (and (agent john E)
          (and (patient mary E)
               (past REAL E))))))
```

9.4 Combinatory Logic

It is an observation that natural language can be viewed as a formal language that lacks the notation of explicit logical variables; they are only implicit. A branch of the lambda calculus is called Combinatory Logic [?] in which there are no lambda variables.

We will define our own set of combinators that combine logical expressions so that the meaning of an expression will be abstracted and represented by its behaviour when regarded as a lambda function. Our variable free version of the lambda calculus will be called combinatory logic (COL). In this spirit, we can define some high level combinatorical operators

```
every :== P:Q: forall(X: implies (P X) (Q X)).
some  :== P:Q: exists(X: and (P X) (Q X)).
```

so the sentence *John loves every woman* is expressed in COL and then translated to FOL

```
(every woman (love john))
```

```
==>
```

```
forall(X: implies (woman X) (love john X))
```

Every man that lives loves a woman

is represented in COL as

```
(every (rel man live) (trans (some woman) love))
```

and is similarily translated to the FOL expression

```
forall(X: implies (and (man X) (live X))
                  (exists (Y:and (woman Y) (love X Y))))
```

when we use some of the defintions

```
andp      :==  Y:Z:X: and (Y X) (Z X).
orp       :==  Y:Z:X: or (Y X) (Z X).
impliesp  :==  Y:Z:X: implies (Y X) (Z X).
notp      :==  Z:X:   not (Z X).
cis       :==  Z:X:Y: (Z Y) X.
trans     :==  Z:X:Y: Z (X Y).
idem      :==  X:     X.
rel       :==         andp.
truep     :==  X:     true.
```

9.5 Proper Treatment of Quantifiers

Montague in the paper "Proper Treatment of Quantifiers" (PTQ) ([**?**]) gave a uniform treatment of names (e.g. John) and noun phrases (every man) as in

John loves Mary and *Every man loves Mary*

In stead of letting *'love'* be a dominant predicate with nouns as arguments, we let the noun phrase be the dominant expression, taking a verb phrase as an argument. In general,

the meaning of an entity is defined as the set of properties that the entity has.

[1] However, it has also a practical side. A set may be represented simply as a discriminating function. Thus the " set of all properties that John has "
can be paraphrased as

$$\{P|P \text{ is a property and John has } P\},$$
in short
$$\{P|P(John)\}.$$

---

[1]

This principle is actually the "identity of indiscernabilities" devised by Leibnitz.
It means that if two entities have exactly the same set of properties, they must be considered equal.

The set notation, on the other hand, is nothing more than the extension of the corresponding lambda expression

$$\lambda P.P(John).$$

Compare this to the SOL notation `(P: P john)`

Especially the combinator

```
proper :== X:Y: (Y X)
```

makes the noun phrase "John" be represented simply as

```
(proper john)
```

`proper` can be thought of as a *Property Set Function*, i.e. the set of all properties that an argument has. So

```
(proper john big)
```

can be paraphrased as " big is a member of Johns property set".

Likewise, the meaning of the noun phrase " *every man* " is literally the set of properties that every man has. In set notation,

$\{P|\forall x(man(x) \Rightarrow P(x))\}$

In SOLON, this is simply expressed as `(every man)` .
y A final example shows what logic can do with poetry:

What is in a name that we call a rose,
By any other name would smell as sweet
...
(W. Shakespeare)

is stated prosaically as

```
(some rose)
```

which is literally the set of properties that some rose has.

9.6 The Event Complement Construction

A normal sentence in English contains a verb, a subject, and a set of verb complements that may be direct objects, indirect objects, prepositional phrases, adverbial phrases or subordinate statements. We can in this respect regard the subject as one of the verb complements. In view of events, we may regard the verb as a classification of the type of event, and all the verb complements (including the subject) as "event complements" that specify the event further. We will also remember that all events happen in one or another of the possible worlds.

We can also include complements to the nouns (reduced relatives) into this scheme by using relative clauses of the kind

" *a statue in the park* " is interpreted as " *a statue that is in the park*"

In order to better describe event complements, we use the following basic notions

Fluent and *Event Complement*.

A fluent, in general, is a function from situations to truth values. ([**?**]).

John died describes an event, where the agent is "John" and the action is "die". There are two fluents:

`(action die)` and `(agent john)`

This is because in a certain event E, the following propositions hold:

`(action die) (E)` and `(agent john) (E)`

An event complement is the property set of a fluent, and is the building block of the semantics of SOLON. If F is a fluent, then `(proper F)` is an event complement.

```
proper F    ==>  K: K F
           <==>  K: K (x: F x)
```

Thus, we have two event complements

```
proper (action die) ==> K:(K (x: (action die) x))
proper (agent john) ==> L:(L (y: (agent john) y)).
```

We seek to combine the two event complements into a new event complement:

```
M: (M (z: (and (action die z) (agent john z)))).
```

This is done by an operator `coupl`, defined below.

```
(coupl (x: and (action die x)) (y: (agent john) y)).
```

We can define some other operators (`verb, subject,object`) that makes it even more readable.

This combination is achieved as the result of

```
johndie1 :== coupl (verb die) (subject (proper john))
```

using some of the definitions

```
coupl    :== C1:C2:Z: C1 (F:C2 (G:Z (andp F G)))
comp     :== A:Q:Z:   Q (X:Z (A X)).
verb     :== V:       comp action (proper V)
subject  :==          comp agent.
object   :==          comp patient.
```

We could also replace

`(subject (proper john))` by

`(subject (every man))` and get a meaningful result.

It may look as if the use of the event complement construction is an overkill, but in the section Quantifiers and Scopings, we shall demonstrate that it solves problems with quantifiers and scopings in a uniform manner.

In a strange sense, we have reintroduced the verb as a head concept, that subordinates the noun phrase (complements), thus going back to a "pre PTQ construction."

Fixing a Sentence in a World

The combination of event complements represents a sentence as a fluent property set, (which is of type Boolean function). In order to make a proposition (of type Boolean) out of it, we bind it to an event and make it into a closed first order formula. That is demonstrated by the definition

```
somehow :== proper (some event)

(somehow johndie1) ⟹


exists(E: (and (event E)
        (and (agent john E) (action die E))))
```

In general, we need to fix an event temporally to a specific world. We assume that each world `W` has associated a time of the present that an event can be compared to.

```
(past W E)          E happened in the past in W
(present W E)       E happens in the present of W
(future W E)        E will happen  in the future of W
```

Adverbial Phrases

Adverbial phrases like prepositional phrases, for example

"before 1200"

is formalised as an event complement using the construction

```
(comp before (proper 1200))
```

"John died before 1200"

is formulated as

```
(coupl johndie1 (comp before (proper 1200)))
```

Quantifiers and Scopings

When we introduce quantified expressions, we get a scoping problem. In the sentence *"Every man sees a guitar"*, there may either be one guitar for each man, or they could see the same guitar. While the choice of the most thrustworthy of several independent interpretations is dependent on syntax, semantic and context, the representation language must be able to represent each alternative precisely.

The example *"Every man sees a guitar"* is formulated by coupling two of the event complements in either of two ways

```
look1 :== (coupl (verb see)
          (coupl (subject (every man))
                 (object (some guitar)))).

look2 :== (coupl (verb see)
          (coupl (object (some guitar))
                 (subject (every man)))).
```

We will soon see that the order of scoping of the verb complements is the same as the sequence of appearence in the "(coupl)" expression.

```
(coupl A B)
```

means that A outscopes B. This greatly simplifies the representation of complex scoping matters. In the first case, each man sees a different guitar, in the second case, there is one guitar that they all see. We will demonstrate that by translating `look1` and `look2` to FOL:

" for each man there is a guitar s.t.... "

```
(somehow look1) ⟹

forall(X: implies (man X)
  (exists(Y: and (guitar Y)
    (exists(E: and (event E)
      (and (action see E)
      (and (agent X E)
           (patient Y E))))))))
```

" there is a guitar s.t. for each man ...

```
(somehow look2) ⟹

exists(Y: and (guitar Y)
  (forall(X: implies (man X)
    (exists(E: and (event E)
      (and (action see E)
      (and (patient Y E)
           (agent X E))))))))
```

9.6.1 Compositional Semantics

The semantics of natural language is by definition undefined (that is why we call it natural). For Montague and others working in the frameworks descending from that tradition, the intermediate logical language was merely a matter of convenience which could in principle be dispensed with provided the principle of compositionality was observed.

Compositional semantics is a way of insisting of meaningful subexpressions. It means that the semantics of any phrase is a combination of the semantics of its subphrases. It does not depend on any other phrase before, after or encompassing the given phrase.

Compositional semantics is also necessary because without it, we would not be able to define the meanings of an infinite set of sentences by using a finite set of meaning definitions.

We see that when we use event complements as building blocks, compositional semantics is realised through the `coupl` operator.

It is also easy to make a parser for natural language NRL that is defined in terms of compositional semantics. There is almost a one-one correspondence between

Every man sees a guitar

and the SOLON version

```
look3 :== coupl (subject (every man))
                (coupl (verb see)
                       (object (some guitar)))
```

It can be shown that `coupl` is associative, idempotent, non-commutative and having a unity element (`proper truep`).

This `coupl` construction has some resemblance of Quasi Logical Form ([**?**]), where the scoping of noun phrases is left undecided. It is customary to leave the scoping as a a separate issue, transforming an expression in Quasi Logical Form to First Order Logic, using transformation rules and scoping priorities. In SOLON, this is expressed through a permutation of the event complements. In fact every combination of coupling of the complements gives a meaning, and most scoping regimes can easily be achieved by a mere reordering of complements. The same flexibility also goes for prepositional phrase complements.

9.6.2 Subordinate Sentences

The worlds are connected to each other by constructions similar to subordinate statements. Thus, if *"John says that Mary kissed Fred"*, we get a subordinate world where the kiss did happen. This imaginary world may or may not coincide with the real world, depending on the truthfulness of the speaker. The real world is represented by a known constant REAL. The sentence

"John says that Mary kissed Fred"

is modelled as follows:

There is a story W1 that John said in a real event E1 that happens in the present. In the story W1, there was an event E2 in its past where Mary kissed Fred.

```
exists(W1: and
    (and (isa story W1) (and

(exists(E1: and (event E1)
    (and (present REAL E1)
    (and (action say E1)
    (and (agent john E1)
         (patient W1 E1))))))

(exists(E2: and (event E2)
    (and (past W1 E2)
    (and (action kiss E2)
    (and (agent mary E2)
         (patient fred E2))))))
)))
```

The distribution of world parameters makes the SOLON operators slightly more complex than explained in this text. The reader may refer to the appendix "A combinatory kernel of Consensical Grammar" However, the main principles remain the same, most importantly, the compositional semantics. Statements about knowledge and belief are treated by sentence subordination, where the the subordinate sentence is modelled as a property of the subordinate world.

9.6.3 Collective and Distributive Interpretations

It is well known that there is a difference, besides the scopings, in the interpretations of

1. Three men carried a woman
2. Three men carried a piano

even if there is only *one woman and one piano*.

In the first case, there were 3 events where 3 men carried a woman, one after another. We call this a *distributive interpretation*.
In the second case, there was only one event, in which three men participated, carrying the same piano. We call this a *collective interpretation*.
It is a pragmatic issue when the most reasonable interpretation is collective (e.g. when we know that a piano is too heavy for one person). However, both alternatives must be expressible in SOLON. We will show by examples how the scoping and distributive interpretations are formulated.
Three men carried a piano (1 piano, 3 men, 3 events)

```
carry1 :==  (coupl (verb carry)
                   (coupl (object (some piano))
                          (subject (every man))))
```

Three men carried a piano together. (1 piano, 3 men, 1 event)

```
carry2 :==  coll (coupl (verb carry)
                        (coupl (object (some piano))
                               (subject (every man))))

coll     :== Q:     proper (fix Q).
fix      :== Q:E:   Q (proper E).
```

Distributive: The events are dependent on the men, one for each.

```
  somehow carry1 ⟹

 (exists(Y: and (piano Y)
    (forall(X: (implies  (man X)
       (exists(E: and (event E)
                 (and (action carry E)
                 (and (patient Y E)
                 (agent X E))))))))))
```

Collective: There is one event, independent on the men.

```
  somehow carry2 ⟹

(exists(E: and (event E)
  (exists(Y: and (piano Y)
    (forall(X: implies   (man X)
       (and (action carry E)
       (and (patient Y E)
            (agent X E)))))))))
```

By a simple technicality, we have made one global event E under which all the actions are related.

9.7 Conclusions

We have in this chapter shown how the framework of a natural language understanding system can be based on a combinatory lambda calculus. Some problems of natural language semantics have found its logical and practical solutions.

Chapter 10

BusTUC - A natural language bus route oracle

10.1 Introduction

A natural language interface to a computer database provides users with the capability of obtaining information stored in the database by querying the system in a natural language (NL). With a natural language as a means of communication with a computer system, the users can make a question or a statement in the way they normally think about the information being discussed, freeing them from having to know how the computer stores or processes the information.

The present implementation represents a major effort in bringing natural language into practical use. A system is developed that can answer queries about bus routes, stated in natural language texts, and made public through the Internet World Wide Web (`http://www.idi.ntnu.no/bustuc/`).

Trondheim is a small city with a university and 140000 inhabitants. Its central bus systems has 42 bus lines, serving 590 stations, with 1900 departures per day (in average). That gives approximately 60000 scheduled bus station passings per day, which is somehow represented in the route data base.

The starting point is to automate the function of a route information agent. The following example of a system response is taken from an actual request over telephone to the local route information company:

```
Hi, I live in Nidarvoll and tonight i must
reach a train to Oslo at 6 oclock.
```

and a typical answer would follow quickly:

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Bus number 54 passes by Nidarvoll skole at 1710
and arrives at Trondheim Railway Station at 1725.
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

In between the question and the answer is a process of lexical analysis, syntax analysis, semantic analysis, pragmatic reasoning and database query processing.

One could argue that the information content could be solved by an interrogation, whereby the customer is asked to produce 4 items:

```
station of departure,
station of arrival,
earliest departure time and/or
latest arrival time
```

It is a myth that natural language is better way of communication because it is "natural language". The challenge is to prove by demonstration that an NL system can be made that will be preferred to the interrogative mode. To do that, the system has to be correct, user friendly and almost complete within the actual domain with respect to language and expertise.

10.2 Previous Efforts, CHAT-80, PRAT-89 and HSQL

The system, called BusTUC is built upon the classical system CHAT-80 ([**?**]). CHAT-80 was a state-of-the-art natural language system that was impressive on its own merits, but also established Prolog as a viable and competitive language for Artificial Intelligence in general. The system was a brilliant masterpiece of software, sophisticated and extremely efficient. The natural language system was connected to a small query system for international geography. The following query could be analysed and answered in less than half a second:

```
Which country bordering the Mediterranean borders a
country that is bordered by a country whose population
exceeds the population of India?
```

(*The answer 'Turkey' has become incorrect as time has passed. The irony is that Geography was chosen as a domain without time.*)

The ability to answer ridiculously long queries is of course not the main goal. The main lesson is that complex sentences are analysed with a proper understanding without sacrificing efficiency. Any superficial pattern matching technique would prove futile sooner or later.

10.2.1 Making a Norwegian CHAT-80, PRAT-89

At the University of Trondheim (NTNU), two students made a Norwegian version of CHAT-80,called PRAT-89 ([**?**],[**?**]). (Also, a similar Swedish project SNACK-85 was reported).

The dictionary was changed from English to Norwegian together with new rules for morphological analysis. The change of grammar from English to Norwegian proved to be amazingly easy. It showed that the langauges were more similar than one would believe, given that the languages are incomprehensible to each other's communities.

After changing the dictionary and grammar, the following Norwegian query about the same domain could be answered correctly in a few seconds.

```
Hvilke afrikanske land som har en befolkning stoerre
enn 3 millioner og mindre enn 50 millioner og er nord
for Botswana og oest for Libya har en hovedstad som
har en befolkning stoerre enn 100 tusen.
```

(*A translation is beside the point of being a long query in Norwegian.*)

10.2.2 HSQL - Help System for SQL

A Nordic project HSQL (Help System for SQL) was accomplished in 1988-89 to make a joint Nordic effort on interfaces to databases.

The HSQL project was led by the Swedish State Bureau (Statskontoret), with participants from Sweden, Denmark, Finland and Norway [**?**]. The aim of HSQL was to build a natural language interface to SQL databases for the Scandinavian languages Swedish, Danish and Norwegian. These languages are very similar, and the Norwegian version of CHAT-80 was easily extended to the other Scandinavian languages. Instead of Geography, a more typical application area was chosen to be a query system for hospital administration. We decided to target an SQL database of a hospital administration which had been developed already.

The next step was then to change the domain of discourse from Geography to hospital administration, using the same knowledge representation techniques used in CHAT-80. A semantic model of this domain was made, and then implemented in the CHAT-80 framework.

The modelling technique that proved adequate was to use an extended Entity Relationship (ER) model with a class (type) hierarchy, attributes belonging to each class, single inheritance of attributes and relationships.

Coupling the system to an SQL database.

After the remodelling, the system could answer queries in "Scandinavian" to an internal hospital database as well as CHAT-80 could answer Geography questions. HSQL produced a Prolog-like code FOL (First Order Logic) for execution. A mapping from FOL to the data base Schema was defined, and a translator from FOL to SQL was implemented. The example

```
 Hvilke menn ligger i en kvinnes seng?
( " Which men lie in a woman's bed? " )
```

would be translated dryly into the SQL query:

```
SELECT DISTINCT T3.name,T1.sex,T2.reg_no,T3.sex,
       T4.reg_no,T4.bed_no,T5.hosp_no,T5.ward_no
FROM PATIENT T1,OCCUPANCY T2,PATIENT T3,
     OCCUPANCY T4,WARD T5
WHERE (T1.sex='f') AND (T2.reg_no=T1.reg_no) AND
      (T3.sex='m') AND (T4.reg_no=T3.reg_no) AND
      (T4.bed_no=T2.bed_no) AND (T5.hosp_no=T4.hosp_no) AND
      (T5.ward_no=T4.ward_no)
```

10.2.3 The Understanding Computer

The HSQL was a valuable experience in the effort to make transportable natural language interfaces. However, the underlying system CHAT-80 restricted the further development.

After the HSQL Project was finished, an internal reseach project TUC (the Understanding Computer) was initiated at NTNU to carry on the results from HSQL. The project goals differed from those of HSQL in a number of ways, and would not be concerned with multimedia interfaces . On the other hand, portability and versatility were made central issues concerning the generality of the language and its applications. The research goals could be summarised as to

- Give computers an operational understanding of natural language.
- Build intelligent systems with natural language capabilities.
- Study common sense reasoning in natural language.

A test criterion for the understanding capacity is that after a set of definitions in a Naturally Readable Logic, NRL, the system's answer to queries in NRL should conform to the answers of an idealised rational agent.

```
Every man that lives loves Mary. John is a man. John lives.
Who loves Mary?
==> John
```

NRL is defined in a closed context. Thus interfaces to other systems are in principle defined through simulating the environment as a dialogue partner.

TUC is a prototypical natural language processor for English written in Prolog. It is designed to be a general purpose easily adaptable natural language processor. It consists of a general grammar for a subset of English, a semantic knowledge base, and modules for interfaces to other interfaces like UNIX, SQL-databases and general textual information sources.

10.2.4 The TABOR Project

It so happened that a Universtity Project was starteded in 1996, called TABOR (" Speech based user interfaces and reasoning systems "), with the aim of building an automatic public transport route oracle, available over the public telephone. At the onset of the project, the World Wide Web was fresh, and not as widespread as today, and the telephone was still regarded as the main source of information for the public. Since then, the Internet has become the dominant medium, and it is as likeley to find a computer with Internet connection, as finding a telephone, or a local busroute table for that matter.

It was decided that a text based information system should be built, regardless of the status of the speech rocgnition and speech synthesis effort, which proved to lag behind after a while.

(Beforethe project was finished, the cellular phone revolution started, giving more prominence for the speech approach, but more recently, we have got the short message services (SMS) which tips the picture back in favour of text).

The BusTUC system

The resulting system BusTUC grew out as a natural application of TUC, and an English prototype could be built within a few months (J.Bratseth [**?**]).

Since the summer 1996, the prototype was put onto the Internet, and been developed and tested more or less continually until today. The most important extension was that the system was made bilingual (Norwegian and English) during the fall 1996.

In spring 1999, the BusTUC was finally adopted by the local bus company in Trondheim (A/S Trondheim Trafikkselskap), which set up a server.

Until today, over 100.000 questions have been answered, and BusTUC seems to stabilize and grow increasingly popular.

10.3 Anatomy of the bus route oracle

The main components of the bus route information systems are:

- A parser system, consisting of a dictionary, a lexical processor, a grammar and a parser.
- A knowledge base (KB), divided into a semantic KB and an application KB
- A query processor, containg a routing logic system, and a route data base.

The system is bilingual and contains a double set of dictionary, morphology and grammar. Actually, it detects which language is most probable by counting the number of unknown words realated to each language, and acts accordingly. The grammars are surprisingly similar, but no effort is made to coalesce them. The Norwegian grammar is slightly bigger than the English grammar, mostly because it is more elaborated but also because Norwegian allows a freer word order.

10.3.1 Features and figures of BusTUC

For the Norwegian systems, the figures give an indication of the size of the domain: 420 nouns, 150 verbs, 165 adjectives, 60 prepositions, etc.

There are 1300 grammar rules (810 for English) although half of the rules are low level lexical type rules.

The semantic net described below contains about 4000 entries.

A big name table of 3050 names in addition to the official station names, is required to capture the variety of naming. A simple spell correction is a part of the system (essentially 1 character errors).

The pragmatic reasoning is needed to translate the output from the parser to a route database query language . This is done by a production system called Pragma, which acts like an advanced rewriting system with 580 rules.

In addition, there is another rule base for actually generating the natural language answers (120 rules).

The system is mainly written in Prolog (Sicstus Prolog), with some Perl programs for the communication and CGI-scripts.

At the moment, there are about 35000 lines of programmed Prolog code (in addition to route tables which are also in Prolog).

The server is a 300 MHz PC with Linux . Average response time is usually less than 1 second, but a timeout of 3 seconds is imposed.

Sicstus Prolog proved to be extremely efficient and reliable for the application.

10.3.2 The Parser System

The Grammar System

The grammar is based on a simple grammar for statements, while questions and commands are derived by the use of movements. The grammar formalism which is called Consensical Grammar, (CONtext SENSItive Categorical Augmented Logic Grammar) is an easy to use variant of Extraposition Grammar ([**?**]), which is a generalisation of Definite Clause Grammars. As for Extraposition grammars, a grammar is translated to Definite Clause Grammars, and executed as such.

A characteristic syntactic expression in Consensical Grammar may define an incomplete construct in terms of a “difference ” between complete constructs. When possible, the parser will use the subtracted part in stead of reading from the input, after a gap if necessary.

The effect is the same as for Extraposition grammars, but this format makes it much more intuitive.

Examples of grammar rules.

```
statement(P) --->
   noun_phrase(X,VP,P),
   verb_phrase(X,VP).

statement(Q) --->
   verb_complements0(VC),  % initial optional verb_complements
   statement(Q) /          % may be  inserted  after a gap
        verb_complements0(VC).

whoseq(P) --->   % whose dog barked?
   [whose],
   noun(N),
   whoq(P) \ ([who],[has],[a],noun(N),[that]). % without gap

whoq(P) --->
   [who],
   whichq(P) \ ([which],[person]).

whichq(which(X)::P) --->
   [which],
   statement(P) \ the(X).
```

Example:

```
Whose dog barked?
```

is analysed as if the sentence had been

```
Who has a dog that barked?
```

which is analysed as

```
Which person has a dog that barked?
```

which is analysed as

```
for which X is it true that
the (X) person has a dog that barked?
```

where the last line is analysed as a `statement`.

Movement is easily handled in Consensical Grammar without making special phrase rules for each kind of movement. The following example shows how TUC manages a variety of analyses using movements:

```
Max said Bill thought Joe believed Fido Barked.

Who said Bill thought Joe believed Fido barked?
   ==> Max
Who did Max say thought Joe believed Fido barked?
    ==> Bill
Who did Max say Bill thought believed Fido barked?
    ==> Joe
```

The parser

The experiences with Consensical grammars are a bit mixed however, due to the wide variety of modes of expressions, the incredible ambiguity and the sheer size of the covered language.

Many principles that would prove elegant for small domains turned out to be to costly for larger domains.

The disambiguation is a major problem for small grammars and large languages, and was solved by the following guidelines:

- a semantic type checking was integrated into the parser, and would help to discard sematically wrong parses from the start.
- a heuristics was followed that proved almost irreproachable: The longest possible phrase of a category that is semantically correct is in most cases the preferred interpretation.
- due to the perplexity of the language, some committed choices (cuts) had to be inserted into the grammar at strategic places. As one could fear however, this implied that wrong choices being made at some point in the parsing could not be recovered by backtracking.

These problems also made it imperative to introduce a timeout on the parsing process of 3 seconds. Although most sentences, would be parsed within a second, some legal sentences of moderate size actually need this time.

10.3.3 The semantic knowledge base

Adaptability means that the system does not need to be reprogrammed for each new application.

The design principle of TUC is that most of the changes are made in a tabular semantic knowledge base, while there is one general grammar and dictionary. In general, the logic is generated automatically from the semantic knowledge base.

The nouns play a key role in the understanding part as they constitute the class or type hierarchy. Nouns are defined in an `a-kind-of` hierarchy. The hierarchy is tree-structured with single inheritance. The top level also constitute the top level ontology of TUC's world.

In fact, a type check of the compliances of verbs, nouns adjectives and prepositions is not only necessary for the semantic processing but is essential for the syntax analysis for the disambiguation as well. In TUC, the legal combinations are carefully assembled in the semantic network, which then serves a dual purpose.

These semantic definitions are necessary to allow for instance the following sentences

```
The dog saw a man with a telescope.
The man saw a dog with a telescope.
```

to be treated differently because `with telescope` may modify the noun `man` but not the noun `dog`, while `with telescope` modifies the verb `see`, restricted to person.

10.3.4 The Query Processor

Event Calculus

The semantics of the phrases are built up by a kind of verb compelements, where the event play a central role. The text is translated from natural language into a form called TQL (Temporal Query Language/ TUC Query Language) which is a first order event calculus expression, a self contained expression containing the literal meaning of an utterance.

A formalism TQL was defined, inspired by the Event Calculus by Kowalski and Sergot ([**?**]).

The TQL expressions consist of predicate, functions, constants and variables. The textual words of nouns and verbs are translated to generic predicates using the selected interpretation.

The following question

```
Do you know whether the bus goes to Nidarvoll on Saturday ?
```

would give the TQL expression below. Typically, the Norwegian equivalent

```
Vet du om bussen går til Nidarvoll på lørdag ?
```

gives exactly the same code.

```
test::                          % Type of question
 isa(real,program,bustuc),      % bustuc is a real program
 isa(real,bus,A),               % A is a real bus
 isa(real,saturday,B),          % B isa saturday
 isa(real,place,nidarvoll),     % nidarvoll isa place
 event(real,D),                 % D is real event
 know(whether,bustuc,C,D),      % C is a statement known at D
 event(C,E),                    % E is an event in C
 action(go,E),                  % the action of E is 'go'
 agent(A,E),                    % the agent of E is  A
 comp(to,place,nidarvoll,E),    % E is related to Nidarvoll
 comp(on,time,B,E).             % E is related on Saturday
```

The event parameter plays an important role in the semantics. It is used for various purposes. The most salient role is to identify a subset of time and space in which an action or event occured. Both the actual time and space coordinates are connected to the actions through the event parameter.

Pragmatic reasoning

The TQL is translated to a route database query language (BusLOG) which is actually a Prolog program. This is done by a production system called Pragma, which acts like an advanced rewriting system with 580 rules.

In addition, there is another rule base for actually generating the natural language answers (120 rules).

10.4 Conclusions

The TUC approach has as its goal to automate the creation of new natural language interfaces for a well defined subset of the language and with a minimum of explicit programming.

The implemented system has proved its worth, and is interesting if for no other reason. There is also an increasing interest from other bus companies and route information companies alike to get a similar system for their customers.

Further work remains to make the parser really efficient, and much work remains to make the language coverage complete within reasonable limits.

It is an open question whether the system of this kind will be a preferred way of offering information to the public.

If it is, it is a fair amount of work to make it a portable system that can be implemented elsewhere, also connecting various travelling agencies.

If not, it will remain a curiosity. But anyway, a system like this will be a contribution to the development of intelligent systems.

Chapter 11

GeneTUC

This chapter describes some of the work done on the GeneTUC system. It recounts some of the differences between GeneTUC and the original TUC, and what enhancements have been made on GeneTUC, and subsequently TUC, during the project. [1]

11.1 History

GeneTUC stems from the BusTUC system [**?**, **?**]. Work on the system was initiated in January 2000 as a student project [**?**]. The original framework was augmented with a moderate number of words from the biomedical domain, regarding gene - protein interactions. All concepts were entered manually, using Medline abstracts as a "training set", trying to incrementally expand GeneTUC's capabilities on a per-sentence basis.

Later, databases containing gene and protein names and their synonyms were imported from the HUGO[2] Gene Nomenclature database and the SwissProt Annotated protein sequence database[3], respectively. This massively increased the size of GeneTUC's permanent database. The permanent base now contains names of more than 10,000 genes and more than 5,000 proteins. A list of adjectives and adverbs was imported from the WordNet[4] [**?**] lexical database.

The development has been towards creating a very general semantic base, allowing for constructs normally not regarded as meaningful[5], but being grammatically correct. This has been done under the assumption that the contents of the input, which is taken from the Medline corpus, has been proof-read and contains no or few meaningless sentences. Furthermore, irrelevant information will not interfere with the essentials of the semi-permanent database. When the semantic and grammatic base is sufficiently "trained", one might constrain the semantics, filtering out "noise" in the input.

11.2 Goals

The goals for the GeneTUC project can roughly be divided into two parts. The first part is the goals related to further development of the TUC architecture. The second part is the goals related to understanding texts related to genetics.

[1] This chapter is adapted from the Masters thesis of Anders Andenæs [**?**]

[2] The Human Genome Organisation, http://www.hugo-international.org/hugo/

[3] http://www.expasy.ch/sprot/sprot-top.html

[4] http://www.cogsci.princeton.edu:80/~wn/

[5] e.g., genes are *agents* (see below), thus they can speak, as all agents can.

11.2.1 TUC-related

The TUC framework has thus far only given rise to one application, the BusTUC bus route oracle. One of the key objectives of this project was therefore to assess how easily the TUC architecture could be transferred to another domain, far removed from bus routes. TUC is designed to be domain-independent system, suggesting that porting the framework to a new domain is feasible. Porting the system must be done without interfering with the domain-independent parts of the framework, notably the grammar and part of the semantics. This to maintain some sort of “sideways compatibility” between the applications, i. e., all successful parses in GeneTUC should have a BusTUC equivalent. An example:

> A protein is a marker to predict the outcome.

becomes:

> A bus is a vehicle to pass the airport.

In this manner, development of one TUC application is a development of the entire TUC architecture.

As TUC is still very much a work in progress, the grammar and the domain-independent semantics are not complete. By entering another domain, new errors and omissions are detected; errors and omissions not easily found when concentrating solely on one limited domain (bus route queries). The new application thus aids in the development of the entire framework.

GeneTUC’s vocabulary exceeds TUC’s by at least one, probably several, orders of magnitude. Hence, GeneTUC is a good test of how well the architecture scales in terms of size. Will the increased vocabulary and semantic net cause the application to run appreciably slower? As the results will show, this is not the case.

GeneTUC is also set to evaluate how well the TUC architecture is ported to one of the most complex knowledge domains of our time. Research and development efforts in molecular biology and genetics provide us with new results and knowledge on a daily basis. This information need not only be collected, it must also be stored and maintained once it is extracted.

Finally, the extended use of the system helps debug the framework, uncovering errors not related to the grammar or semantics, i.e., errors related to the program execution. Some types of sentences cause the program to stop execution; this behaviour has to be checked.

11.2.2 Genetics-related

The primary goal in respect to genetics and molecular biology is to create a NLP capable of extracting factual assertions from a large corpus of literature. These assertions could be simple, like:

> Here we demonstrate, using a cell free system, that at low concentrations of heparin, FGF4 binds only to FGFR-2, while much higher heparin levels are required for binding to FGFR-1

giving something in the lines of (excerpt):

> bind/fgf4/’fgfr-2’/A

or more complex assertions like:

> Risk factors for vascular disease in general and stroke in particular had no visible influence on CRP levels .

becoming (excerpt):

```
(A isa influence, B isa level,
adj/visible/A/C, adj/crp/B/D,
nrel/on/influence/level/A/B,
has/risk/influence/sk(58)/A,
event/real/E) => false
```

This ultimately leads to a large database consisting of assertions extracted from free-text sources, such as Medline citations. Using the NL interface, all this information would be readily accessible and a valuable asset to geneticists worldwide. GeneTUC, with its knowledge of both written language and logic, could also be used as an NL interface to existing knowledge bases. Potentially, it can become a universal interface to a large number of distributed knowledge bases.

The information GeneTUC extracts is an augmentation of the efforts to map the human genome, such as Celera's[6] and the HGP [**?**]. These projects are well on their way, but a simple mapping of the genome does not eliminate the need for researching into how genes and proteins interact in the organism. Additionally, much science effort has already been put into this field of research over the years. This knowledge needs to be assembled and organised for easy access, a task which GeneTUC might very well perform.

11.3 Adapting TUC

Bus route queries are most often posed in a direct manner, along the lines of[7]:

> When is the next bus from the train station to the airport?

Plain and simple as it may seem, this question does contain some subtleties. The asker expects to get an answer referring to a certain bus departure from the train station, even though the question makes no reference to the word "departure" or any related words, whatsoever. Also, the time reference is a bit vague. The "next" bus may refer to the first bus departing after this instant or the next bus after an earlier mentioned departure. Still, by applying a few conventions, extracting the meaning of this interrogative sentence is not too hard.

Scientists seldom feel obliged to keep the language in their articles direct and simple, and the tendency is even more apparent in the article abstracts. The abstracts are kept compact, forcing the authors to cram as much information as possible into a small number of sentences. The result is often long sentences densely packed with information, with cascaded subordinate clauses and extended use of ellipsis and anaphora. This affects the readability of the texts, for humans and computers alike.

Another challenge is coping with the different styles of writing of each author. An abstract is typically less than fifteen sentences, and in a large corpus, many different authors will have contributed, each having her own way of presenting her material. GeneTUC must therefore cope with both direct language, as well as more elaborate ways of expression.

11.3.1 Key Relationships

Starting the GeneTUC project, it appeared to be wise to focus on a few key relationships. A cancer researcher was consulted [**?**], and an incomplete list of the most common interactions between genes and proteins was produced, shown in Figure 11.1.

As the project has evolved, these relations have been kept at the centre of attention. But adding new relations to the list require minimal effort, and GeneTUC will try to extract as much knowledge as it possibly can from all sentences, no matter what. What might be considered, is a mechanism for mapping other, synonymical relations injectively into those in Figure 11.1. Information retrieval is simplified if the active vocabulary is kept small but expressive.

[6] http://www.celera.com

[7] Not that BusTUC never encounters ambiguous or complex questions.

protein binds DNA/protein
protein dimerizes with protein
protein interacts with protein
gene codes protein
protein represses gene
protein recruits gene
protein complexes with protein
protein regulates gene/protein
protein inhibits gene/protein
protein associates with protein
protein phosphorylates protein
protein dephosphorylates protein
protein inactivtes gene/protein
protein induces gene/protein

Figure 11.1: Key relationships in the GeneTUC system.

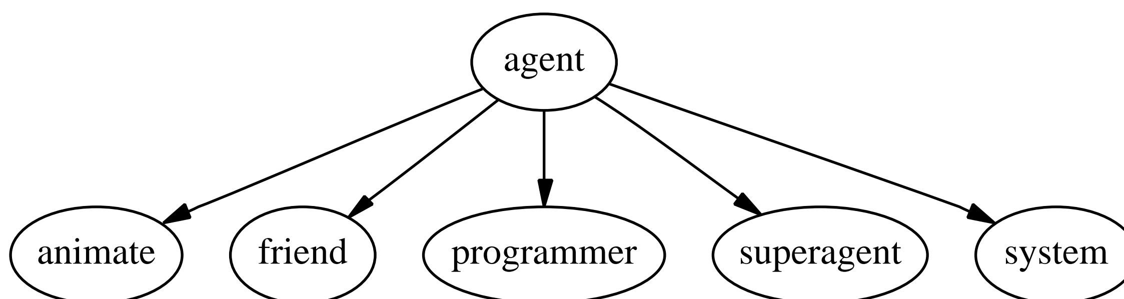

Figure 11.2: TUC's agent subclasses.

The reasons for focusing on interactions between genes and proteins rather than focusing on the genes themselves, are many. The most important of which, is that the genes and proteins direct the processes in the living cell. Understanding how these interactions are conducted is tantamount for understanding how the organism works.

11.3.2 Agents

In TUC's ontology, only agents and their subclasses are considered to actively pursue goals or trying to attain effects. Agents are thus thought of as mainly humans and animals, and the roles they may fill. However, TUC also denotes a system, in the computer system sense, an agent. Although seemingly self-flattering, this is required for answering questions regarding TUC's own operation. The immediate subclasses of agent in the original TUC is shown in Figure 11.2.

In the biomedical terminology, a number of concepts are treated as though they are agents, behaving actively. In the following example, a mutant protein *prevents* adipogenesis, the genetic command for fat production:

> Furthermore, the mutant protein prevented thiazolidinedione-induced adipogenesis in 3T3-L1 cells, whereas expression of recombinant wild-type PPARgamma2 promoted adipogenesis.

Preventing something from happening is normally considered an act of intent, and it would not be unreasonable to say that only sentient beings are capable of acting on intent. One could expand the semantic base saying that proteins are allowed to prevent, and leave the proteins elsewhere in the ontology. This would require a large number of such additions to be made, not only for quite a few protein - action verb combinations, but also for DNA, RNA, genes and so forth. Rather,

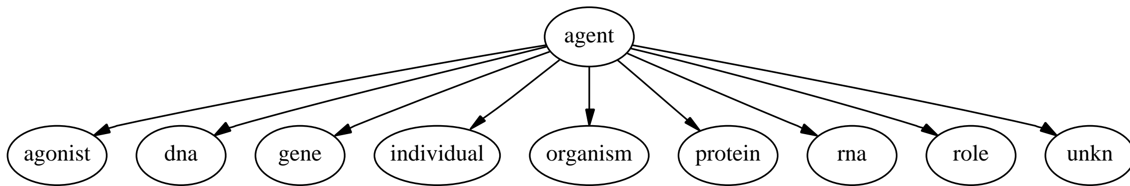

Figure 11.3: Those subclasses of GeneTUC's agent concept not found in TUC.

the notion of agents was extended somewhat, leaving us with a larger class of agents, some of which are shown in Figure 11.3. Note that many of the concepts are placed in multiple loci in the ontology (not shown in the figure).

11.3.3 Unfamiliar Words

Early versions of GeneTUC required all words encountered in the input corpus to be known beforehand. This developed to be somewhat of an Achilles' heel for the system. Scientists come up with new names for genes, proteins and substances when they are discovered. Keeping up with the nomenclature does not seem feasible. The number of *hapax legomena*[8] in the test corpus was very high. As the vocabulary was extended, the number of such words remained alarmingly high, as the cost-benefit ratio for adding such words is very low. On inspection, the hapaxes were found to mainly consist of gene and protein names.

Fukuda et al. ([**?**]) suggest a method for identifying previously unknown protein names in a text, using an inference mechanism and knowledge on how such substances are named. Although highly successful in its domain, the method is not easily modified to recognising protein and gene names both, and keeping them apart. In Figure 11.3, a concept "unkn" has been introduced as an agent. This concept encompasses all words not recognised by GeneTUC's lexical analysis. According to the presumption that most unknown words in our input are proteins, genes or subclasses thereof, making agent their common ancestor is a satisfactory solution.

Even still, the gene and protein name databases should be kept as up-to-date as possible, as the quality of the output suffers when introducing unrequired generality. The classification of unknowns as agents is merely a temporary remedy and not a permanent solution. In addition, not all unknowns are genes or proteins, some are not even nouns. Therefore, this method must be employed with some caution.

11.4 Other Changes

The strategy chosen for expanding the semantics, and consequently the grammar, was based on the inclusion-exclusion principle. In short, the semantics was made very general at first, accepting almost all possible combinations of words, although the sentence still had to be grammatically correct. Later, the semantics have been gradually constrained, hopefully giving a higher quality on the output.

11.4.1 Vocabulary

The preliminary version of GeneTUC [**?**], contained about 4,800 words, not including domain-specific proper names, i.e., gene and proteins names and their aliases. It was therefore clear that the vocabulary needed expansion, preferably import of dictionaries from existing sources, rather than manual entering of single words.

[8]From Greek, meaning "words mentioned once", i.e., words encountered only one time in a corpus.

The WordNet[9] was chosen, due to its availability and size. The complete lists of adjectives and adverbs were imported directly into GeneTUC, adding 43,000 new words to the vocabulary. The words were added using the highest level of generality, that is all adjectives can act on all nouns and all adverbs can act on all verbs, making the semantics less strict. Although this calls for re-evaluation of the semantics on a later stage, it abides to the inclusion-exclusion principle stated above.

Adjectives and adverbs are easily imported, their low semantic significance (in this context) makes it convenient to accept unneeded generality. Nouns and verbs are harder, if the added generality of the semantics is to be kept under reasonable constraints. Unnecessary generality in the ontology is undesirable, thus the adding of nouns will most likely have to be done by hand. Likewise, verbs have to be added manually to ensure that only the right concepts can act as subjects and objects for a given verb. 1,400 words have been added manually to the vocabulary.

11.4.2 Standard Complements

Prepositional and adverbial phrases act as modifiers on the noun, verb and adjectives (see Chapter 2). As numerous such complements may modify each noun, verb or adjective, having to enter them all in by hand is very unwelcome.

Subsequently, the notion of standard complements was resorted to. This means that GeneTUC allows for a number of complements of the form "<Prep> <Concept>", where <Prep> is a lexeme from the class of prepositions, and <Concept> is a lexeme from the ontology, e.g., "of thing" and "to place".

11.5 Conclusions

GeneTUC has matured over the months since its beginning in [**?**]. The original system, essentially just BusTUC without the bus tables, had a very restricted vocabulary (less than 5000 words). This vocabulary was primarily aimed at bus route queries and everyday matters, such as "When is the next bus to the airport?" and "What time is it?". A restriction set upon the development of GeneTUC is that it is to maintain "sideways" compatibility with BusTUC, i.e., the systems are to remain equivalent, bar some of the vocabulary and semantics. Experience has shown that this restriction is not always convenient; separating the two applications further would have been desirable on numerous occasions. A compromise has been to introduce a number of flags which enable or disable different features of the system.

11.5.1 TUC

Making the GeneTUC system has some secondary effects on the general development of the TUC architecture. Enhancing GeneTUC's performance in most directions implies enhancing that of TUC's, due to the restriction on sideways and backwards compatibility (with the original TUC framework). Some of the vocabulary and semantics are, however, domain-specific.

Thus far we have seen remarkably few changes to the grammar. A few errors have been discovered, some of which having severe consequences, but all in all the grammar has kept up with the new semantics in a remarkable way. Some flaws needing to be corrected have been uncovered, and the grammar is still not complete, but it has shown to be very robust and covering a great number of sentence patterns.

Increase in the grammar size has been relatively small, mostly because there have been made very few changes to the grammar. The scalability of the grammar has therefore not really been assessed.

However, the architecture seems to scale very well with the increased vocabulary and semantics. The larger permanent database has not made the application run slower, and the increase

[9] Available at http://www.cogsci.princeton.edu/~wn/

in memory usage is well within what is acceptable. With further growth, it may be convenient to distribute the semantics into more files, instead of piling it all up into one huge file. Sicstus Prolog does not, unfortunately, allow for a single predicate[10] in multiple files, but there are ways to circumvent this restriction.

Even though it has not been impeded by the growth in size, execution speed is still an issue. The parsing time has been improved, but is still too high for high-volume input (our training set is a minute subset of the the abstracts found in the Medline). On the other hand, input needs only be run through the system once, relaxing the speed requirement.

11.5.2 GeneTUC

Inclusion of external dictionaries and manually adding thousands of words to the vocabulary has sent GeneTUC's results soaring over the last months. Some of this success has come at the expense of a more general semantics, which is obviously not good or wanted. However, it is the author's belief that restricting the semantics later in a top-down manner is easier and more appropriate than creating strict semantics right from the start, mainly due to the better overview the top-down method gives.

In terms of appropriateness, the high growth rate of the number of successful parses imply that GeneTUC may become a useful tool for NL processing of biomedical texts. Nonetheless, the currently low rate of success suggests that much work still lies ahead before the results are comparable to other systems in the same domain. The performance on unseen material is acceptable, which means that using a standard training set and test set technique will successfully develop the application further.

It seems right to focus on the abstracts of articles. Article titles have too little content to provide useful information, and are often ungrammatical (lacking a verbal). Complete articles contain scores of information, but it would be difficult to separate interesting knowledge from "noise". Abstracts have the ideal combination of high information density and relevance, along with linguistic compliance.

A direct comparison of GeneTUC and the competing systems is hard, because our source material regarding these are scarce in terms of numbers, bar recall and precision according to MUC [**?**]. At present, GeneTUC's recall is substantially lower than that of the others; 8.3% versus 51% for ARBITER [**?**] and 29% for Highlight [**?**] (numbers for EDGAR not available). GeneTUC's precision has not been measured.

One must also bear in mind that the results presented in this report does not do GeneTUC proper justice. The real test of the NL understanding strategy comes when IE is taken beyond extracting simple facts from independent sentences.

[10] e.g., all adjectives.

Appendices

List of Appendices.

A scanner for Prolog parser

Metaparser for phrase structured languages

Semantic agreement check

Early parsing in Prolog

Feature unification in Prolog

A micro version of SHRDLU

A micro version of CHAT-80

A combinatory kernel of NRL

Sample questions to Bustuc

Sample sentences for GeneTUC

Scanner for Prolog grammars

The scanner is listed under the reposititory directory

```
http://www.idi.ntnu.no/~natlang/readin.prolog
```

The details of the listing is not important however.
The purpose of the scanner is to convert lines of texts to lists of atomic symbols.

Workings of the scanner:

The scanner has two entry points:
scanner1 and scanner2.

scanner1 will give a prompt (E:) and then accept one line of text. At the end of line, the scanner will be activated and return an atom list.

scanner2 is similar, but may accept text over several lines. First when a line is ended with a terminator character ('.', '!' or '?'), the scanner will return the atom list.

Prolog programs using this scanner will work as in the following example:

```
parseS(SyntS) :-
    scanner1(List),
    s(SyntS,List,[]).
```

This is invoked as

```
?- parseS(ST).

E: john loves mary

ST = s(np(proper_noun(john),vp(verb(loves),np(proper_noun(mary)))))
```

if that was the output.

Metaparser for phrase structured languages

See the programs under the Resource directory (PRO)

```
metaparse.prolog  %% context free

cgmi.prolog       %% context sensitive

readin.prolog
```

```
/*

Example of use:

Include the metagrammar into the parser

Ensure that the prolog program readin.pl is available

% sicstus
?-run.

E: the cat likes a fish .

sentence
   ...


yes
?-

*/
```

Semantic Agreement Check

Se the program

```
semagree.prolog
```

under the resource directory. It goes together with the program

```
metaparse.prolog
```

Early parsing in Prolog

A program containing a demo of Early parsing in Prolog is very well described im Jurafsky&Martin ch 10.4 .

An implementation can be found in the Resource directory

```
earlyparse.prolog
```

The implemantation is based on the Production system Proxy, which is found in

```
proxy.prolog
```

Proxy is described in the NOTES directory as the note

```
Proxy.txt
```

To run the system, do as follows:
Copy the programs mentioned and change extension to .pl
There is a sample grammar within that can be modified.

```
% sicstus
?-[proxy,earlyparse].
?-trace := 2.
?-run.

.......
```

The output is also listed in

```
earlyrun
```

Probabilistic Context Free Parsing

Parsing with Probabilistic Context Free Grammars (PCFG) is very well described in Jurafsky & Martin: SaLP.

In the following, we shall demonstrate the same principles in the context of a top down Meta-parser for CFG grammars

The program is also available under the Repository directory under name

```
pcfg.prolog
```

Program listing.

```
%% FILE pcfg.pl
%% SYSTEM NATLANG
%% CREATED TA-010128
%% REVISED TA-010204

% Prolog metaparser for Contexft Free Languages

:-op(1100,xfy,--->).
:-op(1150,xfx,prob).

metaparse(S,List):-
    parse(S,_,PSTree,List,[]),
    prettyprint(PSTree).

parse(S,Prob,prod(S,Prob,PSTree)) -->
    {S ---> PS prob P1 },
    parse(PS,Prob1,PSTree),
    {Prob is P1 * Prob1}.

parse((First,Rest),Prob,(FirstTree,RestTree)) -->
   parse(First,Prob1,FirstTree),
   parse(Rest,Prob2,RestTree),
   {Prob is Prob1*Prob2}.


parse([Word],1.0,[Word]) --> [Word].

parse([],1.0,[]) --> [].

parse({Cond},1.0,{Cond}) --> {Cond}.
```

```
prettyprint(ST):-
    nl,
    pretty(0,ST),
    nl.


pretty(N,prod(X,Prob,Y)):- !,
    tab(N),write(X),write(' '),writeprob(Prob),nl,N2 is N+2,
    pretty(N2,Y).

pretty(N,(X,Y)):- !,
    pretty(N,X),
    pretty(N,Y).

pretty(N,ST):-tab(N),write(ST),nl.

writeprob(P):- format("   PROB ~2E ",P).

% Bigger example

sentence       ---> noun_phrase,verb_phrase       prob 0.8 .
sentence       ---> verb,noun_phrase,comparison   prob 0.2 .

noun_phrase    ---> noun                      prob 0.2 .
noun_phrase    ---> proper_noun               prob 0.3 .
noun_phrase    ---> article,noun              prob 0.4 .
noun_phrase    ---> composite_noun            prob 0.1 .

verb_phrase    ---> verb,object           prob 0.7 .
verb_phrase    ---> verb,comparison           prob 0.3 .

object     ---> noun_phrase               prob 1.0 .

comparison     ---> comparator,noun_phrase    prob 1.0 .

composite_noun ---> proper_noun,noun          prob 1.0 .

verb ---> [time]   prob 0.1 .
verb ---> [flies]  prob 0.4 .
verb ---> [like]   prob 0.5 .

comparator ---> [like]    prob 1.0 .

noun ---> [time]          prob 0.5 .
noun ---> [flies]         prob 0.2 .
noun ---> [arrow]         prob 0.3 .

proper_noun ---> [time]   prob 0.1 .
proper_noun ---> [john]   prob 0.9 .

article ---> [an]         prob 1.0 .

%------------------------------------
```

```
run:-
    metaparse(sentence,[time,flies,like,an,arrow]),fail.
```

```
sentence    PROB 1.15E-03
  noun_phrase    PROB 1.00E-01
    noun    PROB 5.00E-01
      [time]
  verb_phrase    PROB 1.44E-02
    verb    PROB 4.00E-01
      [flies]
    comparison    PROB 1.20E-01
      comparator    PROB 1.00E+00
        [like]
      noun_phrase    PROB 1.20E-01
        article    PROB 1.00E+00
          [an]
        noun    PROB 3.00E-01
          [arrow]


sentence    PROB 3.46E-04
  noun_phrase    PROB 3.00E-02
    proper_noun    PROB 1.00E-01
      [time]
  verb_phrase    PROB 1.44E-02
    verb    PROB 4.00E-01
      [flies]
    comparison    PROB 1.20E-01
      comparator    PROB 1.00E+00
        [like]
      noun_phrase    PROB 1.20E-01
        article    PROB 1.00E+00
          [an]
        noun    PROB 3.00E-01
          [arrow]


sentence    PROB 6.72E-05
  noun_phrase    PROB 2.00E-03
    composite_noun    PROB 2.00E-02
      proper_noun    PROB 1.00E-01
        [time]
      noun    PROB 2.00E-01
        [flies]
  verb_phrase    PROB 4.20E-02
    verb    PROB 5.00E-01
      [like]
    object    PROB 1.20E-01
      noun_phrase    PROB 1.20E-01
        article    PROB 1.00E+00
          [an]
        noun    PROB 3.00E-01
          [arrow]


sentence    PROB 9.60E-05
```

```
verb    PROB 1.00E-01
  [time]
noun_phrase    PROB 4.00E-02
  noun    PROB 2.00E-01
    [flies]
comparison    PROB 1.20E-01
  comparator    PROB 1.00E+00
    [like]
  noun_phrase    PROB 1.20E-01
    article    PROB 1.00E+00
      [an]
    noun    PROB 3.00E-01
      [arrow]
```

Feature Unification in Prolog

The program is also available under the Repository directory under name

```
features.prolog
```

The program listed here , with a sample grammar, is a rudimentary implemenation of the basics of feauture unification in Prolog (FUP).

The addition is that an operator === implements feature expression unification as an extra condition. Besides the program for prettyprinting features, the essence lies in the predicate unify_feature.
Feature unification allows feature structure lists to be instantiated during the parsing process.

A limitation is that the feature structures as such are not unified. Rather, it is feature expression that are unified. That means that in priniciple, only one attribute value pair is unified at a time. The three golden rules are:

- if the attributes are identical, the values are unified recursively.
- if the attributes are identical, but the values are not unifiable, the whole unification fails
- if the attributes are different, they are placed as alternatives

Technicallly, the feature lists is kept as open ended lists that allow these extensions, and the features structures are "grown" during the parsing.

Whatever, you should study the listing and test it out to get a feeling.

The Feature Unification program

```
%% File features
%% SYSTEM NATLANG
%% CREATED TA-010208
%% REVISED TA-010210

% FUP Feature Unification in Prolog

% The examples implements as closely as possible the
% examples in Jurafsky&Martin S&LP Ch. 11.1-11.3


:-op(900,xfx,===).
:-op(400,yfx,'::').
```

```
:-op(300,xfy,':').

%% Head Features and Subcategorisation


/*   Legend

     g0(G0) --> g1(G1),... gn(Gn),
     {Gi :: f1:f2:...:fk === Fval}, ...


  gi      Syntactic category as in DCG

  Gi      Feature structure of g

  fi      feature

  Gi :: f1:f2:...:fk     Feature expression

  f1:f2:...:fk     Feature path

  ===     Feature unification symbol

  Fval    Feature value     Atom or Feature expression

*/

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:-compile(readin).

run :- scanner1(List),
       parse(List).

parse(List) :-
    s(S,List,[]),
    writefeature(S).

FX1 === FX2 :-
    unify_feature(FX1,FX2).


% Unify Feature Expressions

unify_feature(FS1::FP1,end):-
   feature(FP1,FS1,F1),
   !,
   F1=end. %% no alternatives

unify_feature(FS1::FP1,Atom):-
   atomic(Atom),
   feature(FP1,FS1,F1),
   !,
   F1=[Atom|_]. %% allow alternative
```

```
unify_feature(FS1::FP1,FS2::FP2):-
   feature(FP1,FS1,F1),
   feature(FP2,FS2,F2),
   !,
   F1=F2.

feature(Head:Body,LList,X):-
    member(Head:Rest,LList),
    !,
    feature(Body,Rest,X).

feature(Head,LList,X):-
    member(Head:X,LList).

member(X,V):-var(V),!,V=[X|_].
member(X,[Y|Z]):-X=Y;member(X,Z).

% Prettyprint Feature

writefeature(F):-
    nl,nl,
    writef(0,F),
    write('-------------'),
    nl.

writef(_N,[]):-!.

writef(N,[A]):-
     !,
     writef(N,A).


writef(N,A:B):-
    !,
   tab(N),
   write(A),write(':'),
   nl,
   N3 is N+3,
   writef(N3,B).

writef(N,[F|G]):-
    !,
   tab(N),
   write('['),nl,
      N3 is N+3,
      writefl(N3,[F|G]),
   tab(N),
   write(']'),nl.

writef(N,A):-
    tab(N),write(A),nl.

writefl(_,[]):-!.
```

```
writefl(N,[F|G]):-
    !,
   writef(N,F),
   writefl(N,G).
```

Sample FUP grammar

```
s(S) --> aux(Aux),np(NP),vp(VP),
    {S :: cat  === s},
    {S :: head === VP ::head},
    {Aux :: head:agreement === NP :: head:agreement},
    {VP :: head:agreement:number === pl}.


s(S) --> np(NP),vp(VP),
    {S :: cat  === s},
    {S :: head === VP :: head},
    {NP :: head:agreement === VP :: head:agreement}.


np(NP) --> det(Det),nominal(Nominal),
    {NP ::  cat  === np},                  % ???
    {NP ::  head === Nominal :: head},
    {Det :: head:agreement === Nominal :: head:agreement}.


np(NP) --> nominal(Nominal),
    {NP :: head === Nominal :: head}.

aux(Aux) --> [do],
    {Aux :: head:agreement:number === pl},
    {Aux :: head:agreement:person === 3}.

aux(Aux) --> [does],
    {Aux :: head:agreement:number === sg},
    {Aux :: head:agreement:person === 3}.

det(Det) --> [this],
    {Det :: head:agreement:number === sg}.

det(Det) --> [these],
    {Det :: head:agreement:number === pl}.

vp(VP) --> verb(Verb),
    {VP :: head === Verb :: head},
    {VP :: head:subcat === intrans}.

vp(VP) --> verb(Verb),np(_NP),
    {VP :: head === Verb :: head},
    {VP :: head:subcat === trans}.
```

```
vp(VP) --> verb(Verb),np(_NP1),np(_NP2),
    {VP :: head === Verb :: head},
    {VP :: head:subcat === ditrans}.

%% Lexicon part

verb(Verb) --> [serve],
    {Verb :: head:agreement:number === pl},
    {Verb :: head:subcat           === trans},
    {Verb :: head:subcat:first:cat === np},
    {Verb :: head:subcat:second    === end}.

verb(Verb) --> [serves],
    {Verb :: head:agreement:number === sg},
    {Verb :: head:subcat           === trans},
    {Verb :: head:subcat:first:cat === np},
    {Verb :: head:subcat:second    === end}.

verb(Verb) --> [exists],
    {Verb :: head:agreement:number === sg},
    {Verb :: head:subcat           === intrans},
    {Verb :: head:subcat:first     === end}.

verb(Verb) --> [leaves],
    {Verb :: head:agreement:number  === sg},
    {Verb :: head:subcat            === trans},
    {Verb :: head:subcat:first:cat  === np},
    {Verb :: head:subcat:second:cat === pp},
    {Verb :: head:subcat:third      === end}.

verb(Verb) --> [want],
    {Verb :: head:agreement:number   === pl},
    {Verb :: head:subcat             === vpto},
    {Verb :: head:subcat:first:cat   === vp},
    {Verb :: head:subcat:first:form  === infinitive},
    {Verb :: head:subcat:second:cat  === np},
    {Verb :: head:subcat:third       === end}.


noun(Noun) --> [flight],
    {Noun :: head:agreement:number === sg}.

noun(Noun) --> [flights],
    {Noun :: head:agreement:number === pl}.

noun(Noun) --> [breakfast],
    {Noun :: head:agreement:number === sg}.

nominal(Nominal) --> noun(Noun),
    {Nominal :: head === Noun :: head}.
```

Example of use

```
% sicstus
vier% sicstus
SICStus 3.8.4 (sparc-solaris-5.7): Mon Jun 12 18:41:59 MET DST 2000
Licensed to idi.ntnu.no
| ?- [features].
...............

| ?- run.

E: does this flight serve breakfast


[
   cat:
      s
   head:
      [
         agreement:
            number:
               pl
         subcat:
            [
               trans
               first:
                  cat:
                     np
               second:
                  end
            ]
      ]
]
-------------

yes



E: this flight serves breakfast


[
   cat:
      s
   head:
      [
         agreement:
```

```
            number:
               sg
         subcat:
            [
               trans
               first:
                  cat:
                     np
               second:
                  end
            ]
      ]
]
-------------

E: these flight serves breakfast

no
|
E: these flights serves breakfast
no

E: these flights serve breakfast


[
   cat:
      s
   head:
      [
         agreement:
            number:
               pl
         subcat:
            [
               trans
               first:
                  cat:
                     np
               second:
                  end
            ]
      ]
]
-------------

yes
| ?-

E: these flights serves

no
| ?-
```

```
E: this flight exists


[
   cat:
      s
   head:
      [
         agreement:
            number:
               sg
         subcat:
            [
               intrans
               first:
                  end
            ]
      ]
]
-------------

yes
|
E: this flight exists breakfast

no
| ?-
```

A micro version of SHRDLU

The following chapter will explain and list a program that implements a tiny subset of the famous program SHRDLU. It is included, both because it is an interesting application, and because it demonstrates many of the themes in computational semantics that are described in that chapter.

The programs described are also available on the repository with the file

```
shrdlu.prolog
```

The program consists of 4 parts:

- A Prolog grammar `shrdlu` for a language subset that covers the box world, and translates the language into a logic knowledge base.
- A situation calculus planner `sitcalc` that can perform simple planning operations in the box world.
- A plan executor that can perform the actions of the plan, and change the internal knowledge base accordingly. This is actually not implemented here.
- A box world `boxcalc` operation definition that defines the semantics of the operators.
- A scanner `readin` that converts text into lists of atoms. This is common to all applications, and is not described here.

The situation calculus planner will not be described in detail, as it is related to a course in knowledge systems. It is just included because it gives the language implementation an interesting target application. The AI inclined student is of course encouraged to study it and extend it to other applications.

Overview of shrdlu with examples

```
E: c is free and c is on a and a is on the floor.

((isa(c,box)and clear(c))and isa(c,box)and isa(a,box)and on(c,a))
  and isa(a,box)and isa(thefloor,floor)and on(a,thefloor)



E: b is on the floor and b is free.

(isa(b,box)and isa(thefloor,floor)and on(b,thefloor))
 and isa(b,box)and clear(b)
```

```
E: how come a is on b.

how(isa(a,box)and isa(b,box)and on(a,b))


Trying:
. . .


start
move(c,a,floor)
move(a,thefloor,b)


: how come a is on b and b is on c.

how((isa(a,box)and isa(b,box)and on(a,b))and
 isa(b,box)and isa(c,box)and on(b,c))


Trying:
. . . .



start
move(c,a,floor)
move(b,thefloor,c)
move(a,thefloor,b)


E: q.

yes
| ?-
```

Program listing shrdlu

```
:- op(900,xfx,  =>).
:- op(799,xfy,  :). % infix lambda
:- op(701,xfy,  or ).
:- op(700,xfy,  and ).
:- op(600,fx,   not).

:- dynamic fact/1,  %% Dynamic fact
           skolem/1.
```

```
s(P) --> statements(P).

s(P) --> command(P).

% s(P) --> question(P). % imlicit with determiner = which


statements(P) -->
   statement(P1),
   restm(P1,P).

restm(P1,Q) -->
   [and],
   statement(P2),
   restm(P1 and P2,Q).

restm(P,P) -->  terminator.

command(how(P)) --> [how],can,statements(P).




statement(SemS) -->
    noun_phrase(SemNP),
    verb_phrase(SemVP),
    {apply(SemNP,SemVP,SemS)}. % SemS = SemNP(SemVP)

noun_phrase(SemNP) -->
    determiner(SemDet),
    noun_expression(SemNX),
    {apply(SemDet , SemNX , SemNP)}.

noun_phrase(SemNP) --> % box b
    noun(SemNO),
    name_exp(SemNA),
   {include(SemNA,SemNO,SemNP)}.


noun_phrase((NA:P): isa(NA,NO) and P   ) --> %  b
    name(NA),
    {isa(NA,NO)}.

name_exp((X:P):P) --> name(X).
name_exp((X:P):P) --> number(X).

noun_expression(SemNX) -->
    adjnoun(SemAN),
    rel_clause(SemRelC),
    {compose2(SemAN ,SemRelC  , SemNX)}.

adjnoun(SemAN) -->
```

```
    adj(SemA),
    adjnoun(SemAN1),
    {conjoin(SemAN1,SemA,SemAN)}.

adjnoun(P) -->
   noun(P).



verb_phrase(XP) --> [is],
    adj(XP).

verb_phrase(XP) -->
    intrans_verb(XP).

verb_phrase(VP) -->
    trans_verb(XYP),     % loves
    noun_phrase(NP),     % mary
   {compose(XYP,NP,VP)}. % VP = XYP o NP

verb_phrase(VP) -->
    noun_phrase(NP),     % mary
    trans_verb(XYP),     % loves
   {compose(XYP,NP,VP)}.


rel_clause((X:P1):(P1 and P2)) -->
    [that],
    verb_phrase(X:P2).

rel_clause((_:P):P) --> [] .

determiner((X:P1):(X:P2): (forall(X : P1=>P2))) --> [every] .

determiner((X:P1):(X:P2): (exists(X : P1 and P2))) --> [a] | [the] .

determiner((X:P1):(X:P2): (which(X: P1 and P2))) --> [which] .


terminator --> ['.'] | ['?'] | ['!'] .

name(thefloor) --> [the],[floor]. %% Technical

name(X)  --> [X],{isa(X,_Box)}. % to be defined in the database

number(N) --> [nb(N)].   % the form of number produced by readin.pl



%%% Compositional Operators %%%%%%%%%%%%%%%%%%%%

apply(A:B, A, B).  % apply
```

```
conjoin(X:P, X:Q, X:(P and Q)). % conjunct

compose(X:Q, Q:P, X:P). %% chain rule

compose2((X:P) , (X:P):Q , X:Q).

include((X:VP):P,X:N,(X:VP):(N  and  P)).


%%%%%%%%%  DICTIONARY  %%%%%%%%%%%%%%%%%%%%%%%%%%


noun(X: isa(X,box)) --> [box].

% noun(X: floor(X)) --> [floor]. %% made into a name for p.p.

trans_verb(X:Y:exceeds(X,Y)) --> [exceeds].
trans_verb(X:Y:has(X,Y))     --> [has].
trans_verb(X:Y:loves(X,Y))   --> [loves].

trans_verb(P) --> be,prep(P).

intrans_verb(X:exists(X)) --> [exists].
intrans_verb(X:lives(X)) -->  [lives].

intrans_verb(A) --> be,adj(A).

adj(X: not P) --> [not],adj(X: P).
adj(X: not clear(X)) --> [occupied].

adj(X: blue(X))  -->  [blue].
adj(X: big(X))   -->  [big].
adj(X: clear(X)) -->  [clear].
adj(X: clear(X)) -->  [free]. % synonym

adj(X: empty(X)) -->  [empty].


prep(X:Y:on(X,Y)) --> [on].
prep(X:Y:on(X,Y)) --> [above].
prep(X:Y:on(Y,X)) --> [below].

be --> [is].
be --> [be].
be --> [are].
% etc

can --> [can].
can --> [come].

%%%%%%%%%  DATA BASE  %%%%%%%%%%%%%%%%%%%%%%%%%%%

%%  Names  %%
```

```
isa(a,box).
isa(b,box).
isa(c,box).

isa(thefloor,floor). %% Technical




reset :-
    retractall(fact(_)),
    retractall(skolem(_)).

run :-

       repeat,
       nl,
       ask(user,L),
       ( L=[q|_],! ;  %% q.  means quit
         (analyse(L),
       fail) ).

analyse([co,F|_]):-
    nl,
    compile(F),
    !.

analyse([spy,F|_]):-
    nl,
    (spy F),
    !.

analyse(L) :-
s(P,L,[]),
   !,
nl,
prettyprint(P),
nl,
evaluate(P).

analyse(_L) :-
nl, write('     E r r o r '), nl.


prettyprint(P) :-
numbervars(P,23,_), % start from X
write(P),nl,
false;  % release bindings
true.


%% Rudimentary evaluator
```

```
%% Evaluator of queries

evaluate(how(P)):-
    !,
    translate(P,Q),
    solve(Q,Solution),
    writeplan(Solution),
    execute(Solution).

evaluate(which(X:P)) :- !,   % interrogative sentences (queries)
( evalq(P),
  write(X),nl,
  false;
  true ).



%% Evaluator of statements


evaluate(A):-
   evals(A).
evaluate(X).                  % others

evals(A and B) :- !, evals(A),evals(B).
evals(exists(X:Y)):- !,bind(X),evals(Y).
evals(A) :- remember(A).


bind(sk(N)):-
    retract(skolem(N)),
    N1 is N + 1,
    assert(skolem(N1)).

remember(X):-fact(X),!
             ;
             assert(fact(X)).


evalq(exists(_:P)) :- !,evalq(P).

evalq(forall(_: P=>Q)) :- !,evalq(not (P  and  (not Q))).

evalq(X  and  Y) :- !,evalq(X),evalq(Y).

evalq(not X) :- \+ evalq(X).

evalq(X) :-fact(X).



%%%%%%%%%  TEST EXAMPLE  %%%%%%%%%%%%%%%%%
```

```
translate(P,Q):-
    trans(P,Q). %% dummy

trans(exists(_:P),Q) :- !,
    trans(P,Q).

trans(forall(_: P=>Q),R) :- !,
    trans(not (P  and  (not Q)),R).

trans(X  and  Y,X1 and Y1) :- !,
    trans(X,X1),trans(Y,Y1).

trans(not X,not X1) :-
    trans(X,X1).

trans(X,X).

%%%%%%%%%%%%%%%%%%%%%%%

execute(_):- !.
% Shall perform the operations on the internal state (fact)
% Not implemented

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Axioms for box moving

```
%% Facts true in all states

fact00(clear(floor)).

% Effect Axioms

consequence(on(X,Z),move(X,Y,Z),
    clear(X)  and
    on(X,Y)  and
    clear(Z)  and
    dif(X,floor)  and
    dif(X,Y)  and
    dif(Y,Z)  and
    dif(X,Z)).

consequence(clear(Y),move(X,Y,Z),
    clear(X)  and
    on(X,Y)  and
    clear(Z) and
    dif(X,floor)  and
    dif(X,Y)  and
    dif(Y,Z)  and
```

```
    dif(X,Z)).


% Frame Axioms

invariant(isa(_,_),_):-!.
invariant(blue(_),_):-!.
% .. etc


invariant(on(U,_V),move(X,_Y,_Z)) :-
    dif(U,X).
invariant(clear(U),move(_X,Y,Z)):-
    dif(U,Y),dif(U,Z).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A Planner based on the Situation Calculus

```
%% Implementation is dependent on
%% the coroutining facilities in SICStus Prolog,
%% The predicate dif/2 tests for non unifiablity,
%% but delays the goal if not decidable.



%% Axioms for planning

holds(Condition1 and Condition2, State) :- !,
    holds(Condition1,State),
    holds(Condition2,State).

holds(not Condition,S) :- !,
    \+ holds(Condition,S).

holds(Condition,_) :-
     always(Condition),    % axiomatic,
     !.                    % situation independent


holds(Condition,do(State,Action)) :-
    holds(Condition,State),           % true already
    invariant(Condition,Action).      % no action made it false

holds(Condition,do(State,Action)) :-
    consequence(Condition,Action,Preconditions),  % an action made it true
    holds(Preconditions,State).


holds(Condition,_) :-
    fact00(Condition).              % true in all situations
```

```
holds(Condition,start) :-
    fact(Condition).

always(true).
always(dif(X,Y)) :- dif(X,Y).     % axiom of difference

%% Axioms for search strategy
%  Depth First Iterative Deepening

try(_,start) :- write('. ').


try(N,do(State,_)) :-
   M is N-1,
   M >= 0,
   try(M,State).


solve(G,S) :-                    Max = 10,
    nl,
    write('Trying: '),nl,
    try(Max,S),
    holds(G,S),
    !.


%% Utilities

listall(X,P) :- P,write(X),nl,false;nl,nl.

writeplan(start):-
    !,
    nl,
    write('Nothing needs to be done').

writeplan(P):-
   nl,
   wp(P).

wp(do(S,P)):-
   !,
   nl,
   wp(S),
   wp(P).

wp(S1):-
   write(S1),nl.
```

CHAT-80 revisited

This section will review the classical system CHAT-80, and make a reimplementation in Consensical grammar.

We will combine the context sensitive categorial grammar with a computational semantics that is defined erlier. However, to make the system simple, we shall focus on the code generation, and leave out agreement checks of all kinds.

The grammar is grossly simplified and takes no agreement checks into account. The grammar viewed as a context sensititive grammar is meant to be a comparable subset of the grammar of Chapter 3.

The programs are also available in Repository (chat80.prolog, cgmi.prolog, readin.prolog,chatrun)

```
sentence(Sem) --->
    question(Sem),
    terminator.
sentence(Sem) --->
    statement(Sem),
    terminator.

statement(SemSt) --->
     noun_phrase(SemNP),
     verb_phrase(SemVP),
    {apply1(SemNP,SemVP,SemSt)}.

question(Sem) ---> which_question(Sem).
question(Sem) ---> what_question(Sem).
question(Sem) ---> yn_question(Sem).


% which country does china border ?

which_question(Sem) --->
         [which],
         noun_phrase(SemNP)\ [which],
         aux,
         statement(Sem)/noun_phrase(SemNP).

% which country borders  china ?

which_question(Sem) --->
         [which],
         noun_phrase(SemNP)\ [which],
         statement(Sem)\noun_phrase(SemNP).
```

```
what_question(which(X:SemX)) --->
     [what],[is],noun_phrase(Sem),
     {Sem= ((X:true):SemX)}.


who_question(Sem):-
    which_question(Sem)\([which],[agent]).

yn_question(test(Sem)) ---> aux, statement(Sem).




noun_phrase(Sem) ---> proper_noun(Sem).
noun_phrase(Sem) ---> proper_number(Sem).

noun_phrase(SemNP) --->
    determiner(SemDet),
    nominal_expression(SemX),
    {apply1(SemDet,SemX,SemNP)}.

nominal_expression(SemX) --->
    nominal(SemNom),
    noun_modifiers(SemMod),
    {apply3(SemNom,SemMod,SemX)}.

noun_modifiers(SemMods) --->
    noun_modifier(SemMod1),
    noun_modifiers(SemMods1),
    {apply3(SemMod1,SemMods1,SemMods)}.

noun_modifiers(_:true) ---> [].

noun_modifier(Sem) ---> prep_phrase(Sem).
noun_modifier(Sem) ---> rel_clause(Sem).
noun_modifier(Sem) ---> gerund_phrase(Sem).
noun_modifier(Sem) ---> whose_phrase(Sem).



gerund_phrase(SemG) --->
    verb_ing(Border),
    noun_phrase(SemNP),
    rel_clause(SemG)\
       ([that],[Border],noun_phrase(SemNP)).

whose_phrase(Sem) ---> [whose],noun(SemN),
     rel_clause(Sem) \
     ([that],[has],[a],noun(SemN),[that]).


nominal(Sem) ---> noun(Sem). % i.e. noun*
```

```
verb_phrase(X) ---> [is],prep_phrase(X).

prep_phrase(SemPP) --->
    preposition(SemP),
    noun_phrase(SemNP),
    {apply2(SemNP,SemP,SemPP)}.


verb_phrase(SemVP) --->
    trans_verb(SemTV),
    noun_phrase(SemNP),
   {apply2(SemNP,SemTV,SemVP)}.

verb_phrase(Sem) ---> intrans_verb(Sem).



rel_clause(SemRel) --->
     [that],
      statement(SemSt) / noun_phrase(SemNP),
     {apply1(SemNP,SemSt,SemRel)}.

rel_clause(SemVP) ---> [that],verb_phrase(SemVP).


noun(X:isa(X,City)) --->
     [City],
     {class(City)}.


preposition(Y:X:of(X,Y)) ---> [of].
preposition(Y:X:in(X,Y)) ---> [in].



determiner((X:P1):(X:P2):
     which(X: (P1 and P2))) ---> [which]. % ProForma
determiner((X:P1):(X:P2):
    exists(X: (P1 and P2))) ---> [a].
determiner((X:P1):(X:P2):
    exists(X: (P1 and P2))) ---> [the].
determiner((X:P1):(X:P2):
    forall(X: (P1 => P2))) ---> [every].

proper_noun((Turkey:P):P) --->
    the0,[Turkey],{isa(Turkey,_)}.
proper_number((N:P):P) ---> [nb(N)].

trans_verb(X:Y:border(X,Y))  ---> verb_ed(border).

trans_verb(Y:X:border(X,Y))  ---> [border].
trans_verb(Y:X:border(X,Y))  ---> [borders].
trans_verb(Y:X:exceed(X,Y))  ---> [exceed].
trans_verb(Y:X:exceed(X,Y))  ---> [exceeds].
trans_verb(Y:X:have(X,Y))    ---> [have].
```

```
trans_verb(Y:X:have(X,Y))    ---> [has].

intrans_verb(X:exist(X)) ---> [exist].
intrans_verb(X:exist(X)) ---> [exists].


verb_ed(border) ---> [is],[bordered],[by].

verb_ing(border) ---> [bordering].

aux ---> [does].

the0 ---> [the].
the0 ---> [].

terminator ---> ['.'].
terminator ---> ['?'].


%% Compositional Semantic Predicates

apply1(X:P,X,P).

apply2((Y:P):Z, (Y:X:P), X:Z).

apply3(X:P, X:Q, X:(P and Q)).



%% Access Utilities


class(thing).
class(C):- ako(City,_).


isa(X,C) :- ako(D,C), isa(X,D).

exist(X):-isa(X,_),!,  % the verb exist

of(population=X,India):-   %% ad hoc for attributes
   have(India,(population=X)).

exceed(Pop=X,Pop=Y):-
    !,
    have(_C,(Pop=X)),  %% Ad hoc
    have(_I,(Pop=Y)),  %% instantiations
    X > Y. %% ad hoc

exceed(Pop=X,Y):- number(Y),
    !,
    have(_C,(Pop=X)),  %% Ad hoc
    X > Y. %% ad hoc
```

```
border(X,Y):-
    border1(X,Y)
    ;
    border1(Y,X).
```

A combinatory kernel for consensical grammar

The intereste reader may consult the program files under the Resource directory (PRO)

```
nrl.prolog      % main program for nrl

consec.prolog   % metaparser for consensical grammar

lilac.prolog    % linear lambda calculus evaluator

readin.prolog   % standard scanner

sologram.prolog % consensical grammar for nrl

solon.prolog    % lambda function definitions
```

Sample questions to BusTUC

what date is it ?

what is a bus stop near nth ?

when do the next buses go from NTH to town?

when does bus number 5 leave from nardo?

where can i get on bus 5?

What can you do?

bus from town?

how do i get from steinanveien to lade?

what is the end station of 52 ?

I want to go from Moholt to Downtown at 1400?

How can I get from NTNU-Lade to Fagerlia?

when is the earliest bus from nrdo to vestlia ?

is there a bus going downtown ?

when does bus 8a pass Gløshaugen after 16.18?

when does the next 2 busses leave from nth to vestlia?

when does the next two busses leave from nth to vestlia?

What time does the bus arrive in Nardo after 3:00 ?

What time does the bus arrive in Nardo after noon ?

which cities are there ?

When does the next bus for dalgård leave gløshaugen ?

When does the bus go from NTNU to the city centre ?

What time does the bus arrive in Nardo after 3:00 ?

which buses from nth arrive at dragvoll before 1300 ?

which bus must i take from nth to be at dragvoll before 1300 ?

which bus must i take from nth to be at universitetet dragvoll before 1300 ?

which bus stop between hovedterminalen and pirterminalen ?

what date is it tomorrow ?

which buses to dragvoll depart from nth after 1000 ?

when does a bus that arrives at pirterminalen before 930 go to hovedterminalen ?

when does a bus that arrives at pirterminalen before 930 go from hovedterminalen ?

when does a bus that arrives at pirterminalen before 930 pass hovedterminalen ?

what is the meaning of life ?

what is the meaning of life today?

is there any meaning of life ?

what is it ?

why are we here?

I want to be at Sluppen before 1200 am .

buses through the train station .

When does the bus to Steinan appear at NTNU?

When does the bus to Steinan show up at NTNU?

Can you not lie when you say I can t get to Vestlia and Nardo?

When is next departure from Nidarø .

which buses goes from the train station to festningen?

buses to the train station .

When is the earliest bus from Nardo to Vestlia?

where do I catch a bus to Trondheim?

good morning.

Where can I catch a bus from Nardo to Vestlia before 1300 but after 900?

when can I go by Vestlia riding bus 5c?

what is bustuc ?

what is your name?

how are you ?

Show me the bus stop nearest Kolstad.

Show me the bus stop nearest to Kolstad.

Show me the day nearest to Kolstad.

how much does a single trip cost?

how expensive is a single trip ?

When does the bus leave for Vaernes Airport?

when is next bus for munkholm ?

how do I get to the airport from the city center .

what is the fastest way to Moholt Studentby from Vestlia ?

where does bus 1 go?

when can I go by Vestlia and Nardo by bus 5c?

does Bus 5e go by Nardo?

Is there a quick Bus from Nardo to Vestlia before 1300 but after 900?

Is there a bus currently from Nardo to Vestlia before 1300 but after 900?

what bus is possible from Nardo to Vestlia before 1300 but after 900?

Is there a bus I can ride from Nardo to Vestlia before 1300 but after 900?

Is there a bus I can catch from Nardo to Vestlia before 1300 but after 900?
what bus am I able to take from Nardo to Vestlia before 1300 but after 900?
am i able to catch a bus from nardo to vestlia before 1300 but after 900 ?

When am I able to catch a bus from Nardo to Vestlia before 1300 but after 900?
which bus passes heimdal ?

when goes the next bus after 1750 to sentrum from Berg ?

when can I go by Vestlia and Nardo by bus?

can I go by Vestlia and Nardo by bus?

When did the last bus arrive from Lade?

how I go by Vestlia and Nardo by bus?

when is the next departure ?

how long does bus 3 take from Hallset to nodre ?

Which bus runs between the station and the harbour?

which bus runs between the station and the port ?

how long time does bus 5 use to go from nth to nardo ?

how many buses travel per day?

how far does the bus travel?

how far does bus 5 travel?

how long does the longest trip last?

what bus do I have to take if I want to be at nth before 1200 am ?

what bus do I have to take if I want to be at nth at 1200 am ?

Can I get to Tors veg and Nardo on the same bus ?

Can I get to Tors veg and Lade on the same bus ?

What is the way to get from Vestlia to Moholt Studentby?

What is a way to get from Vestlia to Moholt Studentby?

What is the fastest way to get from Vestlia to Moholt Studentby?

what bus is the best?

What are you today ?

when is next bus from Mediahuset to Vestlia .

from Blaklia to nardo at 1800 hours today .

if I want to go to Nardo, is there a bus?

If I take the bus to Nardo, can I leave before 600?

When I take the bus to Nardo, can I leave before 600?

how can you say i cant get to vestlia and nardo ?

Is there a Bus with handicap seating from Nardo to Vestlia before 1300
but after 900?
what bus should I take from hotel royal garden to NTNU 25th of june 0700.

how long does the route that passes by nardo take ?

Examples from Abstracts about Molecular Genetics

We studied the effect of dietary olive and corn oil on high-density lipoprotein (HDL) metabolism in golden Syrian hamsters.

The animals were fed a semipurified diet containing 0 percent cholesterol and 40 energy percent in the form of either olive or corn oil for a period of nine weeks.

The binding capacity of 125I-labelled HDL to liver membranes was 33 percent higher in the hamsters fed corn oil instead of olive oil (571 plus or minus 29 versus 429 plus or minus 24 ng HDL protein per mg membrane protein, P less than 0, n equals 4).

HDL protein kinetics were studied with 125I-HDL using a constant infusion technique.

Both HDL fractional catabolic rate (0 plus or minus 0 versus 0 plus or minus 0 or h, P less than 0, n equals 5) and transport rate (2 plus or minus 0 versus 1 plus or minus 0 mg per hour, P less than 0, n equals 5) were about 2-fold higher in the hamsters fed corn oil.

The rate of plasma cholesterol esterification by lecithin: cholesterol acyltransferase (LCAT) was essentially the same for the two diets.

It is concluded that the low HDL level in the hamsters fed corn oil diets is linked with increased HDL binding and degradation in the liver and possibly other tissues.

Due to increased HDL protein turnover, the capacity for reverse cholesterol transport is increased in hamsters fed corn oil despite the relative low HDL concentrations.

We measured the binding affinity of low density lipoprotein (LDL) for the LDL receptor in patients with various types of hyperlipoproteinemia and investigated the effects of LDL lipid composition and particle size on receptor affinity.

LDL (1 less than d less than 1) was isolated by sequential ultracentrifugation from the serum of normolipidemic controls and patients with hyperlipoproteinemia.

Patients with type IIa hyperlipoproteinemia had LDL with a similar receptor affinity to that of normal LDL.

However, patients with hypertriglyceridemia (type IIb and type IV hyperlipoproteinemia) had LDL with a low receptor affinity, and the degree of the reduction in affinity paralleled the severity of the hypertriglyceridemia.

The LDL of hypertriglyceridemic patients was rich in protein and triglycerides, had a low content of cholesterol and phospholipids, and was smaller than normal, thus resembling the atherogenic lipoprotein known as small, dense LDL.

These abnormalities were observed even in patients with mild hypertriglyceridemia regardless of their serum cholesterol levels.

The degree of alteration in LDL lipid composition and particle size was strongly associated with the reduction of LDL receptor affinity.

We also examined the effects of two lipid-lowering agents (bezafibrate and probucol) on the characteristics of LDL.

LDL receptor affinity was only improved when the lipid composition and particle size were normalized by drug therapy.

Although it has been reported that decreased cholesteryl ester transfer protein (CETP) activity results in the formation of small LDL, plasma CETP activity was normal in the hyperlipoproteinemic patients and the normalization of LDL characteristics by drug therapy was not accompanied by an increase of CETP activity.

Our results suggested that an abnormal lipid composition and or or small particle size might cause a decrease in the receptor affinity of LDL.

These structural and functional abnormalities were reversed by drug therapy, underlining the importance of treating hypertriglyceridemia for the prevention of atherosclerosis.

C-reactive protein (CRP) is a useful prognostic factor in coronary heart disease. It has not been previously studied in acute cerebro-vascular events, which was the topic of the present study.

Patients admitted to the hospital for an acute cerebro-vascular event were prospectively investigated.

C-reactive protein was determined nephelometrically.
Infection or inflammation were excluded clinically and with an erythrocyte sedimentation rate less than 30 mm per hour.

Computed tomography or nuclear magnetic resonance imaging of the brain was performed.

According to initial brain imaging and the clinical course the 138

patients were divided into five groups: 20 with transient ischemic attack, 20 with reversible neurological deficit lasting less than 2 weeks, 61 with completed stroke and restitution, 16 with stroke without restitution and 21 with cerebral hemorrhage.

Median CRP values (range) were 3 (2-13), 3 (2-39), 4 (2-73), 3 (3-44) and 3 (2-104 mg per l), respectively with no significant differences between groups in a non-parametric test (Kruskal-Wallis).

Risk factors for vascular disease in general and stroke in particular had no visible influence on CRP levels.

No relationship was found between time interval since onset of symptoms and CRP measurement, suggesting that an acute cerebro-vascular event has little influence on CRP values.

CRP is not a useful marker to predict the outcome of an acute cerebro-vascular event on hospital admission.

This is in contrast to acute coronary events.

Postprandial hypertriglyceridemia represents an independent risk factor for coronary artery disease.

In the postprandial state, elevated levels of triglyceride-rich lipoproteins (TRL) are minor acceptors of HDL-cholesteryl ester (CE) transferred by CETP in normolipidemic subjects: indeed, LDL particles represent the major CE acceptors.

In order to evaluate further the potential atherogenicity of lipoprotein particles characteristic of the postprandial phase in normolipidemic subjects, we determined the quantitative and qualitative features of apoB- and apoAI-containing lipoproteins over an 8-h period following consumption of a mixed meal.

During postprandial lipemia, we observed a significant decrease (-12 percent) in plasma AI concentration (138 plus or minus 4 and 156 plus or minus 4 mg per dl, at 3 h and baseline, respectively, P less than 0).

Additionally, plasma LDL was reduced by 5 percent (247 plus or minus 12 mg per dl at 3 h versus 260 plus or minus 15 mg per dl at baseline; P less than 0) 3 h following meal intake.

Moreover, a significant reduction (-10 percent) occurred in the CE or TG ratio in LDL at 2 h postprandially (8 plus or minus 2 at 2 h versus 9 plus or minus 3 at baseline; P less than 0).