Masaryk University
Faculty of Informatics

Algorithm for Analysis and Translation of Sentence Phrases

Bachelor's thesis

Roman Lacko

Brno, 2014

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Roman Lacko

Advisor: RNDr. David Sehnal

Acknowledgement

I would like to thank my family and friends for their support. Special thanks go to my supervisor, RNDr. David Sehnal, for his attitude and advice, which was of invaluable help while writing this thesis; and my friend František Silváši for his help with the revision of this text.

Abstract

This thesis proposes a library with an algorithm capable of translating objects described by natural language phrases into their formal representations in an object model. The solution is not restricted by a specific language nor target model. It features a bottom-up chart parser capable of parsing any context-free grammar. Final translation of parse trees is carried out by the interpreter that uses rewrite rules provided by the target application. These rules can be extended by custom actions, which increases the usability of the library. This functionality is demonstrated by an additional application that translates description of motifs in English to objects of the MotiveQuery language.

Keywords

Natural language, syntax analysis, chart parsing, interpreting, MotiveQuery

Contents

1 **Introduction** . . . 3
2 **State of the art** . . . 5
2.1 Dictionaries and translators . . . 5
2.2 Automatic summarization . . . 5
2.3 Dialogue systems . . . 6
2.4 Libraries for NLP . . . 7
2.5 MotiveQuery natural language tools . . . 8
3 **Phrase analysis and translation** . . . 9
3.1 Formal languages . . . 9
3.1.1 Alphabet, word and language . . . 9
3.1.2 Grammar . . . 10
3.2 Chomsky language hierarchy . . . 10
3.2.1 Type 0 – Phrase grammars . . . 11
3.2.2 Type 1 – Context-sensitive grammars . . . 11
3.2.3 Type 2 – Context-free grammars . . . 11
3.2.4 Type 3 – Regular grammars . . . 11
3.2.5 Relation to formal languages . . . 11
3.3 Derivation trees . . . 12
3.4 Automata . . . 12
3.4.1 Abstract automata . . . 12
3.4.2 Deterministic Finite Automaton . . . 13
3.4.3 Pushdown Automaton . . . 13
3.4.4 Turing machine . . . 14
3.5 Basics of speech and language processing . . . 14
3.5.1 Levels of knowledge in language understanding . . . 15
3.5.2 Phonetics and phonology . . . 15
3.5.3 Morphology . . . 15
3.5.4 Syntax . . . 16
3.5.5 Semantics . . . 16
3.5.6 Pragmatics . . . 16
3.6 Phrase analysis . . . 16
3.6.1 Compiler and interpreter . . . 16
3.7 Lexical analysis . . . 17
3.8 Syntax analysis . . . 17
3.8.1 Earley parser . . . 18
3.8.2 Cocke–Younger–Kasami algorithm . . . 19
3.8.3 Chart parsing . . . 20
3.8.4 Head-driven chart parsing . . . 23
3.8.5 Tree-adjoining grammar parser . . . 24
3.8.6 LR parser . . . 25
3.9 Semantic analysis . . . 25
3.10 Code generation . . . 25
3.11 Issues with natural language processing . . . 26

3.11.1 Natural languages are not context-free . . . 26
3.11.2 Ambiguity . . . 26
3.11.3 Metaphors and metonymys . . . 27
3.11.4 Resolution . . . 27
3.12 MotiveQuery . . . 28
3.13 Summary . . . 28
4 **Implementation** . . . 29
4.1 .NET Framework . . . 29
4.2 Overview of the solution . . . 29
4.3 The Mercury library . . . 30
4.3.1 Nucleus . . . 30
4.3.2 Scanning . . . 31
4.3.3 Syntax . . . 32
4.3.4 Interpreting . . . 34
4.3.5 Note on the semantic analysis . . . 39
4.3.6 Analyser . . . 39
4.4 The Mercury.Formats library . . . 39
4.4.1 GrammarParser . . . 39
4.4.2 RewriteRulesParser . . . 40
4.5 The MQNLParser demo application . . . 40
4.6 Unit tests . . . 41
5 **Results and discussion** . . . 42
5.1 Performance . . . 42
5.2 Comparison with MotiveQuery Natural Language Analyser . . . 44
5.3 Limitations and possible solutions . . . 44
6 **Conclusion** . . . 45
Appendices . . . 49
A **Demo applications** . . . 50
A.1 The MercuryParser application . . . 50
A.2 The MQNLParser application . . . 51
A.3 The PerfWrapper application . . . 53
B **Test inputs (examples)** . . . 54
B.1 Test A inputs . . . 54
B.1.1 Test B and C inputs . . . 54
C **Action templates and basic actions** . . . 55
C.1 Action templates . . . 55
C.2 Basic actions (excerpt) . . . 55
D **Contents of the attached CD** . . . 56

1 Introduction

Natural language processing (NLP) is a field of computer science, computational linguistics and artificial intelligence that studies computational properties of natural languages. Its main task is to allow humans to use these languages in the communication with computers, either in written or spoken form. The field spans various areas of research, for instance *automatic text summarization*, *machine translation*, *speech recognition* and *knowledge representation* [1]. In comparison with programming languages, which are designed to give exact instructions to a processing unit, natural languages are without a doubt more complex. One of the most notable problems is *ambiguity* and computers' lack of intuition. While there are techniques that can address the former, the latter requires intelligence of humans to be solved [2].

Even though not all of these problems have been addressed yet, scientists in this field work extensively to improve processing algorithms that NLP utilizes. It is only natural that it is indeed more convenient to tell the computer what to do in English (or any other language for that matter) rather than typing commands of formal languages. Besides, even a slight mistake in the formal language usually leads to an error, which users rarely enjoy. However, there is no NLP formalism that can be used universally, as there are several factors that affect the choice. For instance, devices with low computational capabilities (like mobile phones) usually feature fast algorithms at the cost of precision.

This thesis is focused on general translation of English phrases describing a certain object into its formal representation. The actual language and the final representation depends on the target application; therefore the solution is not restricted by a specific grammar. Instead, it is able to work with any *context-free language* whose grammar is provided by the target application. The result of this effort is the Mercury library written in the C# programming language. It features lexical analyser (tokenizer), bottom-up chart parser and parse tree interpreter.

The usability of this library is demonstrated by an additional application that translates phrases describing chemical motifs into expressions of the *MotiveQuery* language. This language is being developed at the Faculty of Informatics of Masaryk University in cooperation with the *National Centre for Biomolecular Research* (NCBR) and *Central-European Institute of Technology* (CEITEC). The MotiveQuery language is based on the Python programming language and is used to describe structural motifs in proteins and nucleic acids. It is a part of the *WebChemistry* platform [3]. The proposed application is designed to simplify the input of queries; for instance, the query "find all CYS residues" would be translated into MotiveQuery as `"ResidueSet[CYS]"`. Since MotiveQuery has not been finished yet and it may introduce new features in the future, the application can be easily extended by new rules.

Brief description of the thesis' structure:

State of the art lists several natural language processing tools, whether they are universal or designed for a specific task (such as dialogue systems).

Theoretical background covers basics of formal languages and automata theory, followed by fundamentals of speech and language processing. It also features basic parsing techniques and functions of a compiler, which serves as a skeleton of the implementation. Finally, the chapter briefly describes the MotiveQuery language.

Implementation chapter describes the Mercury library. It starts by an overview of the entire solution, data structures and then focuses on each stage of analysis separately. The description of the library is divided into lexical analysis, syntax analysis and final translation (interpreting) of parse trees. Moreover, it briefly describes an additional library that allows user to write rules and grammars as text files in convenient format.

Results include performance evaluation tests and comparison between the solution and an existing MotiveQuery Natural Language Analyser.

Conclusion summarizes the results of this thesis.

2 State of the art

Since *Natural Language Processing* is a constantly evolving field whose results can be applied in many other areas of computer science, there are plenty of tools that are used to analyse and process natural languages. Some of the tools mentioned here have been selected from the master's thesis of Radek Lukáš [4].

2.1 Dictionaries and translators

Electronic translation dictionaries and automatic translators are possibly the most used NLP tools. The former are usually simpler to implement, because they only need to provide semantic equivalents of the given word in an another language. On the other hand, translators are generally more interesting and complex. Quality of the output is slightly worse in comparison with human translators; however, electronic translators can be used to help inexperienced speakers to better understand foreign languages.

Google Translate

Google Translate is a free online translation service based on *statistical machine translation* that essentially uses huge parallel corpora with texts translated by humans. The service searches for patterns in the translated text and tries to infer the best translation for the input from user.

It is available at http://translate.google.com/.

2.2 Automatic summarization

Nowadays, there is a huge amount of textual information available it is almost impossible for people to cope with all of it. Automatic summarization provides techniques that can create a summary from text while retaining as much important information from the original document as possible [5]. Some automatic summarizers can be taught using *machine learning*.

SMMRY

SMMRY (pronounced SMUR-EE) is an online summarizer based on *extraction* of the most important sentences. The core algorithm essentially ranks the sentences according to their importance (e.g. number of contained keywords) and filters out the rest leaving only few sentences from the original input. Apart from many other online summarizers, this one is also accessible via its *Application Programming Interface* (API) with responses encoded in *JavaScript Object Notation* (JSON) which makes SMMRY easy to use by other websites or online applications.

SMMRY is available at http://smmry.com/.

Open Text Summarizer (OTS)

OTS is an open-source library for text summarizing that supports UTF-8 encoding and more than 25 languages. It can be linked with some word preprocessors, such as *AbiWord* and *KWord*, and comes with a console tool that (apart from text summarization) is also able to output HTML documents with important sentences highlighted.

Instructions on how to get OTS in C programming language for UN*X-based systems are available at http://libots.sourceforge.net/, C# port[1] for .NET (or Mono) platform can be found at http://ots.codeplex.com/.

2.3 Dialogue systems

Also referred to as *conversational agents*, these are primarily used for simulating a dialogue with humans in natural language. The dialogue itself can be spoken or written, although the former is more difficult due to the fact that *speech recognition* is one of *AI-complete* problems, which are extremely difficult [2].

Artificial Intelligence Markup Language (AIML)

AIML is a dialect of *Extensible Mark-up Language* (XML) that was used to extend chatbot[2] *Eliza* into *Artificial Linguistic Internet Computer Entity*, shortly *A.L.I.C.E.* It has won Loebner Prize Competition in Artificial Intelligence three times, last time in 2010.

A.L.I.C.E. web interface is available at http://alice.pandorabots.com/ and its AIML sources at https://code.google.com/p/aiml-en-us-foundation-alice/.

Voice XML

Voice XML is a *World Wide Web Consortium* (W3C) standard for specifying dialogue systems, usually for automated customer services. It features speech synthesizer, digitalized audio and voice recording. A program that interprets Voice XML document (or other similar document types) is sometimes called *voice browser*.

JVoiceXML is a free Voice XML interpreter implementation in Java available at http://jvoicexml.sourceforge.net/. VoiceGlue, available at http://www.voiceglue.org/, features *Asterisk* software implementation of a telephone branch exchange.

Abodit NLP

Abodit NLP is a conversational agent .NET library designed for specific application domains such us customer services or *home automation* (lighting, heating control etc.). It can be controlled either by voice interface or remotely via SMS, SmartPhone applications and so on. The library also uses a word database similar to *WordNet*[3].

1. Original source code rewritten in another programming language.
2. A conversational agent application.
3. WordNet is a lexical database for English that records various semantic relations between words.

The documentation of Abodit NLP can be found at http://nlp.abodit.com/, the library is available for download from NuGet package manager at https://www.nuget.org/packages/AboditNLP.

Cleverbot

Even though Cleverbot is not a tool, it represents a significant application of NLP and artificial intelligence. It is an online chatbot whose responses are not programmed a priori, but it uses *machine learning* to remember human inputs in huge corpora, where it also searches for keywords when assembling its own response. Therefore Cleverbot can easily learn frequently used phrases, quotes and even song lyrics.

Cleverbot is available at http://www.cleverbot.com/.

2.4 Libraries for NLP

Apart from projects focused on certain tasks within NLP, there are also libraries that provide generic tools that can be used as parts of a larger project. There are no formal requirements for what a NLP library should contain, but the most used libraries contain tokenizer[4], syntax analyser (*parser*) and additional language processing tools such as *Parts-of-speech tagger* (POS) and *WordNet* access.

Natural Language Toolkit (NTLK)

NTLK is a suite of open source libraries for Python programming language. These libraries contain many useful modules such as tokenizer, POS tagger, four parsers and even semantic interpreter based on *first order logic*. It also features an interface that makes it easy to access over 50 lexical databases and corpora, including WordNet.

Documentation and library suite is available at http://www.nltk.org/.

The Stanford Natural Language Processing Group

The Stanford NLP Group provides *statistical* NLP toolkits. It is being developed at Stanford University and supports many languages, for instance English, Chinese and German. Apart from tokenizer, probabilistic parser and similar "standard" tools it also features phrase-based translation system *Phrasal* with *coreference resolution system*.

While the original suite is written in Java, there are available ports for other languages (Python, Perl, etc.) as well.

The suite is available for download at http://nlp.stanford.edu/software/.

4. An algorithm that divides input into sequence of significant character groups (called *tokens*), which usually match words and punctuation.

Apache OpenNLP

Apache OpenNLP is another library suite for natural language processing. Similar to the Stanford NLP Group, OpenNLP also contains tools like tokenizer, POS tagger and parser, but it uses *maximum entropy* and *perceptron based machine learning* approach.

It is available at https://opennlp.apache.org.

Antelope Framework

Antelope is a comprehensive NLP framework for .NET and Mono platforms. It features dependency and semantic parser, coreference resolution and multithreading support. The framework is free for academic and non-commercial purposes; however, registration is required in order to download it.

The documentation and registration form can be found at https://www.proxem.com/en/technology/antelope-framework/.

2.5 MotiveQuery natural language tools

MotiveQuery is a language designed to describe and analyse structural motifs. We are going to discuss it in more detail later in the section 3.12.

MotiveQuery Natural Language Analyser

MotiveQuery Natural Language Analyser is a .NET library that is able to parse phrases in English into queries in the MotiveQuery language. This result of the master's thesis of Radek Lukáš [4] is the only tool available that offers this functionality.

However, the language the analyser can recognize is restricted to a set of deterministic context-free languages which, as we will see later, is hardly enough for natural languages. For instance, "*zinc atoms*" and "*atoms of zinc*" are semantically equivalent phrases, but the tool will recognize only the first one. This is because it treats the word "*atoms*" as a *function* and thus it cannot be used in different language constructs, such as the second phrase.

3 | Phrase analysis and translation

Basically, **communication** is a process of sharing information between its participants. Those do not have to be necessarily humans; animals and machines can communicate in their own way as well. However, in order for communication to be successful, both parties need to follow the same *rules* of communication. For instance, when humans speak to each other, they usually use the same common **language**. Any other human who does not know the language will probably not understand and thus will not learn the information. Communication between computers is actually very similar, except they use a different kind of language (usually a **communications protocol**) and electric signals or radio waves as a medium, neither of which humans understand very well.

This poses an interesting question – how can computers communicate with humans? First computers did not understand human languages (and they still do not); but without a doubt it would be more convenient to *tell* the computer what to do instead of pressing buttons and typing text. Therefore, around the year 1950, computer scientists started looking for a way to teach computers human languages. It was then that natural language processing (NLP) was born.

In this chapter we are going to start with a brief introduction to formal languages and automata theory. This will provide us useful concepts also used in NLP, which we are going to look at next. There we are going to introduce a compiler and interpreter model; this will serve as a skeleton of the resulting algorithm. It will be followed by depiction of a few parsing techniques, such as Earley parser and chart parsing; and some advanced techniques introduced in less detail. Finally, we will have a look at issues that natural language processing tries to cope with.

3.1 Formal languages

With only a few exceptions, natural languages are without a doubt very complex. Indeed, they "suffer" from many anomalies, *ambiguity*, *polysemy* and other issues (we are going to discuss them later in the section 3.11). Human brain is *somehow* able to cope with it as it learns the language. On the other hand, computers are not very happy with that as their construction requires more formal approach. This is why we should have a look at formal languages before we proceed to the natural language processing.

3.1.1 Alphabet, word and language

Hopcroft in [6] defines an **alphabet** as a *finite*, non-empty set of symbols denoted by Σ. **Word** (or **string**) is any finite sequence of those symbols. An **empty word** is a special word of length 0 usually denoted by ϵ or λ. Given word w, by $|w|$ we mean its length. Σ^* denotes a set of all words over an alphabet Σ, where * is the *Kleene star* [6]. Finally, a **formal language** is an arbitrary subset of Σ^*.

3.1.2 Grammar

Even though every alphabet must be finite, languages do not need to be. Describing an infinite language by its enumeration would definitely (yet infinitely) do, but there is a more convenient way – a **formal grammar**. It is a 4-tuple $\mathcal{G} = (N, \Sigma, P, S)$ where its elements are defined as follows:

- N is a set of **nonterminal symbols**; it must hold that $N \neq \varnothing$ and $N \cap \Sigma = \varnothing$,
- Σ is an alphabet, in grammar also referred to as a set of **terminal symbols**,
- $P \subseteq V^*NV^* \times V^*$ is a *non-empty* set of **production rules** where $V = N \cup \Sigma$,
- $S \in N$ is a **starting symbol** or **root nonterminal**.

We are going to denote nonterminal symbols (shortly **nonterminals**) by Latin upper case letters $(A, B, \dots)$ and terminal symbols (**terminals**) by lower case letters $(a, b, \dots)$. For terminal symbols consisting of more than one character (like in natural languages) we are going to use non-proportional font, i.e. `a cat`. By Greek letters $(\alpha, \beta, \dots)$ are denoted strings consisting of the symbols of V. Instead of (α, β) notation of the elements of P we are going to use a more convenient notation $\alpha \to \beta$. Also, we may abbreviate $\{\alpha \to \beta_1, \dots, \alpha \to \beta_n\}$ into $\alpha \to \beta_1 \mid \cdots \mid \beta_n$ when necessary.

Derivation is a relation $\Rightarrow \subseteq V^* \times V^*$ that lets us generate words from grammar. Let $\rho \Rightarrow \sigma$ if and only if for some $(\alpha \to \beta) \in P$ and $\tau, \nu \in V^*$ it holds that $\rho = \tau\alpha\nu$ and $\sigma = \tau\beta\nu$. In other words, we derived $\tau\beta\nu$ from $\tau\alpha\nu$ by applying the rule $\alpha \to \beta$. A reflexive and transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$. While $\alpha \Rightarrow \beta$ denotes a single rule application, $\alpha \Rightarrow^* \beta$ denotes zero or more applications. In addition, let $S \Rightarrow^* \alpha$; if α does not contain nonterminal symbols, we say that α is a **sentence**. Otherwise we call it a **sentential form**.

Suppose that we always select the leftmost nonterminal in a sentential form to be replaced according to some rule. This derivation is called **leftmost derivation**. Analogously the derivation that selects the rightmost nonterminal is called **rightmost derivation**. Hopcroft in [6] introduces notation $\underset{lm}{\Rightarrow}$ and $\underset{rm}{\Rightarrow}$ for leftmost and rightmost derivations respectively.

If the result of derivation is a sentence, it proves that the sentence belongs to a language generated by the grammar. A string is derived **ambiguously** if the grammar allows at least two different derivations of the string. A grammar that generates some string ambiguously is **ambiguous** [7].

A language generated by grammar $\mathcal{G}$ (denoted by $L(\mathcal{G})$) is a set $\{w \in \Sigma^* \mid S \Rightarrow^* w\}$ where S is the root nonterminal of $\mathcal{G}$.

3.2 Chomsky language hierarchy

In 1956 Noam Chomsky divided formal grammars into four groups according to the form of rules they contain [8]. These groups are called **type 0** to **type 3** grammars.

3.2.1 Type 0 – Phrase grammars

Grammars of type 0 (sometimes referred to as *phrase grammars*) are not restricted and are essentially (in their expressive power) equal to *Turing machines* [9]. They include all formal grammars. In automata theory these languages are also known as *recursively enumerable languages* [6]. Those are not to be confused with *recursive languages*, which are decided by an *always-halting* Turing machine.

3.2.2 Type 1 – Context-sensitive grammars

Context-sensitive grammars are sometimes referred to as *CSG*. Rules of these grammars are restricted in the following way: for each rule $\alpha \to \beta$ from set P it must hold that $|\alpha| \leq |\beta|$. The only exception is rule $S \to \epsilon$, but then there can be no rule that contains S on its right-hand side. In other words, sentential forms can never *shrink* when derived from a context-sensitive grammar.

3.2.3 Type 2 – Context-free grammars

Abbreviated as *CFG*, they inherit the restriction of type 1 grammars. Furthermore, left-hand side of each rule must consist of *exactly one* nonterminal symbol. This means that all rules are of the form $A \to \alpha$ where $A \in N$ and $|\alpha| \geq 1$ because of the inherited restriction. This form leads to a very useful representation of derivations called **derivation trees**, which we are going to discuss in the next section.

An extension to context-free grammars removes the ϵ rule constraint, thus allowing rules of the form $A \to \epsilon$ for any $A \in N$. As shown in [7] and [10], any CFG with ϵ-rules is still equivalent to some context-free grammar without these rules. The mentioned literature also features algorithms that can eliminate ϵ-rules from any context-free grammar.

3.2.4 Type 3 – Regular grammars

Regular grammars are the most restricted type of the hierarchy. The rules can be of two forms only; either $A \to xB$ or $A \to x$ where $A, B \in N$ and $x \in \Sigma$. Type 3 grammars inherit the exception from type 1 grammars as well.

3.2.5 Relation to formal languages

As each grammar generates a formal language, these languages are also divided into four groups. A language is called *regular* if there exists a regular grammar that generates it. Similarly we can define *context-free* (*CFL*), *context-sensitive* (*CSL*), and *phrase* languages that are generated by CFG, CSG and phrase grammars respectively.

The restrictions of type 0 to type 3 grammars create the following hierarchy of languages:

$$\text{regular} \subset \text{context-free} \subset \text{context-sensitive} \subset \text{phrase}$$

3.3 Derivation trees

A **derivation tree** represents the structure of a sentence generated by some formal grammar. We are going to focus on derivation trees generated by context-free grammars. A derivation tree represents rule applications that took place when a sentence was being generated. However, the order of application does not need to be clear from the tree itself.

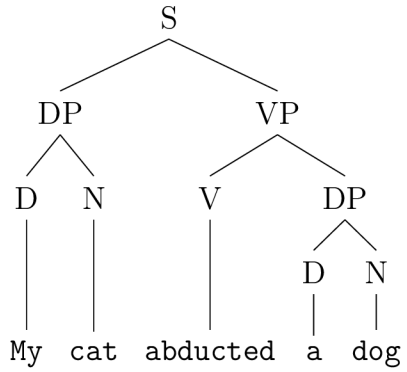


Figure 3.1: An example of a derivation tree

Let us consider a context-free grammar $\mathcal{G} = (N, \Sigma, P, S)$. The sentence generation starts as a string w containing only S. Therefore the corresponding tree has a root node of value S. If the rule $A \to X \dots Z$ ($A \in N$ and $X, \dots, Z \in N \cup \Sigma$) was applied on a nonterminal A somewhere in the string w, the corresponding node with value A would have child nodes of values X to Z. When the generating is done, the leaves of the tree make up sentence w if read left to right.

For instance, consider the tree in the figure 3.1. Clearly, the first rule that has been applied is $S \to DP\ VP$. In the second step, one of $DP \to DT$ or $VP \to V\ DP$ (the actual order would not effect the result) has been applied and so on. The result of the derivation is a sentence `"My cat abducted a dog"` as the symbols that make it up are stored in the leaves of the tree.

3.4 Automata

Now that we have discussed formal languages and the idea of generating them from grammars, let us have a look at the opposite process. Given language $L(\mathcal{G})$ and sentence w, we would like to know whether $w \in L(\mathcal{G})$. An abstract **automaton** is a mathematical model that facilitates this functionality.

3.4.1 Abstract automata

There are many types of automata designed for specific grammar types. They vary in construction, components and expressive power. We will start with the general idea.

A common automaton usually consists of an infinite **tape** with symbols of an input sentence, a **tape head** that reads the symbols, a finite set of **states**, a **transition table**

that encodes *transition function* and may contain some additional components. One of the states is the **initial** (or **start**) **state**, from which the automaton starts a computation.

Automaton reads symbols and transitions into different states according to its transition table. If the automaton stops in a **final state**, we say it has *accepted* the input sentence w (meaning $w \in L$); otherwise we say it has *rejected* w ($w \notin L$). The actual conditions for acceptance vary with the type of automaton, and with some types the automaton does not need to stop for all possible inputs.

3.4.2 Deterministic Finite Automaton

This type of automaton recognize the set of regular languages (type 0) [6]. Formally, a deterministic finite automaton (DFA) is a 5-tuple $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite, non-empty set of states,
- Σ is a set of tape symbols (alphabet),
- $\delta : Q \times \Sigma \to Q$ is a transition function,
- $q_0 \in Q$ is a start symbol and
- $F \subseteq Q$ is a set of final states.

There are a few extensions to this model. For example, we may declare transition function as $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ (the power set of states), and get a **non-deterministic finite automaton** (*NDFA*). Another extension is $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$, which allows an automaton to make *ϵ-transitions*, that is, to change its state without reading a symbol. We can also use more tapes or more tracks on one tape, or perhaps a two-sided infinite tape; however, in the end all these machines are of equal expressive power [10].

3.4.3 Pushdown Automaton

In addition to components of the DFA, a pushdown automaton introduces a data structure called **stack**, which can *push* a new element on the top of the stack or *pop* the element from the top. The term *pushdown* refers to the fact that the automaton cannot perform operations on elements that are under the top element.

Formally, a PDA is a 7-tuple $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, Z, F)$, where Q, Σ, q_0 and F are equivalent to those in the definition of DFA and in addition

- Γ is an alphabet of stack symbols,
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}_{\mathcal{F}in}(Q \times \Gamma^*)$ is a transition function, where $\mathcal{P}_{\mathcal{F}in}(A)$ denotes an arbitrary *finite* subset of the power set $\mathcal{P}(A)$,
- $Z \in \Gamma$ is an initial stack symbol.

The transition function takes into account the symbol on the top of the stack and can modify it in three ways; let $q, r \in Q$, $a \in \Sigma$ and $X, Y \in \Gamma$, then

- $\delta(q, a, X) \ni (r, \epsilon)$ removes X from the top of the stack,
- $\delta(q, a, X) \ni (r, Y)$ replaces[1] X by Y on the top of the stack
- $\delta(q, a, X) \ni (r, YX)$ adds Y to the top of the stack.

1. Notice that $X = Y$ is not forbidden and would mean that the stack was not changed.

The PDA is able to recognize the set of context-free languages [6]. It is clearly non-deterministic by default; however, we can restrict for some $q \in Q, a \in (\Sigma \cup \{\epsilon\})$ and $X \in \Gamma$ that $|\delta(q, a, X)| \leq 1$ and if $(\exists a \in \Sigma)(\delta(q, a, X) \neq \varnothing)$ then $\delta(q, \epsilon, X) = \varnothing$. This will yield **deterministic PDA** (or *DPDA*). Unlike finite state automaton and its non-deterministic version, DPDA recognizes a *proper* subset of languages recognized by PDA [6], which results in a special subset of context-free languages called **deterministic context-free languages** (DCFL). A grammar that generates deterministic CFL is called similarly – deterministic context-free grammar (DCFG).

3.4.4 Turing machine

Because the **Turing machine** is far beyond the scope of this thesis, we will omit its formal description (which can be found in [6], [7] or [11]). The Turing machine is more complex than finite and pushdown automata and is able to recognize the set of phrase languages also called **recursively enumerable languages**. By default, the machine does not need to halt if the input word is *not* from the language it recognizes. Turing machines that halt for *all* inputs are called *always-halting* Turing machines. They recognize a subset of recursively enumerable languages called *recursive languages*.

If the tape of a Turing machine is limited by the number of tape tiles, we call it a **linear bounded automaton** which is able to recognize the set of context-sensitive languages [12].

3.5 Basics of speech and language processing

With basics of formal languages we are now able to have a look at natural languages. They are clearly different from the formal ones. Probably the most notable difference is their complexity. Natural languages usually feature properties such as ambiguity, polysemy, abstract concepts (i.e. "friendship") and non-projectivity. We will discuss them later in this chapter.

The language we will focus on is (British) English; although some features discussed may be more common in other languages.

Natural language processing (NLP) is a field of computer science that is focused on communication between computers and humans in human languages. Some authors (i.e. Jurafsky and Martin in [1]) consider NLP a part of **speech and language processing** along with **computational linguistics**, **speech recognition** and **speech synthesis**. In the second chapter we have already covered a few major tasks of NLP (automatic summarization and dialogue systems). Some of the other goals (but not all of them) are:

Natural language generation
: focuses on converting raw computer data to human language.

Natural language understanding
: is essentially an opposite process to the previous one; it tries to convert human sentences into "computer-friendly" structures (for instance first-order logic expressions).

Parsing
is a process of creating a **parse tree** for a given sentence.

Speech recognition
is a process of analysing speech and determining its textual representation.

Question answering
looks for answers for queries given in human language.

3.5.1 Levels of knowledge in language understanding

Natural language analysis is usually not performed by one formalism alone, but is rather broken down into several *levels* on which separate analyses are carried out. The literature differs in the number, but James Allen in [13] introduces the following five levels of knowledge relevant to natural language processing:

Phonetic and phonological level examines the relation between words and sounds.

Morphological level considers internal structure of words.

Syntactic level concerns itself with internal structure of sentences and their correctness.

Semantic level infers meaning of words and combines it into sentence meaning.

Pragmatic level examines relation between expressions and context; how it effects the interpretation of the sentence.

Allen also introduces an additional sixth level that focuses on general knowledge about the world (for instance that cats do not fly or that an orange is a fruit). It is sometimes called **world knowledge** and is essential when trying to understand speech.

3.5.2 Phonetics and phonology

Both phonetics and phonology focus on the sound aspect of human language. **Phonetics** is concerned with speech production as physiological process (which organs participate in it, how they work and so on), sound transmission and its perception. The speech sounds are usually referred to as **phones**. **Phonology** focuses on systematic organization of sounds as *abstract units* called **phonemes** of the language and how the affect the understanding of sentences.

3.5.3 Morphology

Morphology identifies and analyses language's smallest units of meaning called **morphemes** (such as word roots, affixes or implied contexts) [13]. In synthetic languages (such us German, Czech or Japanese) morphemes are used to express grammatical categories (tense, mood, person, ...) by modifying a bare word form. This modification is called *inflection*. Inflection of verbs is referred to as *conjugation*, while inflection of nouns, pronouns and adjectives is called *declension*. The process for reducing inflected words to their root is called **stemming**. For instance, stemming the word "fisher" would yield the root "fish".

Morphological analysis also covers **Parts-of-Speech** tagging (or **word-category disambiguation**), that is, assigning grammatical tags to the parts of the sentence. These tags can be simple (i.e. `NOUN`, `VERB`) or complex with subcategories like person or tense.

3.5.4 Syntax

Noam Chomsky defines syntax as "the study of the principles and processes by which sentences are constructed in particular languages" [14].

Syntax analysis is concerned with the process of checking whether an input sentence is correct in a given language. We have already discussed formal automatons, which essentially do just that. Algorithms that only perform the check are called **recognizers**. Yet sometimes it is the structure of the sentence that interests us, usually in the form of **derivation trees**, sometimes also called **parse trees**. Algorithms that create these structures are called **syntax analysers** or **parsers**.

3.5.5 Semantics

Semantics focuses on relation between *denotations* (meaning) of words and phrases. It is much more complex than previous levels. For instance, consider the following sentence composed by Noam Chomsky: "Colorless green ideas sleep furiously" [14]. It is clearly syntactically correct, but it makes no sense. Algorithmic semantic analysis of natural language is even more difficult, considering abstract concepts like *friendship* or *hatred*.

3.5.6 Pragmatics

Pragmatics explores how context affects interpretation of a sentence. Unlike semantics, it takes into account other factors that may alter the meaning. Consider the following sentence: "I killed that process". The straightforward meaning would be that someone has confessed to a killing of an abstract series of actions. That clearly makes no sense. More likely, the speaker was talking about terminating a computer process by operating system as a result of receiving the `kill` signal from the user. Hence, "to kill" in *this context* means "to terminate" a process.

3.6 Phrase analysis

The goal of this thesis is an algorithm capable of phrase analysis and its translation into an object model. The required behaviour is apparently close to that of compilers and interpreters. A **compiler** is a computer program that translates source code written in some programming language into another language, often binary. An **interpreter** is similar, but they differ in their output.

3.6.1 Compiler and interpreter

Both compilers and interpreters are programming language processors. While the compiler outputs a program in another (formal) language, interpreter directly executes operations from the source program. A compiler performs two operations: *analysis* and *synthesis* [15].

The **analysis** part creates a grammatical structure from the source code which then poses as an *immediate representation* of the program. If the syntactic or semantic check of the structure fails, an error is reported and user (in this case programmer) should use it to fix the problem.

The **synthesis** part constructs the output program from the structure provided by the analysis part. Interpreter generally lacks this step as it directly executes the program in its immediate representation.

The whole compilation process (without optimization) consists of the stages depicted in the figure 3.2 [15]. We are going to discuss these stages in more detail; though with natural languages as the source rather than programming languages.

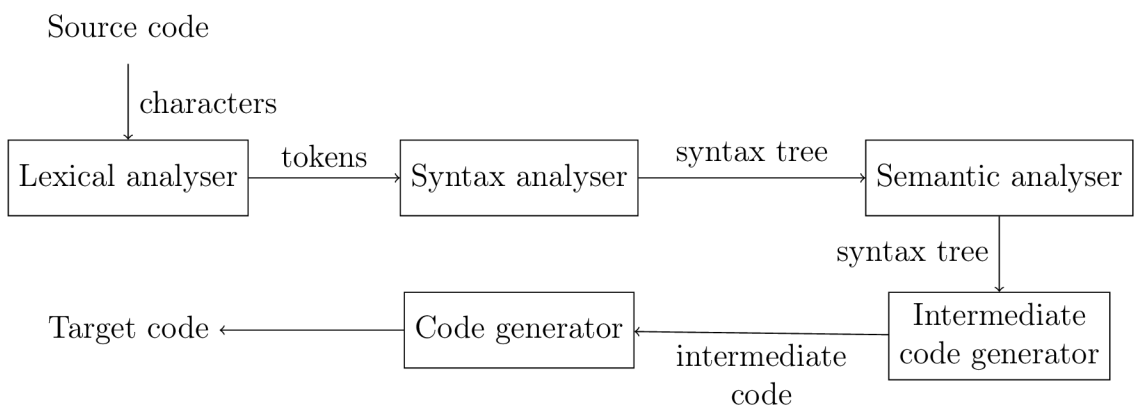


Figure 3.2: Stages of the compilation process adapted from [15]

3.7 Lexical analysis

Lexical analyser reads source code and recognizes meaningful sequences of characters called **lexemes**. To each lexeme it assigns properties (for instance *type* or *starting position*) and yields the pair (`lexeme`, `properties`) as **token**. The lexical analysis can also omit useless code segments like comments.

3.8 Syntax analysis

Syntax analyser (shortly **parser**) creates a syntactic structure from a stream of tokens produced by lexical analyser. The structure is often represented by a **syntax tree** (sometimes called **parse tree**). There are many types of parsers that recognize different language types of Chomsky hierarchy. We are going to focus on *phrase-structure grammar* (usually context-free) parsers that can be used to parse natural languages rather than programming languages. They are usually divided into the following groups [16]:

Top-down parsers try to infer the derivation that created the sentence. Their name originates from the fact that they construct trees from their root nodes. This approach is said to be *goal-directed* because it attempts to match the result sentence.

Bottom-up parsers begin with the sentence and try to reduce it back to the root nonterminal, thus reverting the derivation process. Bottom-up approach is said to be *data-directed*, for it starts with a concrete sample of goal data – the sentence.

3.8.1 Earley parser

Earley parser is a simple top-down parser invented by Jay Earley in his dissertation [17]. It can parse all context-free languages. The algorithm is based on the *dynamic programming* approach and runs in $\mathcal{O}(n^3)$ time in general and $\mathcal{O}(n^2)$ for unambiguous grammars (n being the length of the input string).

In his dissertation, Early introduces a **dot notation** for the states of the parser. Let $A \to \alpha\beta$ be a rule of a context-free grammar $\mathcal{G}$. Then $(A \to \alpha \cdot \beta, i)$ denotes a *state* (or *edge*), where string α has already been parsed, β is expected and i is the position in the input where the matching began. If $\beta = \epsilon$, we say that the state is *completed*.

Let $S(k)$ denote a set of states for input position k. Union of all $S(k)$ sets is sometimes referred to as **chart**. The parser initialization adds a state $(P \to \cdot S, 0)$ to the $S(0)$. Rule $P \to S$ is called a **top-level rule** and it is artificial; neither P nor the rule are in the original grammar.

Next the parser executes the operations described below until no new states are created. If the sentence is made of n symbols and $S(n)$ contains $(P \to S\cdot, 0)$ in the end, the sentence will be accepted by the parser.

Algorithm 1: Earley's algorithm adapted from [1] and [18]

```
1  Function Earley(𝒢, w) → {True, False}
       Input: context-free grammar 𝒢 = (N, Σ, P, S), sentence w ∈ Σ*
2      let S(0) ... S(|w|) of type SET;
3      S(0) ← {(P → ·S, 0)};
4      for k ← 0 to |w| do
5          foreach s ∈ S(k) do
6              match s with
7                  (A → α · Bβ, j) and B ∈ N ? Predict(s);
8                  (A → α · aβ, j) and a ∈ Σ ? Scan(s);
9                  otherwise Complete(s);
10     return (P → S·, 0) ∈? S(|w|);
11 Function Predict((A → α · Bβ, j))
12     foreach (X → ξ) ∈ P where X = B do
13         S(k) ← S(k) ∪ {(X → ·ξ, k)};
14 Function Scan((A → α · aβ, j))
15     if w_k = a then
16         S(k + 1) ← S(k + 1) ∪ {(A → αa · β, j)};
17 Function Complete((A → α·, j))
18     foreach (B → β · Cγ, i) ∈ S(j) where A = C do
19         S(k) ← S(k) ∪ {(B → βA · γ, i)};
```

Prediction takes place when a nonterminal symbol is expected. Formally, for each state in $S(k)$ of the form $(X \to \alpha \cdot Y\beta, j)$ and for each rule $Y \to \gamma$ in grammar the operation creates a state $(Y \to \cdot\gamma, k)$.

Scanning reads a terminal from the input. For each state $(X \to \alpha \cdot a\beta, j)$ in $S(k)$ scanning creates a state $(X \to \alpha a \cdot \beta, j)$ in $S(k+1)$ provided that a is the j-th input terminal.

Completion is performed when a completed state was fetched. For every state of the form $(X \to \gamma\cdot, j)$ in $S(k)$ and for each state $(Y \to \alpha \cdot X\beta, i)$ in $S(j)$ the operation creates a state $(Y \to \alpha X \cdot \beta, i)$ in $S(k)$.

Derivation trees can be reconstructed by inspecting the operations that were used. We are going to discuss this process in general later in this chapter.

3.8.2 Cocke–Younger–Kasami algorithm

CYK (or sometimes *CKY* algorithm) is another type of parser based on the bottom-up approach. It was invented independently by Tadao Kasami (1965), Daniel Younger (1967) and John Cocke (1970). It is highly efficient with time complexity $\mathcal{O}(n^3)$ ([16]); however it requires the input grammar to be in **Chomsky normal form**.

Chomsky normal form (CNF) permits only production rules of the form $A \to a$ or $A \to B\ C$ where $A, B, C \in N$ and $a \in \Sigma$. Every context-free grammar can be converted into CNF as shown in [6] and [10]. However, the drawback of this form is that important syntactical structures expressed by rules of different forms are lost. Hence the resulting parse tree processing is more complex.

The CYK parser starts with the input sentence $w \in \Sigma$. It creates a table V of $|w|^2$ sets. The initialization assigns nonterminals to each terminal in w. For instance, if $w_i = a$ and $A \to a$, then $A \in V[i, 0]$. Next it considers every substring of length 2 to $|w|$, partitioning it into two parts and checking if there is a rule $A \to B\ C$ where one partition matches B and the other matches C. When it is done, parser accepts w if $S \in V[1, |w|]$.

Algorithm 2: CYK algorithm adapted from [16]

```
1 Function CYK(𝒢, w) → {True, False}
    Input: context-free grammar 𝒢 = (N, Σ, P, S) in CNF, sentence w ∈ Σ*
2   let V matrix |w| × |w| of type SET;
3   for k ← 1 to |w| do
4       V[k, 1] ← {A | (A → w_k) ∈ P};
5   for i ← 2 to |w| do                                  /* substring length */
6       for j ← 1 to |w| − i + 1 do                      /* substring start */
7           for k ← 1 to q − 1 do                        /* partitioning */
8               V[j, i] ← V[j, i] ∪ {A | (A → B C) ∈ P ∧ B ∈ V[j, k] ∧ C ∈ V[j + k, i − k]};
9   return S ≟ V(1, |w|);
```

Parse trees can be created very easily by modifying the algorithm. In the initialization section, if $A \to a$ is used, then a tree with root A and one leaf a will be created. While in the inner `for` loop, if $A \to B\ C$ rule is used, previously created trees for B and C will be joined with a new root of value A.

3.8.3 Chart parsing

Chart parsing is credited to Martin Kay [19] and can be seen as abstraction (or modification) of Early and CYK parsers. It introduces variety of techniques used extensively in natural language processing because they are simple to implement and can cope even with ambiguous context-free grammars.

In this section we are going to focus on *top-down* and *bottom-up* chart parsers. They usually consist of two data structures – a **chart** and an **agenda**. Both contain labelled **edges** (analogous to Earley's *states*). The chart contains processed edges while the agenda contains newly discovered ones that have not been processed yet. Input tokens are separated by graph vertices usually labelled 0 to n with n being the length of the input (in [16] they are labelled 1 to $n+1$).

The **edge** is formally a triple $[A \to \alpha \cdot \beta, i, j]$ where $A \to \alpha\beta$ is a rule of a context-free grammar, i, j $(0 \leq i \leq j \leq n)$ are indices of nodes between which the input tokens were parsed. The segment α represents the parsed part of the input while β is what remains to be (and in the worst case may never be) parsed. Edges where $\beta = \epsilon$ (having the form $[A \to \alpha\cdot, i, j]$) are called **passive**; other edges are called **active**.

The chart and agenda are usually initialized by a small set of edges. Then new edges are created from it using **inference rules**. These rules are of the form "If the chart contains edges $E_1, E_2, \ldots$ it must also contain E". Sikkel in [18] introduces a notation $E_1, E_2, \cdots \vdash E$. Clearly the algorithm computes a **transitive closure** of the rules over the chart [16]. The inference rules are similar to those of Earley parser:

Completion (fundamental rule)
$[A \to \alpha \cdot B\beta, i, j], [B \to \gamma\cdot, j, k] \vdash [A \to \alpha B \cdot \beta, i, k]$.

Scanning
$[A \to \alpha \cdot a\beta, i, j] \vdash [A \to \alpha a \cdot \beta, i, j+1]$ if a is the j-th terminal in the input.

Prediction
depends on the parser direction:

top-down $[A \to \alpha \cdot B\beta, i, j] \vdash [B \to \cdot\gamma, j, j]$ for each rule $(B \to \gamma) \in P$.

bottom-up $[A \to \alpha\cdot, i, j] \vdash [B \to \cdot A\beta, i, i]$ for each rule $(B \to A\beta) \in P$.

By default, chart parsers are non-deterministic in the sense that edges from agenda and inference rules are selected in no particular order. However, the actual implementations usually specify the order; for instance by implementing agenda as a **queue**.

Initialization process also depends on the direction of the parser. The top-down chart parser initializes its agenda with edges $[S \to \cdot\alpha, 0, 0]$ for each rule $S \to \alpha$ in P. The bottom-up chart parser creates edges $[A \to \cdot a\alpha, i, i]$ for each rule $(A \to a\alpha) \in P$ provided that

Algorithm 3: Transitive closure (general chart parser) algorithm from [16]

```
1 Function GeneralChartParser(𝒢, w) → {True, False}
    Input: context-free grammar 𝒢 = (N, Σ, P, S), sentence w ∈ Σ*
2   Initialize(chart);
3   Initialize(agenda);
4   while agenda ≠ ∅ do
5       select e from agenda;
6       foreach f ∈ ApplyInference(e) where f ∉ ( chart ∪ agenda ) do
7           Insert(agenda, f);
8       Insert(chart, e);
9   return [S → α·, 0, |w|] ∈? chart for any (S → α) ∈ P;
```

a is the i-th symbol of the input. In addition, if the grammar contains an ϵ-rule $A \to \epsilon$, bottom-up initialization creates edges $[A \to \cdot, i, i]$[2] for all $0 \leq i \leq |w|$ (w being the input sentence). In both cases the initial chart is empty.

Consider a CF grammar $\mathcal{G}$ with the following set of rules:

$$\begin{array}{lll} S \to DP\ VP & DP \to D\ N & VP \to V \\ D \to \texttt{The} & N \to \texttt{cat} & V \to \texttt{purrs} \end{array}$$

Clearly, the only sentence $L(\mathcal{G})$ contains is `"The cat purrs"`. The bottom-up parser described above would perform steps as shown in the table 3.1.

n	Inference	Rule	n	Inference	Rule
[0]	initial	$[D \to \cdot\texttt{The}, 0, 0]$	[7]	prediction	$[5] \vdash [VP \to \cdot V, 2, 2]$
[1]	initial	$[N \to \cdot\texttt{cat}, 1, 1]$	[8]	completion	$[3], [6] \vdash [DP \to D \cdot N, 0, 1]$
[2]	initial	$[V \to \cdot\texttt{purrs}, 2, 2]$	[9]	completion	$[5], [7] \vdash [VP \to V\cdot, 2, 3]$
[3]	scanning	$[0] \vdash [D \to \texttt{The}\cdot, 0, 1]$	[10]	completion	$[4], [8] \vdash [DP \to D\ N\cdot, 0, 2]$
[4]	scanning	$[1] \vdash [N \to \texttt{cat}\cdot, 1, 2]$	[11]	prediction	$[10] \vdash [S \to \cdot DP\ VP, 0, 0]$
[5]	scanning	$[2] \vdash [V \to \texttt{purrs}\cdot, 2, 3]$	[12]	completion	$[10], [11] \vdash [S \to DP \cdot VP, 0, 2]$
[6]	prediction	$[3] \vdash [DP \to \cdot D\ N, 0, 0]$	[13]	completion	$[9], [12] \vdash [S \to DP\ VP\cdot, 0, 3]$

Table 3.1: Bottom-up chart parsing of `"The cat purrs"`

The edges contained in the final chart are shown in the figure 3.3. Edge [13], that is $[S \to DP\ VP\cdot, 0, 3]$, covers the whole input. This means that the rule $S \to DP\ VP$ can generate everything between nodes 0 to 3, which happens to be the result sentence. Hence, the parser would accept the input.

To be able to create parse trees, chart parsers usually feature additional structure that stores information about every inference rule applied. Parse trees can be constructed using

2. ϵ symbol is left out as $[A \to \cdot\epsilon, i, i]$ would make no sense for parsing.

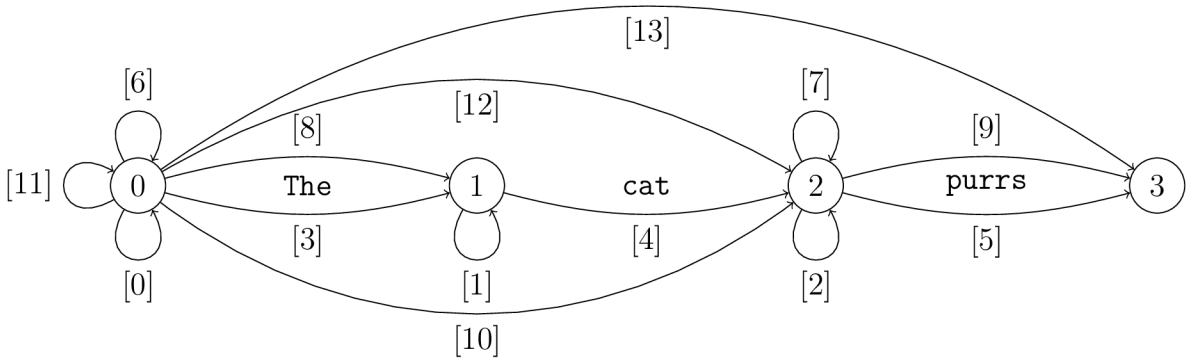


Figure 3.3: A chart of a bottom-up parser for sentence "The cat purrs"

dynamic programming; that is, for each inference we can create a new tree using those that were constructed before.

Initialization of $[A \to \cdot\alpha, i, i]$ is represented by a single root node of value A.

Completion that uses edges $[A \to \alpha \cdot B\beta, i, j], [B \to \gamma\cdot, j, k]$ with trees T_A and T_B respectively will yield a new tree by adding T_B to the T_A as its right-most child.

Scanning of edge $[A \to \alpha \cdot a\beta, i, j]$ will yield the same tree created for the source edge except with leaf a added as its new right-most child.

Prediction is analogous to the initial inference.

The table 3.2 depicts these steps (a dashed edge from a node to a string α represents the edges to all subtrees created for symbols in α).

Inference	Parameters	Result
Initial and **Prediction**	Using $A \to \alpha$	$[A \to \cdot\alpha, i, i]$ A
Completion	$[A \to \alpha \cdot B\beta, i, j]$ $[B \to \gamma\cdot, j, k]$ A B α γ	$[A \to \alpha B \cdot \beta, i, k]$ A α B γ
Scanning	$[A \to \alpha \cdot a\beta, i, j]$ A α	$[A \to \alpha a \cdot \beta, i, j+1]$ A α a

Table 3.2: Parse tree construction for chart parsers

As we mentioned earlier, chart parsers are able to recognize all context-free languages. If the derivation of the input sentence is ambiguous, at some point an edge that already

exists will be inferred by completion; however, the source edges may be different. Hence the aforementioned auxiliary structure must be able to store more than one inference record and each record must be examined when creating parse trees.

Chart parsing is clearly an easy and effective way to parse context-free languages. The implementation is not very complex and can easily handle ϵ-rules and ambiguity. However, one of its limitations is that while recognition part is polynomial ($\mathcal{O}(n^3)$ time complexity), the tree creation is **exponential**. This is easy to see; consider a grammar $S \to S\ S \mid x$, the grammar of **Catalan's problem** [20]. For an input consisting of n terminals there are C_{n-1} derivation trees, where C_n is the n-th **Catalan number** defined as $C_n = \frac{1}{n+1}\binom{2n}{n}$.

Grammars with ϵ rules pose another kind of problem. Consider a grammar with rules $S \to S\ S \mid x \mid \epsilon$. It is a valid grammar, but any sentence it generates can be derived in **infinitely many** ways. To derive a sentence of n symbols, we simply apply $S \to S\ S$ as many times as we like (but at least n times). Then we apply $S \to x$ exactly n times (on any S); the rest will be cancelled out by $S \to \epsilon$. As we stated in the section 3.2, ϵ rules can be eliminated algorithmically. However, natural languages almost never use constructs like the one mentioned before, and for this reason it is speculative whether the elimination should be implemented or not.

In the following sections we are going to look at other interesting parsing techniques; however, we are not going to cover them in full detail as they are more complex to implement.

3.8.4 Head-driven chart parsing

The general idea of head-driven chart parsing was introduced by Kay in [21]. It can be seen as a generalization of chart parsers.

These parsers require a context-free **head** grammar (CFHG), which is an extension to the context-free grammar. Sikkel in [18] defines a context-free head grammar as the 5-tuple $\mathcal{G} = (N, \Sigma, P, S, h)$ where (N, Σ, P, S) is a context-free grammar. In addition h is a function $P \to \mathbb{N}$ for which it must hold that $h(A \to \epsilon) = 0$ and $1 \leq h(A \to \alpha) \leq |\alpha|$ if $\alpha \neq \epsilon$. The head of the rule $p \in P$ is then the $h(p)$-th symbol of the right-hand side of p. Heads in rules are usually underlined, for instance $A \to \alpha\ \underline{C}\ \beta$.

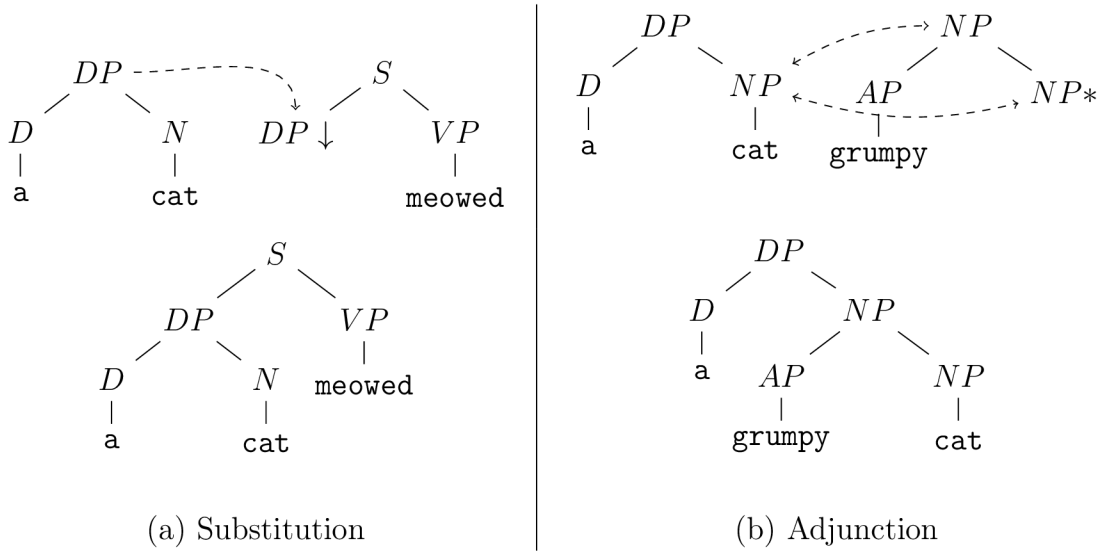
The edge used by the parser is a tuple $[A \to \alpha \cdot \beta \cdot \gamma, i, j]$ where i, j are known from chart parsers and $(A \to \alpha\beta\gamma) \in P$. The head of the rule is contained in β. There are more inference rules [18] (for shifting each of the dots) and are more complex than rules of chart parsers from the previous section. However, the number of edges created is lower because the parsing starts from the head of a rule, which is the most significant part; and not from the left-most symbol.

3.8.5 Tree-adjoining grammar parser

Tree-adjoining grammars were described by Joshi, Levy and Takahashi in [22] as Tree-adjunct grammars. They represent a different approach to parsing with the notion of cross-dependencies. Grune in [16] gives an example of a Dutch sentence "`Daar doe ik niet aan mee`" (I do not participate in that). The words `meedoen` (participate) and `daaraan` (to that) have been split up. It is very similar to German separable verbs, for instance `zuschauen` (to watch) and "`Ich schaue sie zu`" (I see them). These sentences show examples of cross-dependencies, as parts of words are scattered across the sentence.

The Tree-adjoining grammar is a 5-tuple $\mathcal{G} = (N, \Sigma, S, \mathbf{I}, \mathbf{A})$ [23]. N, Σ and S are known from CFGs. Moreover, $\mathbf{I}$ is a finite set of **initial trees** and $\mathbf{A}$ is a finite set of **auxiliary trees**. The set $\mathbf{I} \cup \mathbf{A}$ is called a set of **elementary trees**. Nodes of trees are labelled by non-terminals, leaf nodes by terminals or ϵ. Leaves can also contain nonterminal symbols, but these must be marked with $\downarrow$ and are supposed to be substituted. Auxiliary trees have exactly one nonterminal leaf with the same value as its root. This leaf is called the **foot** and is usually marked by $*$.

New trees are derived by **adjunction** or **substitution**. Former "splits" some interior node, replacing it by an auxiliary tree whose foot will get substituted by the remaining subtree of the split node. Substitution replaces a marked node in the initial or previously derived tree by another tree. These operations are depicted in the figure 3.4.



(a) Substitution (b) Adjunction

Figure 3.4: Substitution and adjunction examples

As shown in [24], TAG can generate all context-free languages. In addition it can generate languages that are not CFL, for instance $\{a^n b^n c^n \mid n \in \mathbb{N}\}$; though it is not as strong as type 1 grammars. The set of languages recognized by TAG parsers is usually called **mildly context-sensitive** (MCSL) or **tree-adjoining languages** (TAL).

There are several TAG parsers, for instance see [24]. However, these are usually more time-complex in comparison with context-free parsers (time complexity class of $\mathcal{O}(n^6)$).

3.8.6 LR parser

LR parser is a bottom-up parser invented by Donald Knuth in 1965 [25]. The **L** expresses the fact that the parser reads input tokens from left to right, **R** means that the parser creates a reversed **rightmost derivation**. The right-most derivation chooses the right-most nonterminal to apply a rule on when deriving a sentence.

Radek Lukáš in his thesis [4] chose this parser as he implemented an algorithm for parsing natural language phrases to MotiveQuery (described later). This parser can effectively (in $\mathcal{O}(n)$ time) parse deterministic context-free languages. It is very convenient for programming languages, but too weak for natural languages. For this reason, we are not going to discuss it any further.

3.9 Semantic analysis

Compilers and interpreters perform semantic analysis to check whether the input program is consistent with language definition. The most important part of semantic analysis is *type checking* [15], where the analyser inspects operators and their operands to confirm that their types match. For instance, consider an expression `"xyz" / 3;` of the C programming language. In this case, operands of `/` are string `"xyz"` (C type `const char[4]`) and an integer `3` (C type `int`). Syntactically this expression is correct. However, `/` is not defined for these argument types; hence the semantic analysis would fail for any type that fits or is *implicitly convertible* to `int` or `const char[4]`.

Implicit type conversion, also known as **coercion**, is one of syntax tree modifications analyser is allowed to perform. Consider an expression `3 + 2.0`. The first operand is of type `int`, the second is `double`. In this case the compiler may convert `3` to `double` of value `3.0` so both operands would be of the same type.

As we have indicated earlier, semantic analysis of natural language infers the language-independent meaning of a syntactical structure. This meaning is captured in a formal system called the **meaning representation** [1]. An algorithm that can create such systems must *know* facts about this world that are not logical, but rather empiric. For instance, one could say that "birds fly". However; the fact that penguins are birds that cannot fly is a kind of information that must be observed empirically. The field of artificial intelligence that studies ways how to store this knowledge is called **knowledge representation and retrieval**.

3.10 Code generation

Usually a compiler contains two code generators – an **immediate code generator** and a **code generator**. The former simply rewrites the high-level code representation (i.e. parse tree) into some low-level representation (i.e. bytecode). The latter rewrites this low-level representation into final target code. It is required that result code is as efficient as possible, high quality and must preserve the semantic meaning of the original source code [15].

The goal algorithm of this thesis should be able to transform a parse tree into a formal representation in some target object model. This process should be easy to extend or modify, so the transformation itself should not be hard-coded, but rather rule-based with tree pattern matching. For this reason it will be implemented using *interpreter* approach, which will rewrite parse trees into object models according to rules of the target language.

3.11 Issues with natural language processing

As we stated earlier, natural languages are not deterministic. In fact, they are highly ambiguous. Consider an English sentence "Put that over there." This sentence is without a doubt syntactically correct; but with no further information about the situation (who the speaker is talking to, what object he is talking about and so on), we would hardly get useful results of higher levels of analysis. This is but one of many problems we can encounter when trying to analyse sentences of human languages.

3.11.1 Natural languages are not context-free

It is very likely that no natural language is regular. For instance, Noam Chomsky in 1957 showed this about English [14]. After that, the "context-freeness" of natural languages had been debated until 1985 when Shieber showed that German is not a context-free language [26].

This fact makes natural language processing (especially its understanding) more difficult. Context-free languages have convenient properties and are generally recognizable in $\mathcal{O}(n^3)$ time as we have seen earlier. But even if we consider mildly context-sensitive languages, it is very plausible that this formalism is not strong enough to describe all natural languages (for instance, see [27]).

However, for the practical purposes and reasons described above, NLP usually "approximates" natural languages to be context-free or mildly context-sensitive.

3.11.2 Ambiguity

Ambiguity, **vagueness** or less common **inexactness** is a property of some object that means it can be interpreted in two or more ways. In linguistics, there are several types of ambiguity and we are going to look at some of them.

"Lead" rhymes with "read", but "lead" also rhymes with "read"

Homograph is a word that shares the spelling with another word, but their meaning is different. **Homophones** are pronounced the same but differ in meaning. Words that are homographs and homophones at the same time are called **homonyms** [28]. The table 3.3 clarifies these definitions. For instance, "lead" (a verb) and "lead" (the metal) are homographs, "meat" and "meet" are homophones; neither are homonyms though. True homonyms are for example "book" (a noun) and "book" (a verb).

The disambiguation is usually inferred from the sentence where the word is used.

	spelling	**pronunciation**	**meaning**
homographs	**same**	*does not matter*	**different**
homophones	*does not matter*	**same**	**different**
homonyms	**same**	**same**	**different**

Table 3.3: Homographs, homophones and homonyms

"A ship shipping ship that ships shipping ships."

Lexical ambiguity occurs when a sentence contains similar words with different meanings. As with homonyms, the actual meaning of the word must be inferred from the context or the position in the sentence. The example in the title contains different forms of word "ship", which can mean a large boat or transport on a ship.

"Stolen painting was found by a tree"

Syntactical ambiguity happens when a sentence has an ambiguous syntactical structure. Formally it means that there exist more than one parse tree for the sentence. Humans usually resolve this ambiguity easily. The title example offers two interpretations: either the painting was found lying near a tree or some tree went for a walk and it just so happened it stumbled upon a stolen painting. Clearly, the latter interpretation is nonsense since trees cannot walk. However, should the disambiguation be performed algorithmically, the algorithm would need to *know* this empiric fact.

An interesting approach to cope with syntactical ambiguity is **statistical parsing**. Statistical parsers assign probabilities to grammar rules, after parsing they compute probabilities of generated trees and choose the most likely ones.

3.11.3 Metaphors and metonymys

A **metaphor** is a figurative use of words. In the section about pragmatics we stated an example "I killed that process" which also happens to be a metaphor. The usage of word "kill" is not literal, but obviously figurative.

A **metonymy** is an object referenced by an expression of similar, yet different meaning. This expression is often shorter than the name of the object. For instance, the sentence often featured in American films "I'm calling Washington" does not mean that the speaker was going to call every person that has ever lived in Washington. The name of the city refers to the Government of the United States of America.

3.11.4 Resolution

The previous list of problems that natural languages introduce is far from complete. Some of these problems (usually syntactically ambiguous sentences) are even hard to solve for humans. There are also efforts to address these issues (like statistical parsing or reference resolution), but unfortunately they make the processing of natural languages slightly more complex as these approaches usually introduce a high level of heuristics.

As the translation of phrases to objects is what concerns us, we are not going to address these problems by any special approach. Instead we are going to allow ambiguous interpretations, since disambiguation would be too complex. The post-processing that would chose the best interpretation will be left up to the designer of the target code.

3.12 MotiveQuery

"MotiveQuery (MQ) is a user friendly chemical language primarily designed for defining structural motifs" [29], where structural motifs are structures that can be found in polymeric biomolecules such as proteins or nucleic acids [30]. While there are many tools that can be used to analyse structural motifs (for instance, Radek Lukáš in [4], chapter 4.4 provides a brief listing), they usually treat motifs as plain, 2D structures. As a consequence, they are unable to describe complex structures based on their 3D geometry and properties. MotiveQuery, on the other hand, can do so.

The language is being developed at Faculty of Informatics, Masaryk University in collaboration with the *National Centre of Biomolecular Research* and *Central European Institute of Technology*. Its syntax is based on the Python programming language. MotiveQuery is used in several applications provided by the WebChemistry platform [3] such as SiteBinder and MotiveExplorer.

The basic elements of MotiveQuery are the `Motive`, a set of atoms, and `Motives`, a sequence of `Motive`. It also features basic query functions (`Atoms`, `Residues` etc.), filtering (`Filter`, `Count`), topology (`Path`, `Star`) and many more [29].

3.13 Summary

In this chapter we have discussed theoretical background of natural language analysis. We have also described compiler and interpreter-based approaches to this process, which we are going to use as a skeleton of the resulting algorithm. It will contain a simple extensible lexical analyser, a chart parser using the bottom-up approach from the chapter 3.8.3 (with parsing time complexity in $\mathcal{O}(n^3)$ and exponential tree creation in the worst case) and a rule-based interpreter roughly described in the chapter 3.10 (as we are going to see later, its time complexity is in $\mathcal{O}(nm)$ where n is the number of parse tree *nodes* and m is the number of rewrite rules). It will also be able to perform semantic analysis; but since the target language is not supposed to be natural, we are not going to implement semantic analysis in linguistic sense. Rather we are going to use compiler-based tree modification approach.

All of these components will be wrapped by a single analyser. It will have modular form, so any component could be replaced by a different implementation to match specific target language needs.

The solution will also feature a demo application that will prove the concept in the domain of biochemistry. It will be able to analyse motifs specified by natural language phrases and translate them to MotiveQuery using its application programming interface.

4 | Implementation

In this chapter we are going to describe an algorithm created as a result of this thesis. Its source code is attached to this thesis on a compact disc along with the library documentation. The actual implementation is written in the C# programming language, while the demo application for MotiveQuery is written in F#. Both of these languages were developed by Microsoft Corporation as a part of its **.NET** framework, which we are going to look at briefly in the next section.

4.1 .NET Framework

.NET is an application framework consisting of the **common language runtime** (CLR) and .NET Framework class library.

Common language runtime is an implementation of **common language infrastructure** (CLI), an open specification by Microsoft. The fundamental concept of CLR is language-neutral code management. The runtime manages memory and resources, thread execution and other system services. It was also designed to enhance performance with a feature called just-in-time compilation (JIT).

.NET Framework class library is a collection of general-purpose algorithms, data containers and types, network and file system services that pose as basic structures on which an application can be built easily.

The framework is language independent, so applications running on .NET can be written in any language as long as it can be compiled into CIL. The concept of common language featured in .NET introduces a significant benefit: language interoperability. More information about the .NET Framework can be found on web pages of Microsoft Developer Network [31].

There is also an open-source implementation of CLI and .NET tools called **Mono**.

4.2 Overview of the solution

The solution consists of two libraries and three demo applications. The core library **Mercury** contains key algorithms and data structures. **Mercury.Formats** contains parsers that read grammars and rewrite rules written as plain text and output instances of objects used by the core library. Demo applications are described in appendices A.

Thought **Mercury.Formats** will be discussed *after* the core library, we are going to introduce the format it recognizes throughout this chapter. This will allow us to write complex structures more conveniently.

4.3 The Mercury library

The Mercury library features key algorithms for preprocessing and parsing the input and interpreting the parse tree. All classes are stored in namespace called `Mercury` or inner namespaces.

Each namespace shares its name with a folder it is stored in; for instance, classes in `Mercury.Syntax` can be found in the folder `Syntax`.

There are only few classes contained directly in the `Mercury` namespace:

`Analyser` is the *wrapper* class that composes tokenizer, parser and interpreter into one single class that automatizes all steps of the analysis.

`AnalyserResult` is a class that stores detailed results of analysis.

`Defaults` is a *static* class that cannot be instantiated, but contains default values and convenience methods for the rest of the library.

`Extensions` static class contains all the extension methods[1] used in the library.

`LanguageInfo` is a class that stores all information about a language required by the Analyser class, for instance a grammar and a set of interpreter (rewrite) rules.

The library contains the following namespaces (we are going to discuss some of them in more detail separately):

`Exceptions` contains all the exceptions that are thrown in the library,

`Interpreting` is the most important namespace of the library, because it contains implementation of the `Interpreter` class,

`Nucleus` contains generic data structures.

`Scanning` covers lexical analysis in the `Tokenizer` class.

`Syntax` contains classes responsible for syntax analysis, that is `Grammar`, `Parser` and auxiliary classes.

The components of the library avoid using concrete implementations whenever possible. They rather use *interfaces* and the actual implementation is provided by the target application. The application can in fact create its own component, and as long as it complies with interface requirements, other library components will work with it seamlessly. For instance, the `Syntax` namespace provides `IParser` interface and other components refer to it rather than to `ChartParser` class, which is an implementation of that interface.

4.3.1 Nucleus

We begin with the `Mercury.Nucleus` namespace as it contains data structures used by the entire library. It features data collections such as `IMultiDictionary` with a few classes implementing it and the `Tree` class, which interests us as it is used to represent parse trees.

1. Special static methods that are called as if they were regular methods of a type being extended.

Trees

`Tree` class in `Nucleus.Trees` namespace represents a tree data structure. It is primarily used to represent parse trees and therefore is optimized for this purpose.

As we have seen in the chapter 3.8.3, chart parsers build trees either as bare nodes with no children or by combining previously created trees into new ones. Because of this, the `Tree` class does not offer any way to determine its parent node. If it did, it would pose a performance issue as reusing a tree twice would require to copy it entirely.

Another property of the `Tree` class is **immutability**. That is, it cannot be modified after it has been created. This allows us to introduce an optimization technique called **memoization**, which remembers the result of an expensive computation and reuses it when needed again later [32].

The convenient string representation of an instance of the `Tree` is also important. If the tree has no children, it simply returns the string representation of the stored value. Otherwise it outputs the node value and string representations of its children separated by spaces; all of this enclosed by square brackets. For instance, the left bottom tree in the figure 3.4 would be represented by the string `"[S [DP [D a] [N cat]] [VP meowed]]"`.

4.3.2 Scanning

The namespace `Mercury.Scanning` covers lexical analysis, the very first step of the entire analysis. It contains an algorithm represented by the `Tokenizer` class that breaks down raw string input into a sequence of the `Token` class instances.

Token

The `Token` class features the following properties:

`Value`	contains the lexeme recognized in the input string,
`Type`	represents the category of the lexeme,
`Position`	is the number of input characters *before* this token.

There are four token types defined by the `TokenType` enumeration: `String`, `Symbol`, `Number` and `Keyword`. Their meaning is determined by the `Tokenizer` implementation and it can be customized very easily. This implementation clearly does not offer features like grammatical categories or root words because it would be too difficult to incorporate all of them. Moreover, in the case the target language does not need this data, it would burden the analysis with redundant steps. However, the `Token` class can be inherited and extended by these additional properties in a target application that uses this library.

Tokenizer

The lexical analysis is referred to by the rest of the library by its `ITokenizer` interface. It declares only one method, `Tokenize`, which takes an instance of `string` and returns a sequence of tokens; in C# represented by `IEnumerable<Token>`.

This interface is implemented by a customisable class `Tokenizer`. Lexeme recognition is done using **Microsoft Regular Expression Language**, whose description can be found in [33]. The algorithm can recognize integral and floating point numbers, words, symbols and quoted strings. Simplified conditions for types of tokens are as follows:

`Number`	**all** of the following: a) begins with `+`, `-` or a digit, b) can contain one dot (`.`), which must be enclosed by digits, c) contains only symbols `+`, `-`, `.`, digits and nothing else
`Keyword`	**all** of the following: a) is any string defined in `keywords` parameter (see below), b) is separated from other tokens by at least one white space and is **un**quoted,
`Symbol`	is any non-alpha numeric symbol,
`String`	anything else.

These conditions can be modified by constructor parameters. The most customisable constructor takes the following arguments:

`alpha`	is a sequence of characters that will be treated as alphanumeric symbols,
`keywords`	specifies strings that should be tagged by `Keyword` token type,
`options`	can set additional flags, like `IgnoreNumbers` or `IgnoreCase`.

4.3.3 Syntax

This library namespace covers syntax analysis. It contains all classes required to parse a sequence of tokens provided by the previous step. The result of this level is a sequence of parse trees represented by `Tree<Symbol>` instances.

Symbol

The `Symbol` class represents elements of $N \cup \Sigma \cup \{\epsilon\}$ defined in the section 3.1.2. It features the properties `Name`, `Type` (determins which set the symbol belongs to) and `Token` (contains a copy of an input token or `null` if the symbol is a nonterminal symbol).

To distinguish symbols in a grammar, there are some conditions that must hold for symbol names. For instance, nonterminals begin with an upper case letter, `@` or `_` and contain only these symbols or digits. On the other hand, terminals are either numbers, words beginning with lower case letters or single characters like `+` or `.` The instance of `Symbol` representing ϵ symbol is constructed by passing the `Epsilon` symbol type as one of the parameters.

Rule

The immutable class `Rule` represents a grammar rule. It consists of two properties, `LHS` and `RHS` which contain the left-hand and the right-hand sides respectively. When constructed, an algorithm checks that `LHS` contains nonterminal symbol and `RHS` contains either exactly one epsilon or at least one other symbol.

The string representation of the rule corresponds to the formal one. Instead of $\rightarrow$ a sequence `->` is used. For instance, $A \rightarrow aXb$ rule is represented by `A -> a X b`.

Grammar

The class `Grammar` corresponds almost precisely to the formal representation of a context-free grammar (N, Σ, P, S) as it contains properties `Nonterminals`, `Terminals`, `Rules` and `Root`. The difference is that `Grammar` in addition contains `Epsilon` property as every grammar can specify its own representation of the epsilon symbol. Grammars can be created using the `GrammarBuilder` class.

Parser

The syntax analysis is represented by the `ChartParser` class. Its interface `IParser` requires method `Parse`, which takes a sequence of tokens and returns the `ParserResult` class instance.

Though `ChartParser` class does indeed represent the analysis, it only serves as a wrapper for parser. The **bottom-up chart parser** from section 3.8.3 is actually implemented in the `ChartParserInternal` class. For better performance, the `Grammar` instance passed to the constructor is rebuilt again to create an instance of the internal class `ExtendedGrammar`. It contains precomputed lists of rules with certain properties, so inference rules do not need to waste time filtering all the rules as they can fetch only those that are relevant for them.

The implementation of parse tree creation also differs from the theoretical proposal. While in the theory dynamic programming is featured, the implementation uses *divide et impera* approach with memoization of results. The reasoning behind this is that chart parser can falsely predict edges which will never be used in the result; however, dynamic programming approach would still waste time creating trees for them. Instead, the actual implementation creates trees only for the edges used in the result.

The drawback of the `ChartParserInternal` implementation is that, due to the large amount of cached data, it is not thread-safe. In other words, calling `Parse` method from two different threads at the same time could yield incorrect results or cause a runtime error. Because of this, the `ChartParserInternal` is not publicly accessible and is accompanied by the `ChartParser` class which serves as a thread-safe wrapper for the internal parser.

Scanner

Syntax analysis can additionally perform initial scanning of input tokens using an implementation of the `IScanner` interface. The idea the scanner represents is similar to that of **parts-of-speech tagger**. A scanner takes a sequence of tokens, examines it and then yields additional rules that are not in the original grammar because of efficiency or generality. The parser temporarily incorporates these additional rules into its grammar.

The `BasicScanner` class featured in the library converts token types to rules. For example, for a token `57` with type `Number` the scanner would yield `@Number -> 57`. To make

these rules different from grammar rules, the scanner prefixes its nonterminals with the symbol `@`. Thus token types are represented by nonterminals `@String`, `@Symbol`, `@Number` and `@Keyword`.

4.3.4 Interpreting

The most interesting and complex part of the analysis is interpretation. Up until now, all steps were straightforward and had a clear goal. This layer of analysis receives parse trees from the syntax analyser, but the target language is an instance of a type that is not known a priori. Therefore almost all classes implemented in the `Interpreting` namespace are *generic* and the target language is represented by their type parameter.

Rewrite rules

Rewrite rules describe how parse trees should be "rewritten" to the final form. There are many types and forms of rewrite rules; in fact, the grammar rules can be seen as one of these forms. This implementation uses rewrite rules whose left-hand side selects a part of a parse tree and right-hand side states what should be done with it. The left-hand side is built of recursive structures called **elements**. The right hand side consists of **actions** that yield objects of the target language as their return value.

To distinguish these rules from the grammar rules, their string representation contains the sequence `"==>"` as the delimiter of the left and the right side.

Elements

There are four basic elements, a `Constant`, a `Variable`, a `Wildcard` and a `Structure`. The class hierarchy is depicted in the figure 4.1.

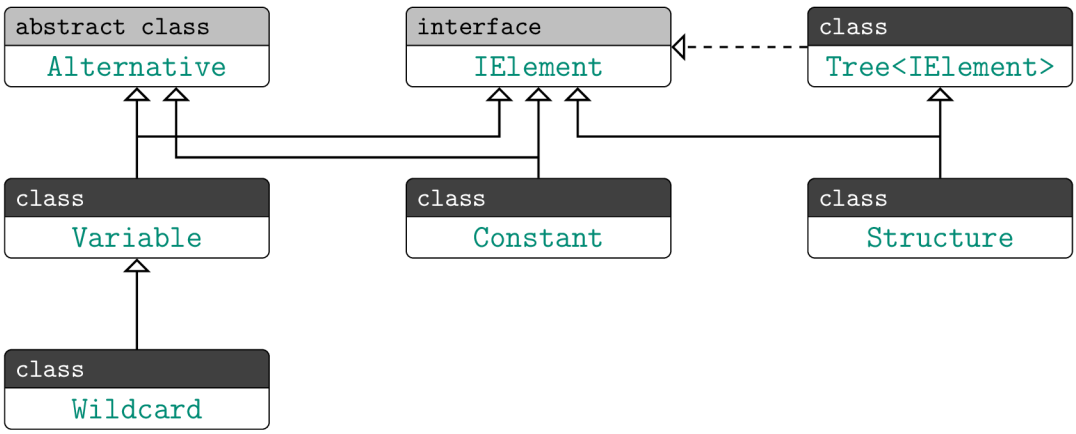


Figure 4.1: The hierarchy of elements

Alternative

`Alternative` is an abstract class that serves as a basis for `Constant`, `Variable` and `Wildcard`. Although it is not considered an element itself, it indicates which symbols derived classes match with. The class provides the `SymbolMask` property which specifies symbol sets of the grammar and the `Symbols` list that contains exact names of symbols.

The string representation of the class is of the form `symbols:mask`. The `mask` part can contain three letters, `N` for nonterminals, `T` for terminals and `e` for epsilon symbol, but can be empty as well. The second part contains symbol names separated by `|`. If there are no symbols, the colon **must** be omitted. For instance, `a|b:N` matches every nonterminal and terminals `a` or `b`, while alternative `a` matches only the terminal `a`.

Constant

The only difference between `Constant` and `Alternative` is that the former implements `IElement` interface and additional methods like `Equals` and `GetHashCode`. Their string representation is same as well.

Variable

`Variable` is a named element that instructs the interpreter to remember the matched value. It can be later referenced from the right-hand side of a rewrite rule by its name. Its string representation is of the form `#name:alternative`.

For instance, `#x:T` is a variable of the name `x` that matches any terminal. To make the representation convenient, the pattern `#name` is a shorthand for `#name::NTe`.

Wildcard

`Wildcard` is a special variable that matches zero or more nodes. Its string representation is of the form `*name:alternative`, but since it should be primarily used to match unimportant values, both `name` and `alternative` parts can be omitted. In that case a random unique name will be generated.

Structure

The `Structure` element represents a convenient way to match whole trees. It is derived from the `Tree<IElement>` class, so it can be easily mapped to a parse tree. The string representation is inherited from the parent class, that is, `[X y...]` where `X` is the root element and `y...` are string representations of child elements separated by spaces.

Consider the example tree in the figure 3.1. The structure `[S #l #r]` would match the subtree `[S [DP...] [VP...]]` and variables `l` and `r` would contain values of the `DP` and `VP` subtrees respectively.

Another example `[* * [#x:N *] *]` would match any tree that has at least one child whose root symbol is the `N` nonterminal. This subtree can be then referenced by the name `x`. In case of the tree in the figure 3.1, there are clearly two such tress.

Actions

The right hand side of rewrite rules indicates the actions that should be done if the left-hand side matched some part of a parse tree. The generic abstract class `InterpreterAction<T>` represents a declarative approach to this task. Each action has the following properties:

`Name`	contains the name of the action used in its string representation,
`Arity`	specifies the minimum and maximum number of parameters,
`ArgumentTypes`	is an array of `ArgumentType` values specifying parameter types this action can accept,
`ReturnType`	indicates the type of values this action returns,
`AllowsWildcard`	indicates whether a named wildcard can pose as an argument of the action.

The action arity consists of two numbers. They are interpreted in the way that if an action has arity (x, y) where $x < y$, then the action *requires* the first x arguments and other $y - x$ arguments are *optional*.

All classes deriving `InterpreterAction<T>` must implement the method `Invoke` method that carries out the evaluation. It should return an instance of `object` class and has the following parameters:

`arguments`	is the array of actual arguments provided,
`eval`	is a delegate that recursively invokes the interpreter,
`context`	is a supplementary object that can store additional values.

Although this class is generic, the result is of the `object` type. This is because actions are **not** restricted to only return values of the generic type parameter; they can also return values of other types like `int` or `string`. Because of this, the target application should not derive its own actions from this class, but rather extend one of the following classes.

`InterpreterAction<T,U>` is an extension to the previous class. It features an additional type parameter `U`, which represents the return type of the action. The conversion of `U` to the `ArgumentType` flag is provided automatically by the implementation. Instead of `Invoke` (which this class implements), an implementation of `Evaluate` method must be provided by the deriving class. It takes the same parameters as `Invoke`, but returns a value of the type `U` to avoid type conversions in the custom code.

`GenericAction<T>` represents actions that infer their return type only after they are provided with the formal parameters. In addition to `Invoke`, this class requires an implementation of the method `InferReturnType`. It takes two parameters, an `expected` return type and an array `actual` containing provided argument types.

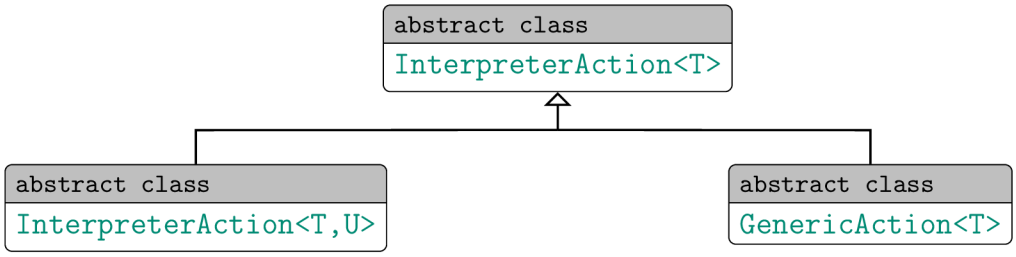


Figure 4.2: The hierarchy of actions

Argument types

Actions were designed to be simple to use and expressive enough to provide a convenient way of parse tree processing. On the other hand, these constructs are not supposed to replace a full programming language. There are no keywords, built-in control structures nor any way to declare own types as programming languages usually offer. The type system is simple, as the `ActionType` is actually a flag array where each primitive type has its own bit. Because of this, the type checking is carried out using elementary binary arithmetic.

The available values of `ActionType` are `Integer`, `Real`, `Boolean`, `String`, `Symbol`, `Tree` and `TValue`. The first four values represent types `int`, `double`, `bool` and `string` respectively. The `Tree` and `Symbol` represent parse trees and grammar symbols stored in their nodes. The interpreter implementation can *implicitly convert* the `Tree` value to `Symbol` by extracting the root value. Finally, the `TValue` represents an object of the generic type parameter `T`. The flag array also offers two additional flags that cannot be used as types on their own, `Null` and `Lazy`. Their meaning will be discussed later.

Action calls

As stated before, actions can take arguments. They are defined by formal parameters, classes that implement `IFormalParameter`. The basic types are `ConstantParameter` which holds constants and `VariableParameter` which specifies the name of a variable. Since actions do not contain their formal parameters, they are stored together in a structure called `ActionCall`. This class represents the **application** of an action to its parameters, but not the actual evaluation. To allow nested actions, `ActionCall` also implements the `IFormalParameter` interface.

Since the actions have declarative semantics (rather than imperative), the string representation of action calls is based on the syntax of the Scheme programming language. For instance, `(f #x (g #y 3))` represents an action `f` with two arguments, variable `x` and another action call `(g #y 3)`. The second call has again two argument, a variable `y` with an integral constant 3.

Type check

The rewrite rule constructor requires an instance of `Structure` and `ActionCall`. The constructor uses the `RewriteRuleValidator` class to validate the rule. This class checks that each variable is declared only once, only existing variables are used on the right-hand side and that actions and formal parameters are of compatible types.

Tree matching

The whole process of interpreting begins by calling the method `Interpret` with a parse tree as its argument. The interpreter would try to match the tree with any rule it is equipped with. These rules are fetched in the order in which they were declared. As a side effect of matching the interpreter constructs the list of variables it encounters along with their values.

Evaluation

The result of a successful matching is a list of variable instances. The interpreter would recursively inspect action calls on the right-hand side of the rule, and for each action call it would create the `Evaluator` class instance. This class is similar to the `ActionCall`, but variable parameters are replaced by actual values. Because the type system of actions is primitive and there are no collections, wildcard variables are *expanded*. That is, if a wildcard matched n nodes, they would be passed to the action as n optional arguments. Since action calls can be nested, evaluators mirror this behaviour and can be nested too.

The `Evaluator` class carries out the actual evaluation when its internal property **Data** is accessed. This process also takes into account additional `ArgumentType` flags we mentioned earlier, `Null` and `Lazy`. Before calling the `Invoke` method of its action, it goes over the values of arguments. If any argument evaluates itself to `null` and has no additional flags, the evaluator will immediately stop and return `null` as well. However, if the argument is marked with the `Null` flag, the value is ignored. Moreover, an argument marked with the `Lazy` flag will not be evaluated at all; the evaluation will be carried out automatically as a side effect when the action accesses the argument. Hence the evaluator guarantees that if the action is called, arguments without additional flags are not `null`.

Recursive evaluation and context

The method `Invoke` of the `InterpreterAction<T>` class takes three parameters. We have already discussed the `arguments` array. The second parameter is a `delegate` `eval`, which encapsulates the `Interpret` method of the interpreter that processes the tree. It can be used to recursively interpret trees from action. However, this method may return `null` if there is no interpretation. Because of this, it is safer to use nested `Eval` action (see the list of basic actions in the appendix C) as the `null` value would be handled by the evaluator.

The third parameter is an instance of the `InterpreterContext` class. It is suitable to be extended with custom properties in the target application. These properties can be used by actions to store values that would be otherwise lost or difficult to pass to other actions.

Action templates and basic actions

The library comes with action templates that were designed to simplify the implementation of custom actions. These templates are classes that encapsulate *lambda functions* and use them to implement abstract methods.

The library also contains predefined general-purpose actions that can, for instance, concatenate strings, evaluate arithmetic or boolean expressions. These can be used to pre-process arguments of custom actions. There are 49 of them, appendix C lists 8 examples and the rest can be found in the documentation.

4.3.5 Note on the semantic analysis

As we have stated before, semantic analysis is not the crucial part of this algorithm. However, the interpreter is generic enough to provide this functionality with almost no additional code.

The `SemanticInterpreter` class is implemented according to the *composite design pattern* and encapsulates two interpreters. The first one is instantiated with the `Tree<Symbol>` type parameter and therefore can simulate semantic analysis similar to that of compilers. The second interpreter is generic and performs the translation.

4.3.6 Analyser

The components we mentioned before work almost like robots on a conveyor belt. The tokenizer divides the raw string input into the stream of tokens, which is then analysed by a scanner. The additional rules it creates along with the tokens are then examined by the parser, which constructs parse trees. Finally, these trees are interpreted by the interpreter and the result is provided to the target application.

However, there is a bit of extra work required. Any of the components can fail and return a result that indicates the failure, which is not meant to be processed further. The auxiliary code would therefore have to inspect every result of each component manually.

Because of this, the library provides the `Analyser` class which takes care of this. It encapsulates all the components (referenced by interfaces to sustain generality) and performs all the steps automatically one by one. After each analysis it inspects the result and reacts to any failure. In addition to this, the analyser measures duration of parsing, translation and the entire process.

4.4 The Mercury.Formats library

This additional library provides classes that can parse context-free grammars with ϵ rules and rewrite rules specified in text files. Their names are `GrammarParser` and `RewriteRulesParser` respectively. In this chapter we also are going to summarize the format of entities presented throughout the previous section.

4.4.1 GrammarParser

The `GrammarParser` class is responsible for parsing grammar files. These consist of two parts: an optional header and a list of rules. In addition, grammar files can contain comments, lines where the first non-white-space character is `#`.

The header contains one or both of the following commands:

`root`	specifies the root symbol, which must be a nonterminal,
`epsilon`	specifies a string that will represent the ϵ symbol.

Rules are written in the similar way as the formal notation, that is, rule $A \rightarrow aXb$ can be written as `A -> a X b`. The shorthand format is also allowed, using the | symbol as the separator.

4.4.2 RewriteRulesParser

The format of files the `RewriteRulesParser` recognizes contain only comments (same as the comments in grammar files) or rewrite rules. Each rewrite rule is of the format `LHS ==> RHS`. The left-hand side is always a structure, the right hand side is always an action call.

The format of the elements (and the alternative) is as follows:

`Alternative` is declared as `list:mask` where `list` contains specific symbols separated by `|` and the `mask` contains any of `N`, `T` or `e` characters that specify grammar sets.

`Constant` is syntactically equivalent to the `Alternative`.

`Variable` has the format `#name:alternative`.

`Wildcard` can be declared as `*name:alternative`.

`Structure` is a recursive element of the format `[X y...]` where `X` is an `Alternative` and `y` is a list of zero or more elements separated by space.

Syntax of actions is adapted from the Scheme programming language. The pattern is `(f args)` where `f` is a name of an action and `args` is a list of zero or more arguments. Arguments can contain other actions, variables or constants (integer, floating point number, boolean or string). Variables are specified as `#name` without an alternative. Since they are always prefixed by `#`, there is no need to quote string constants. While `"cat dog"` would yield one constant, `cat dog` would be recognized as two constants.

4.5 The MQNLParser demo application

The application demonstrates the use of the Mercury library. It is written in the F# programming language in order to use advantages of functional paradigm, such as brevity, *declarative programming* and *lazy evaluation*.

Rules are specified in the module `LanguageData`, while custom actions are implemented in the module `Actions`. There are almost 300 grammar rules and more than 100 rewrite rules. The former module also declares `MQNLAnalyser`, which incorporates all the components using `QueryBuilderElement` type as the intermediate language and `Query` as the target language. The application recognizes the following MotiveQuery functions: `Atoms`, `Residues`, `Inside`, `Tunnels` and others.

Let us have a look at the following examples with actual program outputs (with inserted line breaks to fit the width of the page):

```
> Any zinc, oxygen or carbon atom.
  1 result (T 3 ms) (P 2 ms) (I 0 ms)
~ AtomSet[C,O,Zn]

> Yield all residues that contain at least dozen zinc or oxygen atoms.
  1 result (T 9 ms) (P 9 ms) (I 0 ms)
~ Filter[ResidueSet[],Lambda[(m0),LessEqual[12,Count[AtomSet[O,Zn],
      Symbol["m0"]]]]]

> Find all zinc atoms, oxygen atoms, amino acids with the aromatic charge
  or any residue.
  1 result (T 19 ms) (P 18 ms) (I 2 ms)
~ Or[AminoAcids(ChargeType=Aromatic)[],
      AtomSet[O],AtomSet[Zn],ResidueSet[]]

> Find all tunnels in residues starting from any oxygen atom with
  bottleneck radius eight and one thousandths and
  probe radius five point sixteen.
  1 result (T 28 ms) (P 21 ms) (I 6 ms)
~ Tunnels(ProbeRadius=5.16,InteriorThreshold=1.25,BottleneckRadius=8.001)
      [ResidueSet[],AtomSet[O]]
```

4.6 Unit tests

Each stage of the Mercury library development was accompanied with tests that helped to reveal errors in the implementation. These tests are available in the `Tests.Mercury` namespace. The amount of code blocks these tests cover is shown in the table 4.1. According to the said table, the mean test coverage of the lexical analysis (`Scanning`), syntax analysis (`Syntax`) and translation (`Interpreting`) is about 72%. Other blocks contain mainly maintenance code.

	Covered		Not covered	
Namespace	Blocks	% Blocks	Blocks	% Blocks
`Mercury`	125	36.66%	216	63.34%
`Mercury.Exceptions`	35	43.21%	46	56.79%
`Mercury.Interpreting`	1358	60.17%	899	39.83%
`Mercury.Nucleus`	250	57.33%	186	42.66%
`Mercury.Scanning`	172	78.54%	47	21.46%
`Mercury.Syntax`	956	79.34%	249	20.66%
`mercury.dll`	2896	63.80%	1643	36.20%

Table 4.1: Unit test code coverage of the Mercury library

5 Results and discussion

5.1 Performance

The performance of the solution has been tested on several grammars with different inputs. There are many factors that have impact on the performance. For instance, the number of ϵ rules and the ambiguity of grammar heavily affects the parsing performance. The results presented in this chapter can be obtained from an auxiliary application **PerfWrapper** described in the appendix A.3.

All the tests were performed on a computer with the following specifications:

CPU Intel Core i3 M350, 2.27 GHz,
RAM 4 GB, DDR3,
OS Microsoft Windows 8.1 Pro, 64-bit,

The first test is mainly focused on parsing performance, as it features the grammar for Catalan's problem, which creates exponential number of trees. The actual grammar and rewrite rules can be found in the appendix B. An example of the input is `"x x x"`, which can be parenthesized as `"((x x) x)"` or `"(x (x x))"`. This means there are two parse trees for the input. Interpreter converts these trees to the same parenthesized notation. However, the actual test is performed on inputs with thousands of possible parse trees.

The results are depicted in the table 5.1 while the figure 5.1 visualises the values, with black bars indicating the time required to parse all the trees and gray dots representing the number of parse trees.

Input		Nodes	Total time		Time per tree	
length	Trees	per tree	Parsing	Interpreting	Parsing	Interpreting
8	429	23	5 ms	8 ms	10 ns	19 ns
9	1430	26	15 ms	28 ms	11 ns	20 ns
10	4862	29	51 ms	101 ms	11 ns	21 ns
11	16796	32	212 ms	358 ms	13 ns	21 ns
12	58786	35	897 ms	1167 ms	15 ns	20 ns
13	208012	38	3696 ms	4269 ms	18 ns	21 ns
14	742900	41	15546 ms	15112 ms	21 ns	20 ns
15	2674440	44	61585 ms	53515 ms	23 ns	20 ns

Table 5.1: Performance results for test A

This shows that even though parsing time complexity is $\mathcal{O}(n^3)$ (where n is the number of input tokens), the parse tree creation is indeed exponential as we stated in the chapter 3.8.3. The table also indicates that interpreter time complexity is linear ($\mathcal{O}(nm)$ in the worst case where n is the number of tree nodes and m is the number of rewrite rules) as

the values remained relatively close to 20 ns while parsing time has increased from 10 ns to 23 ns.

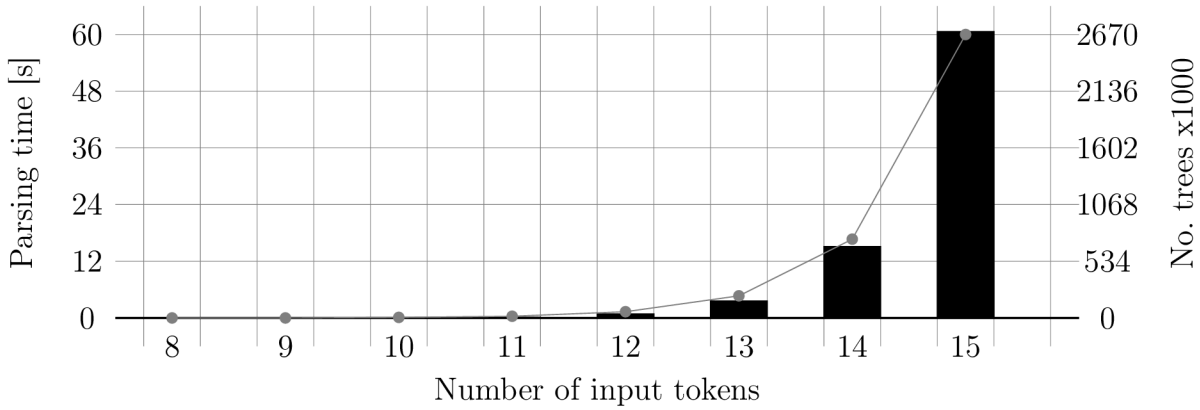


Figure 5.1: A relationship between the number of input tokens, the number of trees and total parsing time

The second test measures the performance of the **MQNLParser** application. The PerfWrapper application executes these tests 10 times and returns the average value. This reduces the impact of lazy evaluation just-in-time compiling, which can significantly slow down the evaluation of the very first query. Inputs prefixed with B have only one parse tree while those prefixed with C have more than one. All the inputs that are tested here are listed in the appendix B.

The following example represents the query B7:

```
Yield all CYS or HIS residues or any amino acid that contains exactly
two oxygen atoms.
```

with the result in MotiveQuery

```
Or[Filter[AminoAcids[],Lambda[(m0),Equal[2,Count[AtomSet[O],Symbol["m0"]]
  ]]],ResidueSet[CYS,HIS]]
```

Other example queries can be found in the appendix B.

	Time				Time		
Input	Parsing	Interpreting	Total	Input	Parsing	Interpreting	Total
B0	7 ms	0 ms	8 ms	B6	62 ms	0 ms	64 ms
B1	8 ms	1 ms	13 ms	B7	45 ms	1 ms	47 ms
B5	21 ms	20 ms	43 ms	B8	42 ms	27 ms	71 ms
B3	24 ms	26 ms	61 ms	B9	121 ms	19 ms	148 ms
B4	34 ms	0 ms	34 ms	C0	35 ms	33 ms	87 ms
B5	22 ms	3 ms	27 ms	C1	23 ms	17 ms	41 ms

Table 5.2: Results for MQNLParser performance test

5.2 Comparison with MotiveQuery Natural Language Analyser

MotiveQuery Natural Language Analyser (henceforth MQNLAnalyser) is a library developed by Radek Lukáš in his master's thesis [4]. Similarly to MQNLParser, it can analyse English phrases and translate them into MotiveQuery expressions. The actual coverage is difficult to compare as MotiveQuery is still being developed and after a year new features have been added (*meta functions*), deprecated or entirely removed (*Contains*).

As a consequence, we are only going to focus on the general properties. The MQNLAnalyser uses LR parser, which makes it faster, but is restricted to deterministic context-free grammars only. Possible ambiguous interpretations are handled by its interpreter. Also, the output of the analysis is a plain `string` which has to be parsed again to be usable in MotiveQuery tools. In contrast to this, Mercury library uses a bottom-up chart parser which enables it to handle ambiguous grammars with ϵ rules. These are without a doubt better suited for natural languages; yet slower to analyse. The interpretation is deterministic, generic and the analysis returns instances of target language.

The difference between both approaches also affects extensibility. To add a new MotiveQuery function, MQNLParser requires new grammar, rewrite rules and optionally implementations of new actions. The MQNLAnalyser can be extended by adding new entries in its extensible mark-up files (XML) that describe the language. However, some language constructs (like verbs, sequence of motives etc.) are "hard-wired" in both the parser and the interpreter. This approach is indeed required by the parser construction; but since natural languages often feature flexible constructs, it makes the extensibility slightly more complex.

5.3 Limitations and possible solutions

We have mentioned in the section 3.11 that natural languages are definitely not context-free and perhaps not even mildly context-sensitive. This limits the use of the Mercury library as well, so it can be only used to analyse languages that *approximate* the natural ones.

There are naturally certain additional improvements that can be made. Grammars with ϵ rules pose a big performance problem. As the number of these rules grows, the parser performance drops significantly due to the large amount of predicted edges. One of the possible solutions would be to implement an algorithm that eliminates these rules (from [7] or [10]). However, the elimination of ϵ symbols from rewrite rules would be required as well. This would be very difficult to accomplish as rewrite rules are recursive structures. Other solution would be to replace the chart parser with a head-driven chart parser (3.8.4). Even though it is more complex and requires head grammars, it is generally faster as it starts the analysis from the most significant lexemes (heads).

Another issue is "debugging" of grammars in the Mercury library. The only thing that can be used to search for a problem is the chart, which is usually hard to analyse. It would be far more convenient to have a module that can try to recover from syntax errors by analysing the chart algorithmically and pointing out the exact problem.

6 | Conclusion

The main aim of this thesis was to design and implement an algorithm capable of analysis and translation of natural language phrases into an object model. This was not a trivial task, as natural languages are extremely complex and their computational processing is still an evolving field of computer science. As a consequence, this thesis deals with an approximation of natural languages to a set of formal ones called the context-free languages.

A brief introduction to formal languages is provided in the third chapter, along with an insight of natural language processing. It introduces an interpreter-based model of translation, which is later used as a skeleton of the resulting algorithm. Apart from that, it discusses several parsing techniques that can be used to analyse context-free languages along with their pros and cons. We have chosen the bottom-up chart parser as it is simple to implement and reasonably efficient at the same time.

The fourth chapter discusses the Mercury library, the result of this thesis. The most important parts are syntax analysis and parse tree translation. The latter is represented by a rewrite-rule-based interpreter. It features a parse tree matching algorithm and introduces a simple, yet extensible and powerful formalism that can process these trees. The whole translation is not restricted to any concrete type of a result, but rather features generic components. These can be instantiated with any type that can represent the target language reasonably well.

As the section with results indicates, the library can indeed handle context-free languages and translate them into custom object types. This is well demonstrated by the MQNLParser application. It uses the Mercury library to translate motifs (chemical structures) described by English phrases into the MotiveQuery builder expressions.

Future development of the library could focus on better handling of syntax errors, typeahead suggestions or support for languages that would require morphological analysis.

Bibliography

[1] Daniel Jurafsky and James H. Martin. *Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition*. Prentice Hall, Upper Saddle River, 2nd edition, 2008.

[2] Stuart C. Shapiro. Artificial Intelligence. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 54–57. John Wiley & Sons, Inc., New York, 2nd edition, 1992.

[3] Webchemistry: Services and applications. Accessed: 2014-05-14. URL: http://webchem.ncbr.muni.cz/Platform/.

[4] Radek Lukáš. Specifikace molekulárních motivů pomocí přirozeného jazyka [online]. Diplomová práce, Masarykova univerzita, Fakulta informatiky, 2013. Accessed: 2014-04-26. URL: http://is.muni.cz/th/389911/fi_m/.

[5] Gerard Salton, Amit Singhal, Mandar Mitra, and Chris Buckley. Automatic text structuring and summarization. *Information Processing & Management*, 33(2):193 – 207, 1997.

[6] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 2nd edition, 2003.

[7] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[8] Noam Chomsky. Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124, September 1956. doi:10.1109/TIT.1956.1056813.

[9] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[10] Ivana Černá, Mojmír Křetínský, and Antonín Kučera. *Formální jazyky a automaty I*. Elportál. Masarykova univerzita, Brno, 1st edition, 2006. Accessed: 2014-04-26. URL: http://is.muni.cz/elportal/?id=703389.

[11] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

[12] Peter S. Landweber. Three theorems on phrase structure grammars of type 1. *Information and Control*, 6(2):131–136, 1963.

[13] James Allen. *Natural Language Understanding*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, 2nd edition, 1995.

[14] Noam Chomsky. *Syntactic Structures*. Mounton, The Hague, 1957.

[15] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2 edition, 2006.

[16] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Monographs in Computer Science. Springer, 2007.

[17] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970. doi:10.1145/362007.362035.

[18] Klaas Sikkel. *Parsing schemata – a framework for specification and analysis of parsing algorithms*. Springer, 1997.

[19] Martin Kay. Algorithm schemata and data structures in syntactic processing. In Barbara J. Grosz, Karen Sparck-Jones, and Bonnie Lynn Webber, editors, *Readings in Natural Language Processing*, pages 35–70. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1986.

[20] Heinrich Dörrie. *100 Great Problems of Elementary Mathematics: Their History and Solution*. Dover Books on Mathematics Series. Dover Publications, 1965.

[21] Martin Kay. Head driven parsing. *Proceedings of Workshop on Parsing Technologies*, pages 52–62, 1989.

[22] Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. Tree adjunct grammars. *Journal of Computer and System Sciences*, 10(1):136–163, 1975.

[23] Miguel A. Alonso, Éric Villemonte de La Clergerie, Vítor J. Diaz, and Manuel Vilares. Relating tabular parsing algorithms for LIG and TAG. In John Carroll, Giorgio Satta, and Harry Bunt, editors, *New Developments in Parsing Technology*, volume 23 of *Text, Speech and Language Technology*, pages 157–184. Kluwer Academic Publishers, 2005.

[24] Laura Kallmeyer. *Parsing Beyond Context-Free Grammars*. Cognitive Technologies. Springer, 2010.

[25] Donald Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.

[26] Stuart M. Shieber. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8(3):333–343, 1985.

[27] Gregory M. Kobele. Mild context sensitivity is not enough, 2006. Accessed: 2014-05-02. URL: http://home.uchicago.edu/~gkobele/files/2006-Kobele06MildContextSensitivityIsNotEnough.pdf.

[28] James B. Hobbs. *Homophones and Homographs: An American Dictionary, 4th ed.* McFarland, 2006.

[29] Motivequery language reference. Accessed: 2014-05-08. URL: http://webchem.ncbr.muni.cz/Wiki/MotiveQuery_Language_Reference.

[30] Voet, J.G. Voet, and C.W. Pratt. *Fundamentals of Biochemistry: Life at the Molecular Level*. Wiley, 2013.

[31] Overview of the .NET framework. Accessed: 2014-05-06. URL: http://msdn.microsoft.com/en-us/library/zw4w595w(v=vs.110).aspx.

[32] Peter Norvig. Techniques for automatic memoization with applications to context-free parsing. *Comput. Linguist.*, 17(1):91 – 98, March 1991.

[33] Regular expression language - quick reference. Accessed: 2014-05-06. URL: http://msdn.microsoft.com/en-us/library/az24scfc(v=vs.110).aspx.

Appendices

A | Demo applications

A.1 The MercuryParser application

MercuryParser is a demo application that can be used to test the features the Mercury library. Though it can be instructed via command-line arguments, the default mode when run without arguments is *interactive* (the same effect has the `/interactive` option). List of available options can be displayed by running the program with the `/?` option.

Interactive commands are always prefixed with a colon (`:`). Inputs that are not prefixed with this symbol are considered inputs for the analyser. Some of the commands are:

`:a` | `:actions`	shows rewrite rules,
`:c` | `:chart`	shows chart,
`:g` | `:grammar`	shows grammar,
`:i` | `:interpts`	shows interpretations,
`:l` | `:load`	loads grammar and rewrite rules (see below),
`:q` | `:quit`	quits the program,
`:t` | `:trees`	shows trees.

The `:load` command requires one argument, the name of a file that contains a context-free grammar in the format described in 4.4.1. It can also take an optional argument, the name of a file with rewrite rules.

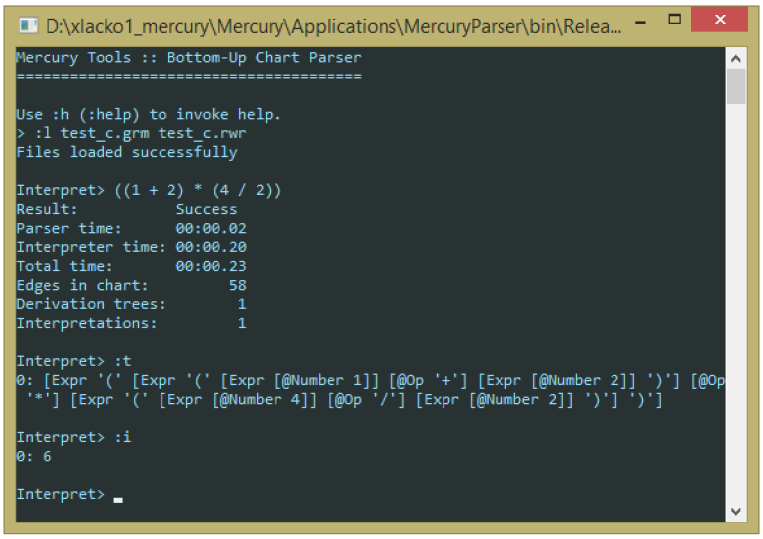


Figure A.1: A view of the MercuryParser application

A.2 The MQNLParser application

The MQNLParser application was briefly described in the section 4.5. The application recognizes the following commands:

`grammar`	prints the grammar,
`rwrules`	prints rewrite rules,
`exit`	quits the application.

All other inputs will be analysed. Slow response time of the first analysis is not a bug, it is a consequence of F#'s lazy evaluation and just-in-time compiling.

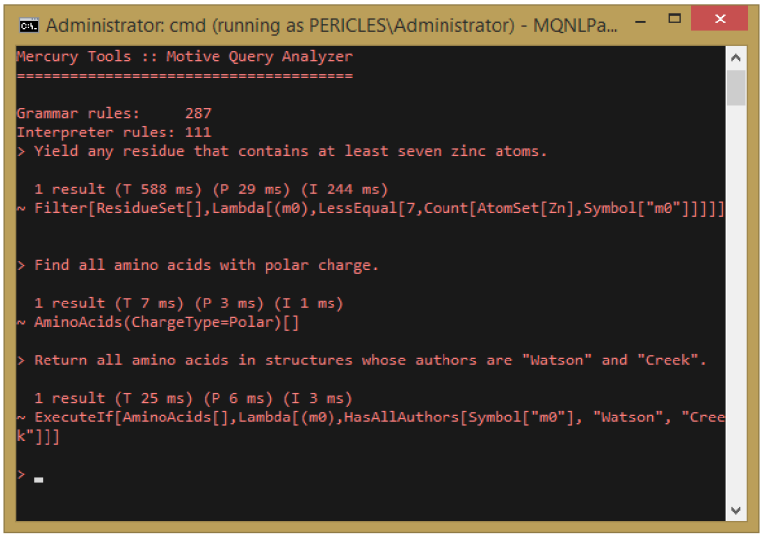


Figure A.2: A view of the MercuryParser application

To extend the language that the application recognizes one could use operators defined in the `DeclOp` module and add new rules to the `LanguageData` module. These operators allow the grammar to be brief and compiled with the entire application without the need of additional parsing. The `Language` module also contains *metafunctions* that can create several rules at once using phrasal patterns (see below). There are many of these operators, but the code is documented and function of each operator is sufficiently explained.

Examples:

Phrases (concept example)

```
pp2 NOUN            PREP     ADJ       OBJ
pp2 "AminoAcid"     "PAgent" "Charge"  "NCharge"
```

creates a preposition phrase with object of the form `DET NOUN PREP ENUM(ADJ) DET OBJ`. It can match sentences like `"(any) (amino acid) (with) (polar or aromatic) (charge)"`. Parenthesis separate the constituents, i.e. `PREP = with`.

Meta functions

```
metaEnum Q                    == EnumQ -> Q ;  EnumQ -> Q Conj EnumQ
metaEnum @AtomName
```

constructs enumeration for `Q` where `Conj` represents conjunctions. The example with `@AtomName` can for instance match `zinc or oxygen`.

```
lex NAME        [X; Y; ...]  == NAME -> X | Y | ...
lex "NParent"   [ "parent"; "structure"; "motive"; "motif" ]
```

is a convenience structure. The name suggests that it should be used to construct the *lexicon*.

Grammar operators

```
"A" ---> ["B";"C"]            == A -> B C
"A" -->> [["B";"C"]; ["D"]]   == A -> B C | D
```

Element operators

```
!~   "A"                      == A          %% constant
!~~ [ "A"; "B" ]              == A|B        %% constant
!&"x"                         == #x::NT     %% variable
"x" &~   "A"                  == #x:A       %% variable + alt.
"x" &~~ ["A";"B"]             == #x:A|B     %% variable + alt.
!~"x" |-- [ !~"A"; !~"B" ]    == [x A B]    %% structure
w(); eps(); any()             == *; e; NTe  %% wildcards, epsilon
```

Action operators

```
f  |< [x; y; z]              == (f x y z)  %% action call
f ^|<  x                     == (f x)      %% action call
   !<  f                     == (f)        %% action call
   !* "x"                    == #x         %% variable parameter
ip 3    ; rp 3.0             == 3    ; 3.0 %% integer and real const.
bp true ; sp "xyz"           == true ; xyz %% bool and string const.
ev'   "x"                    == (eval #x)  %% shorthand
```

Complex examples

```
!~"Filter"  |-- [!&"m"; w(); "p" &~ "Predicate"]
            ===> (filter |< [ev' "m"; ev' "p"])
[Filter #m * #p:Predicate] ==> (filter (eval #m) (eval #p))

!~"@Number" |-- [!&"n"]
            ===> (toqbv ^|< parsei ^|< name ^|< !*"n")
[@Number #n] ==> (toqbv (parseinteger (name #x)))

!~"Count"   |-- [!~"NCount"; "ms" &~ "EnumMSeq"; !~"NCount"]
            ===> (cntlmb ^|< qbeor ^|< ev' "ms")
[Count NCount #ms:EnumMSeq NCount] ==> (cntlmb (qbeor (eval #ms)))
```

A.3 The PerfWrapper application

"PerfWrapper" is an abbreviation of "Performance Wrapper". This applications wraps the parser and interpreter in a loop which is useful for profiling with Microsoft Visual Studio Profiling Tools.

When run without command-line arguments, it assumes it is being run by the profiler and executes the loop. The parameters of profiling are defined by `private static` variables:

`rsname`	resource name,
`count`	number of iterations,
`perfmod`	specifies the component, can be `Parser` or `Interpreter`.

The application expects that files `<rsname>.grm` (grammar), `<rsname>.rwr` (rewrite rules) and `<rsname>.in` (inputs) are built within it as **embedded resources**.

To run tests from section 5.1 there are command-line arguments `/a` for the first test (A) and `/b` for the second test (B and C).

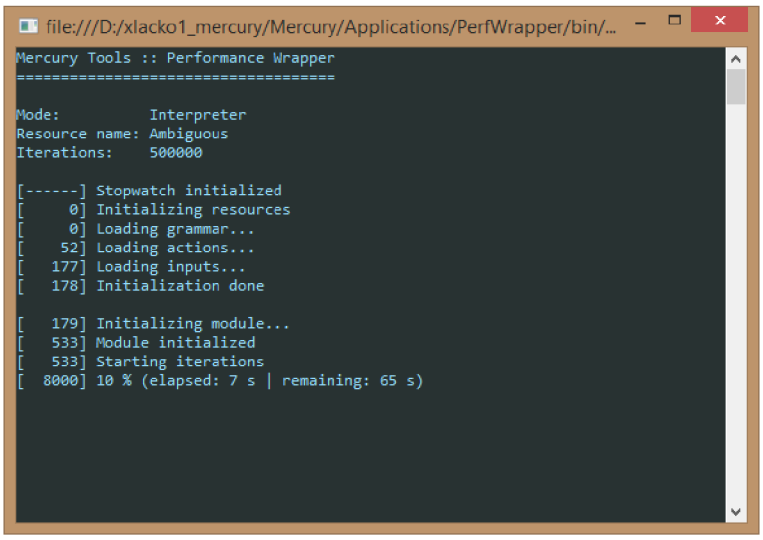


Figure A.3: A view of the PerfWrapper application in the profiling mode

B | Test inputs (examples)

B.1 Test A inputs

The grammar for the first test is specified in 5.1 (`catalan.grm`)

```
S -> S S | x
```

The rewrite rules for the test (`catalan.rwr`) are

```
[S #l #r] ==> (concat "(" (eval #l) " " (eval #r) ")")
[S     x] ==> (string x)
```

Example analysis (from MercuryParser)

```
Input: "x x x x"

Interpretations:
0: ((x x) (x x))                    1: (x (x (x x)))
2: (x ((x x) x))                    3: ((x (x x)) x)
4: (((x x) x) x)
```

B.1.1 Test B and C inputs

```
in bc.in
B0 Any zinc or oxygen atom.
B1 All amino acids with polar charge.
B2 Find any atom with its IDs in range from seven to sixty.
B3 Yield all residues that contain no more than five zinc
   or oxygen atoms.
B4 Return all atoms of hydrogen, lithium, sodium, potassium,
   oxygen and carbon.
B5 Return all amino acids in structures whose authors are
   "Watson" and "Creek".
B6 Find all zinc atoms, oxygen atoms, amino acids with the
   aromatic charge or any residue.
B7 Yield all CYS or HIS residues or any amino acid that contains
   exactly two oxygen atoms.
B8 Find all tunnels in residues starting from any oxygen atom
   with interior threshold five and seven tenths.
B9 Fetch any residue that contains exactly one hundred and
   twenty-three million, four hundred and fifty-six thousand,
   seven hundred and eighty-nine oxygen atoms.
C0 Return all residues that contain oxygen atom, zinc atom or
   any amino acid.
C1 Fetch any residue that contains a hundred oxygen atoms.
```

C | Action templates and basic actions

Action templates and basic were briefly introduced in section 4.3.4. In this appendix we are going to use Haskell function type syntax, i.e. `a -> b -> c` is a type of function that takes values of type `a` and `b` and returns a value of type `c`.

C.1 Action templates

Action templates are instantiated with lambda functions and the rest is implemented automatically. The Mercury library offers the following functions:

`OperationAction`	lambda type `U -> U -> V` Implements operations on two values.
`FoldingAction`	lambda type `V -> U -> V` and `seed` of value `V` "Folds" the values list using the lambda and the initial value of `seed`.
`ConvertAction`	lambda type `V -> U` Applies the lambda on a value and returns its result.
`ConstantAction`	takes a value of a generic type Creates a constant from a value that is not compatible with the basic type system.
`ArithmeticAction`	lambda type `dynamic -> dynamic -> dynamic` Implements arithmetic operations, similar to `FoldingAction` but does not require an initial value.

Example – a function that extracts the name from a symbol:

```
new ConvertAction<T, Symbol, string>("Name", x => x.Name);
```

C.2 Basic actions (excerpt)

```
<a> is any type
(ParseInteger   String) -> Integer  %% Parses integer
(Add        Integer...) -> Integer  %% Sums all its arguments
(String         <a>   ) -> String   %% equivalent of ToString()
(Join  String   <a>...) -> String   %% Joins values with the string
(And        Boolean...) -> Boolean  %% Logical multiplication
(Symbol           Tree) -> Symbols  %% Extracts the root symbol
(Try           <a> <a>) -> <a>      %% Evaluates its first argument,
                                    %% if it catches an exception,
                                    %% returns the second
(Error          String) -> <a>      %% Throws an exception
```

The full list of all 49 actions can be found in the documentation.

D | Contents of the attached CD

- `Documentation` – code documentation generated by Doxygen,
- `Examples` – example grammars and rewrite rules,
- `Executables` – compiled applications and libraries,
- `Mercury` – source codes in as a Microsoft Visual Studio 2013 solution

`Applications`	contains MercuryParser, MQNLParser (F#) and PerfWrapper,
`Mercury`	contains the Mercury library,
`Mercury.Formats`	contains the Mercury.Formats library,
`Tests`	nit tests

- `Thesis` – copy of this document

List of Tables

3.1 Bottom-up chart parsing of "`The cat purrs`" *21*
3.2 Parse tree construction for chart parsers *22*
3.3 Homographs, homophones and homonyms *27*
4.1 Unit test code coverage of the Mercury library *41*
5.1 Performance results for test A *42*
5.2 Results for MQNLParser performance test *43*

List of Figures

3.1 An example of a derivation tree *12*
3.2 Stages of the compilation process adapted from [15] *17*
3.3 A chart of a bottom-up parser for sentence "The cat purrs" *22*
3.4 Substitution and adjunction examples *24*
4.1 The hierarchy of elements *34*
4.2 The hierarchy of actions *36*
5.1 A relationship between the number of input tokens, the number of trees and total parsing time *43*
A.1 A view of the MercuryParser application *50*
A.2 A view of the MercuryParser application *51*
A.3 A view of the PerfWrapper application in the profiling mode *53*