BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

AUTOMATIC TRANSLATION AND COMPARISON OF LUMERICAL AND MEEP SIMULATIONS

AUTOMATICKÝ PŘEKLAD A SROVNÁNÍ SIMULACÍ MEZI MEEP A LUMERICAL

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **DANIEL MAČURA**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ MILET, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2025

Abstract

This thesis aims to provide a comparison between Ansys Lumerical and Meep FTDT simulation tools. A transpiler from Lumerical Scripting Language to the Meep Python interface was implemented. FDTD simulation were first automaticaly results are compared with analytical results.

Abstrakt

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

Keywords

CEM, FDTD, Meep, Ansys Lumerical, Transpiler, Source-to-Source Compiler

Klíčová slova

Sem budou zapsána jednotlivá klíčová slova v českém (slovenském) jazyce, oddělená čárkami.

Reference

MAČURA, Daniel. *Automatic translation and comparison of Lumerical and Meep simulations.* Brno, 2025. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Milet, Ph.D.

Automatic translation and comparison of Lumerical and Meep simulations

Declaration

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana X... Další informace mi poskytli... Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . .
Daniel Mačura
March 31, 2025

Acknowledgements

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant apod.).

Contents

List of Figures

Chapter 1

Note to the Reader

Through the thesis, the expression „iff“ refers to „if and only if“.
Except if explicitly stated otherwise, the following notation conventions apply throughout the thesis. In the context of regular languages, lower case Latin alphabet letters, such as $a, b, c, \ldots$, refer to terminal symbols. Upper case Latin alphabet letters, such as $A, B, C, \ldots$, denote nonterminal symbols.

Chapter 2

Introduction

In modern society humans are reliant on a multitude of technologies such as internet, mobile phones, television, radio, microwave ovens, camera sensors, lasers, light emitting diodes, electrical motors, medical imaging systems and many others. All of these are electromagnetic devices, thus without a doubt electromagnetism plays a key role in day-to-day life.

Electromagnetic field theory is the underpinning framework used to study the effects of electromagnetic phenomena at scales where quantum effects are negligible. It studies the interactions between electric charges and currents (Currents are often referred to as electric charges in motion). Maxwell's equations, a set of fundamental coupled partial differential equations form the cornerstone of classical electromagnetism. Closed form analytical solutions are complex, know for only simple cases and usually inapplicable in real world use.

However with the rapid increase of available computational power, the use of analytically simple but computationally taxing methods arose as they became more available. These method provide solutions to more general problems compared to their counterparts. The branch of electromagnetics that focuses on such methods is know as computational electromagnetics (CEM). CEM allows for simulating more complex problems and verifying designs before production of prototypes. They provide key insight in to the operation of electromagnetic devices and even certain information that may be unattainable by classical analytical methods. More over with the ability to tweak parameters and resimulate, CEM brings about an advent of design optimization in electromagnetic devices.

Today, there are many such tools and software packages, some commercial and others open-source. It is important to be able to verify results against other implementations, enabling researchers to verify that results are not artifacts of a particular solver but are physically meaningful. Furthermore tools may offer different features and constrains, thus interoperability is desired. This thesis explores the possibility of translating simulation code between two such tools, Ansys Lumerical and Meep.

[[Add motivation for thesis, talk about importance of optics simulations, talk about uses, show examples. Talk about the need to compare simulation software.]]

Chapter 3

Review of Relevant Literature

This chapter aims to give the reader a broad understanding of the subjects of matter. It provides an overview of the physics behind the aforementioned simulations and an introduction to the basic concepts of compiler design and the underlying formal language theory. This chapter will not delve into the specifics of each subject at hand but will provide the reader with the foundations necessary to comprehend the rest of this thesis.

3.1 Computational Electromagnetics

Curl theorem

Divergence theorem

Maxwell's equations

[[show all 4 equations]] [[derive partial form form integral form]]

Finite Difference Time Domain

3.2 Formal Language Theory

We use languages like English or Slovak every day to communicate information, these languages are referred to as *natural languages*. The Oxford English Dictionary defines language as a system of spoken or written communication used by a particular country, people, community. How ever a more rigorous definition of languages and the tools to operate on them is required. That is what the notion of formal language theory will introduce in the following sections.

3.2.1 Alphabets and Languages

[[add text about languages and alphabets, use English as example]]

Definition 3.2.1 (Alphabet)**.** Let Σ be an *alphabet*, a finite nonempty set of symbols, letters. Then Σ^* defines the set of all sequences w:

$$w = a_1 a_2 a_3 \dots a_{n-1} a_n, \in \Sigma \text{ for } n \in \mathbb{N}$$

The sequence of symbols w is called a *word*. The length of a word is given by the number of symbols a, symbolically $|w| = n$. The word with a length of 0 is called an *empty word* denoted as ϵ.

Definition 3.2.2 (Language)**.** The set L where $L \subseteq \Sigma^*$ is know as the *formal language* over the alphabet Σ.

The words $L = \{\epsilon, a, b, aa, ab, bb\}$ are some words of a language L over the alphabet $\Sigma = \{a, b\}$. Other examples for languages over the alphabet $\Sigma = \{a, b\}$ might include:

- $L_1 = \{\epsilon\}$
- $L_2 = \{a\}$
- $L_3 = \{aaa\}$
- $L_4 = \left\{a^i, b^i; i \in \mathbb{N}\right\}$

Notation conventions

- Iteration
  Let $a^i; a \in \Sigma$ and $i \in \mathbb{Z}$ be the *iteration* of a character a, where $|a^i| = i$.
  Example bellow:
  - $a^0 = \epsilon$
  - $a^1 = a$
  - $a^2 = aa$
  - $a^i = a_0 a_1 a_2 \ldots a_i; i \in \mathbb{N}$
- Concatenation
  Let $w \cdot w^{'}; w, w^{'} \in \Sigma^*$ be the *concatenation* of the words w and $w^{'}$.
  $w = a_1 a_2 a_3 \ldots a_n; w^{'} = a^{'}_1 a^{'}_2 a^{'}_3 \ldots a^{'}_m; n, m \in \mathbb{Z}$ then
  $w \cdot w^{'} = ww^{'} = a_1 a_2 a_3 \ldots a_n a^{'}_1 a^{'}_2 a^{'}_3 \ldots a^{'}_m$
- **[[Kleene star]]**
- Symbol count
  The number of occurrences of a in w, where $a \in \Sigma, w \in \Sigma^*$, noted as $|w|_a$.

3.2.2 Grammars

Linguists refer to grammar as a set of rules that specify how a natural language is formed. Due to the polysemous and vague nature of natural languages, it is not suited for the description of unambiguous systems. The inherit need to strictly describe languages introduced various language defining mechanisms. Many of these mechanisms are interchangeable, and may describe the same languages. How ever, not all mechanisms are able to describe languages formed by other mechanisms.

The concept of a grammar is a powerful tool for describing languages[**?**, p. 52]. A simple sentence in English consists of a single independent clause. A clause is typically formed by a subject and a predicate. This can be written as follows.

$$\langle \mathit{clause} \rangle \rightarrow \langle \mathit{subject} \rangle \, \langle \mathit{predicate} \rangle$$

We can further define the $\langle subject \rangle$ and $\langle predicate \rangle$. One of the possible subjects is a noun phrase, and the predicate may be a simple verb.

$$\langle subject \rangle \rightarrow \langle determiner \rangle \, \langle premodifier \rangle \, \langle noun \rangle \, \langle postmodifier \rangle$$

$$\langle predicate \rangle \rightarrow \langle verb \rangle$$

Associating the $\langle determiner \rangle$ with the article „the“, $\langle premodifier \rangle$ with „fast“ or „slow“, $\langle noun \rangle$ with „athlete“, $\langle postmodifier \rangle$ to „from England“ or to an empty string and finally associating the $\langle verb \rangle$ to „won“ or „lost“ allows us to write multiple sentences like „The fast athlete from England won“ or „The slow athlete lost“. These sentences are considered to be *well formed*, as they resulted from following the grammatical rules.

The premise is consecutively replacing $\langle clause \rangle$ until only irreducible blocks of the language remain. Generalizing this idea brings about the concept formal grammars.

Definition 3.2.3 (Grammar)**.** [**?**] Let an ordered quadruple G define a grammar such that: $G = (N, \Sigma, P, S)$, where:

1. N is a finite set of *nonternminal* symbols
2. Σ is an alphabet finite set of *terminal* symbols, such that $N \cap \Sigma = \varnothing$
3. P is a finite set of rewriting rules known as *productions*, ordered pairs (α, β). P is a subset of the cartesian product of $\alpha = (N \cup \Sigma)^* \, N \, (N \cup \Sigma)^*$ and $\beta = (N \cup \Sigma)^*$

   Productions are denoted as $\alpha \rightarrow \beta$. If there are multiple productions with the same left hand side (α), we can group their right hand sides (β).

   $\alpha \rightarrow \beta_1, \alpha \rightarrow \beta_2$ may be written as $\alpha \rightarrow \beta_1 | \beta_2$
4. S is the starting symbol of the grammar G, where $S \in N$

Productions $\alpha \rightarrow \beta$ symbolize, that given the words $V, W, x, y \in (N \cup \Sigma)^*$, the word V can be rewritten as follows: $V \Rightarrow W$ iff there are words, which satisfy the following condition, $V = x\alpha y \wedge W = x\beta y$ and $\alpha \rightarrow \beta \in P$.

Definition 3.2.4 (Derivation)**.** $V \overset{*}{\Rightarrow} W$ iff there is a finite set of words

$$v_0, v_1, v_2, \ldots, v_z; z \in \mathbb{Z}$$

such that $v_0 = V$ and $v_z = W$ where each is rewritten from the previous word. Such a sequence of applications of productions is called a derivation. The length of a derivation is given by z.

Grammars are often represented using formalisms, these formalisms often set restrictions on the left and right hand sides of productions. These restrictions further impose limits on the set of languages a grammar may produce, this is known as the *expressive power* of a grammar.

3.2.3 Chomsky hierarchy

When working with formal grammars, the need to compare their expressive power arose. Linguist Noam Chomsky introduced the now so called *Chomsky hierarchy*[**?**]. A set of four classes, each more expressive than the previous, see **??**.

The intricacies of each class are not necessary in the context of this thesis, how ever relugar laguages will be used heavily throughout the rest of this thesis, so they will be explained more in depth.

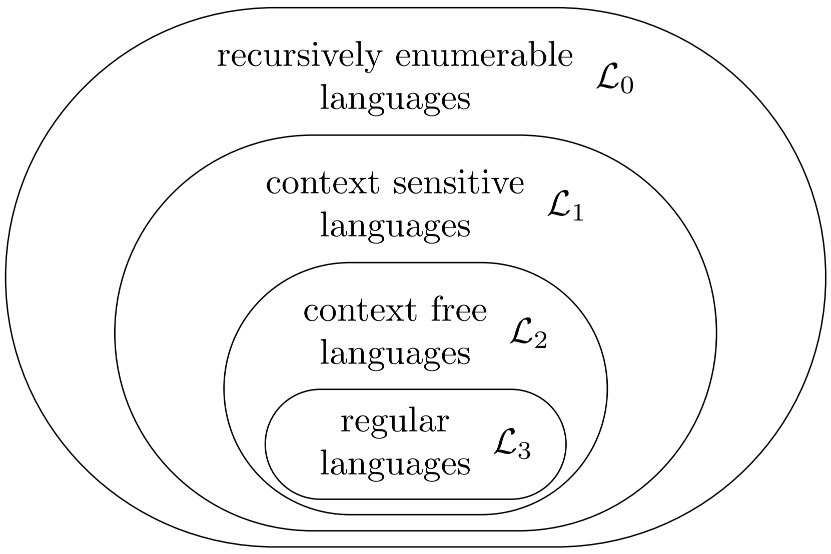


Figure 3.1: Description of Chomsky hierarchy.

Grammar	Language	Restrictions
$\mathcal{L}_3$	Regular	[[**todo**]]
$\mathcal{L}_2$	Context Free	
$\mathcal{L}_1$	Context Sensitive	
$\mathcal{L}_0$	Recursively Enumerable	

Table 3.1: An overview of the Chomsky hierarchy classes.

3.2.4 Regular Languages

As shown above, regular languages are the inner most part of the Chomsky hierarchy, thus they are the most constricted. How ever, regular languages play a crutial role in lexical analysis, more precisely pattern matching. These constrictions lead to many useful closure propetries and decisability properties, mainly membership, which will be introduced shortly.

Regular languges may be defined in multiple equivalent manners such as through finite automata or regular expressions.

Finite Automata

Definition 3.2.5 (Deterministic Finite Automata)**.** DFAs are formally defined as 5-tuples

$$M = (Q, \Sigma, \delta, q_0, F),$$

where Q is a finite set of states. Σ is a finite set of symbols, also know as a input alphabet. δ is a transition function between states defined as $\delta : Q \times \Sigma \to Q$. q_0 is a initial state, $q_0 \in Q$. And F is a set of final states, $F \subseteq Q$.

If starting in the state q_0 and moving left to right over the input string such that during each move a single symbol is consumed from the input string and a transition function for the corresponding state and symbol exists. If after all symbols from the input string have been read and the dfa stopped in a final state, the string is accepted. This is called a deterministic finite acceptor. **[[rephrase nicer]]**

Regular Expressions

Definition 3.2.6 (Regular Expressions)**.**

[[show equivalence DFA = NFA = $\mathcal{L}_3$]]

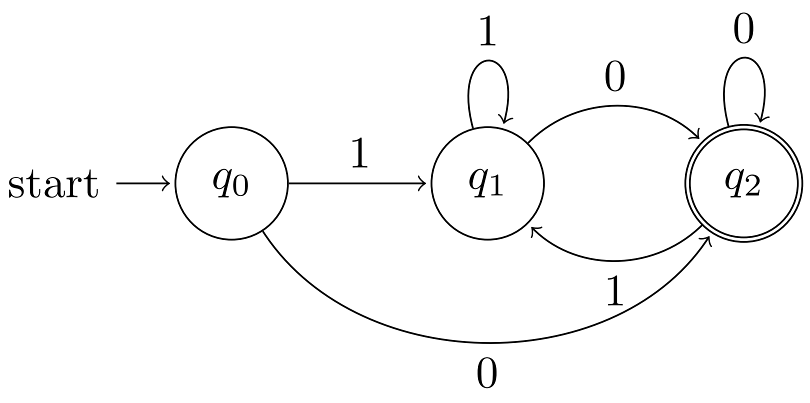


Figure 3.2: Description of a compiler as a single unit.

3.3 Compilers

Today's world undeniably relies on software more than ever before. It is imperative for developers to be able to efficiently write software. Software today is written in programming languages, high-level human-readable notations for defining how a program should run. However, before a program can be run on a system, it has to be translated (or compiled) into low-level machine code, which computers can run. The computer program that facilitates this translation is called a *compiler*. See **??**.

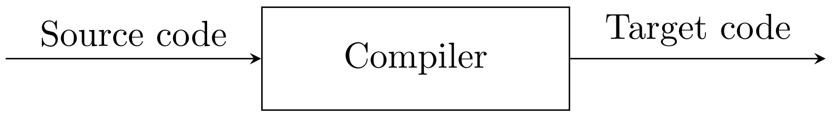


Figure 3.3: Description of a compiler as a single unit.

Compilers are complex programs. It is helpful to break them down into parts, each handling different tasks, which are chained together to form a compiler. Modern compilers are composed of many phases, such as the Lexical Analyzer, Syntax Analyzer, Semantic Analyzer, Intermediate Code Generator, Machine-Independent Code Optimizer, Code Generator, Machine-Dependent Code Optimizer [**?**, p. 5], however this chapter covers the components relevant to this thesis. See relevant stages in **??**.

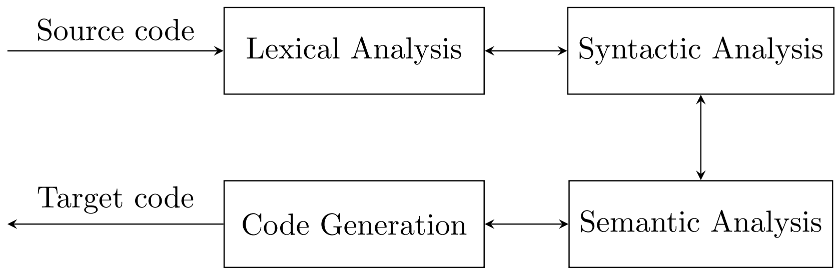


Figure 3.4: Overview of relevant compiler stages.

Lexical Analysis

The first stage of a compiler is lexical analysis, also known as a *lexer* or *tokenizer*. For the remainder of this thesis, *lexer* will refer to the lexical analysis stage of a compiler. The lexer consumes a stream of characters, the *source code*, and returns a stream of *tokens*. A *token* is a lexically indivisible unit, for example, the Python keyword `return`, you cannot divide it further, for example, into `re turn`. Each token is comprised of characters. The rule that defines which combination of characters constitutes a given token is called a *pattern*. The sequence of characters matching a pattern is called a *lexeme*, the *lexeme* is stored along with the token as a value.

Regular Expressions

A compact way to represent the patterns accepting tokens are *regular expressions*, which were introduced in **??**. Regular expressions are an algebraic definition of patterns, they specify *regular languages*, $\mathcal{L}_3$

Syntactic Analysis

LL Parser

Semantic Analysis

Code Generation

Chapter 4

Transpiler Implementation

This chapter introduces the implementation of the source to source compiler designed to convert Lumerical's scripting language to Python combined with the Meep library.

4.1 Choice of Tools [[rename]]

When writing any program, it is important to choose the right tools for the job. That comes down not only to choosing a programming language but also what libraries and packages to utilize and which components to implement to better suite the project's constraints. The source language, Lumerical's scripting language allows you to automate tasks and analysis such as manipulating simulation objects, launching simulations, and analyzing results [**?**]. This how ever is not suitable for implementing a transpiler. Looking at the target language, Python is a general-purpose high-level interpreted language. Writing the transpiler in Python will allow easier editing and debugging due to it's high-level nature. Following that, utilizing the fact that the implementation language is the same as the target language allows me to use built in modules for generating the target code. Python's ecosystem allow allows the use of tools such as Sphinx and Pytest for writing technical documentation and tests respectively.

4.1.1 Sphinx

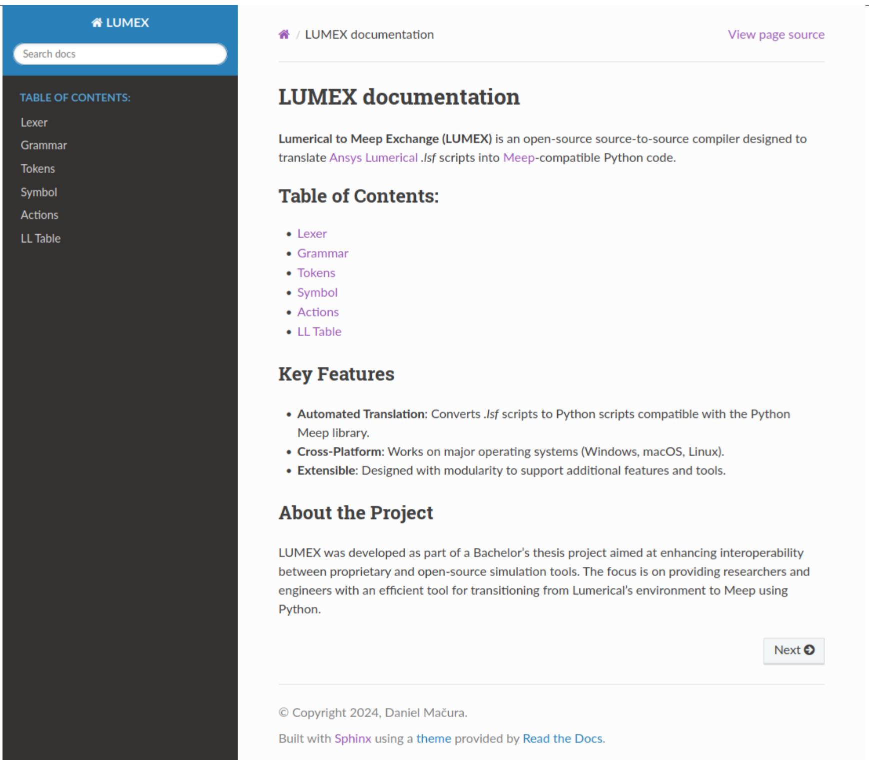
Sphinx is a tool that automatically generates documentation by converting plain text source files into multiple output formats[**?**]. Sphinx was chosen because it facilitates the extraction of docstring style comments from Python code with may be enriched by the addition of ReStructured Text [**?**]. Multiple output formats such as Portable Document Format (PDF), LaTeXsource code and Hypertext Markup Language (HTML) are supported. Sphinx also includes multiple extensions which allow the parsing of LaTeXinto Scalable Vector Graphics (SVG) which are easily rendered in the web.

This helps provide the user insight into the technical operation of the transpiler and allows him to search and view information in a concise manor.

4.1.2 Pytest

Pytest is a widely used testing framework for Python.

[[mention python, sphinx, pytest, explain motivation for implementation of own transpiler]]



LUMEX documentation

Lumerical to Meep Exchange (LUMEX) is an open-source source-to-source compiler designed to translate Ansys Lumerical *.lsf* scripts into Meep-compatible Python code.

Table of Contents:

- Lexer
- Grammar
- Tokens
- Symbol
- Actions
- LL Table

Key Features

- **Automated Translation**: Converts *.lsf* scripts to Python scripts compatible with the Python Meep library.
- **Cross-Platform**: Works on major operating systems (Windows, macOS, Linux).
- **Extensible**: Designed with modularity to support additional features and tools.

About the Project

LUMEX was developed as part of a Bachelor's thesis project aimed at enhancing interoperability between proprietary and open-source simulation tools. The focus is on providing researchers and engineers with an efficient tool for transitioning from Lumerical's environment to Meep using Python.

Next

© Copyright 2024, Daniel Mačura.

Built with Sphinx using a theme provided by Read the Docs.

Figure 4.1: Screen-capture of the resulting documentation.

4.2 Grammar

Before being able to translate between two languages, it is paramount to understand their grammar. Many languages provide definitions of grammars in a metasyntax format such as Backus-Naur form(BNF), extended Backus-Naus form(EBNF), Wirth syntax notation(WSN), augmented Backus–Naur form (ABNF), parsing expression grammar (PEG) or Zephyr Abstract Syntax Definition Language (ASDL) [?]. Python grammar is defined by a mixture of EBNF and PEG as used in the CPython parser [?]. Additionally Python abstract symbol tree (AST) module also defines an abstract grammar in ASDL[?].

Lumerical, as of writing this thesis, does not publicly provide any such specification for Lumerical's scripting language. Thus it is necessary to analyze the language and derive such a grammar at least for the scope of this thesis.

[[provide steps to reverse-engineer grammar]] [[add grammar listing here or in appendix?]]

4.3 Lexical Analysis

After defining a grammar, the first stage in creating a transpiler is the lexer. There are multiple methods in which a lexer may be modeled, the most straight forward is with the use of a DFA formalism and a loop over all characters. Always checking if there is a valid production from the current character to the look-ahead character and greedily continuing and backtracking to the last successful match.

Algorithm 1: State Machine Based Lexer

```
current_state ← None
last_accepting_state ← None
while lookahead_character != EOF do
    if has_transition(current_state, lookahead) then
        current_state ← next_state(current_state, lookahead)
        get_next_character()
        if is_accepting_state(current_state) then
            last_accepting_state ← current_state
    else
        if no_accepting_state_visited then
            return None ;                                          // Failure
        else
            revert_to_previous_accepting_state_and_revert_input_characters
            return current_state ;                                 // Success
```

This implementation is fine, how ever with an ever growing number of states, which have to be hard-coded, the solution may become rather complex. How ever there is a cleaner method. Since DFA may be transformed into regular expressions, it is sufficient to try to match all regular expressions against the input and return the longest match while removing it from the input.

Algorithm 2: Regular Expression Based Lexer

```
longest_match ← None
for all_regular_expressions do
    if regular_expression_matches_input then
        if regular_expression_is_longer_than_longest_match then
            longest_match ← current_regular_expression
if longest_match != None then
    return matching remove_matched_lexeme_from_input
else
    return None ;                                                  // Failure
```

[[mention lexer regex implementation]]

4.4 LL Parser

[[mention LL table calculation according to theory]]

4.5 Code Generation

[[]] **[[handling context specific actions such as the selection -> introduction of runtime class]] [[write about transformation of ast]]**

Chapter 5

Evaluation of Results

[[talk about methodology]] [[show comparison of leumerical code, generated python and handwritten -> conclusion probably not best method]] [[test examples analytic vs FDTD]]

Chapter 6

Conclusion

[[compare lumerical and meep]] [[test result TBD]]