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 TRANSPILATION 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,

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

Reference
MAČURA, Daniel. *Automatic transpilation 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 transpilation 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
November 18, 2024

Acknowledgements

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

Contents

1 Note to the Reader **3**

2 Introduction **4**

3 Review of Computational Electromagnetics and Compilers **5**

3.1 Computational Electromagnetics 5

3.2 Formal Languages 5

3.2.1 Alphabets and Languages 5

3.2.2 Regular Languages 7

3.3 Compilers 7

4 Transpiler Implementation **9**

5 Evaluation of Results **10**

6 Conclusion **11**

Bibliography **12**

List of Figures

3.1 Figure describing the compiler as a single unit. 7
3.2 Figure describing the compiler stages. 8

Chapter 1

Note to the Reader

Through the thesis, the expression „iff“ refers to „if and only if“.

Chapter 2

Introduction

[[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 Computational Electromagnetics and Compilers

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 languages. 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

[[assuming magnetic monopoles dont exist]] [[show all 4 equations]] [[derive partial form form integral form]]

Finite Difference Time Domain

Mie Scattering

3.2 Formal Languages

[[add text explaining need of formal languages]]

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 λ.

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

The words $L = \{\lambda, a, b, aa, ab, bb\}$ 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:

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

Notation conventions

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

Languages are often defined by grammars.

Definition 3.2.3 (Grammar)**.** [1] 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 $\alpha \to \beta$. If there are multiple productions with the same left hand side (α), we can group their right hand sides (β).

   $\alpha \to \beta_1, \alpha \to \beta_2$ may be written as $\alpha \to \beta_1 | \beta_2$
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.

Chomsky Hierarchy **[[introduce chomsky hierarchy so you can later show DFA = NFA = $\mathcal{L}_3$]]**

3.2.2 Regular Languages

Automata

Regular Expressions

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

Definition 3.2.5 (Finite State Machine).

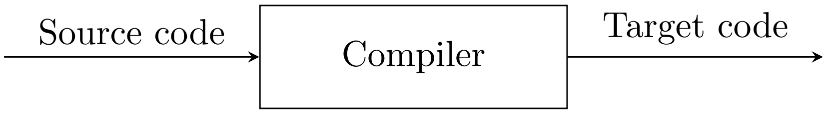
Definition 3.2.6 (Regular Expressions).

3.3 Compilers

Compilers are complex algorithms, *programs*, that translate a human readable input known as *source code* into a representation more suitable for computers to run. The introduction of compilers

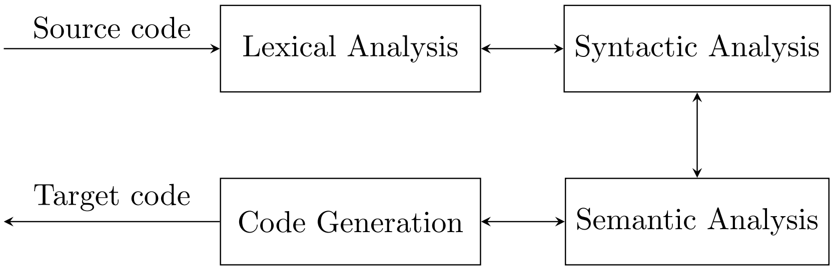
Today's world undeniably relies on software more than ever before. It is imperative for developers to be able to efficiently write software. Software today is written in programming languages, high-level human-readable notations for defining how a program should run. However, before a program can be 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 fig. 3.1.

Figure 3.1: Figure describing the 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* [2], however this chapter covers the components relevant to this thesis. See relevant stages in fig. 3.2.

Figure 3.2: Figure describing the 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 definition 3.2.6. 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

[[mention lexer regex implementation]] [[mention LL table calculation according to theory]] [[write about transformation of ast]]

Chapter 5

Evaluation of Results

[[talk about methodology]] [[test examples analyticall vs FDTD]]

Chapter 6

Conclusion

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

Bibliography

[1] Salomaa, A. *Formal languages*. USA: Academic Press Professional, Inc., 1987. ISBN 0126157502.

[2] Aho, A. V.; Lam, M. S.; Sethi, R. and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN 0321486811.