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

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

165352

Institut: Department of Computer Graphics and Multimedia (DCGM)
Student: **Mačura Daniel**
Programme: Information Technology
Title: **-**
Category: Compiler Construction
Academic year: 2024/25

Assignment:

1. Study and familiarize yourself with FDTD simulations. Review the tools Ansys Lumerical and Meep. Understand the differences between the scripting languages LSF and Python with the Meep library. Investigate the algorithms used in code translation by the transpiler.
2. Propose an algorithm and implement a one-way source-to-source translation of equivalent commands within a single simulation.
3. Evaluate the extent to which automatic translation is suitable, justifying your reasoning for each example.
4. Create test examples and translate them into equivalent examples in the second tool. Simulate the examples and measure the accuracy in comparison to examples that can be easily solved analytically. Measure the speed in relation to accuracy and the number of cores used.
5. Evaluate the results obtained and suggest possible extensions.

Literature:

- Teixeira, F.L., Sarris, C., Zhang, Y. *et al.* Finite-difference time-domain methods. *Nat Rev Methods Primers* **3**, 75 (2023). https://doi.org/10.1038/s43586-023-00257-4
- Alfred Aho, Jeffrey Ullman, Ravi Sethi, Monica Lam, Compilers: Principles, Techniques, and Tools 2nd Edition, 2006, ISBN 032148681

Requirements for the semestral defence:
Points 1 and 2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Milet Tomáš, Ing., Ph.D.**
Head of Department: Černocký Jan, prof. Dr. Ing.
Beginning of work: 1.11.2024
Submission deadline: 14.5.2025
Approval date: 12.11.2024

Abstract

This thesis aims to develop a source-to-source compiler that translates Ansys© Lumerical scripts into Meep scripts. To this end, a broad explanation of the physics behind the FDTD method, along with a background of formal languages, is provided. Key implementation details are discussed, including the design of the transpiler and relevant language features. After creating the transpiler itself, this thesis also sets out to compare the results of the two simulation tools. The comparison addresses the accuracy of the simulations, as well as the performance of the two tools coupled with their parallelization capabilities. The provided open source code may serve as a framework for future scientific endeavors in this area.

Abstrakt

Cieľom tejto práce je vyvinúť kompilátor, ktorý prekladá skripty Ansys© Lumerical do skriptov Meep. Na tento účel sa v práci podáva základné objasnenie formálnych jazykov, ako aj všeobecný opis fyzikálnych princípov, na ktorých je postavená metóda FDTD. Ďalej sa diskutuje o kľúčových implementačných detailoch vrátane návrhu transpilátora a príslušných funkcií jazyka. Po vytvorení samotného transpilátora sa táto práca zameriava aj na porovnanie výsledkov oboch simulačných nástrojov. Porovnanie sa týka presnosti simulácií, ako aj výkonnosti oboch nástrojov v spojení s ich paralelizačnými schopnosťami. Výsledný open source kód môže slúžiť ako rámec pre nadväzujúce výskumné činnosti v tejto oblasti.

Keywords

computational electromagnetics, finite difference time domain, Meep, Ansys© Lumerical, transpiler

Klíčová slova

výpočtová elektromagnetika, metóda konečných diferencií v časovej doméne, Meep, Ansys© Lumerical, transpilátor

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.

Rozšířený abstrakt

V modernej dobe je ľudská populácia závislá od veľkého množstva technológií, napr. internetu, mobilných telefónov, televízie, rádia, mikrovlnných rúr, senzorov kamier, laseru, svetelných diód, elektromotorov, lekárskych zobrazovacích systémov a mnohých ďalších technológií. Všetky uvedené technológie sú postavené na princípe elektromagnetizmu, ktorý nepochybne zohráva kľúčovú úlohu v každodennom živote.

Elektromagnetické simulácie majú kľúčový význam pri navrhovaní moderných technológií, počnúc mikrofotónovými čipmi a končiac komunikačnými anténami a senzormi. Umožňujú detailnú analýzu šírenia elektromagnetických vĺn v zložitých štruktúrach bez vynaloženia nákladov na fyzické prototypy. Na tieto simulácie sa používajú dve vývojové prostredia: Ansys Lumerical© a Meep. Lumerical je komerčný softvér s grafickým rozhraním a vstavanými optimalizačnými nástrojmi, pričom Meep je softvérový balík open source s možnosťou skriptovania v Pythone. Napriek spoločnému fyzikálnemu základu (riešenie Maxwellových rovníc) majú oba systémy rozdielne formáty a jazyky pre opis simulácií. Neexistencia priamej kompatibility skriptov medzi týmito platformami komplikuje prenášanie existujúcich modelov a znalostí. Hlavný cieľ tejto bakalárskej práce spočíva vo vytvorení nástroja LUMEX, ktorý automatizuje preklad simulačných skriptov zo softvéru Ansys Lumerical do skriptov pre Meep. Táto práca sa taktiež zaoberá porovnaním simulačných nástrojov so zameraním na presnosť a rýchlosť simulácií.

Pri návrhu riešenia boli použité poznatky z oblasti výpočtovej elektromagnetiky, formálnych jazykov a návrhu prekladačov. Z fyzikálneho hľadiska simulácie vychádzajú z Maxwellových rovníc, ktoré sú numericky riešené pomocou metódy konečných diferencií v časovej doméne (FDTD). Z pohľadu softvérovej techniky sa implementácia opierala o koncepty lexikálnej a syntaktickej analýzy skriptov. V tejto práci bola definovaná vlastná gramatika na opis syntaxe skriptov Lumerical. Lexikálny analyzátor rozkladá vstupný text na základné tokeny, zatiaľ čo syntaktický analyzátor overuje syntaktickú štruktúru a vytvára abstraktný syntaktický strom (AST), ktorý reprezentuje logiku programu.

Na základe získaného syntaktického stromu vykonáva LUMEX generovanie cieľového kódu pre Meep. Každá časť pôvodného skriptu je mapovaná na zodpovedajúce volania v Meep. Napríklad, definície geometrických objektov, zdrojov elektromagnetického poľa či okrajových podmienok sú transformované do syntaktických konštrukcií knižnice Meep. Pri generovaní kódu je potrebné zohľadniť rôzne konvencie: Lumerical napríklad používa vlastné spôsoby na definíciu mriežky a jednotiek, ktoré je potrebné adekvátne previesť do prostredia Meep. Výsledný Python skript zachováva logiku pôvodnej simulácie, pričom využíva možnosti a knižnice Meep. Keďže ide o jednosmerný preklad, výsledný skript je generovaný výlučne v podobe kompatibilnej s Meep.

Výsledky testov ukázali, že simulácie správne preložených príkladov v Meep dosahujú veľmi podobnú presnosť ako v pôvodnom prostredí Lumerical. Priestorové rozloženie elektrického poľa a jeho časový priebeh zodpovedali analytickým výsledkom. Pokiaľ ide o výpočtovú výkonnosť, zistilo sa, že rozdiely závisia od konkrétnych prípadov. Lumerical vykazuje vyššiu výkonnosť vo väčšine prípadov, avšak s nižšou presnosťou. V prípade simulácií, kde by bola dosiahnutá rovnaká presnosť, by uvedený rozdiel rýchlostí nemusel byť taký významný. Ďalej bola skúmaná réžia pri spúšťaní simulácií na viacerých jadrách. Meep vykazuje lepšiu škálovateľnosť.

Táto práca je dôkazom, že automatizovaný preklad simulácií medzi prostredím Lumerical a Meep je zmysluplný a pre používateľov má praktický prínos. Prekladač LUMEX dokázal zachovať podstatné časti simulácií zo vstupných skriptov a reprodukovať výsledky s vysokou vernosťou. Kvôli rozdielom v syntaxi a implementácii medzi uvedenými dvoma

platformami bol však preložený kód výrazne komplexnejší. Medzi hlavné limity tohto riešenia patrí fakt, že niektoré pokročilé príkazy a knižničné prvky Lumerical (napríklad špeciálne modely materiálov či optimalizácie) nemajú priame ekvivalenty v prostredí Meep. Túto výskumnú tematiku je v budúcnosti možné rozšíriť o širšiu škálu príkazov Lumerical, optimalizáciu generovaného kódu a využitie vyššej úrovne abstrakcií pre sprehľadnenie výsledného kódu. Výsledky tejto práce zdôrazňujú nielen dôležitosť interoperability medzi simulačnými prostrediami, ale zároveň slúžia ako pevný základ pre ďalší rozvoj v tejto oblasti.

Automatic translation and comparison of Lumerical and Meep simulations

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Tomáš Milet, Ph.D.
Supplementary information was provided by Mgr. Sivie Luisa Brázdilová, Ph.D. and Mgr. Martin Čermák, Ph.D.
I have provided an exhaustive list of all literary sources, publications and other sources used during the preparation of this thesis.

.....
Daniel Mačura
May 13, 2025

Acknowledgements

I want to express my deepest gratitude to my supervisor, Ing. Tomáš Milet, Ph.D., for his guidance and support throughout my journey. I also want to extend my most sincere thanks to Mgr. Sivie Luisa Brázdilová, Ph.D., who recommended this topic to me and provided me with invaluable insights. I would also like to thank Mgr. Martin Čermák, Ph.D., for his expert guidance and patient explanation of physics principles. I would like to extend my gratitude to my colleagues at onsemi—and the company as a whole—for providing access to Lumerical and the necessary servers to run the simulations. I am also profoundly grateful to my whole family, who have always greatly supported me in my studies. And last but not least, I would like to thank my friends, particularly my roommate, for his support and encouragement.

Contents

List of Figures

Acronyms

1D One Dimension. 16

2D Two Dimensions. 16

3D Three Dimensions. 6, 10, 16, 29

ABNF Augmented Backus-Naur form. 28

ASDL Zephyr Abstract Syntax Definition Language. 28, 29

AST Abstract Syntax Tree. 25, 29, 41

BC Boundary Condition. 11, 54

BNF Backus-Naur form. 28, 37

CEM Computational Electromagnetics. 4–7, 10

CFG Context Free Grammar. 22–24, 29

CFL Courant-Friedrichs-Lewy. 10, 22, 23

DFA Deterministic Finite Automaton. 20–22, 35

EBNF Extended Backus-Naur form. 28, 29

EM Electro Magnetic. 5, 11, 12, 15, 16

FDTD Finite-Difference Time-Domain. 5, 9–11, 16, 29, 51, 52, 57

GLR Generalized Left-to-Right Rightmost Derivation. 29

LALR Look-Ahead, Left-to-Right, Rightmost Derivation. 29

LHS Left-Hand Side. 34

LL Left-to-Right, Leftmost Derivation. 23, 24, 27, 29, 30, 33, 36–40

LUMEX Lumerical Meep Exchange. 4, 26, 29, 33, 50, 52, 57

MPI Message Passing Interface. 16, 52, 55

NFA Nondeterministic Finite Automaton. 21, 22

PEC Perfect Electric Conductor. 11

PEG Parsing Expression Grammar. 28, 29

PMC Perfect Magnetic Conductor. 11

PML Perfectly Matched Layer. 11, 52, 54

RHS Right-Hand Side. 34

WSN Wirth Syntax Notation. 28

Chapter 1

Preface

The following notation conventions apply throughout the thesis unless explicitly stated otherwise. In the context pertaining to physics, symbols in **bold** represent vectors and symbols in *italics* represent scalars, unless stated differently. The del operator denoted by the nabla symbol $\boldsymbol{\nabla}$ is used with the dot product and cross product to denote divergence: $\boldsymbol{\nabla} \cdot \mathbf{v}$ and the curl: $\boldsymbol{\nabla} \times \mathbf{v}$; this is explaind in greater detail in section 3.1.1.

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. The expression „iff“ refers to „if and only if“.

Otherwise, fairly standard notation in the applicable fields is used throughout the rest of the thesis.

Chapter 2

Introduction

In modern society, humans are reliant on a multitude of technologies, such as the Internet, mobile phones, television, radio, microwave ovens, camera sensors, lasers, light-emitting diodes, electrical motors, medical imaging systems, and many others. All of these rely on electromagnetic devices; electromagnetism undoubtedly plays a key role in day-to-day life as we know it.

The electromagnetic field theory is the underpinning framework for studying 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 are a series of fundamental coupled partial differential equations that form the cornerstone of classical electromagnetism. However, the closed-form analytical solutions are highly complex and available only for simple cases, making them impractical for most real-world applications.

However, with the rapid advent of available computational power, the use of analytically simple but computationally taxing methods was given an extra boost as they became more available. Compared to their counterparts, these methods provide solutions to more general problems. The branch of electromagnetics that focuses on such methods is termed Computational Electromagnetics (CEM). CEM allows us to simulate complex problems and verify designs before the production of prototypes. They also provide key insight into the operation of electromagnetic devices and even reveal certain information that may be unattainable by classical analytical methods. Moreover, with the ability to tweak the parameters and re-simulate, CEM has caused the advent of design optimization in electromagnetic devices.

Thus, CEM tools have become an indispensable part of the design process in many fields. Today, many such tools and software packages exist, some commercial and others open-source. However, when workflows are built around a specific tool, it may be difficult to switch to another one. This is especially holds true when the tool is proprietary and the source code is not available. It is important to be able to verify the results against other implementations and allow the researchers to check whether the results are mere artifacts of a particular solver or physically meaningful. Furthermore, the tools may offer different features and constraints, making their interoperability desired.

This thesis explores the possibility of translating simulation code between two such tools: Ansys Lumerical© and Meep. To address these challenges, this thesis first introduces relevant concepts in chapter 3, followed by the development of Lumerical Meep Exchange (LUMEX) in chapter 4, a tool that provides a framework for the translation of Lumerical Scripting Language to Meep. Examples of the translation process are provided in chapter 5.

Chapter 3

Review of Relevant Concepts

This chapter aims to give the reader a broader understanding of the relevant subject 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 rather provide the reader with the foundations necessary to comprehend the remaining parts of this thesis.

3.1 Computational Electromagnetics

As this thesis focuses on the implementation of a transpiler, an intricate grasp of CEM is not required; however, a surface-level explanation may benefit the reader, and thus it is provided below.

As explained earlier, CEM is a branch of electromagnetics that focuses on computational methods. This process involves modeling the interactions of Electro Magnetic (EM) fields with objects, typically with the computation of the **E** (electric) and **H** (magnetic) fields or the surface current, in the case of Method of Moments, across the simulation domain. These methods discretize the fields in a step called *meshing*, resulting in the subdivision of the problem domain into many smaller elements. Depending on the simulation method and number of dimensions, this may result in a three-dimensional grid, two-dimensional patches or one-dimensional segments. There are multiple different methods, each having specific use cases. Finite Element Method is useful when facing arbitrary geometry or objects that do not align properly in the Cartesian grid. Method of Moments is a surface-based method that converts the integral forms of Maxwell's equations into matrix equations. Finite-Difference Time-Domain (FDTD) solves Maxwell's equations in the time domain, it is one of the most commonly used methods and also a key part of this thesis. Therefore, it will be introduced in more detail in the following sections [1].

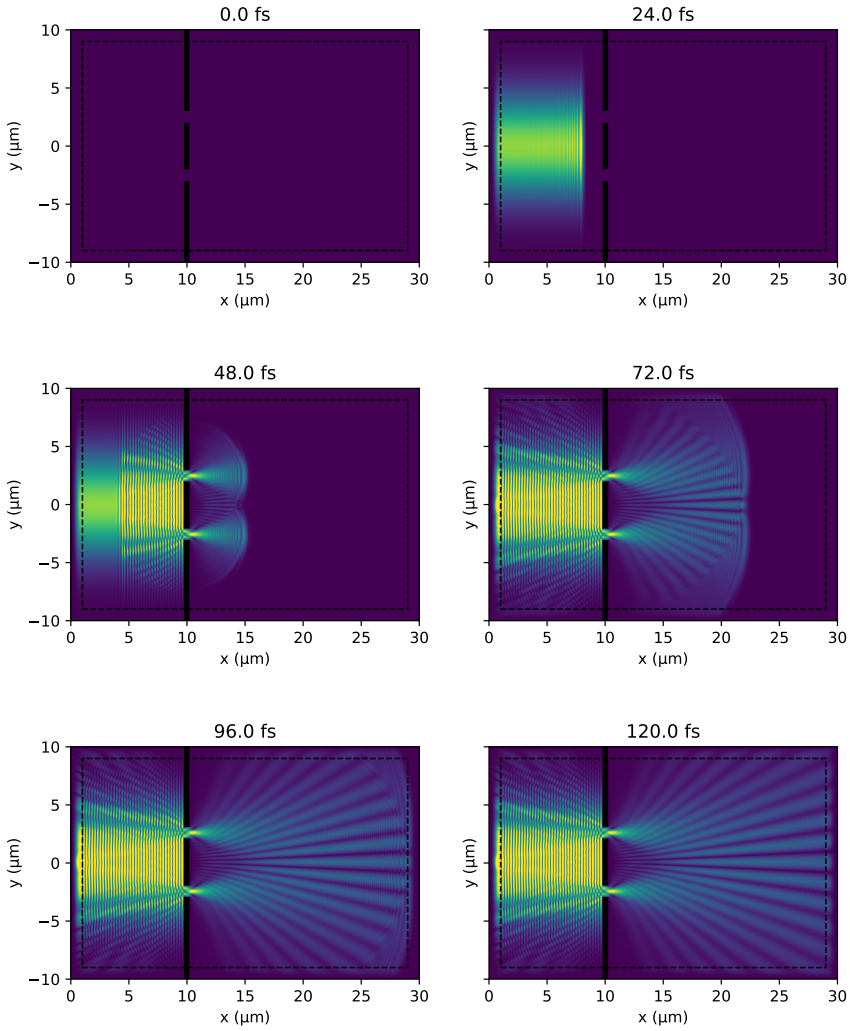


Figure 3.1: Visualization of the electrical field in the famous double slit experiment adapted from [2].

3.1.1 Vector Calculus

A basic understanding of vector calculus is required to understand the equations used in CEM. Vector calculus operates on *vector fields*, which can be imagined as a Three Dimensions (3D) space where every point has a *vector* (represented by three numbers) associated to it. In practice, these fields are discretized into a grid of points; however, we typically assume that the transition between the vectors at different points is continuous, smooth and differentiable.

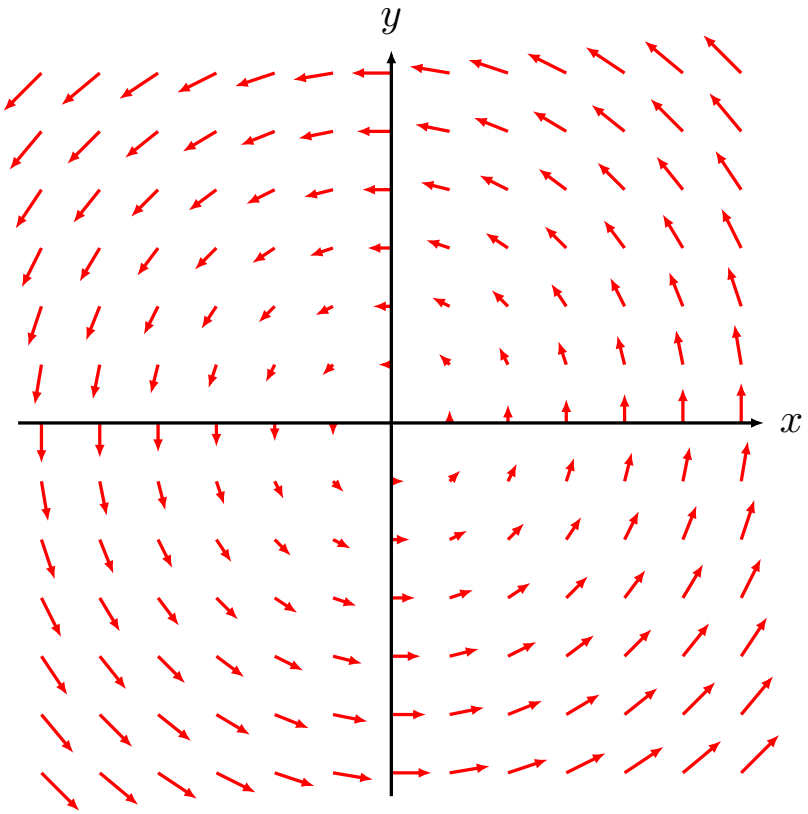


Figure 3.2: An example of a vector field. The arrows represent the direction and magnitude of the vector at each point in the field.

Operators

In general, operators are mappings. The *nabla operator*, denoted by the symbol $\boldsymbol{\nabla}$, is a common operator in CEM. Note that the nabla operator is represented in bold as a vector operator. The nabla operator is defined as:

$$\boldsymbol{\nabla} = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z} \right). \tag{3.1}$$

The nabla operator can be used to define several important operations in vector calculus, such as the *gradient*, *divergence* and *curl*.

Gradient The gradient of a scalar field $f(x, y, z)$ is a vector field that points in the direction of the steepest increase of the scalar field. The gradient is defined as:

$$\boldsymbol{\nabla} f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right). \tag{3.2}$$

Divergence The divergence of a vector field $\mathbf{F} = (F_x, F_y, F_z)$ is a scalar field that measures the rate at which the vector field spreads out from a point. The divergence is defined as:

$$\boldsymbol{\nabla} \cdot \mathbf{F} = \frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} + \frac{\partial F_z}{\partial z}. \tag{3.3}$$

Curl The curl of a vector field $\mathbf{F} = (F_x, F_y, F_z)$ is a vector field that measures the rotation of the vector field around a point. The curl is defined as:

$$\boldsymbol{\nabla} \times \mathbf{F} = \left(\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z}, \frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x}, \frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y} \right). \tag{3.4}$$

3.1.2 Divergence theorem

The divergence theorem states that the sum of the sources and sinks of a vector field within a volume V is equal to the flux of the vector field through the surface S that bounds the volume

$$\iiint_V (\boldsymbol{\nabla} \cdot \mathbf{F}) \, \mathrm{d}V = \oiint_S \mathbf{F} \cdot \mathrm{d}\mathbf{S}. \tag{3.5}$$

Let's unwrap this a bit. The left side of the equation is the volume integral of the divergence of a vector field $\mathbf{F}$ over a volume V. The divergence of a vector field measures the rate at which the field spreads out from a point. The right side of the equation is the surface integral of the vector field $\mathbf{F}$ over the surface S that bounds the volume V. The surface integral measures the total flux of the vector field through the surface. The divergence theorem states that these two quantities are equal, meaning that the total flux through the surface is equal to the sum of the sources and sinks within the volume.

3.1.3 Curl theorem

The curl theorem, also known as the Stokes' theorem, states that the line integral of a vector field around a closed loop L equals the surface integral of its curl over the surface S bounded by the loop.

$$\iint_S (\boldsymbol{\nabla} \times \mathbf{F}) \cdot \mathrm{d}\mathbf{S} = \oint_L \mathbf{F} \cdot \mathrm{d}\mathbf{L}. \tag{3.6}$$

On the right hand side, the vector field $\mathbf{F}$ is integrated over a closed loop L; this is the sum of all vectors along the closed loop L. On the left hand side, the curl of the vector field $\mathbf{F}$ is integrated over the surface S. The curl of a vector field measures the rotation of the field around a point. The curl theorem states that these two quantities are equal, meaning that the total rotation around the closed loop is equal to the sum of the rotations within the surface.

3.1.4 Maxwell's equations

In the 19th century, James Clerk Maxwell formulated a set of four equations that underpin classical electromagnetism and form a coherent theoretical structure. Maxwell's equations in the integral form tend to be used in the calculation of symmetric problems, such as finding the electric field of a charged plane, sphere, etc., whereas the differential form is more suitable for the calculation of numerical problems as it is simple to obtain the magnetic and electric fields at a single point. Ampère-Maxwell law states that magnetic fields are generated by electric currents and a change of the electric field over time. Faraday's law states that a changing magnetic field produces a rotating electric field, and vice versa. Gauss's law for electric flux states that electric charges generate an electric field. Gauss's law for magnetic flux postulates the nonexistence of magnetic charge, i.e., magnetic monopoles. In 1931, Dirac proposed that the discovery of magnetic charge (a magnetic monopole) would

require a modification of Gauss's law for magnetic flux as well as Faraday's law [3]. Despite extensive searches, magnetic monopoles have yet to be detected experimentally.

Ampère-Maxwell law

$$\boldsymbol{\nabla} \times \mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t} \tag{3.7}$$

Faraday's law

$$\boldsymbol{\nabla} \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \tag{3.8}$$

Gauss's law for electric flux

$$\boldsymbol{\nabla} \cdot \mathbf{D} = \rho \tag{3.9}$$

Gauss's law for magnetic flux

$$\boldsymbol{\nabla} \cdot \mathbf{B} = 0 \tag{3.10}$$

The above equations are in the differential form, and they are first-order linearly coupled. They also have equivalent integral forms, which may be produced by applying the divergence integral theorem and curl integral theorem, see sections 3.1.2 and 3.1.3. Refer to [4, Chapter 2.4] for more details about this process.

Constitutive Relations

Since Maxwell's equations contain more unknowns than equations, as shown above, they are undetermined. This is where constitutive relations between field intensities $\mathbf{E}, \mathbf{H}$ and the flux densities $\mathbf{D}, \mathbf{B}$ come into play. For simple mediums, which are *linear*, *isotropic*, and *non-dispersive*, the following equations hold and are a linear approximation.

$$\mathbf{D} = \varepsilon \mathbf{E} \tag{3.11}$$

$$\mathbf{B} = \mu \mathbf{H} \tag{3.12}$$

$$\mathbf{J} = \sigma \mathbf{E} \tag{3.13}$$

Where ε, the permittivity of a medium, is given by the permittivity of free space and the relative permittivity, respectively $\varepsilon = \varepsilon_0 \varepsilon_r$. Similarly, μ, the permeability of a medium, is given by the permeability of free space and the relative permeability $\mu = \mu_0 \mu_r$. A medium is often defined by its *constitutive parameters* ε, μ, σ. A good illustration of this is that a medium is *isotropic* if ε is scalar instead of a tensor, and *homogeneous* if ε is not a function of position.

3.1.5 Finite Difference Time Domain

The FDTD method is a staple among finite difference techniques. Finite difference methods are numerical techniques that approximate the derivatives directly using finite difference quotients. This class of methods is widespread due to its implied simplicity and is widely used for scientific computation.

It is suitable for problems where the electromagnetic wavelengths, along with the geometries, are comparable to the simulation domain. It is a time domain method, thus it may solve for a broad spectrum in a single simulation in a single pass, where it provides a direct numerical approximation of the differential operators in Maxwell's curl equations. The spatial domain is discretized into a staggered grid of *cells*; these will be explained in more depth below. FDTD also discretizes the continuous time into *time steps*, however, there is a

limit to the largest time step in which the method stays numerically stable, $\Delta t < h/\left(c\sqrt{3}\right)$ for 3D, where h and c denote the grid dimension and speed of light in the simulation respectively. This is often referred to as the *Courant limit* after the Courant-Friedrichs-Lewy (CFL) condition.

The user must keep in mind that such a method will never give a precise answer. The accuracy of the simulation is dictated by the resolution. A general rule of thumb in the CEM community is to use at least 10 samples (cells) per the shortest wavelength. Similarly, in his Electromagnetic Simulation Using the FDTD Method, Dennis M. Sullivan states the following: "A good rule of thumb is 10 points per wavelength. Experience has shown this to be adequate, with inaccuracies appearing as soon as the sampling drops below this rate." [5, Page 10].

FDTD provides second-order accuracy with the use of first-order numerical differentiation. Thus, the error may be defined as Error $\propto (\Delta h)^2$. For example, in a 3D simulation, if the spatial (and thus time) resolution is increased twofold, the accuracy increases 4 times. This is due to the fact that Maxwell's curl equations are discretized using central difference approximations, which are inherently second-order accurate.

Another vital part is the staggered grid itself. Kane S. Yee introduced a system of calculating the **E** and **H** fields using the aforementioned central differences in an offset pair of grids. The **E** fields are defined at the edges of a cell while the **H** fields are stored at the centers of the faces, and thus offset spatially by $\frac{h}{2}$. However, since the **E** and **H** fields are defined at different locations, the time step must be staggered as well. The **E** field is calculated at time t and the **H** field at $t + \frac{\Delta t}{2}$. This is known as the Yee cell, and it is a cornerstone of the FDTD methods. The Yee cell is shown in fig. 3.3.

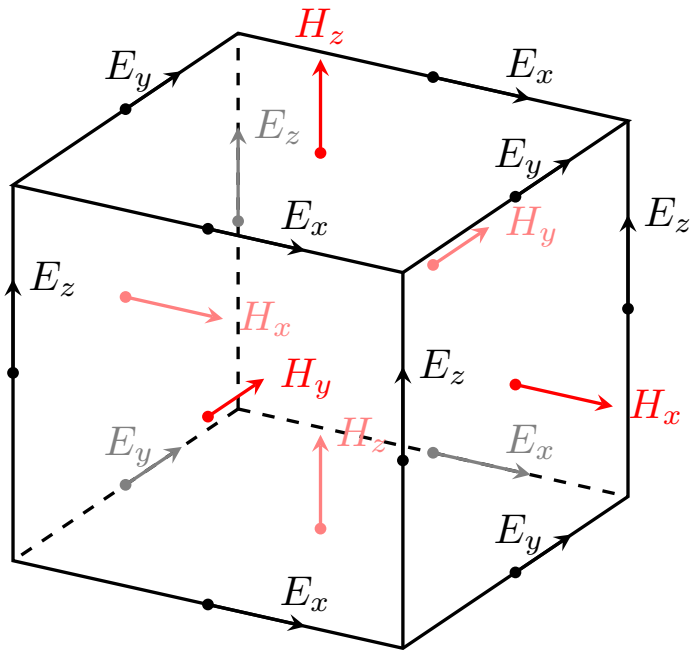


Figure 3.3: Description of the Yee cell.

The basic flow of the FDTD algorithm is as follows:

1. Divide the solution region into a grid of nodes (the Yee cell grid).
2. Approximate the derivatives in Maxwell's equations using finite differences at these nodes.
3. Set the initial conditions for the electric and magnetic fields.
4. Loop over the time steps using the „leapfrogging“ technique:
   (a) Update the electric field components using the values of the magnetic field from the previous half time step.
   (b) Update the magnetic field components using the newly computed values of the electric field.
5. Apply boundary conditions during each time step.
6. Store the field values at desired time points for subsequent analysis.
7. Continue time marching until the desired simulation time is reached.

Figure 3.4: Basic flow of the FDTD algorithm.

3.1.6 Boundary Conditions

As already hinted at in step 5 of the FDTD algorithm, the Boundary Conditions (BCs) are a crucial component of FDTD simulations. The BCs dictate the rules for terminating the simulation domain. There are multiple such types, e.g. Perfect Electric Conductor (PEC) and Perfect Magnetic Conductor (PMC), both acting as mirrors and reflecting their corresponding component of EM waves. *Periodic boundaries* set in place the assumption that one edge of the region matches the opposite edge, thus creating an infinite repeating simulation in that direction.

However, the Perfectly Matched Layer (PML), which was originally introduced in [6], is the most used BC. This is due to its efficiency, good performance in minimizing the reflections, and its overall flexibility [7]. The PML is, however, better thought of as a region rather than a BC, as it requires a certain thickness. According to the Meep documentation, a good rule of thumb is to use a thickness of around half the largest wavelength in the simulation [8].

3.2 Optics

A brief introduction to optics is required to understand the analytical methods used as the ground truth during the evaluation in chapter 5. The following section will introduce the basic concepts of optics, such as reflection, refraction, interference and diffraction.

A monochromatic plane wave in a non-absorbing, homogeneous, non-isotropic and non-magnetic medium is assumed throughout, unless stated otherwise.

When discussing optics, terms such as a medium, ray, and interface often come up. A *medium*, as mentioned above, is a material through which light travels. Mediums may simply be divided into *conductors*, *nonconductors*, which are *dielectrics*, the latter of which will be utilized further on. A *dielectric* medium does not conduct electricity; however, it

may be polarized by an electric field. This leads to further categorization of dielectrics into *polarized* and *nonpolarized* dielectrics. Polarized dielectrics, such as water, are materials with a permanent dipole moment, while nonpolarized dielectrics, such as glass, do not have a permanent dipole moment. The polarization of a dielectric is the deflection of positive and negative charges within the material, which occurs when an external electric field is applied. This polarization leads to the formation of electric dipoles within the material, which can affect the propagation of light through it. The degree of polarization depends on the strength of the electric field and the material's properties [9, Chapter 6].

Light itself may be polarized. Light is an EM wave, where polarization refers to the direction of oscillation of the electric field vector. Light may be polarized in a single direction, such as *linear* polarization, *eliptically* polarized or even *circularly* polarized, where the electric field vector rotation describes an ellipse or circle as the wave propagates. This thesis will utilize the former of the three. Linear polarization is assumed throughout the remainder of this thesis, unless stated otherwise.

To simplify the calculations of a wave's interaction with an interface, it is beneficial to choose a coordinate system which is aligned with the plane of incidence so that the polarization direction may be split into basis vectors.

- *p polarization* (from German *parallel*), where the electric field vector is *parallel* to the plane of incidence.
- *s polarization* (from German *senkrecht*), where the electric field vector is *perpendicular* to the plane of incidence.

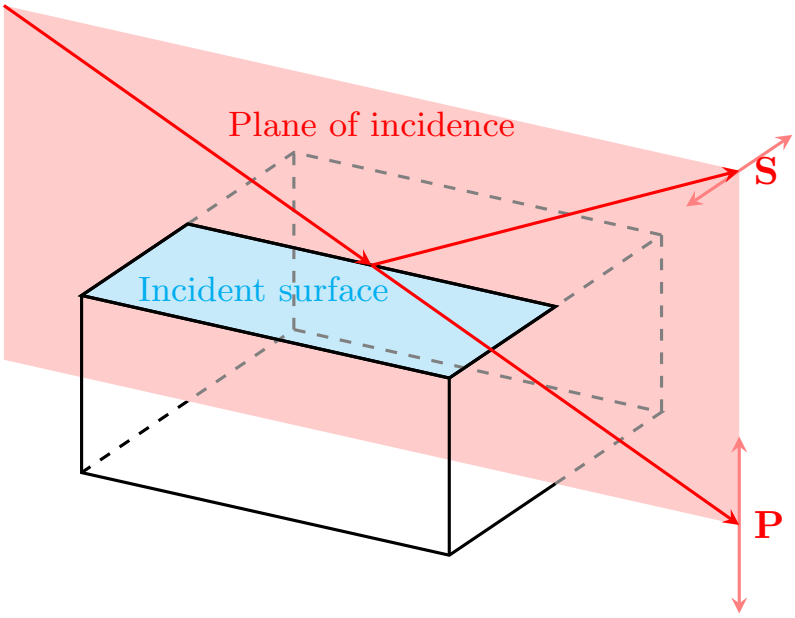


Figure 3.5: The two states of polarization. The plane of incidence is defined by the incident ray and the normal to the surface. The reflected and refracted rays are in the same plane as the incident ray. However, the former is normal to the surface, and the latter is parallel.

However, it is also beneficial to think of light as *rays*, which are an idealized model of light propagation, and represented as a straight line in the direction of its travel. When

a ray travels through an *interface* (the boundary between two different media, such as air and glass), it may be affected by multiple phenomena, which are described in the following sections.

3.2.1 Reflection

Reflection is the redirection of an incident light ray at a boundary, sending it back into its source medium. The most common example of reflection is at a planar interface, where rays are incident on the surface and reflected, allowing objects to be seen. The law of reflection states that the angle of incidence (θ_i) is equal to the angle of reflection (θ_r). This means that if a light ray is incident on a surface at a certain angle, it will be reflected at the same angle. This can be expressed mathematically as:

$$\theta_i = \theta_r \tag{3.14}$$

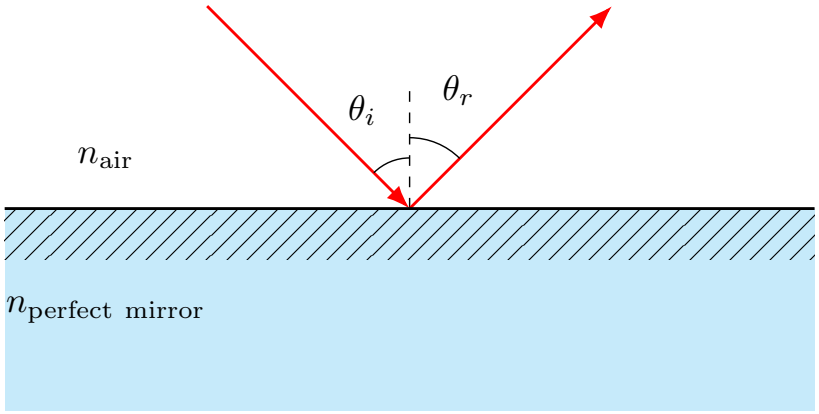


Figure 3.6: A single ray of light incident on a perfect mirror.

3.2.2 Refraction

Refraction is the act of light passing through a medium. The most common example of refraction is a window, where rays are incident on the surface and transmitted through it, allowing objects behind the window to be seen. Snell's law states that the angle of incidence (θ_i) is related to the angle of refraction (θ_t). This means that if a light ray is incident on a surface at a certain angle, it will be refracted as it passes into the second medium. This can be expressed mathematically as:

$$n_1 \sin(\theta_i) = n_2 \sin(\theta_t) \tag{3.15}$$

Where n_1 and n_2 are the refractive indices of the two media. The *refractive index* is a measure of how much light slows down as it passes through a medium. This is shown in fig. 3.7.

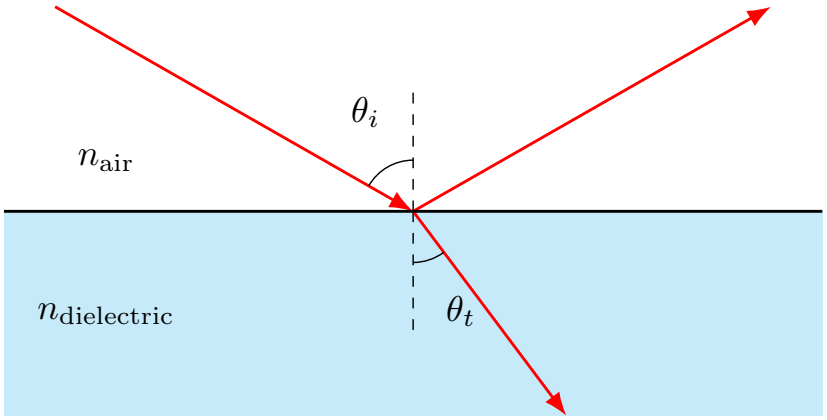


Figure 3.7: A single ray of light incident on a single layer of dielectric. The light ray reflects and refracts at the interface.

Using Snell's law, we can calculate the angle of refraction (θ_t) with both angles and apply Fresnel's equations to calculate the reflectance for p and s polarization:

$$R_p = \left| \frac{n_1 \cos(\theta_i) - n_2 \cos(\theta_t)}{n_1 \cos(\theta_i) + n_2 \cos(\theta_t)} \right|^2 \tag{3.16}$$

$$R_s = \left| \frac{n_1 \cos(\theta_t) - n_2 \cos(\theta_i)}{n_1 \cos(\theta_t) + n_2 \cos(\theta_i)} \right|^2 \tag{3.17}$$

Due to the conservation of energy, the sum of reflectance and transmittance must equal 1 for non-absorbing media. This means that transmittance can be calculated as:

$$T = 1 - R \tag{3.18}$$

A special case is at normal incidence where $\theta_i = 0$, $\theta_t = 0$, and there is no difference between the two polarizations. In this case, reflectance can be calculated as:

$$R = \left| \frac{n_1 - n_2}{n_1 + n_2} \right|^2 \tag{3.19}$$

3.2.3 Internal Reflection

Internal reflection occurs when light travels from a medium with a higher refractive index to a medium with a lower refractive index. The angle of incidence must be greater than the critical angle (θ_c) for total internal reflection to occur. The critical angle can be calculated using Snell's law:

$$\theta_c = \arcsin\left(\frac{n_2}{n_1}\right) \tag{3.20}$$

Where n_1 is the refractive index of the first medium and n_2 is the refractive index of the second medium. This is shown in fig. 3.8 below.

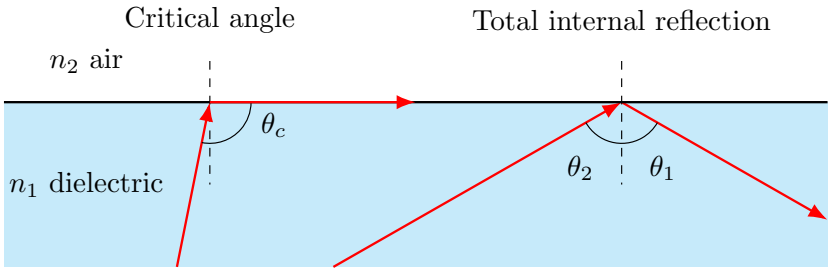


Figure 3.8: Two rays of light, demonstrating the critical angle and total internal reflection.

3.2.4 Thin film interference

The previous sections were building up to the calculation of thin film interference, which will be used as the ground truth in the precision tests in chapter 5. Thin film interference occurs when the light waves reflect off the top and bottom surfaces of a thin film, which causes the two reflected waves to interfere with each other, leading to constructive or destructive interference depending on the thickness of the film, incident angle and the wavelength of the light. For the following example and further tests, we will assume that the light is normally incident on the thin film, which will simplify the calculations.

We first begin by calculating the Fresnel coefficients to get the reflectance of the interfaces.

$$r_{12} = \frac{n_1 - n_2}{n_1 + n_2}, \quad r_{23} = \frac{n_2 - n_3}{n_2 + n_3} \tag{3.21}$$

Following that, and given that the EM wave changes phase when it passes from a medium with a lower refractive index to a medium with a higher refractive index or vice versa, we need to calculate the phase shift. The phase shift is given by:

$$\delta = \frac{2\pi n_2 d}{\lambda} \tag{3.22}$$

Where d is the thickness of the film, λ is the wavelength of light in vacuum and n_2 is the refractive index of the film.

The total complex amplitude of the reflected wave is given by:

$$r_{\text{total}} = \frac{r_{12} + r_{23}e^{2i\delta}}{1 + r_{12}r_{23}e^{2i\delta}} \tag{3.23}$$

Where r_{12} is the Fresnel coefficient for the first interface, r_{23} is the Fresnel coefficient for the second interface and δ is the phase shift.

The intensity of the reflected wave is given by:

$$I_{\text{total}} = |r_{\text{total}}|^2 \cdot I_0 \tag{3.24}$$

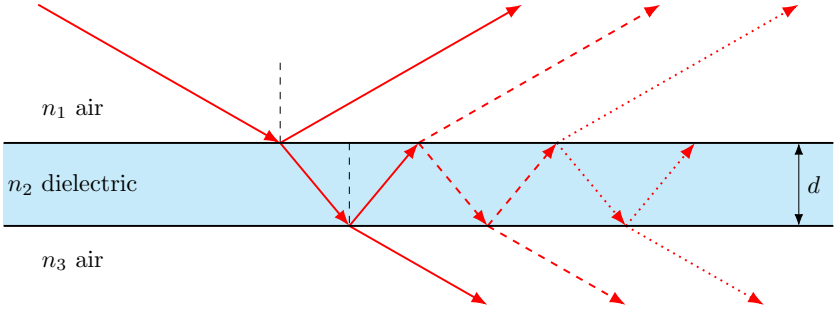


Figure 3.9: Thin film interference. The light wave reflects off the top and bottom surfaces of the thin film, causing the two reflected waves to interfere with each other.

3.3 Meep

Meep — MIT Electromagnetic Equation Propagation — is an open-source program for EM simulations distributed under the GNU General Public License v2.0. It is a scriptable software package using Python, Scheme or C++ APIs. However, it lacks a graphical user interface. Nonetheless, this allows for a combination of multiple computations and a multi-parameter optimization, and even in parallel. Meep handles parallelization across multiple cores and systems using the Message Passing Interface (MPI)[1].

Meep allows simulating in One Dimension (1D), Two Dimensions (2D), 3D and cylindrical coordinates. It can handle arbitrary geometries and anisotropic, dispersive, nonlinear and gyrotropic materials. It also supports a variety of boundary conditions, including Bloch-periodic, perfect-conductor boundary conditions and perfectly matched layers.

Meep offers the ability to exploit symmetries to reduce the computational time and memory requirements. The **E** and **H** fields may be exported and loaded using the HDF5 file format. These may be converted to other formats, such as VTK, and visualized using ParaView[2].

3.4 Ansys Lumerical

Ansys Lumerical FDTD© is a photonics simulation tool that combines FDTD, Optical Multilayer Solvers, Rigorously Coupled-Wave Analysis Solvers, Waveguide Solvers, Finite-difference Eigenmode Solvers and others within a unified design environment[3].

Lumerical provides a computer aided design-like graphical user interface, which is easy to comprehend for engineers using different tools compared to Meep's steeper learning curve. It also provides a Python API and Lumerical scripting language, both of which allow for the automation of simulations. However, due to a priori requirements, the Lumerical scripting language was chosen. Lumerical also provides a library of materials.

Lumerical is capable of attaining normalization of the fields in a single run, whereas Meep requires two runs to normalize the monitors for the calculation of reflectance and

[1] `https://www.mpi-forum.org/`

[2] `https://www.paraview.org/`

[3] Any and all ANSYS, Inc. brand, product, service and feature names, logos and slogans, such as Ansys Lumerical, are registered trademarks or trademarks of ANSYS, Inc. or its subsidiaries in the United States or other countries. All other brand, product, service and feature names or trademarks are the property of their respective owners.

transmittance. Lumerical also provides the possibility to exploit symmetries to reduce the computational time and memory requirements. Furthermore, Lumerical allows for setting a non-uniform mesh, which is useful in the simulations of structures with a large refractive index difference, such as the ones used in this thesis.

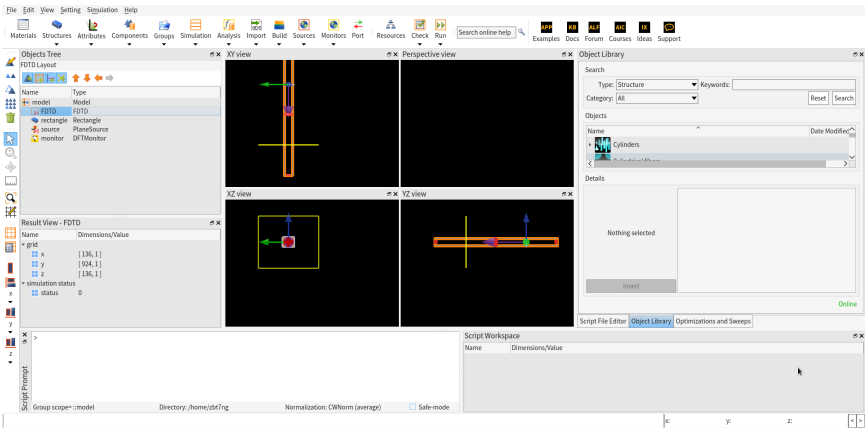


Figure 3.10: Screenshot of Lumerical graphical user interface.

3.5 Formal Language Theory

We use languages such as English, Czech, or Slovak in everyday communication exchanges, and these languages are commonly 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 or community. However, a more rigorous definition of languages and the tools to operate within them is required in this context. These aspects of language will be tackled in the notion of formal language, which is introduced in the sections below.

3.5.1 Alphabets and Languages

Languages, both natural and formal, are constructed from a basic set of orthographic elements. In the case of English, these elements are the letters A to Z, which form the English alphabet. From this alphabet, we can produce words, which in turn form sentences and larger linguistic structures. Similarly, in formal language theory, we start with an alphabet from which we build sequences known as words. These words serve as the foundational units of formal languages, just as words do in natural languages.

Definition 3.5.1 (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*. Word length is given by the number of symbols a, and symbolically annotated as $|w| = n$. The word with a length of 0 is called an *empty word* denoted as ε.

Definition 3.5.2 (Language)**.** The set L where $L \subseteq \Sigma^*$ is defined as a *formal language* over the alphabet Σ.

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

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

Notation conventions

- **Iteration**
  Let $a^i; a \in \Sigma$ and $i \in \mathbb{Z}$ be the *iteration* of a character a, where $|a^i| = i$.
  Examples bellow:
  - $a^0 = \varepsilon$
  - $a^1 = a$
  - $a^2 = aa$
  - $a^i = a_0 a_1 a_2 \dots a_i; i \in \mathbb{N}$
- **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
  $w \cdot w^{'} = w w^{'} = a_1 a_2 a_3 \dots a_n a_1^{'} a_2^{'} a_3^{'} \dots a_m^{'}$
- **Kleene star**
  Let $L \subseteq \Sigma^*$ be a language. The *Kleene star* of L, denoted L^*, is defined as

$$L^* = \bigcup_{i=0}^{\infty} L^i,$$

  where $L^0 = \{\varepsilon\}$ and $L^{i+1} = L \cdot L^i$. Equivalently, L^* is the set of all finite concatenations of zero or more words from L.

- **Symbol count**
  The number of occurrences of a in w, where $a \in \Sigma, w \in \Sigma^*$, denoted as $|w|_a$.

3.5.2 Grammars

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

The concept of a grammar is a powerful tool for describing languages [10, Page 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.

$$\langle clause \rangle \to \langle subject \rangle \langle predicate \rangle$$

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.

$$\langle subject \rangle \to \langle determiner \rangle \langle premodifier \rangle \langle noun \rangle \langle postmodifier \rangle$$

$$\langle predicate \rangle \to \langle verb \rangle$$

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 define a pattern capable of generating an infinite number of clauses/sentences such as „The fast athlete from England won“ or „The slow athlete lost“, which testifies to the principle of Chomskian generative grammar. These sentences are considered to be *well formed* as far as grammar is concerned, as they result from the implementation of grammatical rules.

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

Definition 3.5.3 (Grammar)**.** Let an ordered quadruple G define a grammar such that: $G = (N, \Sigma, P, S)$, where [11]:

1. N is a finite set of *nonternminal* symbols
2. Σ is an alphabet, i.e. a 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 $\beta = (N \cup \Sigma)^*$

   The 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 grammar G, where $S \in N$

Productions $\alpha \to \beta$ 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 $\alpha \to \beta \in P$.

Definition 3.5.4 (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 by formalisms that impose 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.

3.5.3 Chomsky hierarchy

When working with formal grammars, the need to compare their expressive power arose. Linguist Noam Chomsky introduced the so-called *Chomsky hierarchy* [12], i.e. a set of four classes, each more expressive than the previous, see fig. 3.11.

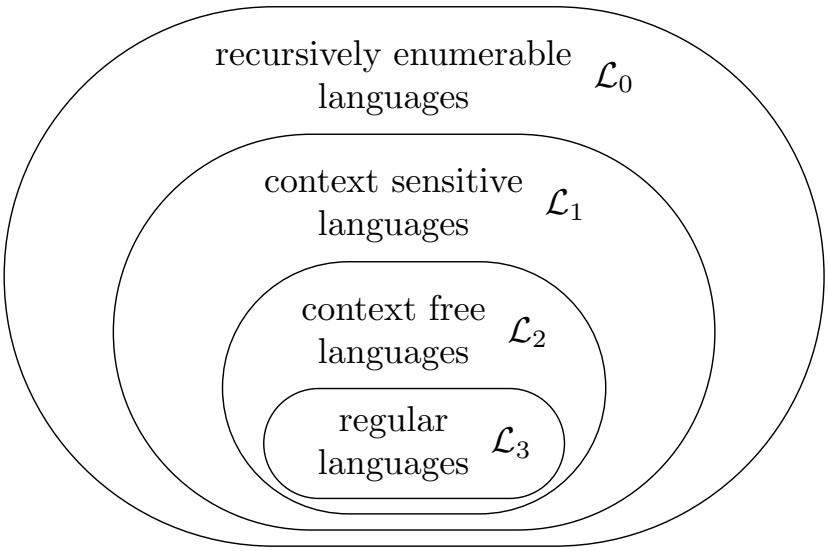


Figure 3.11: Description of the Chomsky hierarchy.

The intricacies of each of said classes are not necessary within the context of this thesis, however, regular and context free languages will be used heavily throughout the rest of this thesis, and explained more in depth.

3.5.4 Regular Languages

As shown above, regular languages are the innermost part of the Chomsky hierarchy, and they are the most restricted. However, regular languages play a crucial role in lexical analysis, more precisely, in pattern matching. These constrictions lead to many useful closure properties and decidability 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 3.5.5 (Deterministic Finite Automata)**.** Deterministic Finite Automatons (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 known as an input alphabet,

- δ is a transition function between the states defined as $\delta : Q \times \Sigma \to Q$,
- q_0 is an initial state for which $q_0 \in Q$ holds true,
- F is a set of final states, $F \subseteq Q$.

If, starting in the state q_0 and moving from 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, all symbols from the input string have been read and the DFA stopped in the final state, the string is deemed accepted. This is called a deterministic finite acceptor.

Similarly, Nondeterministic Finite Automatons (NFAs) are a 5-tuple, however they may have multiple initial and final states and there may be multiple transitions from the same state using the same symbol. For a rigorous formal definition the reader is directed to [11, Chapter 1.4].

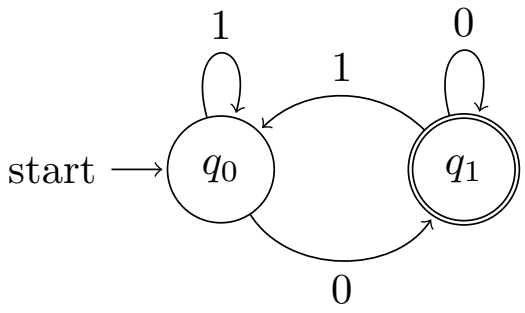


Figure 3.12: A DFA accepting only even binary numbers.

An example of a DFA that accepts only even binary numbers is provided above. For a binary number to be even, the least significant bit must be a 0. When this DFA consumes a 1, it stays or returns to the initial non-accepting state q_0, and when it consumes a 0, meaning the current least significant bit is 0, it moves to an accepting state q_1. Leading zeros are ignored.

Regular Expressions

Definition 3.5.6 (Regular Expressions)**.** Regular expressions are an algebraic notation that describes a regular language.

An example of a regular expression that accepts only even binary numbers is provided below. The expression matches any number of ones and zeros, but must always end with a zero.

$$L1 = \{\text{“0“}\}, L2 = \{\text{“1“}\}\ (L1 \mid L2)^* L1$$

Figure 3.13: A regular expression accepting only even binary numbers.

It holds true that NFAs, DFAs and regular expressions are closed under union, concatenation and the Kleene star. They are equivalent in all intents and purposes and may be converted between each other as shown in fig. 3.14.

Regular Expression	Language	Description
a	{„a“}	The set containing a single character „a“
ε	{„“}	The set containing an empty string
s|t	$L(s) \cup L(t)$	Strings from both languages, union
st	$\{vw \mid v \in L(s), w \in L(t)\}$	All strings formed by concatenating any string from the first language with any string from the second language.
s^*	$\{\text{„“}\} \cup \{vw \mid v \in L(s), w \in L(s^*)\}$	All concatenations of zero or more strings from $L(s)$

Table 3.1: Regular Expressions and their languages.

v

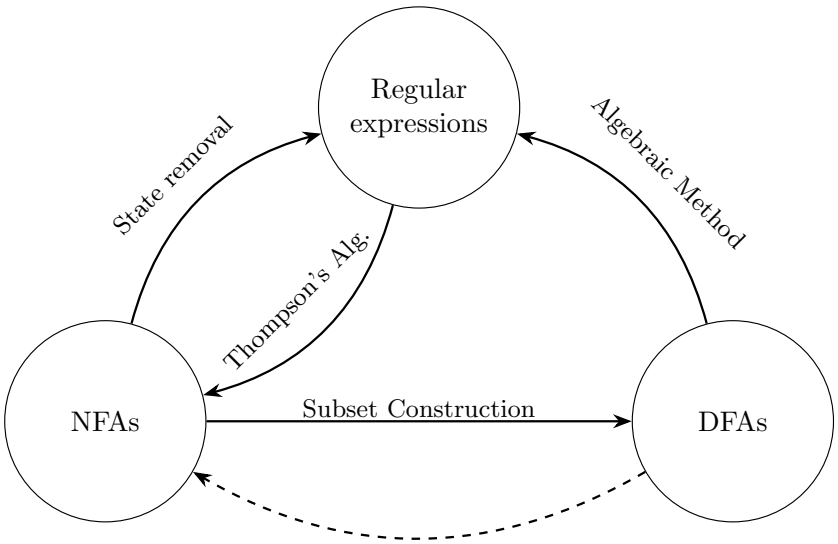


Figure 3.14: A diagram describing the equality of DFAs, NFAs and regular expressions.

Thus, we have shown that all threev devices are equivalent and may be used to describe a $\mathcal{L}_3$ language.

3.5.5 Context Free Languages

CFLs occupy the next level in the Chomsky hierarchy above the regular languages. They are precisely the class of languages generated by Context Free Grammar (CFG) and recognized by pushdown automata. CFLs are fundamental in syntactic analysis, since most programming language syntaxes may be described by CFGs. Although less restrictive than regular languages, CFLs still possesses many useful properties. Most importantly, decidability of membership via parsing algorithms (e.g. CYK, Earley's algorithm) and various closure properties (union, concatenation, Kleene star).

Context Free Grammars

Definition 3.5.7 (Context Free Grammar)**.** A CFG is a 4-tuple

$$G = (V, \Sigma, R, S),$$

where

- V is a finite set of *nonterminal* symbols,
- Σ is a finite set of *terminal* symbols, $V \cap \Sigma = \varnothing$,
- R is a finite set of *productions* of the form $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$,
- $S \in V$ is the *start symbol*.

The language generated by G is

$$L(G) = \{\, w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w \},$$

where $\overset{*}{\Rightarrow}$ denotes the reflexive-transitive closure of one-step derivations.

Pushdown Automata

Definition 3.5.8 (Pushdown Automaton)**.** A (nondeterministic) pushdown automaton (PDA) is a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F),$$

where

- Q is a finite set of states,
- Σ is the input alphabet,
- Γ is the stack alphabet,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ is the transition relation,
- $q_0 \in Q$ is the initial state,
- $Z_0 \in \Gamma$ is the initial stack symbol,
- $F \subseteq Q$ is the set of accepting states.

A string $w \in \Sigma^*$ is accepted if there exists a sequence of moves starting from (q_0, w, Z_0) that eventually reaches some $(q_f, \varepsilon, \gamma)$ with $q_f \in F$.

LL Languages

Left-to-Right, Leftmost Derivation (LL) languages form an important subclass of CFLs suitable for efficient top-down parsing without backtracking. An LL(k) grammar can be parsed by reading the input *L*eft-to-right, constructing a *L*eftmost derivation, with at most k symbols of lookahead.

Definition 3.5.9 (LL (k) Grammar)**.** A CFG G is called $LL(k)$ if for any two distinct productions

$$A \to \alpha \quad \text{and} \quad A \to \beta$$

in R, the sets

$$\text{First}_k(\alpha\,\gamma) \quad \text{and} \quad \text{First}_k(\beta\,\gamma)$$

are disjoint for all $\gamma \in (V \cup \Sigma)^*$ such that $A \overset{*}{\Rightarrow} \gamma A \gamma'$. Here $\text{First}_k(w)$ denotes the set of prefixes of length $\leq k$ of strings in $L(w)$.

In practice, LL (1) grammars are the most widely used. To transform an arbitrary CFG into LL (1) form, two common techniques are employed:

- *Elimination of left recursion*, since left-recursive productions prevent the parser from making a finite decision on which production to apply.
- *Left factoring*, to defer the decision until enough lookahead symbols are seen.

Once the LL(1) parsing table is constructed, a simple deterministic algorithm drives a parsing stack and input buffer, achieving $O(n)$ time complexity for an input of length n.

3.6 Compilers

Today's world is reliant on software more than ever before. It is imperative for the developers to be able to write software efficiently. 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 the computers can run. The computer program that facilitates this translation is called a *compiler*. See fig. 3.15.

Figure 3.15: 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 component phases, such as the Lexical Analyzer, Syntax Analyzer, Semantic Analyzer, Intermediate Code Generator, Machine-Independent Code Optimizer, Code Generator, Machine-Dependent Code Optimizer [13, Page 5]. However, this chapter only covers the components relevant to this thesis. See the relevant stages in fig. 3.16.

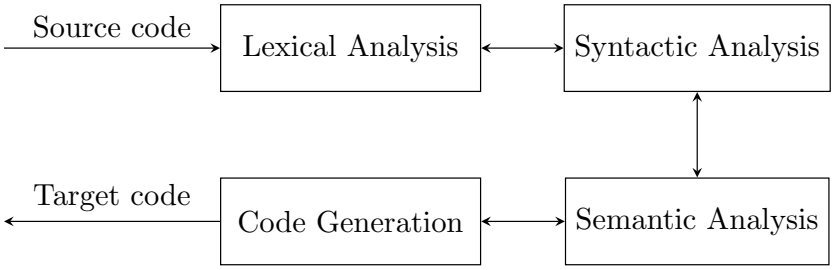


Figure 3.16: Overview of relevant compiler stages.

3.6.1 Lexical Analysis

The first stage of a compiler is lexical analysis, also known as a *lexer* or *tokenizer*. For the remainder of this thesis, the term *lexer* will refer to the lexical analysis stage of a compiler. The lexer consumes a stream of characters, or the *source code*, and returns a stream of *tokens*. A *token* is a lexically indivisible unit, for example, the Python keyword `return` cannot be divided any further, e.g. 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*, which is stored along with the token as a value.

3.6.2 Syntactic Analysis

The second stage receives the tokens produced by the lexer and validates the syntax. This is done by checking the input tokens against the grammar; this part of the compiler is called the *parser*. The parser checks if the tokens form valid sentences in the language; this is called the *acceptor*. The parser may also produce a *parse tree* or Abstract Syntax Tree (AST), which is a representation of the parsing steps taken and the rules applied.

3.6.3 Semantic Analysis

The third stage of a compiler is semantic analysis, however, in practice it is often not a separate stage, but rather intertwined with the parser. This process often involves checking the types and values of variables, ensuring that the variable and function names are declared before use, and checking for other semantic errors. This stage, however, was ommitted as it brings little value since it is expected that the user will provide valid code, which may be tested by first executing it in Lumerical.

3.6.4 Code Generation

The last stage of a compiler is code generation. The AST is processed and it is converted into the target language. This is done via the traversal of the AST and by generating code for each node or group of nodes.

Chapter 4

Transpiler Implementation

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

4.1 Choice of Tools

When writing any program, it is important to choose the right tools for the job. This comes down not only to choosing the programming language but also toc what libraries and packages to utilize and which components to implement to better suit the project's constraints. The source language, i.e. Lumerical's scripting language, allows for the automation of tasks and further analyses, such as manipulating the simulation objects, launching simulations and analyzing the results [14]. This language, however, 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 for easier editing and debugging due to its high-level nature. Consequently, and owing to the fact that the implementation language is the same as the target language, we will also be able to use built-in modules to generate the target code. The Python's ecosystem also includes tools such as Sphinx and Pytest for writing technical documentation and tests, respectively.

4.1.1 Implementation Overview

LUMEX is implemented in Python 3.13.3. It is a command line tool that takes a Lumerical scripting language file as input (specified by an `-f` or `-file` flag) and generates a Python equivalent in the output directory. Meep is not necessary for the functionality of LUMEX. However, it is required for the generated code to run.

The figure below outlines the layout of the submitted work.

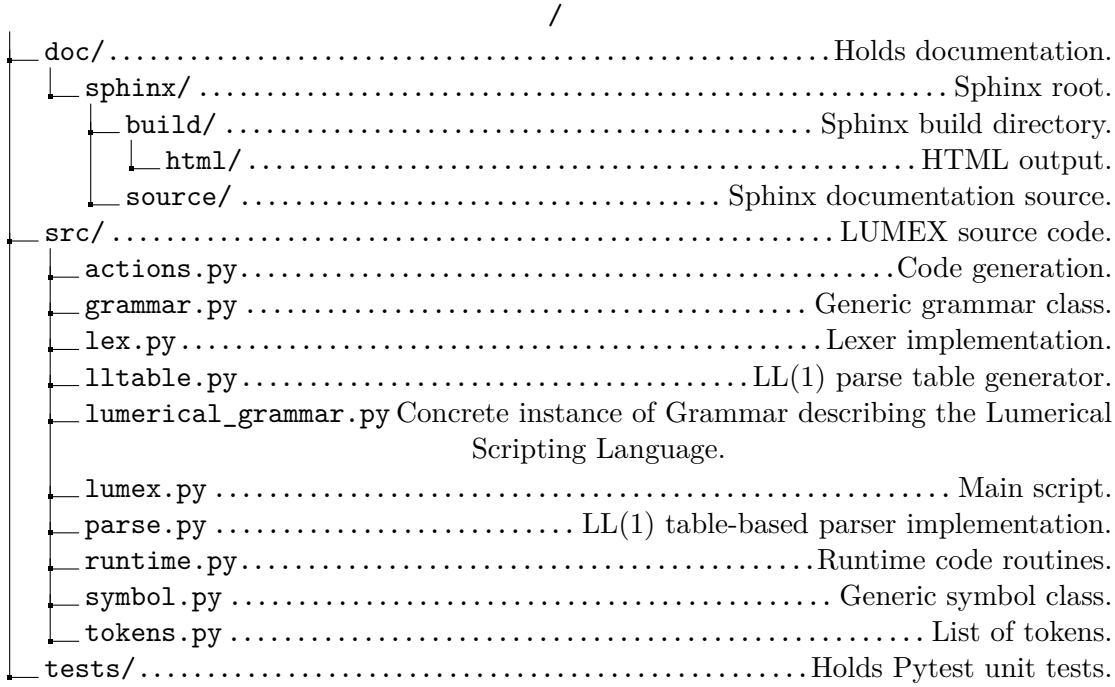


Figure 4.1: Overview of the project file structure.

4.1.2 Sphinx

Sphinx is a tool that automatically generates documentation by converting plain text source files into multiple output formats [15]. Sphinx was chosen because it facilitates the extraction of docstring style comments from the Python code, which may be enriched by the addition of ReStructured Text [16]. Multiple output formats, such as Portable Document Format, LaTeX source code and Hypertext Markup Language, are supported. Sphinx also includes multiple extensions, which allow the parsing of LaTeX into Scalable Vector Graphics, which are easily rendered on the Internet.

This is instrumental in providing the user with the necessary insight into the technical operation of the transpiler, and allows us to search and view information in a concise manner without the need to directly inspect the source code. The documentation is made available to the reader at the following address[1].

[1] `https://www.macura.sk/LUMEX/`

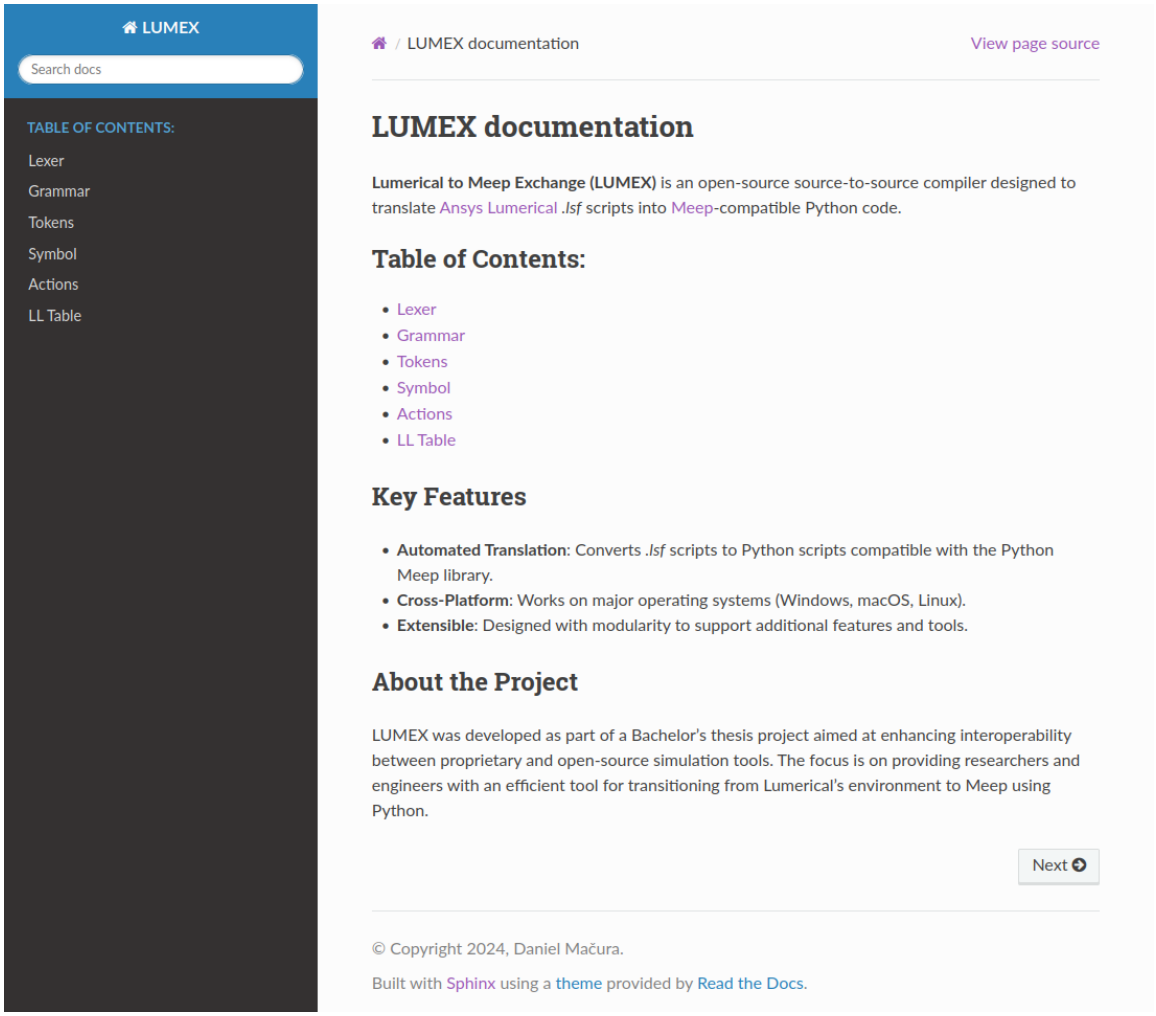


Figure 4.2: Screen-capture of the resulting documentation.

4.1.3 Pytest

Pytest is a widely used testing framework for Python. It is used to write simple as well as scalable test cases. In this thesis, it is used to test the functionality of the transpiler. Test cases are written for the lexer for separate tokens, the parser for generating the parse table, and the code generator for the entire transpilation process. The tests are run automatically, and the results are reported in a human-readable format. The test cases are located in the `tests` directory, and they are run using the command `pytest`.

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-Naur form (EBNF), Wirth Syntax Notation (WSN), Augmented Backus-Naur form (ABNF), Parsing Expression Grammar (PEG) or Zephyr Abstract Syntax Definition Language (ASDL) [17]. The Python grammar is defined

by a mixture of EBNF and PEG, as used in the CPython parser [18]. Additionally, the Python AST module also defines an abstract grammar in ASDL [19].

As of writing this thesis, Lumerical has not publicly provided any such specifications for Lumerical's scripting language. It is therefore necessary to analyze the language and derive a grammar that covers the relevant subset of the language for the scope of this thesis.

Since this thesis focuses primarily on nanophotonic simulation workflows in 3D, only essential commands to set up a simulation will be included, e.g. commands to add blocks, sources, monitors and commands to translate and scale the objects. There may be other parts of Lumerical's scripting language that could be utilized for a nanophotonics simulation. However, those are considered out of the scope of this thesis due to the broad range of Lumerical's scripting language with around 800 commands and keywords [20]. Since Meep is a FDTD solver, only the commands usable in Lumerical FDTD simulations will be considered. While Lumerical's scripting language includes hundreds of commands spanning across optimization, analysis and post-processing (e.g., `runoptimization`, `fitlorentzpdf`, `exportcsvresults`), this work prioritizes the core geometry and simulation workflow: `addrect` (block creation), `addplane`/`addfdtd` (source/solver definition), `addpower` (monitor placement), and `set` (property configuration).

There are infinitely many distinct grammars that describe the same language. By way of example, in a CFG given as $S \to aB$, one may add an intermediate rule, producing $S \to aC, C \to B$. This process may be repeated indefinitely and result in the creation of new grammars that describe the same language. To this end, the grammar described below is just one of the plethora of possible grammars describing a subset of Lumerical's scripting language. While constructing the grammar, it is beneficial to impose additional constraints and only utilize a subset of CFGs, namely a LL grammar. This imposes certain rules as described above, informally speaking: when a parser arrives at a nonterminal, it must be able to decide which production to apply by peeking at the next k symbols; this is an LL(k) grammar.

This will later help with constructing a simpler parser. However, if a language construct were found in Lumerical's scripting language outside of the subset of this thesis, it could not be described and parsed by a LL(1) parser or even a LL(k) parser, and a more powerful parser would be required. This would incur the need to swap out the parser implementation in `parse.py`. Because of this, and for similar reasons, the transpiler was written in a modular fashion, making it a trivial matter. Other common parser types might include the Look-Ahead, Left-to-Right, Rightmost Derivation (LALR) parser, which is stronger than the LL parser, but just as fast, or the Generalized Left-to-Right Rightmost Derivation (GLR) parser, which can even handle ambiguous grammars.

4.2.1 Lumerical Grammar

The following subsection will introduce the grammar, upon which LUMEX is based. Design choices and other considerations will be discussed. The following grammar listings will not include actions and code generating constructs that will be introduced in section 4.5. The full grammar including the actions is provided in Listing A in Appendix A. The grammar is written loosely in a EBNF style where **bold** keywords are terminal symbols and `type-writer` keywords are non-terminal symbols.

Table 4.1: Long Title

1.	`root`	→	`body`
2.	`body`	→	`statement body`
3.		|	`function body`
4.		|	**EOF**
5.		|	ε
6.	`function`	→	**FUNCTION IDENTIFIER** (`parameter_list`) **{** `nested_body` **}**
7.	`nested_body`	→	`statement nested_body`
8.		|	ε

The grammar begins with `root`. This is mainly for code generation purposes, which will be shown later. After that, `body` holds the code, which was split into statements and functions, indicated by their respective nonterminals. This is due to the fact that nested functions cannot be defined. Thus, function bodies use a separate `nested_body` nonterminal, which does not allow a function definition. An astute reader may notice that many nonterminals are written such that `nonterminal` → [nonterminals and terminals] `nonterminal` and `nonterminal` → ε. This concept is referred to as *right recursion*. It allows for the nonterminals to appear 0 or more times, and it can be parsed by an LL parser, unlike the left recursion. This concept will be used throughout the rest of the grammar. The `body` contains an end of file token, which is used as a termination of the transpilation process, and is emitted by the lexer.

Table 4.2: Long Title

9.	`statement`	→	**IDENTIFIER** `identifier_action`
10.		|	`control_structure`
11.		|	`selection`
12.		|	**?** `expression` **;**
13.		|	**ADDFDTD ;**
14.		|	**ADDRECT ;**
15.		|	**ADDSPHERE ;**
16.		|	**ADDPLANE ;**
17.		|	**ADDDFTMONITOR ;**
18.		|	**SET (STRING ,** `expression` **) ;**
19.		|	**BREAK ;**
20.	`selection`	→	**SELECTALL ;**
21.		|	**UNSELECTALL ;**
22.		|	**SELECT (STRING) ;**
23.		|	**SHIFTSELECT (STRING) ;**

The `statement` nonterminal is thematically divided into multiple sections, identifier actions, control structures, selection commands and add commands. Production 12 contains an equivalent of the Python `print` function, which is denoted by prepending a question mark symbol before the expression.

Following that, the productions starting with **ADD** implement the add commands, which are used for adding geometry, sources and monitors to the simulation. Production 18 implements the set command that Lumerical uses to change the properties of the currently selected object. The set command follows the key-value convention, where the key is a string and the value is an expression.

The selection commands are listed under the `selection` nonterminal. This nonterminal consists of commands used to change the currently selected objects in the scene: **SELECTALL** and **UNSELECTALL** are self explanatory, **SELECT** takes a string, which it tries to match with the object name, and all objects with that name become the only selected objects. **SHIFTSELECT** is similar, however, it doesn't unselect the previous objects. Lumerical doesn't provide a command that is the opposite of the **SELECT** command, e.g. a **UNSELECT** command.

Table 4.3: Long Title

24.	`identifier_action`	→	`assignment`	
25.		|	`function_call`
26.	`assignment`	→	= `expression` ;	
27.	`function_call`	→	(`parameter_list`) ;	
28.	`parameter_list`	→	**IDENTIFIER** `parameter_list_prime`	
29.		|	ε
30.	`parameter_list_prime`	→	, **IDENTIFIER** `parameter_list_prime`	
31.		|	ε
32.	`argument_list`	→	`expression` `argument_list_prime`	
33.		|	ε
34.	`argument_list_prime`	→	, `expression` `argument_list_prime`	
35.		|	ε

The `identifier_action` nonterminal is a refactor of the grammar to avoid ambiguity with a single look-ahead token because both the assignment and a function call begin with an **IDENTIFIER**. This is because Lumerical scripting language does not impose any rules that would allow the lexer stage to discern a function name from a variable, e.g. the dollar symbol in front of the variables, as is the case in PHP. The list of parameters and arguments is essentially the same grammatical construct, the only difference being that either the **IDENTIFIER** or `expression` is being repeated while separated by commas.

Table 4.4: Long Title

36.	`control_structure`	→	**IF** (`expression`) { `nested_body` } `elseNT`
37.		|	**FOR** (**IDENTIFIER** = `expression` `loop_condition`)

Continued on next page

Table 4.4: Long Title (Continued)

			{ `nested_body` **}**
38.	`elseNT`	→	**ELSE** `elifNT` **{** `nested_body` **}** `elseNT`
39.		|	ε
40.	`elifNT`	→	**IF** **(** `expression` **)**
41.		|	ε
42.	`loop_condition`	→	**:** `expression` `range_step`
43.		|	**;** `expression` **;** `expression`
44.	`range_step`	→	**:** `expression`
45.		|	ε

The `control_structure` nonterminal is responsible for accepting the conditional statements and loops. Conditional statements begin with an if clause, which evaluates an expression and executes its body of code if the expression is true. Following this, an optional else clause can be included. This clause branches into two possible paths:

1. Additional „else if“: If the **ELSE** token is followed by another **IF** token, it creates a chained „else if“ branch, allowing additional checks for different conditions. This process can repeat, enabling multiple sequential evaluations.

2. Final „else“: If no other conditions follow the **ELSE** token, it serves as the final block. This final else clause executes its code only when all preceding conditions are false.

The For loops are supported in both notation conventions, i.e. `for(x=1:2:100) {}` and `for(x=1; x<= 100; x=x+2) {}`. This is handled by first accepting the **FOR** token until the `loop_condition` nonterminal, then one of two possibilities is accepted:

1. Range notation: If a colon follows the expression, a range is expected. Ranges may either specify the start and end only, or also the step, i.e. *start:stop* or *start:step:stop*. This is where the `range_step` comes into play: if a second comma is accepted, the step variation is accepted.

2. „C“ style: If a semicolon is parsed, the For loop is formatted in the „C“ style, i.e. *x=start; x<= stop; x=x+step*, according to Production 43.v

Table 4.5: Long Title

46.	`expression`	→	`logic_and`
47.	`logic_and`	→	`logic_or` `logic_and_prime`
48.	`logic_and_prime`	→	**AND** `equality` `logic_and_prime`
49.		|	ε
50.	`logic_or`	→	`equality` `logic_or_prime`
51.	`logic_or_prime`	→	**OR** `equality` `logic_or_prime`

Continued on next page

Table 4.5: Long Title (Continued)

52.		|	ε
53.	`equality`	→	`comparison equality_prime`
54.	`equality_prime`	→	`!= comparison equality_prime`
55.		|	`== comparison equality_prime`
56.		|	ε
57.	`comparison`	→	`term comparison_prime`
58.	`comparison_prime`	→	`> term comparison_prime`
59.		|	`>= term comparison_prime`
60.		|	`< term comparison_prime`
61.		|	`<= term comparison_prime`
62.		|	ε
63.	`term`	→	`factor term_prime`
64.	`term_prime`	→	`- factor term_prime`
65.		|	`+ factor term_prime`
66.		|	ε
67.	`factor`	→	`unary factor_prime`
68.	`factor_prime`	→	`/ unary factor_prime`
69.		|	`* unary factor_prime`
70.		|	ε
71.	`unary`	→	**NOT** `unary`
72.		|	`- unary`
73.		|	`primary`
74.	`primary`	→	**INTEGER**
75.		|	**FLOAT**
76.		|	**STRING**
77.		|	**IDENTIFIER**
78.		|	`( expression )`

Expressions are undoubtedly one of the most critical aspects of a grammar. There are two predominant ways to parse expressions with LL parsers, which affect the way the grammar is structured. The first is with the use of a separate bottom-up parser, typically a precedence parser. This involves the creation of a precedence table between the operators and operands based on the Wirth-Weber precedence relation, with shift/reduce operations as values [21]. The parser is called each time an expression is encountered. However, this method increases the complexity of LUMEX, which is why we chose the following approach. This second approach involves hard-coding the precedence of the operators into the grammar. In the grammar shown above, the precedence is ordered from low to high. The parsing process is best shown in an example in fig. 4.3 for the expression $2 * (1 + 3)$.

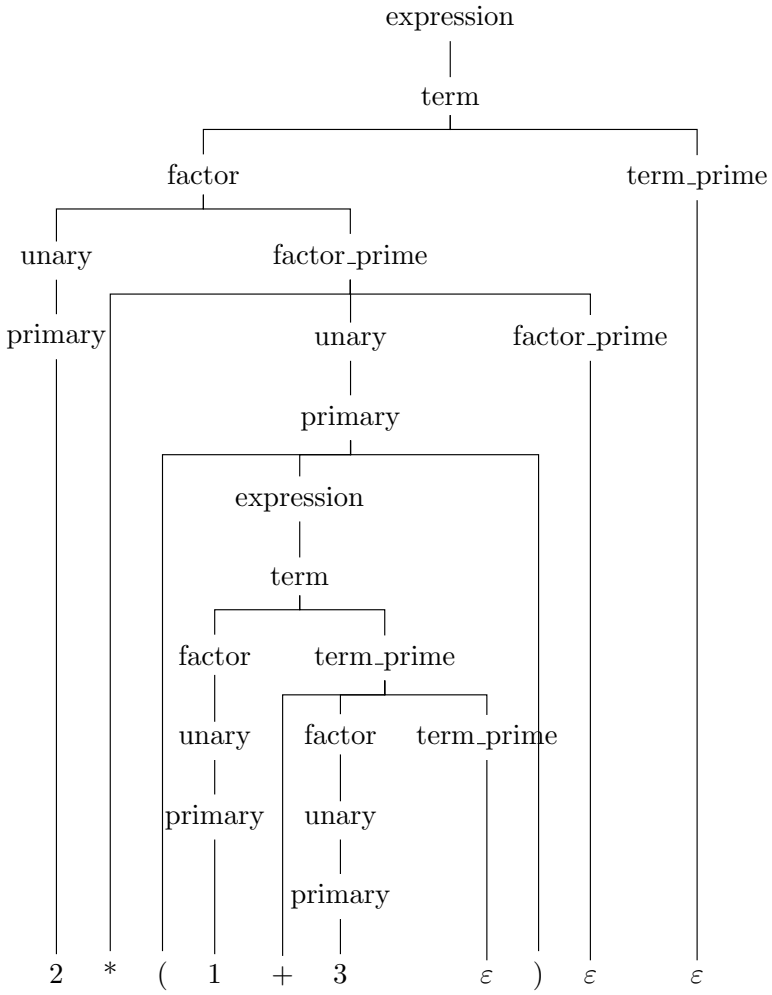


Figure 4.3: Example parse tree for the expression $2 * (1 + 3)$.

The example above shows the parsing process and how a lower-precedence operator is not attempted to be parsed until all higher-precedence levels have been fully reduced (via their `_prime` and ε-branches); the grammar structure ensures, for example, that $*$ binds tighter than $+$, and that $+$ binds tighter than $==$. The parentheses are handled in `primary`, letting the parser suspend this fixed precedence ordering and recursively parse a new full expression inside. To sum up, the ε-productions terminate the repetition on each level, while the layering of nonterminals enforces both the precedence tiers and the correct associativity for every operator.

The grammar is implemented as a standalone class. The `Grammar` class is written in an abstract manner, allowing it to hold information about the Lumerical Scripting Language, while staying extensible. To facilitate this, the `Grammar` class holds a list of productions. The productions are represented as separate `Production` classes holding information about the Left-Hand Side (LHS) and Right-Hand Side (RHS) of the production. The LHS is a single nonterminal, while the RHS is a list of nonterminals, terminals, actions, or a ε symbol. The `Production` class also keeps track of the production number, assigning each

production a unique numerical value in an ascending fashion, and whether the production is nullable.

Circling back to the `Grammar` class, it also provides helper functions, such as `append`, which adds the passed production to the grammar, while also updating the `nullable` attribute of the production. The `Grammar` class also provides the `terminals` and `nonterminals` methods to list all terminals and nonterminals in the grammar.

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 straightforward being the use of a DFA formalism and a loop over all input characters while 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 upon which no applicable production can be found.

Algorithm 1: State Machine Based Lexer.

```
current_state ← None
last_accepting_state ← None
while look-ahead_character != EOF do
    if has_transition(current_state, look-ahead) then
        current_state ← next_state(current_state, look-ahead)
        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; however, with an ever-growing number of states, which have to be hard-coded; the solution may become rather complex. Indeed, a cleaner method exists. Since any DFA may be transformed into a regular expression, 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_token remove_matched_lexeme_from_input
else
    return None ;                                          // Failure
```

This approach is not only more straightforward, but also allows us to model the lexer in a more pythonic manor. Each token is represented as a unique class, which holds the regular expression pattern matching the corresponding token. Each specific token class inherits a common token class, and the lexer only has to enumerate over all children of the token class and search for the one with the longest match, resulting in a greedily matched token.

This is reflected in the `Lexer` class, which provides a stream of tokens. The class holds information about the source code, which is to be processed. This is achieved through the `source` attribute and `cursor` attribute, which is incremented as the source code is parsed. The `Lexer` class implements a `tokens` generator, which repetitively calls `advance` and yields the tokens until all are exhausted, while discarding white space tokens, such as the new line and space token.

The `advance` method returns a single token, or the None value, when the end of the input is reached. Otherwise, it greedily matches the longest token as described in Algorithm 2. This is achieved by iterating over all tokens, which are obtained by retrieving all subclasses of the `Token` class, as each individual token is a child class.

The `Token` class holds information about a token, such as the name, given by the class `__name__` method. The lexeme pattern, which is used to match the token, is stored as a regular expression. The `token` class also stores the lexeme, that it matched. This is used during logging and mainly for storing the value of literals, such as strings, integers and floats, which are used during the code generation phase.

4.4 LL Parser

The LL grammar, specifically LL (1), was chosen for reasons described in section 4.2. There are two common implementations of parsers that accept LL (1) grammars. The first is a recursive descent parser. It is a kind of top-down parser that models the parsed grammar in such a fashion that each nonterminal of the grammar is represented by a procedure in code. The methods recurse and invoke each other, parsing the grammar. This is a simple procedure to implement, however, it often results in long code that is proportional to the size of the grammar, which makes it harder to comprehend and easier to introduce mistakes in the implementation. This does not pair well with the future possible extensions of this transpiler to cover the entirety of Lumerical's scripting language.

The second method is a LL table-based parser. It is a deterministic pushdown automaton utilizing a stack and a parsing table. It continually expands the leftmost nonterminal

in a derivation with a look-ahead of one token to choose the correct production from the parsing table. It starts by reading a token from the lexer and looking at the topmost item on the stack; in the case of a nonterminal, the corresponding production from the parsing table, given by the nonterminal and terminal, is pushed on the stack. If no production is found, a parsing error is raised. On the other hand, if the top stack symbol is a terminal, it must match the input terminal. Otherwise a parsing error is raised. This notion is shown in Algorithm 3 based on [22, Table 4.12.].

Algorithm 3: Top-down table based parsing algorithm.

```
initialize stack with root grammar nonterminal while parse stack not empty do
    if top symbol is nonterminal then
        what_to_push = parse_table[top symbol][current input token]
        if what_to_push == None then
            Syntax error.
        else
            add reversed what_to_push to stack
    else if top symbol is terminal then
        if top symbol == current input token then
            if current input token != end of file then
                get next token
        else
            Syntax error.
```

The theory of a parsing table is pretty straightforward. Each production in the grammar In the parsing table, each row corresponds to a nonterminal and each column to a terminal. The cell at the intersection of a row and column contains the production to be applied. If no production is found, a parsing error is raised. A simple example of a LL(1) grammar is provided in Figure 4.4 and the corresponding parsing table in Table 4.6.

1.	stmt	$\rightarrow$	expr **SEMI**
2.	expr	$\rightarrow$	factor expr_tail
3.	expr_tail	$\rightarrow$	**PLUS** expr
4.		$\mid$	ε
5.	factor	$\rightarrow$	**NUMBER**

Figure 4.4: Simple example BNF grammar that is LL(1).

	SEMI	PLUS	NUMBER
stmt			1
expr			2
expr_tail	4	3	
factor			5

Table 4.6: Example LL(1) parsing table based on the grammar in Figure 4.4.

4.4.1 LL Table Construction

The LL (1) parse tables are constructed using the FIRST, FOLLOW and SELECT sets, which are solved for each nonterminal. The FIRST set of a nonterminal consists of all terminal symbols that can appear at the beginning of any string derivable from that nonterminal.

The FIRST sets are evaluated in an iterative process for the whole grammar. The set is evaluated for each nonterminal present in the grammar, always updating the corresponding set by performing a FIRST closure, until each nonterminal's FIRST set does not change. The theoretical approach it detailed below. In the following sections, ε will be treated as a terminal symbol for simplification, even though it is not, with the corresponding LHS and RHS.

Algorithm 4: The FIRST closure algorithm.

```
if production is nullable then
    return
for symbol in production.RHS do
    if symbol is terminal then
        add symbol to first[production.LHS]
        return
    else
        add first[symbol] to first[production.LHS]
        if symbol is not nullable then
            return
```

Algorithm 4 corresponds to the generalized rule for calculating the FIRST set. For a production in the form of $s \rightarrow \alpha\mathcal{B}\beta$, where s is the nonterminal, for which the FIRST set is being calculated, α are 0 or more nullable[2] nonterminals, $\mathcal{B}$ is a single terminal or nonterminal and β are multiple terminals or nonterminals. Then the FIRST(s) set is the union of FIRST(β) and FIRST(α) sets [22, Chapter 4.7.1].

Similar to the FIRST sets, the FOLLOW sets are computed incrementally until no change in the sets is observed. The FOLLOW contains those terminals, that may follow the nonterminal, for which the FOLLOW set is evaluated.

[2] As a reminder, a nonterminal is *nullable* if it can be transformed to ε by some production, i.e. there has to be at least one production of the form $s \rightarrow \varepsilon$, where s is a nonterminal.

Algorithm 5: The FOLLOW closure algorithm.

```
if production is nullable then
    return
for symbol in production.RHS do
    if symbol is terminal then
        continue
    else
        if next_symbol is terminal then
            add next_symbol to follow[symbol]
        else
            add first[next_symbol] to follow[symbol]
for symbol in reversed production.RHS do
    if symbol is terminal then
        break
    else
        last_nonterminal = symbol
        add follow[production.LHS] to follow[symbol]
        if symbol is not nullable then
            break
if last_nonterminal then
    add follow[production.LHS] to follow[last_nonterminal]
```

Algorithm 5 provides a general method for computing the FOLLOW set. Consider a production expressed as $s \to \ldots a\alpha\mathcal{B}\ldots$, where s and a are nonterminals, α is a possibly empty sequence of nullable nonterminals, and $\mathcal{B}$ is a terminal or nonterminal. FOLLOW(a) is updated with the union of FIRST(α) and FIRST($\mathcal{B}$) [22, Chapter 4.7.2].

Finally, we need to calculate the SELECT sets (also known as PREDICT). Unlike the previous two cases, the SELECT sets are calculated in one pass. The LL(1) SELECT set for a production rule consists of all input symbols that can appear as the next token in the input stream when that production is correctly applied.

Algorithm 6: The LL(1) SELECT algorithm.

```
for production in productions do
    for symbol in production.RHS do
        if symbol is nonterminal then
            add first[symbol] to select[production]
        add follow[production.LHS] to select[production]
        if symbol is nonterminal then
            add follow[production.LHS] to first[production]
            if symbol is not nullable then
                break
        else
            add symbol to select[production]
            break
```

Algorithm 6 outlines a general procedure for computing the LL(1) SELECT set, which corresponds to the following formal definition. When dealing with a non-nullable[3] production of the form $s \to \alpha\mathcal{B}\ldots$, where s is a nonterminal, α is a sequence of one or more nullable nonterminals, and $\mathcal{B}$ is either a terminal or a non-nullable nonterminal (i.e., one that cannot derive ε), possibly followed by additional symbols. The LL(1) SELECT set for this production is the union of FIRST(α) and FIRST($\mathcal{B}$). On the other hand, when a nullable production is encountered in the following form $s \to \alpha$, where s is a nonterminal, α denotes a sequence comprising zero or more occurrences of nullable nonterminals. The union of FIRST(α) and FOLLOW(s) is added to the LL(1) SELECT set for the production [22, Chapter 4.7.3].

After obtaining the LL(1) SELECT sets for the grammar, the construction of the LL parsing table is trivial.

Algorithm 7: Translation of LL(1) SELECT sets to a parse table.

```
initialize table to error states for production in productions do
    for token in select[production] do
        table[production.LHS][token] = production
```

This approach is implemented in the `LLTable` class. The class is instantiated using the grammar for which it calculates the first, follow and predict sets. This is handled in `ComputeFirstSets`, which calls `FirstClosure` until there are no more changes, as described above. Following that, `ComputeFollowSets` is called, which, as in the case of the first sets, calls its respective function `FollowClosure` until no more changes are made to the follow sets. As the last preliminary step, the select sets are computed using the `ComputeSelectSets` function.

4.5 Code Generation

Up to this point, we have applied multiple steps of the transpilation process, such as splitting the input code into tokens, creating a LL(1) parsing table and using it to parse the input code. Thus far, we have verified that the input code is a valid Lumerical Scripting Language[4], however we have not created any resulting Python code. This is where the concept of *actions* is introduced [22].

Actions are symbols added to the grammar. They act as a terminal symbol in the sense that they do not get expanded. However, they do not consume a token from the input upon being matched. On the contrary, actions store logic that is executed at specific points in the grammar, which is used for generating the resulting Python code.

Actions are implemented as a separate `Action` class, and have an abstract method `call`. When the parser processes an action, it automatically invokes this aforementioned `call` method.

[3] The opposite of a nullable production—that is, a production whose right-hand side is not solely ε, nor composed entirely of symbols that can derive ε through some sequence of derivations.

[4] As discussed in section 3.6.3 on p. 28, the input code is expected to be semantically correct. This can be simply verified by executing the code in Lumerical.

4.5.1 Python AST

As mentioned above in section 4.2, Python contains a built-in module called `ast`[5]. This module helps with processing the Python code, generating ASTs out of the Python code and, most importantly, generating the Python code from a AST. For clarity, all AST nodes provided by the Python `ast` module will be prepended with `ast.`.

The Python `ast` module provides a root node named `ast.Module` that has an attribute `body`, which is a list of the module statements. This is the node we will build upon in the parsing process by adding the statements we parse to the `ast.Module`. Thus, extending the Algorithm 3 with the option to accept actions in the grammar, whose full form containing the actions is listed in Appendix A, we get the final version of the top-down table based parser as seen in Algorithm 8 below.

Algorithm 8: Top-down table based parsing algorithm extended with actions.

```
initialize stack with root grammar nonterminal
initialize value_stack with Module node
initialize empty token_stack
while parse stack not empty do
    if top symbol is nonterminal then
        what_to_push = parse_table[top symbol][current input token]
        if what_to_push == None then
            Syntax error.
        else
            add reversed what_to_push to stack
    else if top symbol is terminal then
        if top symbol == current input token then
            if current input token != end of file then
                put token on token_stack
                get next token
        else
            Syntax error.
    else
        invoke action call function with value_stack and token_stack
```

The changes to the previous algorithm are highlighted in red, namely the addition of `value_stack`, `token_stack`, and the action conditional branch. The `value_stack` is used to store the AST node. However, a simple variable is not sufficient because there are cases in which the whole expression is not yet parsed; thus, the last-in, first-out queue is used. This allows the nodes, which are not yet fully parsed, to be pushed onto the stack and popped once they are complete. The `token_stack` is used mainly as a way for the actions to also access the tokens that were parsed. This is not necessary for the tokens such as keywords and other syntactic sugar[6], however, when facing tokens with semantic meaning, such as integers, strings and others, their lexeme carries important information, which is

[5] `https://docs.python.org/3/library/ast.html`
[6] `https://en.wikipedia.org/wiki/Syntactic_sugar`

used during the generation. Both of these are expected by all action `call` methods, even though the `token_stack` is used sparsely and a uniform interface was deemed desirable.

4.5.2 Actions

An exhaustive description of every implemented action would be excessively laborious and contribute little meaningful value. However, an overview of the implementation techniques and key examples will be provided in this subsection.

Anatomy of an Action Actions were generally divided into 3 parts:

- **Extraction Phase:** The stack values are read and popped into local variables.
- **Construction Phase:** The AST node is created using the extracted values.
- **Restoration Phase:** The newly constructed node is returned and the stack is reordered.

StoreToBody The most commonly used action is `StoreToBody`. As mentioned above, the `ast.Module` has an attribute `body`, however, it is not the only one since many others, such as the `ast.If`, `ast.Else`, `ast.For`, each has a `body` attribute. This action expects the `value_stack` to contain a node at the top, which is the node we store to the `body` of the second node below it on the stack. Both of the nodes are popped off, the top node is appended to the `body` list of the nodes below and that node is pushed back on the stack. This allows us to prepare the nodes in advance and then add them to the body of the scope once the whole node has been parsed.

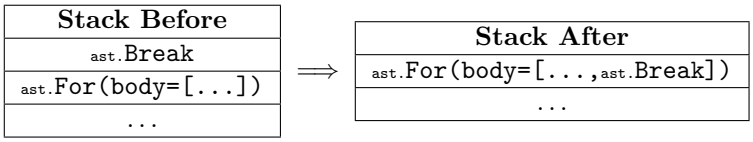


Figure 4.5: LIFO value stack before and after applying `StoreToBody`.

StoreLiteral The `StoreLiteral` is one of the few actions that also utilize the initialization method, simply by accepting a type value derived from the grammar, which is later used for the conversion, which will be described soon. When called, the action first pops a value of the `token_stack`, which is the literal pushed onto the `token_stack` by the parser. Then, an attempt is made to convert the string representation to a built-in Python type as specified in the initialization function. Following this, a `ast.Constant` node is created. This will store the converted value as its `value` attribute and it is pushed on the stack. `StoreLiteral` is called after parsing a literal in the grammar.

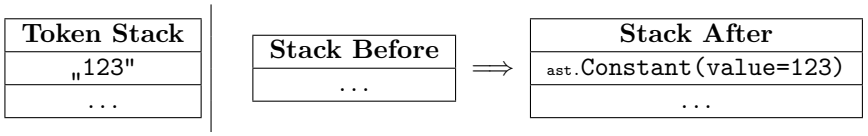


Figure 4.6: LIFO value stack before and after applying `StoreLiteral`.

StoreVariableName `StoreVariableName` is a helper action. It pops a variable name from the token stack and converts it to a `ast.Name` which is used e.g. for assigning the values to variables. The `ast.Name` context is set to store.

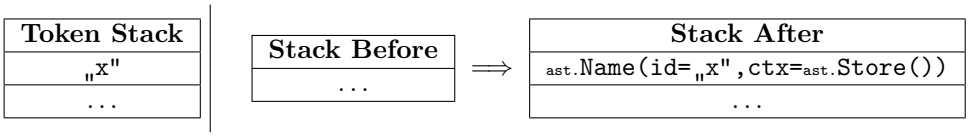


Figure 4.7: LIFO value stack before and after applying `StoreVariableName`.

AssignToVariable The `AssignToVariable` is used to assign a value to the variable. It is called after the whole expression has been parsed. The action expects a value, typically `ast.Const`, on top of the stack. Next, a second node is popped from the stack, and it must be an `ast.Name`. Once both the name and value are acquired, a `ast.Assign` node is created and pushed on the stack.

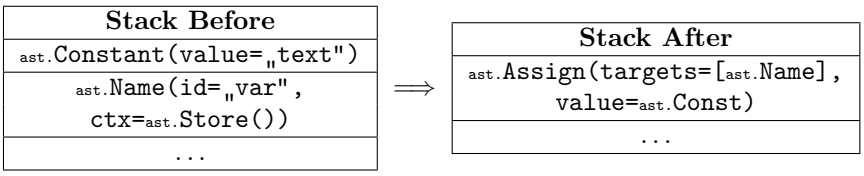


Figure 4.8: LIFO value stack before and after applying `AssignToVariable`.

BinaryOperation Previously, we examined the creation of constants and their assignment to variables, and we also presented an example parse of an expression in Figure 4.3 (page 37). A binary operation, as implied by its name, expects two operands — expressions. The right operand was parsed later, thus it will appear at the top. Following that, the left operand is also popped from the stack. To decide which operator to use, a `ast.operator` is passed into the initialization function of the action. After obtaining both operands and the operator, a `ast.BinOp` node is constructed and pushed on the stack.

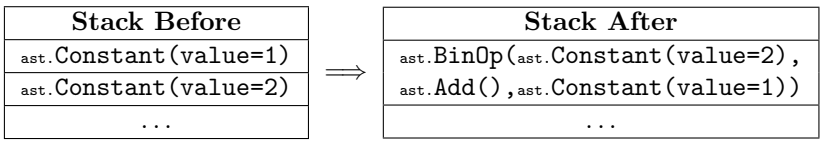


Figure 4.9: LIFO value stack before and after applying `BinaryOperation`.

Boolean logic operators are implemented similarly, however, their operands are added as a list due to Python's design, e.g. `ast.BoolOp(op=self.op, values=[left, right])`. Unary operators are implemented by popping the value of the stack, and pushing it as e.g. `ast.UnaryOp(op=USub(), operand=operand)`.

If The `If` action constructs the first part of a conditional statement, the `if` branch. However, to properly explain how later `elif` and `else` branches are constructed, an overview of how Python `ast` parses conditional statements[7] is necessary. This is best shown in the example below.

[7] `https://docs.python.org/3/library/ast.html#control-flow`

```
If(
  test=a,
  body=[Constant(value=1)],
  orelse=[
    If(
      test=b,
      body=[Constant(value=2)],
      orelse=[Constant(value=3)]
    )
  ]
)
```

(a) AST representation

```
if a:
    1
elif b:
    2
else:
    3
```

(b) Python source code

Figure 4.10: A conditional Python statement and its corresponding AST structure.

The initial `if` branch contains a `test` attribute which has to evaluate to `True` for the first `body` to be executed. If the `orelse` attribute is empty, the previous statement was a single `if`, however, if the attribute contains nodes other than another `ast.If` node, they are executed as part of the `else` path of the conditional statement. On the contrary, if the `orelse` attribute of the initial `ast.If` node contains a nested `ast.If`, it is treated as a `elif`. This can be repeated indefinitely.

With an understanding of the Python `ast` control flow, we now return to the implementation of the `If` action, which pops a value of the stack. This is the expression that will be evaluated as the test. Following this, a `ast.If` node is created, the expression is passed to the `test` attribute and the node is pushed on the stack.

Stack Before
ast.Expression()
...

$\Longrightarrow$

Stack After
ast.If(body=[],orelse=[], test=ast.Expression())
...

Figure 4.11: LIFO value stack before and after applying `If` action.

StoreToElse An attentive reader may have noticed that we need to add nodes not only to the `body` of the `ast.If`, but also to the `orelse`. However, our `StoreToBody` only stores to the `body`, and this is where `StoreToElse` is utilized. That said, there are two cases that have to be handled, the first of which stores a node to the terminal `else`.

To begin with, we pop the target node from the stack, which becomes part of the `orelse`, following that, we pop a second value of the stack; this is a `ast.If` node. If it were an expression, the second case would occur. Once the `ast.If` node and the target node have been obtained, we recursively traverse the `ast.If` node's `orelse` attribute. If `orelse` contains another `ast.If`, the traversal continues until we encounter an `ast.If` whose `orelse` is not itself an `ast.If`. At that deepest level, we append the target node to its `orelse` list and push the `ast.If` node on the stack.

Stack Before
ast.Constant()
ast.If(body=[...], orelse=[...])
...

$\Longrightarrow$

Stack After
ast.If(body=[..., orelse=[..., ast.Constant()])
...

Figure 4.12: LIFO value stack before and after applying `StoreToElse` action in the first case.

In the second case, following the first popped target node, a `ast.Expression` is encountered, which is the test for the `elif`. Thus, we pop the final node, now the `ast.If`. Similar to the procedure above, we recurse to the deepest `ast.If` and we append the target node to its `body` list. This node is pushed on the stack, but the `ast.Expression` cannot be consumed by this action because, in this case, we could not add multiple statements. Therefore, the `ast.Expression` is preserved and pushed onto the stack.

Stack Before
ast.Constant()
ast.Expression()
ast.If(body=[...], orelse=[...])
...

$\Longrightarrow$

Stack After
ast.Expression()
ast.If(body=[..., ast.Constant()], orelse=[...])
...

Figure 4.13: LIFO value stack before and after applying `StoreToElse` action in the second case.

`HandleElse` and `HandleElseCleanup` do not constitute their own section. However, a quick overview of the `HandleElse` is as follows. The action pops an item of the stack, in case a `ast.If`, it is returned and nothing is done. In the case of an `ast.Expression`, a second node, which is now a `ast.If`, is popped, the expression is set as the node's `test` attribute and both are returned, such that the expression is on top. At the end of the whole `if-elifelse` chain, `HandleElseCleanup` is invoked, searching for an `ast.Expression` as the top node and discarding it.

Loops Other forms of control structures include the `for` and `while` loops. Lumerical does not support while loops; however, they substitute them with `for` loops in the form of `for(0; conditional_expression; 0) {}` as per the documentation[8]. However, since Lumerical permits arbitrary modifications of the loop's stop-condition variable within the loop body, the use of Python `for` loops to implement the Lumerical `for` loops is not feasible even though Lumerical provides a range notation. Thus, all loops are implemented as `while` loops in Python. The following actions help to create the `while` loop and properly accept both range and „C“ style forms of loops e.g.:

- `for(x=1:2:100) {}`
- `for(x=1; x<= 100; x=x+2) {}`

However, only the former is depicted below.

CreateEmptyWhile The `CreateEmptyWhile` simply creates an empty `ast.While` and pushes it on the stack as later actions expect a `ast.While` node present. Both loop styles are

[8] `https://optics.ansys.com/hc/en-us/articles/360034928493-for-Script-command`

accepted in the `loop_condition` nonterminal in Production 42 and 43 on page 34. The first type of loops we will tackle is the range style. This is handled by the `CreateRangeCondition`; there are two cases to consider: the range is only composed of two values (*start:stop*) or a step is added (*start:step:stop*). The `end` value is popped of the stack, an `ast.Assign` node is always present with the first part of the range, i.e. the `start` value, which is also popped. A `ast.Compare` node is created to serve as the terminating condition for the `ast.While` node. This involves comparing the value of the `ast.Assign` variable to the `end` value. An `ast.AugAssign` is also created; this serves as the incrementation that happens during each loop iteration with a step of 1 by default. The `test` attribute of `ast.While` is updated and the `ast.Assign` is pushed, followed by `ast.AugAssign` and `ast.While`.

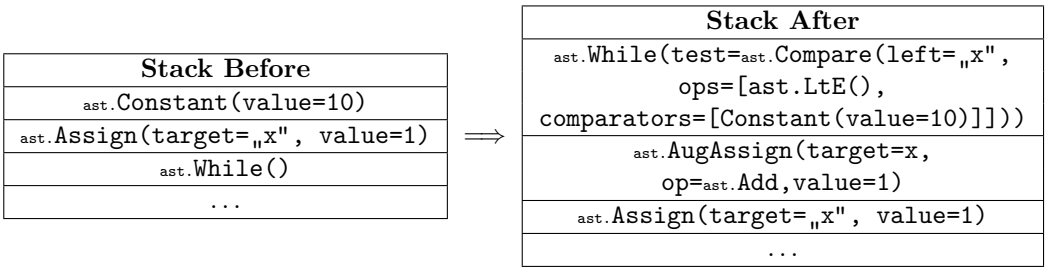


Figure 4.14: LIFO value stack before and after applying `CreateRangeCondition` action.

ExtendRangeCondition The `ExtendRangeCondition` is used when an increment value is parsed; this requires the need to update the previously generated while loop, as what was thought to be the end value is actually the `step` value and the end is now on top of the stack. First, the now correct end expression is popped of the stack, following that both the `ast.While` and `ast.AugAssign` are popped of the stack. The new `ast.AugAssign` value is the `comparator` of `ast.While` and the `comparator` value is the newly acquired `end` value. The `ast.AugAssign` followed by the `ast.While` node is pushed on the stack.

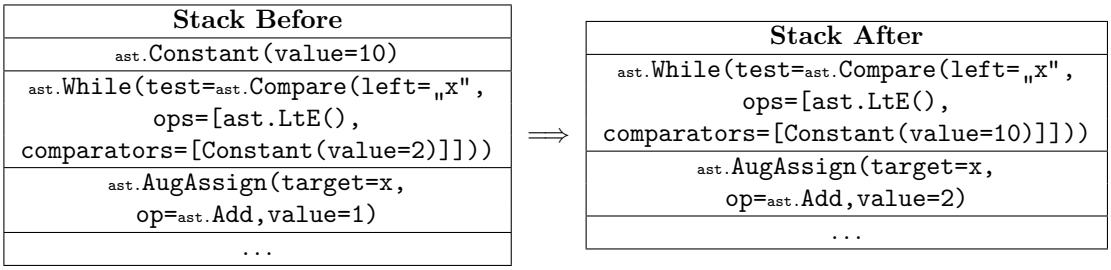


Figure 4.15: LIFO value stack before and after applying `ExtendRangeCondition` action.

The main loop generation is handled by `HandleAllLoops`, which, depending on the style of the loop, arranges the nodes on the stack and pushes them.

Imports The `Imports` handles generating import statements to load Meep and the `runtime` module, which will be introduced soon. This action pushes `ast.Import`, which maps Meep to the common alias `mp`. Following that, a `ast.ImportFrom` node is pushed, which imports the classes `Selector` and `Record` from the `runtime` module.

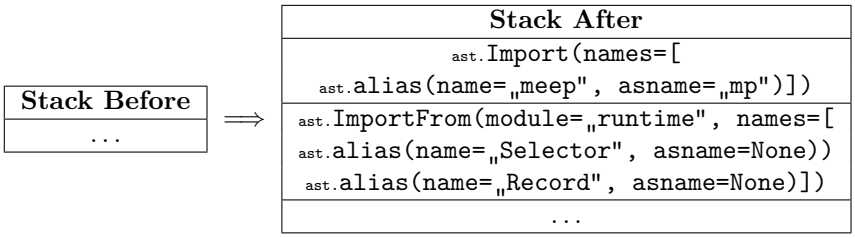


Figure 4.16: LIFO value stack before and after applying `Imports` action.

Selector When a new object is added to the scene, Lumerical assigns it a default name and automatically selects it. Commands that modify the objects will now interact only with the selected object. This cannot be fully predicted at compile-time, thus a `runtime` module is implemented, which contains a `Selector` object that imitates the Lumerical selection process. However, this has to be interpreted.

Each object is stored as a `Record`, which stores the `ast body` attribute and pairs it with the corresponding Lumerical object name and whether or not the object is selected.

This `Selector` class stores a list of `Record`s and implements the following functions:

- `add` — Unselects all previous `Record`s and adds the passed `Record` to the `Selector`.
- `setName` — Sets the passed string as the name for all selected `Record`s.
- `select` — Unselects all and then only selects the `Record`s whose names match the passed string.
- `shiftSelect` — Same as `select`, except it doesnt unselect beforehand.
- `selectAll` — Selects all `Record`s irespective of their name.
- `unselectAll` — Unselects all `Record`s irespective of their name.
- `getSelected` — Returns the `body` attribute of all currently selected `Record`s.

CreateSelector The `CreateSelector` simply creates a variable `selector` that stores an instance of the `Selector` class.

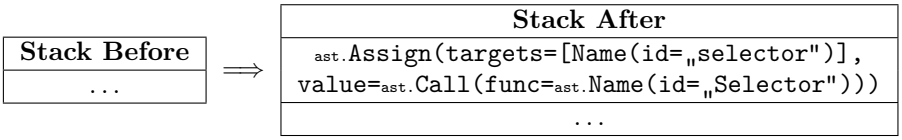


Figure 4.17: LIFO value stack before and after applying `CreateSelector` action.

AddToSelector `AddToSelector` is not a class, but a helper function; however, it is necessary to mention its purpose as it is used in multiple actions. The function takes a tuple of `Record` attributes, consisting of the name, body and the selection boolean. Then a `ast` corresponding to the following code is created and returned; `selector.add(Record(args))`, where `args` are the passed attributes.

AddFDTD After we have imported Meep and created a `Selector`, it is time to handle the conversion of the simulation domain. This is where `AddFDTD` comes in; the Lumerical FDTD solver region is converted to the Meep `Simulation` object. The default size values are passed as a Meep `Vector3` object. The constructed `Simulation` object is then wrapped by `Selector.add` and pushed on the stack.

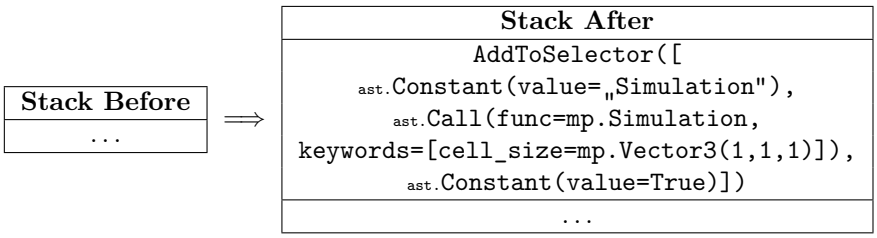


Figure 4.18: LIFO value stack before and after applying `AddFDTD` action.

AddRect The `AddRect` action generates the necessary AST nodes to create a Meep rectangular block with default dimensions and adds it to the `Selector` using the Record. Internally, it constructs an `AddToSelector` node. Once constructed, this node is pushed onto the value stack.

SelectAll, UnselectAll, Select, ShiftSelect The `SelectAll, UnselectAll, Select, ShiftSelect` actions correspond to the functions of the `Selector` class. An example of the `Select` action is provided.

The `Select` action is used to generate an AST node that calls the `select` method on the `selector` variable. It consumes a single `ast.Constant` from the top of the stack, which is the name of the record to be selected.

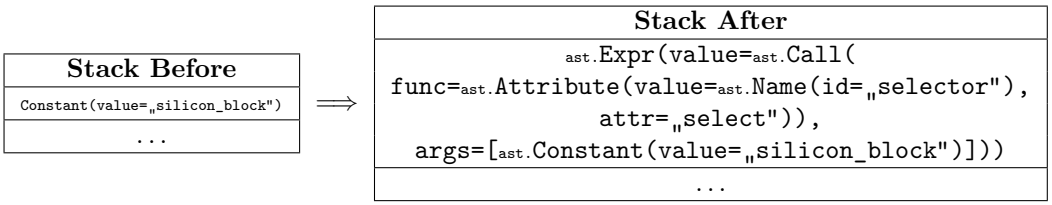


Figure 4.19: LIFO value stack before and after applying `Select` action.

SetProperty The `SetProperty` action is a complex action that handles changing the properties of all types of objects. However, the program flow is similar for the majority of the objects. Thus, we will only show the `SetProperty` action for changing the position. First, the action pops two values from the stack: the object property and the new value. Following that, a reference to the `selector` variable is created. Two helper functions are defined, `create_loop_body` and `create_getSelected_loop`. The former creates a loop body that sets the property of each selected object to the desired value, and the latter creates a loop that iterates over all selected objects.

Finally, a match statement is used to determine the type of the property being set. In this case, the object center. The specified value is passed with the attribute to `create_loop_body`, which returns a `ast.While` node. The `create_getSelected_loop` is called with the `ast.While` node and the result are pushed on the stack.

Chapter 5

Evaluation of Results

Thus far, the thesis has shown the theoretical and practical aspects of the transpilation process while dealing with the key intricacies of Lumerical scripting language and Meep. The following part of this thesis moves on to describe in greater detail the transpilation process applied to the examples and compares the Lumerical and Meep simulations to the analytically obtained results.

5.1 Transpilation

This section will show the transpilation process in action on short examples. If the code encounters an unsupported command, the transpilation halts.

5.1.1 General language constructs

```
1 x = (-1 * (2 - 3)) / 2;
```

(a) Lumerical scripting language

```
1 x = -1 * (2 - 3) / 2
```

(b) Transpiled Python code

Figure 5.1: Translation of simple numerical expression.

```
1 x= 1<=y <= 5;
```

(a) Lumerical scripting language

```
1 x = 1 <= y <= 5
```

(b) Transpiled Python code

Figure 5.2: Translation of simple comparison expression.

```
if(x < 5){
  y = 2;
}
else if (x > 3 and x < 10){
  y = 1;
  if(x > 5){
    ?x;
  }
}
else if (x > 30){
  y = 10;
}
else{
  y = 0;
}
```

(a) Lumerical scripting language

```
if x < 5:
  y = 2
elif x > 3 and x < 10:
  y = 1
  if x > 5:
    print(x)
elif x > 30:
  y = 10
else:
  y = 0
```

(b) Transpiled Python code

Figure 5.3: Translation of nested conditional statements.

```
for(x=1:2:10){
  ?x;
  if(x==5){
    break;
  }
}
```

(a) Lumerical scripting language

```
x = 1
while x <= 10:
  print(x)
  if x == 5:
    break
  x += 2
```

(b) Transpiled Python code

Figure 5.4: Translation of a for loop.

As shown in the previous examples, LUMEX handles the transpilation of general Lumerical scripting language constructs to Python. The conversion results in a 1:1 mapping with similar length and structure.

5.1.2 FDTD language constructs

```
addfdtd;
addrect;
```

(a) Lumerical scripting language

```
import meep as mp
from runtime import Selector, Record
selector = Selector()
selector.add(Record('Simulation',
    mp.Simulation(cell_size=mp.Vector3(1, 1,
    1)), True))
selector.add(Record('Rectangle',
    mp.Block(size=mp.Vector3(1, 1, 1)),
    True))
```

(b) Transpiled Python code

Figure 5.5: Translation of a solver region with a single block.

```
addfdtd;
addrect;
addrect;
set("name", "block");
set("x", 5);
set("x span", 7);
selectall;
set("z span", 11);
```

(a) Lumerical scripting language

```
import meep as mp
from runtime import Selector, Record
selector = Selector()
selector.add(Record('Simulation',
    mp.Simulation(cell_size=mp.Vector3(1, 1,
    1)), True))
selector.add(Record('Rectangle',
    mp.Block(size=mp.Vector3(1, 1, 1)), True))
selector.add(Record('Rectangle',
    mp.Block(size=mp.Vector3(1, 1, 1)), True))
for record in selector.getSelected():
  record.name = 'block'
for record in selector.getSelected():
  record.center = mp.Vector3(5, record.center.y,
      record.center.z)
for record in selector.getSelected():
  if record.record_type == 'Simulation':
    record.cell_size = mp.Vector3(7,
        record.cell_size.y, record.cell_size.z)
  else:
    record.size = mp.Vector3(7, record.size.y,
        record.size.z)
selector.selectAll()
for record in selector.getSelected():
  if record.record_type == 'Simulation':
    record.cell_size =
        mp.Vector3(record.cell_size.x,
        record.cell_size.y, 11)
  else:
    record.size = mp.Vector3(record.size.x,
        record.size.y, 11)
```

(b) Transpiled Python code

Figure 5.6: Translation of a solver region with a single block.

The previous examples showcase the ability of LUMEX to transpile Lumerical specific commands to Meep. However, due to the inconsistency in referencing objects between the two languages, the transpiled code is longer and more complex. Nonetheless, the transpiled code is functionally equivalent to the original Lumerical code.

5.2 Comparison of Lumerical and Meep

The comparison of accuracy and performance of the two FDTD tools is an integral part of this thesis. For this task, the following approach was chosen: for both tests, an identical simulation in each tool was created, each with a plane wave source, a frequency domain monitor and a dielectric block with a constant ε. The only difference was that one test contained a single interface, while the other contained a thin film like interface, see fig. 5.7. Lumerical code was first transpiled to Meep code, followed by the commands unsupported by LUMEX, which were added by hand to ensure parity.

While Lumerical offers more advanced meshing options, a simple uniform rectangular mesh was chosen to keep parity with the Meep simulation. The same resolution, i.e. grid size, was chosen for both simulations, the simulations were set up to complete the same number of steps (achieved by setting the `dt stability factor` until dt matched dt in Meep with a Courant facotr of 0.5), and the frequency range and number of sampling points was chosen. Furthermore, both simulations were forced to compute complex fields. The following results do not rule out the influence of other factors during simulations. Furthermore, it is important to bear in mind that Lumerical may be able to achieve better results with different settings. However, the goal of this comparison is the evaluation of the implementations of the underlying FDTD algorithm.

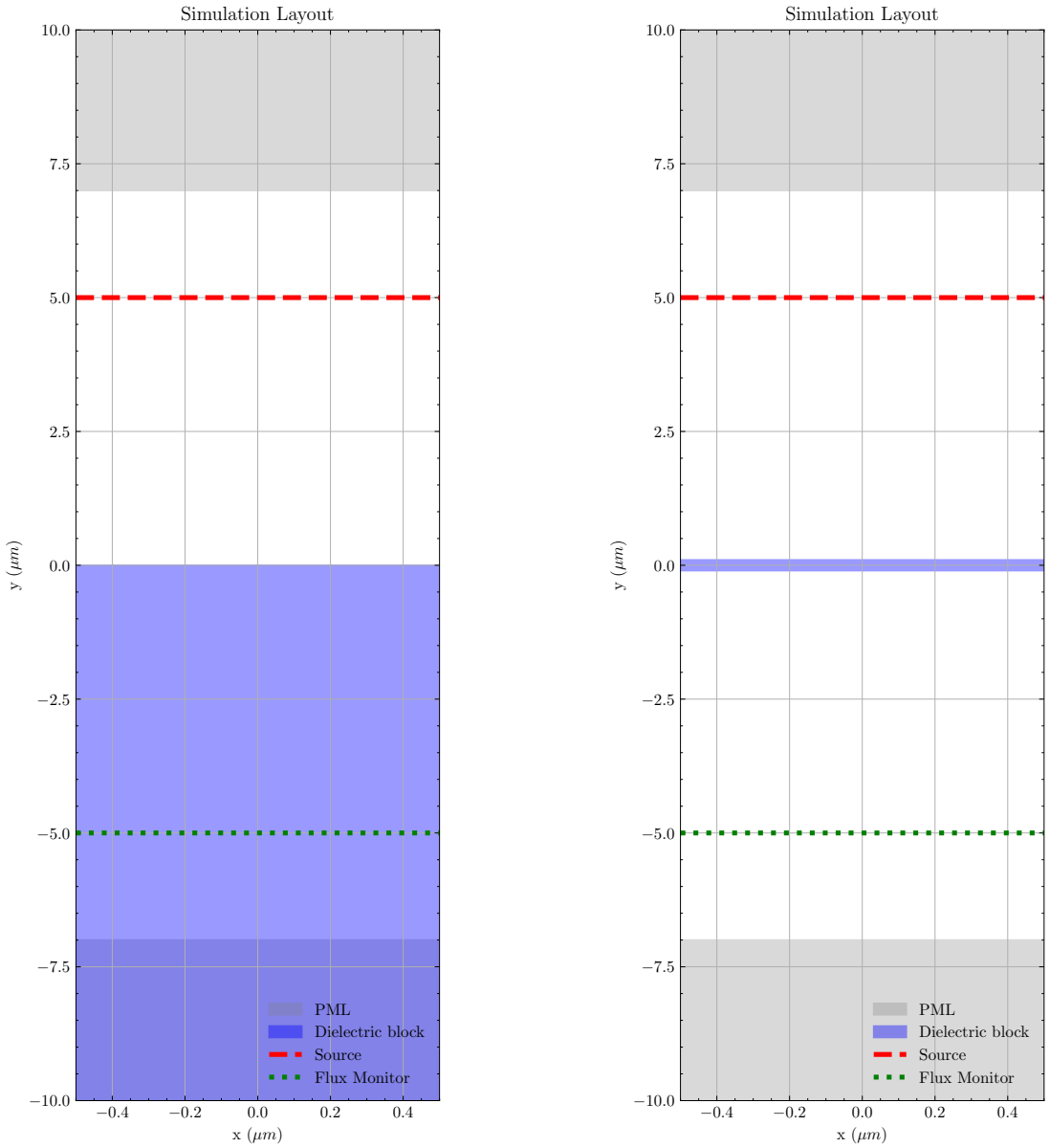
Due to the innate nature of hardware, absolute timings are of little use, and relative timings are presented. All simulations were run using MPI version 4.1 on the same x86_64 host machine with a Intel© Xeon© Gold 6354 CPU clocked at 3.0 GHz.

All the following simulations were run with a monochromatic plane wave in a non-absorbing homogeneous non-isotropic non-magnetic medium with a wavelength range of 0.4 to 0.8 micrometers.

The analytical solution was obtained using the method described in section 3.2.4.

5.2.1 Single Interface

A simulation size of $1 \times 20 \times 0.1$ (x,y,z respectively) was chosen, with a resolution of 50 points per 1 unit of distance. An incident plane wave source was placed at $y = 5$, a monitor at $y = -5$. A linear homogeneous dielectric medium spanned from $y = 0$ to $y = -10$ with a permitivity of $n = 1.4$. PML layers were placed on either end. This layout is shown in fig. 5.7a. The simulations were run for 20000 steps, with a time step of 0.01 and a Courant factor of 0.5, and an end time of 200. Following that, two more simulations were run at a resolution of 100 and 25. The simulations were run on a single core. The results are shown in fig. 5.8.



(a) XY view of the single interface layout.

(b) XY view of the double interface layout.

Figure 5.7: Comparison of simulation layouts for (a) single-interface and (b) double-interface configurations.

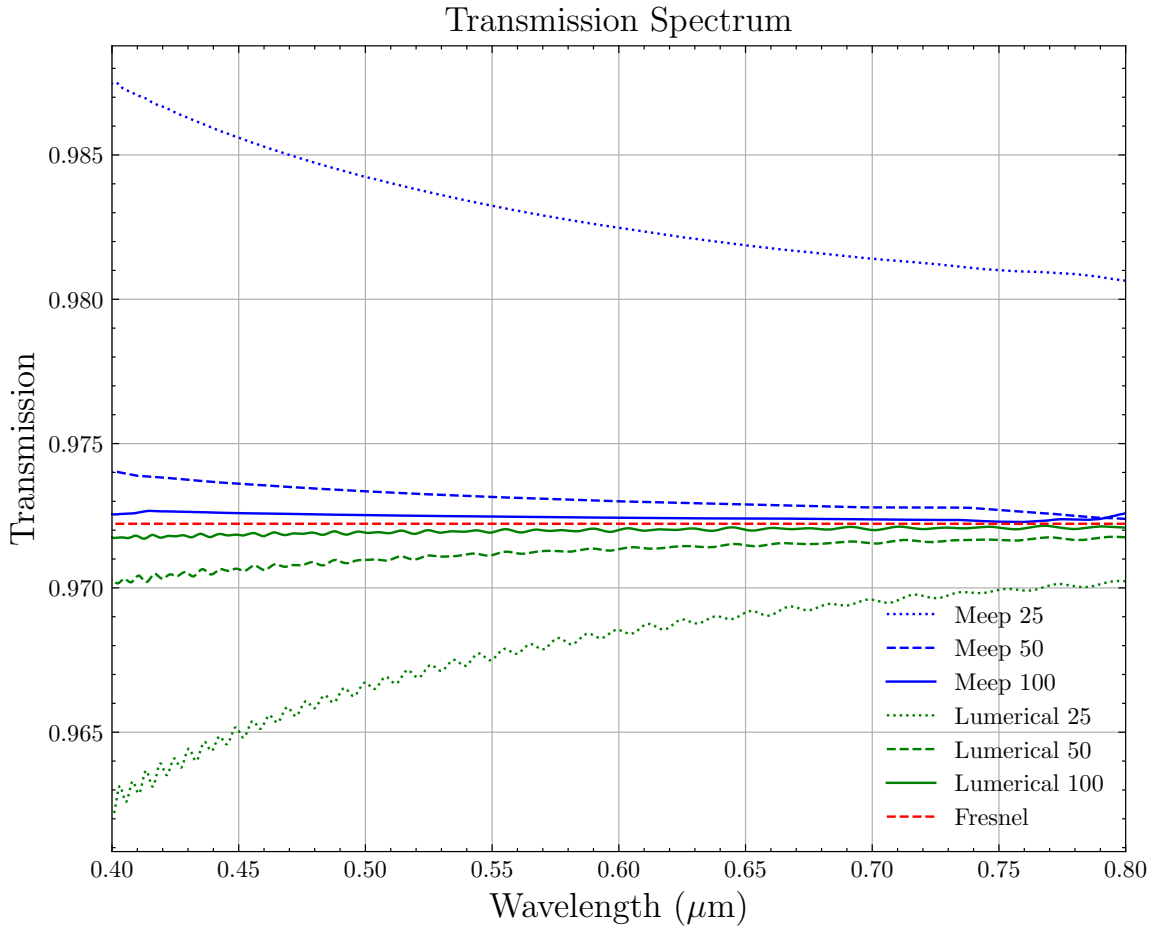


Figure 5.8: Transmission spectrum of the single interface simulation.

A convergence of the results with the analytical solution may be observed. With an increase in resolution, the results are more accurate. However, on average, Lumerical is 1.37 times faster at 25 resolution and up to 3.03 and 4.42 at 50 and 100, respectively. This discrepancy in speed could be attributed to the fact that Meep needs to run two simulations to obtain the normalization.

5.2.2 Thin Film

A simulation size of $1 \times 20 \times 1$ (x, y, z respectively) was chosen, with a resolution of 25, 50 and 100 points per 1 unit of distance per test. An incident plane wave source was placed at $y = 5$ and a monitor at $y = -5$. A linear homogeneous dielectric medium with a permittivity of $n = 3$ and a thickness of 0.2 was placed at $y = 0$. The PML layers were placed on either end and periodic boundaries were chosen for the remaining BCs, which may be observed in fig. 5.7b. The simulations were run for 20000 steps, with a time step of 0.01 and a Courant factor of 0.5, and an end time of 200. The simulations were run on 8 cores.

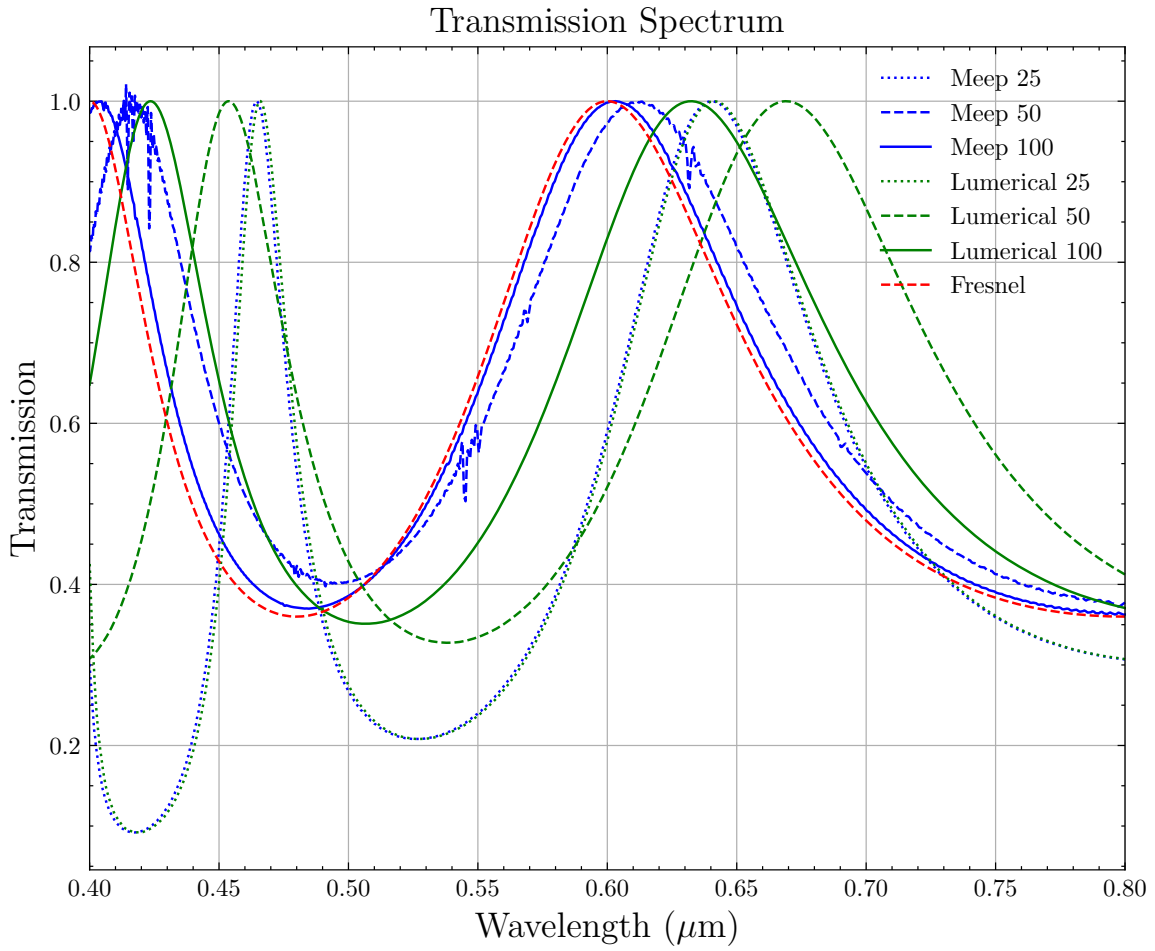


Figure 5.9: Transmission spectrum of the single interface simulation.

In the above test, Lumerical performed 2.5 times faster than Meep at a resolution of 25, 5.2times at 50 and 4.7 times faster at 100. However, it may be noted that Meep was able to achieve a higher accuracy at the same resolution. Thus, when running Lumerical at a higher resolutionc to produce comparable results to Meep, the time difference may not prove to be as significant.

5.2.3 Paralalization Capabilities

Both Meep and Lumerical support parallelization. Meep requires the use of Parallel Meep[1], which utilizes MPI, while Lumerical provides parallelization out of the box. In the following tests, Parallel Meep utilized MPI version 4.1 and Lumerical version 2022 R1.4 with MPICH2[2]. The tests were run on a single x86_64 host machine with a Intel© Xeon© Gold 6354 CPU clocked at 3.0 GHz.

[1] `https://meep.readthedocs.io/en/latest/Parallel_Meep/`

[2] Note: as of 2024 R1, Lumerical no longer supports MPICH2, but rather OpenMPI. `https://optics.ansys.com/hc/en-us/articles/20741668696467-Running-simulations-with-MPI-on-Linux`

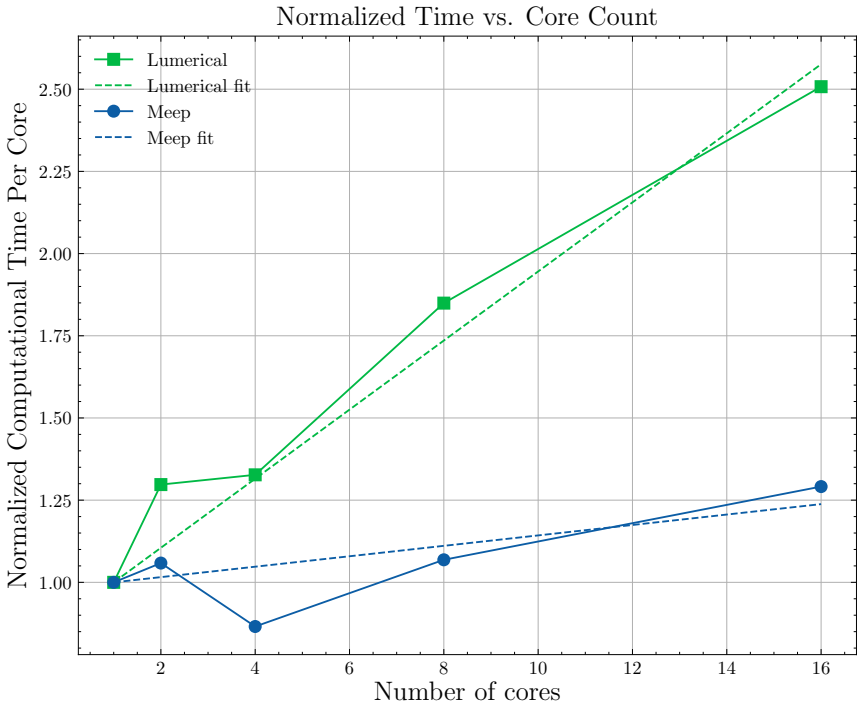


Figure 5.10: Comparison of Lumerical and Meep parallelization capabilities.

It may be observed that with the increase in the number of cores, the overhead is less in Meep than in Lumerical. This may be attributed to the above-mentioned fact that Lumerical utilizes MPICH2 compared to OpenMPI. However, anomalous outliers are present the dataset, for Meep at 4 cores. One valid explanation for this could be the fact that other processes may have been running on the host machine at the time of the test, this is a known issue in the case of Meep[3].

[3] `https://github.com/NanoComp/meep/issues/882`

Chapter 6

Conclusion

This thesis explored the feasibility of automating the translation of simulation scripts between two FDTD tools by studying their structures, languages and underlying computational principles as well as comparing their performance and precision. These goals were successfully achieved.

The thesis began by exploring the underlying FDTD principles in section 3.1 and provided a concise overview of optics in section 3.2. This is followed by a detailed description of Lumerical and Meep in section 3.4 and section 3.3. Following that, the thesis presented an introduction of formal language theory in section 3.5, which built the foundational knowledge for the discussion about the construction of a compiler in section 3.6.

The transpiler implementation followed in chapter 4, which describes the creation of the grammar, lexer, parser and code generator. chapter 5 presented the evaluation of LUMEX, as well as the comparison of Lumerical and Meep in terms of performance and accuracy.

LUMEX is able to transpile Lumerical scripts to Meep. However, the transpiled code is often longer and more complex. This is mainly attributed to the difference in the way objects are referenced in Lumerical and Meep. Other differences between the tools, such as that Meep needs to run two simulations to obtain the normalization make transpilation more complex. Thus, for full simulations, a moderate manual intervention is required. With future work based on this thesis, the LUMEX could be improved to handle more commands and provide error recovery.

Future work could focus on the following areas:

- Expanding command support — The current implementation of LUMEX handles only a small subset of Lumerical commands, requiring the addition of further commands.
- Implementing error recovery — Currently, LUMEX halts transpilation when encountering unsupported commands. Introducing error recovery would allow it to continue while warning the user.
- Adding abstractions — The `runtime` module is currently limited to the `Selector` class. Expanding this with more classes would make the target code more readable.
- Testing — The current tests are limited to a few examples. A more extensive test suite would help ensure the correctness of the transpiler.
- Optimization pass — For cases that are resolvable at compile time, an optimization pass could be added to reduce the complexity of the generated code.

Bibliography

[1] Davidson, D. B. *Computational Electromagnetics for RF and Microwave Engineering* Print and online. 2nd ed. Cambridge, UK: Cambridge University Press, October 2010, Extensively revised. ISBN 9781139492812. Available at: https://doi.org/10.1017/CBO9780511778117. [cit. 2025-05-12].

[2] Taylor, R. A Crash Course using Meep. *Humaticlabs.com: Recreating the Double Slit Experiment with Python and Meep* online. 30. june 2023. ISSN N/A. Available at: https://humaticlabs.com/blog/meep-double-slit/. [cit. 2025-05-08].

[3] Dirac, P. A. M. Quantised Singularities in the Electromagnetic Field. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*. Royal Society, 1931, vol. 133, no. 821, p. 60–72.

[4] Staelin, D. H.; Morgenthaler, A. and Kong, J. A. *Electromagnetics and Applications* online. 1st ed. Cambridge, MA: MIT OpenCourseWare, 2009. Available at: https://phys.libretexts.org/Bookshelves/Electricity_and_Magnetism/Electromagnetics_and_Applications_(Staelin). [cit. !yyyy-mm-dd].

[5] Sullivan, D. M. *Electromagnetic Simulation Using the FDTD Method* Print. 2nd ed. Piscataway, NJ: IEEE Press, May 2013. IEEE Press Series on RF and Microwave Technology, no. 2. ISBN 9781118646632. Available at: https://books.google.com/books/about/Electromagnetic_Simulation_Using_the_FDT.html?id=V39SkmdJQ78C. [cit. 2025-05-12].

[6] Berenger, J.-P. A perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics*, 1994, vol. 114, no. 2, p. 185–200. ISSN 0021-9991. Available at: https://www.sciencedirect.com/science/article/pii/S0021999184711594.

[7] Teixeira, F. L.; Sarris, C.; Zhang, Y.; Na, D.-Y.; Berenger, J.-P. et al. Finite-difference time-domain methods. *Nature Reviews Methods Primers*, 2023, vol. 3, no. 1, p. 75. ISSN 2662-8449. Available at: https://doi.org/10.1038/s43586-023-00257-4.

[8] Team, M. D. What is a good rule of thumb for the PML thickness? *Meep.readthedocs.io: Perfectly Matched Layer* online. 05. may 2025. Available at: https://meep.readthedocs.io/en/master/Perfectly_Matched_Layer/#what-is-a-good-rule-of-thumb-for-the-pml-thickness. [cit. 9. May 2025].

[9] Čermák, M. *Teaching Material for the Electricity and Magnetism (EM) Exercise Course* online. Teaching Material MUNI/FR/0042/2014. Illustrations by Jana

Jurmanová. Brno, Czech Republic: Masaryk University, Faculty of Science, march 2014. Available at: https://is.muni.cz/el/sci/jaro2015/F2050/um/skriptaIS.pdf. Development Fund of MU, project period: 3/2014 – 12/2014.

[10] Linz, P. *An Introduction to Formal Languages and Automata* Print and online. 6th ed. Burlington, MA, USA: Jones & Bartlett Learning, January 2016. ISBN 9781284077247. Available at: https://books.google.com/books?id=pDaUCwAAQBAJ. [cit. 2024-12-16].

[11] Salomaa, A. *Formal Languages* online. 1st ed. New York, NY: Academic Press, August 1987. 322 p. Computer Science Classics, no. 1. ISBN 0-12-615750-2. Available at: https://openlibrary.org/books/OL2373365M/Formal_languages. [cit. 2024-12-16].

[12] Chomsky, N. Three Models for the Description of Language. *IRE Transactions on Information Theory* Print and online. 1st ed. United States: Institute of Electrical and Electronics Engineers (IEEE), September 1956, vol. 2, no. 3, p. 113–124. ISSN 0018-9448. Available at: https://doi.org/10.1109/TIT.1956.1056813. [cit. 2024-12-16].

[13] Aho, A. V.; Lam, M. S.; Sethi, R. and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. 2nd ed. USA: Addison-Wesley Longman Publishing Co., Inc., August 2006. Addison-Wesley Professional Computing Series, no. 2. ISBN 0-321-48681-1.

[14] Ansys Optics. *Lumerical Scripting Language* online. Technical Documentation. Ansys Optics, 2025. Available at: https://developer.ansys.com/docs/lumerical/scripting-language. [cit. 2025-03-30].

[15] Sphinx Documentation Team. *Quickstart — Sphinx Documentation* online. Software Documentation. Sphinx Documentation Team, 2025. Available at: https://www.sphinx-doc.org/en/master/usage/quickstart.html. [cit. 2025-03-30].

[16] Docutils Project. *ReStructuredText Markup Specification* online. Technical Specification. Docutils Project, 2025. Available at: https://docutils.sourceforge.io/rst.html. [cit. 2025-03-30].

[17] Wang, D. C.; Appel, A. W.; Korn, J. L. and Serra, C. S. The Zephyr Abstract Syntax Description Language. In: *Proceedings of the Conference on Domain-Specific Languages (DSL '97)* Print and online. 1st ed. Santa Barbara, California, USA: USENIX Association, October 1997, p. 17–17. DSL '97, no. 1. Available at: http://www.usenix.org/publications/library/proceedings/dsl97/full_papers/wang/wang.pdf. [cit. 2025-05-12].

[18] Python Software Foundation. *Full Python Grammar Specification* online. Technical Specification. Python Software Foundation, 2025. Available at: https://docs.python.org/3/reference/grammar.html. [cit. 2025-03-31].

[19] Python Software Foundation. *Python ast — Abstract Syntax Trees* online. Library Documentation. Python Software Foundation, 2025. Available at: https://docs.python.org/3/library/ast.html. [cit. 2025-03-31].

[20] Ansys Optics. *Lumerical Scripting Language Commands* online. Technical Documentation. Ansys Optics, 2025. Available at:

https://optics.ansys.com/hc/en-us/sections/360005285973-Commands. [cit. 2025-04-05].

[21] Wirth, N. and Weber, H. EULER: A generalization of ALGOL and its formal definition: Part 1. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery, january 1966, vol. 9, no. 1, p. 13–25. ISSN 0001-0782. Available at: https://doi.org/10.1145/365153.365162.

[22] Holub, A. I. *Compiler Design in C*. 1st ed. Englewood Cliffs, NJ: Prentice Hall, January 1990. Prentice-Hall software series, no. 1. ISBN 9780131550452.

Appendix A

Full Implemented Lumerical Grammar

Table A.1: Complete Grammar with Actions

1.	`root`	→	*actions.Imports() actions.StoreToBody()* *actions.CreateSelector() actions.StoreToBody()* `body`
2.	`body`	→	`statement` *actions.StoreToBody()* `body`
3.		|	`function body` **EOF**
4.		|	ε
5.		|	**EOF**
6.	`function`	→	**FUNCTION IDENTIFIER** **(** `parameter_list` **)** **{** `nested_body` **}**
7.	`nested_body`	→	`statement` *actions.StoreToBody()* `nested_body`
8.		|	ε
9.	`nested_else_body`	→	`statement` *actions.StoreToElse()* `nested_else_body`
10.		|	ε
11.	`statement`	→	**IDENTIFIER** *actions.StoreVariableName()* `identifier_action`
12.		|	`control_structure`
13.		|	`selection`
14.		|	**?** `expression` *actions.Print()* **;**
15.		|	**ADDFDTD** *actions.AddFDTD()* **;**
16.		|	**ADDRECT** *actions.AddRect()* **;**
17.		|	**ADDSPHERE** *actions.AddSphere()* **;**
18.		|	**ADDPLANE** *actions.AddPlaneSource()* **;**
19.		|	**ADDDFTMONITOR** *actions.AddDFTMonitor()* **;**

Continued on next page

Table A.1: Complete Grammar with Actions (Continued)

20.		|	**SET (STRING** *actions.StoreLiteral(„str“)* `expression` *actions.SetProperty()* **) ;**
21.		|	**BREAK** *actions.Break()* **;**
22.	`selection`	→	**SELECTALL** *actions.SelectAll()* **;**
23.		|	**UNSELECTALL** *actions.UnselectAll()* **;**
24.		|	**SELECT (STRING** *actions.StoreLiteral(„str“)* *actions.Select()* **) ;**
25.		|	**SHIFTSELECT (STRING** *actions.StoreLiteral(„str“)* *actions.ShiftSelect()* **) ;**
26.	`identifier_action`	→	`assignment`
27.		|	`function_call`
28.	`assignment`	→	**=** `expression` *actions.AssignToVariable()* **;**
29.	`function_call`	→	**(** `parameter_list` **) ;**
30.	`parameter_list`	→	**IDENTIFIER** `parameter_list_prime`
31.		|	ε
32.	`parameter_list_prime`	→	**,** **IDENTIFIER** `parameter_list_prime`
33.		|	ε
34.	`argument_list`	→	`expression` `argument_list_prime`
35.		|	ε
36.	`argument_list_prime`	→	**,** `expression` `argument_list_prime`
37.		|	ε
38.	`control_structure`	→	**IF (** `expression` **) {** *actions.If()* `nested_body` **}** `elseNT`
39.		|	**FOR** *actions.CreateEmptyWhile()* **(** **IDENTIFIER** *actions.StoreVariableName()* **=** `expression` *actions.AssignToVariable()* `loop_condition` **)** **{** `nested_body` *actions.HandleAllLoops()* **}**
40.	`loop_condition`	→	**:** `expression` *actions.CreateRangeCondition()* `range_step`
41.		|	**;** `expression` **;** `expression` *actions.CreateWhileCondition()*
42.	`range_step`	→	**:** `expression` *actions.ExtendRangeCondition()*
43.		|	ε
44.	`elseNT`	→	**ELSE** `elifNT` **{** *actions.HandleElse()* `nested_else_body` *actions.CleanUpElse()* **}** `elseNT`
45.		|	ε
46.	`elifNT`	→	**IF (** `expression` **)**
47.		|	ε

Continued on next page

Table A.1: Complete Grammar with Actions (Continued)

48.	`expression`	→	`logic_and`
49.	`logic_and`	→	`logic_or logic_and_prime`
50.	`logic_and_prime`	→	**AND** `equality` *actions.LogicOperation(ast.And())* `logic_and_prime`
51.		|	ε
52.	`logic_or`	→	`equality logic_or_prime`
53.	`logic_or_prime`	→	**OR** `equality` *actions.LogicOperation(ast.Or())* `logic_or_prime`
54.		|	ε
55.	`equality`	→	`comparison equality_prime`
56.	`equality_prime`	→	**!=** `comparison` *actions.Comparison(ast.NotEq())* `equality_prime`
57.		|	== `comparison` *actions.Comparison(ast.Eq())* `equality_prime`
58.		|	ε
59.	`comparison`	→	`term comparison_prime`
60.	`comparison_prime`	→	> `term` *actions.Comparison(ast.Gt())* `comparison_prime`
61.		|	>= `term` *actions.Comparison(ast.GtE())* `comparison_prime`
62.		|	< `term` *actions.Comparison(ast.Lt())* `comparison_prime`
63.		|	<= `term` *actions.Comparison(ast.LtE())* `comparison_prime`
64.		|	ε
65.	`term`	→	`factor term_prime`
66.	`term_prime`	→	- `factor` *actions.BinaryOperation(ast.Sub())* `term_prime`
67.		|	+ `factor` *actions.BinaryOperation(ast.Add())* `term_prime`
68.		|	ε
69.	`factor`	→	`unary factor_prime`
70.	`factor_prime`	→	/ `unary` *actions.BinaryOperation(ast.Div())* `factor_prime`
71.		|	* `unary` *actions.BinaryOperation(ast.Mult())*

Continued on next page

Table A.1: Complete Grammar with Actions (Continued)

			`factor_prime`
72.		|	ε
73.	`unary`	→	**NOT** `unary`
74.		|	**-** `unary` *actions.UnarySubtract()*
75.		|	`primary`
76.	`primary`	→	**INTEGER** *actions.StoreLiteral(„int“)*
77.		|	**FLOAT** *actions.StoreLiteral(„float“)*
78.		|	**STRING** *actions.StoreLiteral(„str“)*
79.		|	**IDENTIFIER** *actions.StoreVariableName()*
80.		|	**(** `expression` **)**