SHORT TITLE

YOUR THESIS TITLE, WHICH CAN BE AS LONG AS YOU WANT ON THE TITLE PAGE

BY

JANE DOE, B.Eng.

a Report
submitted to the Department You Belong To
and the School of Graduate Studies
of McMaster University
in partial fulfilment of the requirements
for the degree of
Masters of Engineering

Masters of Engineering (YYYY) McMaster University
(Department You Belong To) Hamilton, Ontario, Canada

TITLE:	Your Thesis Title, Which Can Be As Long As You Want On the Title Page
AUTHOR:	Jane Doe B.Eng. (Software Engineering & Game Design), McMaster University, Hamilton, Canada
SUPERVISOR:	Your Supervisor
NUMBER OF PAGES:	xvi, 56

Lay Abstract

A lay abstract of not more 150 words must be included explaining the key goals and contributions of the thesis in lay terms that is accessible to the general public.

Abstract

Scientific Computing (SC) involves analyzing and simulating complex scientific and engineering problems using computing techniques and tools. To improve the understandability, maintainability, and reproducibility of SC software, documentation should be an integral part of the development process. Jupyter Notebook is a popular tool for developing SC software documentation, and is also used to enhance teaching and learning efficiency in engineering education due to its flexibility and benefits. Despite the importance of documentation, it is often missing or poorly executed in SC software for several reasons, including time-consuming.

Drasil is a framework that aims to improve the efficiency of documentation development. By encoding each piece of information of scientific problems once and generating the document automatically, Drasil saves time in the documentation development process. We are interested in generating Jupyter Notebooks in Drasil to expand its applications, including generating educational documents, and extend the values of Jupyter Notebooks to Drasil.

To achieve this, we implement a JSON printer capable of generating Drasil software artifacts, such as SRS, in notebook format. This enables us to generate Jupyter Notebooks in Drasil, and generate educational documents, starting with lesson plans. We develop the structure of our lesson plans and designed the language of lesson plans in Drasil. Additionally, Jupyter Notebooks seamlessly integrate different content types with code, making them ideal for data research. We discuss two approaches and the implementations to combine text and code in our Drasil-generated Jupyter Notebooks.

This paper achieves our main objectives of generating Jupyter Notebooks that integrate both text and code, as well as generating lesson plans using Drasil.

Your Dedication

Optional second line

Acknowledgements

I am deeply grateful to my supervisors, Dr. Spencer Smith and Dr. Jacques Carette, for their support and guidance throughout this project. Their expertise and insights have been invaluable in shaping my research, and I could not have completed this work without their guidance. Their constructive feedback, encouragement, and patience have been truly appreciated, and I feel fortunate to have had the opportunity to work with them.

I would also like to express my gratitude to my colleagues, Jason Balaci, Sam Crawford, and Don Chen, for their willingness to share their expertise and knowledge. Their dedication and talent have inspired and motivated me, and I am grateful to have worked alongside such amazing colleagues.

Lastly, I would like to thank my parents for their unconditional love and unwavering support throughout my academic journey. Their encouragement have given me the confidence to pursue my dreams, and I am forever grateful for their belief in me.

Contents

Lay Abstract **iii**

Abstract **iv**

Acknowledgements **vii**

Notation, Definitions, and Abbreviations **xiv**

Declaration of Academic Achievement **xvi**

1 Introduction **1**

1.1 Background 3

1.2 Problem Statement 6

1.3 Thesis Outline 7

2 Drasil Printer **9**

2.1 How documents are printed in Drasil 10

2.2 Notebook Printer 16

3 Lesson Plans **22**

3.1 Language of Lesson Plans 23

3.2 A Case Study: Projectile Motion 29

4 Code Block Generation 34

4.1 Unit of Contents 35

4.2 Code Block 40

5 Conclusion 45

5.1 Future Work 45

5.2 Conclusion 47

A Your Appendix 50

List of Figures

1.1 Example of a Jupyter Notebook 5

3.2 Review Chapter Created Manually 33

3.3 Review Chapter Generated using Drasil 33

4.4 Snapshot of Example Chapter Generated using Drasil 43

4.5 Snapshot of Example Chapter Created Manually 43

A.6 drasil-printer Dependency Graph 51

List of Tables

2.1 Summary of Packages and Modules in **drasil-printers** 10

3.2 Summary of Notebook Modules 29

List of Codes

2.1 Source Code for Definition of a Printable Document 11

2.2 Source Code for Definition of RawContent 11

2.3 Source Code for Definition of LayoutObj 12

2.4 Source Code for Definition of Spec 12

2.5 Source Code for Definition of Contents 13

2.6 Code for Encoding rectVel 13

2.7 Code for Converting rectVel to a Sentence 14

2.8 Source Code for Converting ModelExpr to Contents 14

2.9 Source Code for Rendering EqnBlock to LaTeX 15

2.10 Source Code for Rendering LayoutObjs into JSON 17

2.11 Source Code for Converting Contents into JSON 18

2.12 Source Code for Rendering a Markdown Table 18

2.13 Source Code for Making Metadata 20

2.14 Source Code for markdownCell 21

2.15 Source Code for Calling markdownCell 21

3.1 Pseudocode for Definition of Document 23

3.2 Source Code for Notebook Core Language 25

3.3 Source Code for LsnDecl 26
3.4 Source Code for Section and the Constructor 26
3.5 Source Code for Chapter Constructors 27
3.6 Source Code for Making Chapters 27
3.7 Source Code for mkNb 28
3.8 Source Code for scalarPos 30
3.9 Source Code for lcrectPos 31
3.10 Source Code for Encoded Review Chapter 32
3.11 Source Code for Forming a Notebook 33

4.1 Source code for Definition of Introduction 36
4.2 Pseudocode for Definition of DocSection 37
4.3 Source Code for printLO' 39
4.4 Source Code for the New Definition of RawContent 40
4.5 Source Code for Rendering CodeBlock to LayoutObj 41
4.6 Source Code for the New Definition of LayoutObj 41
4.7 Source Code for Rendering CodeBlock to LayoutObj 42
4.8 Source Code for Generating a CodeBlock 42
4.9 Source Code for Rendering CodeBlock into JSON 43
4.10 Source Code for Encoding Example Chapter 44

5.1 Source Code for horiz_velo 47

A.1 JSON Code of A Notebook Document 52

Notation, Definitions, and Abbreviations

Notation

a^c	constant acceleration
t	time
v	speed
v^i	initial speed

Definitions

Challenge	With respect to video games, a challenge is a set of goals presented to the player that they are tasks with completing; challenges can test a variety of player skills, including accuracy, logical reasoning, and creative problem solving

Abbreviations

CSS	Cascading Style Sheets
GOOL	Generic Object-Oriented Language
HTML	HyperText Markup Language
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
PDF	Portable Document Format
SC	Scientific Computing
SRS	Software Requirement Specifications

Declaration of Academic Achievement

The student will declare his/her research contribution and, as appropriate, those of colleagues or other contributors to the contents of the thesis.

Chapter 1

Introduction

Scientific Computing (SC) is at the intersection of computer science, mathematics, and science. SC analyses and simulates mathematical methods of complex scientific and engineering problems by using computing techniques and tools. To improve understandability, maintainability, and reproducibility, writing documentation should be part of the process of developing scientific software. The role of documentation is to help people better understand the software and to "communicate information to its audience and instill knowledge of the system it describes" [1]. The significance of software documentation has been presented in many papers by previous researchers. High-quality documentation serves as the foundation for effective communication within software development teams, while also contributing to the overall excellence of the software product [2–4]. Additionally, Smith et al. [5, 6] shows that developing SC software in a document-driven methodology potentially improves the quality of the software.

Jupyter Notebook is a popular approach for documenting SC software, providing a system for creating and sharing data science and scientific computing documentation and code. This nonprofit, open-source application was born in 2014. Jupyter Notebook provides interactive computing across multiple programming languages, such as Python, Javascript, Matlab, and R. A Jupyter Notebook integrates text, live code, equations, computational outputs, visualizations, and multimedia resources, including images and videos. Jupyter Notebook is one of the most widely used interactive systems among scientists. Its popularity has grown from 200,000 to 2.5 million public Jupyter Notebooks on GitHub in three years from 2015 to 2018 [7]. It is used in a variety of areas and ways because of its flexibility and added values. For example, Notebooks can be used as an educational tool in engineering courses, enhancing teaching and learning efficiency [8, 9].

Even though the importance of documentation is widely recognized, it is often missing or poorly realized in SC software because: i) scientists are not aware of the why, how, and what of documentation [10, 11]; ii) it is time-consuming to produce [12]; iii) scientists generally believe that writing documentation demands more work and effort than they would likely yield in terms of the benefits of it [13].

Jupyter Notebook simplifies the process of maintaining SC documentation by enabling explanatory text and code to be combined in a single document. Furthermore, it provides easy sharing of notebooks on platforms like GitHub and exportation to different formats, such as PDF. However, there are also some downsides to employing it. While Jupyter Notebook streamlines documenting code, it can be more challenging to maintain and refactor the code itself, especially when dealing with large datasets or complex code. Debugging and refactoring code across multiple segments, for instance, can be time-consuming and difficult to test. Poor coding practices may lead to poor quality and reproducibility of Jupyter Notebooks [14, 15].

We are trying to increase the efficiency of documentation development by adopting generative programming. Generative programming is a technique that allows programmers to write the code or document at a higher abstraction level, and the generator produces the desired outputs. Drasil is an application of generative programming, and it is the framework we use to conduct this research. Drasil saves us more time in the documentation development process by letting us encode each piece of information of our scientific problems once and generating the document automatically.

In this chapter, we will provide an introduction to Drasil and Jupyter Notebook, including their usage and benefits. Following this, we will delve into the problems that our paper aims to address.

1.1 Background

Chapter 1.1.1 gives a general introduction to Drasil, and Chapter 1.1.2 discusses the features and benefits of Jupyter Notebook.

1.1.1 Drasil

Drasil is a framework that can generate software artifacts, including Software Requirement Specifications (SRS), code (C++, C#, Java, and Python), README, and Makefile, from a stable knowledge base. The goals of Drasil are reducing knowledge duplication and improving traceability [16]. Drasil captures the knowledge through our hand-made case studies. We currently have 10 case studies that cover different physics problems, such as Projectile motion and double Pendulum simulation.

Recipes for scientific problems are encoded in Drasil, which then generates code and documentation for us. Each piece of information only needs to be provided to Drasil once, and that information can be used wherever it is needed. By defining and storing common concepts in a central repository and case-specific concepts in their own packages, Drasil enables the reuse of information across different engineering domains and applications. This feature significantly reduces the time and effort required for software development and documentation, while also improving the consistency and accuracy of the information being used. Later in the chapters, we will discuss the details of how information is encoded and how knowledge is reused in Drasil.

The SRS is built using a template for designing and documenting scientific computing software requirements as created by Smith et al [17]. Drasil is currently capable of generating an SRS in the document languages HTML and LaTeX. We are looking to extend the capability of Drasil by generating Jupyter Notebooks in Drasil.

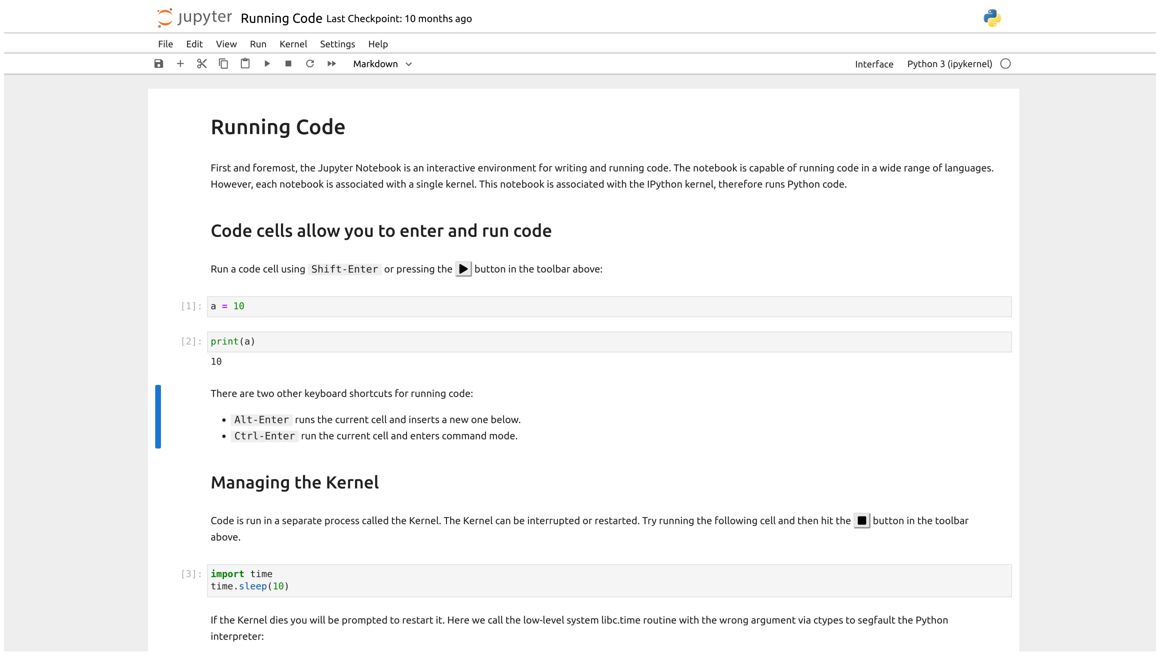
1.1.2 Jupyter Notebook

Jupyter Notebook is an interactive open-source web application for creating and sharing computational science documentation that contains text, executable code, mathematical equations, graphics, and visualizations.

Structure of a notebook document

A Jupyter Notebook has two components: front-end "cells" and back-end "kernels". The notebook consists of a series of cells, which can be code cells, Markdown cells, or raw cells. A cell is a multiline text input field. The notebook follows a sequential flow, where users entering a piece of information, either in the form of text or programming code, into the cells from the web page interface. This information is then sent to the back-end kernels for execution, and the results are return to the user [18]. Figure 1.1 shows an example of a Jupyter Notebook [19].

Figure 1.1: Example of a Jupyter Notebook



Running Code

First and foremost, the Jupyter Notebook is an interactive environment for writing and running code. The notebook is capable of running code in a wide range of languages. However, each notebook is associated with a single kernel. This notebook is associated with the IPython kernel, therefore runs Python code.

Code cells allow you to enter and run code

Run a code cell using `Shift-Enter` or pressing the ▶ button in the toolbar above:

```
[1]: a = 10
```

```
[2]: print(a)
10
```

There are two other keyboard shortcuts for running code:

- `Alt-Enter` runs the current cell and inserts a new one below.
- `Ctrl-Enter` run the current cell and enters command mode.

Managing the Kernel

Code is run in a separate process called the Kernel. The Kernel can be interrupted or restarted. Try running the following cell and then hit the ■ button in the toolbar above.

```
[3]: import time
     time.sleep(10)
```

If the Kernel dies you will be prompted to restart it. Here we call the low-level system libc.time routine with the wrong argument via ctypes to segfault the Python interpreter:

The Value of Jupyter Notebooks

There are several advantages of Jupyter Notebook: sharable, all-in-one, and live code. First of all, the notebook is easy to share because it can be converted into other formats such as HTML, Markdown, and PDF. This is advantageous because it allows someone working on a notebook to share it with others without requesting that they install any additional software and making it easier to collaborate on the projects. Secondly, Jupyter Notebooks combine all aspects of data in one single document, making the document easy to visualize, maintain and modify. In addition, they provide an environment of live code and computational equations. Usually, when programmers are running code on some other IDEs, they have to write the entire program before executing it. However, the notebook allows programmers to execute a specific portion of the code without running the whole program. The ability to run a snippet of code and integrate with text highlight the usability of the notebook. Previous research has demonstrated that Jupyter Notebooks can significantly contribute to reproducibility, reusability, and more effective computational workflows in science [20].

1.2 Problem Statement

Since both Jupyter Notebook and Drasil focus on creating and generating scientific computing documentation, we are interested in extending the values of Jupyter Notebook to Drasil and the kind of knowledge we can manipulate. Following are the three main problems we are trying to solve with Drasil in this paper:

1. Generate Jupyter Notebooks. To acheive this, we will have to generate documents in notebook format. Jupyter Notebook is a simple JSON document with a .ipynb file extension. Notebook contents are either code or Markdown. Therefore, non-code contents must be in Markdown format with JSON layout. Drasil can currently write in HTML and LaTeX. We are building a notebook printer in Drasil for generating documents that are readable and writable in Jupyter Notebook.

2. Develop the structure of lesson plans and generate them. As mentioned, Jupyter Notebook is used as an educational tool for teaching engineering courses. When

it comes to teaching, lesson plans are often brought up because they help teachers to organize the daily activities in each class time. We are interested in teaching Drasil a "textbook" structure by starting with generating a simple physics lesson plan and expanding Drasil's application. We aim to capture the elements of textbook chapters, identify the family of lesson plans, and classify the knowledge to build a general structure in Drasil, which will enable the lesson plan to generalize to a variety of lessons.

3. Generate notebooks that mix text and code. Jupyter Notebook is an interactive application for creating documents that contain formattable text and executable code. However, Drasil doesn't currently support interactive recipes. There is no code in SRS documents, and text and code are generated separately in Drasil. We are looking for the possibility of generating a notebook document that incorporate both text and code, thereby enhancing the capabilities of Drasil and its potential to solve more scientific problems.

1.3 Thesis Outline

Chapter 2 covers the topic of how Drasil generates and prints documents using the Drasil printer, as well as the creation of the notebook printer for generating Jupyter Notebooks in Drasil. Moving on to Chapter 3, we discuss the structure of lesson plans, how we define the lesson plans language in Drasil, and how to generate them with the notebook printer. Chapter 4 discusses various approaches for splitting the contents to mix different types of content, such as text and code, in Jupyter Notebooks with Drasil, as well as the implementation for generating code blocks. Lastly, Chapter 5 provides an overview of the future work and concludes the achievements of this work.

Chapter 2

Drasil Printer

To generate Jupyter Notebooks in Drasil, the first step is to build a printer that can handle notebook generation. As explained in Chapter 1, a notebook is a JSON document composed of code and Markdown contexts, such as text and images. Drasil is currently capable of generating SRS documents in HTML and LaTeX, which are handled by the HTML and TeX printers, respectively. We are adding a JSON printer to Drasil for generating SRS documents in notebook format.

Once we have the user-encoded document (i.e., recipes of the scientific problem), the contents are passed to Drasil's printers for printing. The printer is located in the **drasil-printers**, which contains all the necessary modules and functions for printing software artifacts. The **drasil-printers** module is responsible for transferring the types and data defined in Drasil's source language to printable objects and rendering those objects in desirable formats, such as HTML, LaTeX, or JSON. A list of packages and modules of the printers and their responsibilities can be found in Table 2.1. The majority of the **drasil-printers** already existed before this research; we only added a JSON printer and made a few changes to it for better notebook printing.

This chapter explains how the contents are printed, how the printer works, and the implementation of the JSON printer.

Table 2.1: Summary of Packages and Modules in **drasil-printers**

Package/Module	**Responsibility**
Language.Drasil.DOT	Defines types and holds functions for generating traceability graphs as .dot files.
Language.Drasil.HTML	Holds all functions needed to generate HTML files.
Language.Drasil.JSON	Holds all functions needed to generate JSON files.
Language.Drasil.Log	Holds functions for generating log files.
Language.Drasil.Markdown	Holds functions for generating READMEs alongside GOOL code.
Language.Drasil.Plain	Holds functions for generating plain files.
Language.Drasil.Printing	Transfers types and datas to printable objects and defines helper functions for printing.
Language.Drasil.TeX	Holds all functions needed to generate TeX files.
Language.Drasil.Config	Holds default configuration functions.
Language.Drasil.Format	Defines document types (SRS, Website, or Jupyter) and output formats (HTML, TeX, JSON, or Plain).

2.1 How documents are printed in Drasil

In Drasil, a document that is meant to be printable includes a title, authors, and layout objects, as illustrated in Code 2.1. While the title and authors are simply of type `Sentence`, the layout objects are a collection of various contents.

Code 2.1: Source Code for Definition of a Printable Document

```
data Document = Document Title Author [LayoutObj]
```

The contents of the document are defined as `RawContent` in Drasil's document source language, as shown in Code 2.2. We categorize the contents into various types and deal with them explicitly. For instance, a **Paragraph** is comprised of sentences, and an **EqnBlock** holds an expression of type `ModelExpr`[1].

Code 2.2: Source Code for Definition of RawContent

```
-- | Types of contents we deal with explicitly.
data RawContent =
Table [Sentence] [[Sentence]] Title Bool
| Paragraph Sentence
| EqnBlock ModelExpr
| DerivBlock Sentence [RawContent]
| Enumeration ListType
| Defini DType [(Identifier, [Contents])]
| Figure Lbl Filepath MaxWidthPercent
| Bib BibRef
| Graph [(Sentence, Sentence)] (Maybe Width) (Maybe
  ↪ Height) Lbl
```

To print these raw contents, we transform them into printable layout objects, defined in Code 2.3 in **Language.Drasil.Printing**. Although the types of these layout objects are similar to the types of the raw contents, layout objects are more appropriate for printing because all the information is generalized into a type called `Spec`, as shown in Code 2.4. For example, a printable **Paragraph** contains `Contents`, which is a `Spec` (Code 2.5). The smallest unit that any layout object holds is always a `Spec`, which means that printing always starts from a `Spec`. By generalizing different

[1]`ModelExpr` is a mathematical expression language.

kinds of information that layout objects hold, we can print them more efficiently.

Code 2.3: Source Code for Definition of LayoutObj

```haskell
-- | Defines types similar to content types of
-- RawContent in "Language.Drasil" but better
-- suited for printing.
data LayoutObj =
    Table Tags [[Spec]] Label Bool Caption
  | Header Depth Title Label
  | Paragraph Contents
  | EqnBlock Contents
  | Definition DType [(String,[LayoutObj])] Label
  | List ListType
  | Figure Label Caption Filepath MaxWidthPercent
  | Graph [(Spec, Spec)] (Maybe Width) (Maybe Height)
     ↪ Caption Label
  | HDiv Tags [LayoutObj] Label
  | Cell [LayoutObj]
  | Bib BibRef
```

Code 2.4: Source Code for Definition of Spec

```haskell
-- | Redefine the 'Sentence' type from Language.Drasil
-- to be more suitable to printing.
data Spec = E Expr
          | S String
          | Spec :+: Spec
          | Sp Special
          | Ref LinkType String Spec
          | EmptyS
          | Quote Spec
          | HARDNL
```

Once the conversion of contents from `RawContent` to `Layoutobj` is done, the layout objects can be targeted to produce the desired format in various document languages through language printers.

Code 2.5: Source Code for Definition of Contents

```
-- | Contents are just a sentence ('Spec').
type Contents = Spec
```

Here is an example of how an expression is encoded and printed: Equation 2.1.1 represents the velocity (v) obtained by integrating constant acceleration (a^c) with respect to time (t) in one dimension, which is used in the case study Projectile:

$$v = v^i + a^c t \tag{2.1.1}$$

To encode Equation 2.1.1, which we name `rectVel`, we might write it as shown in Code 2.6, where the type `pExpr` is a synonyms used for `ModelExpr`. Let's unpack this code. The QP module, short for **Data.Drasil.Quantities.Physics**, assigns symbols and units to physical concepts used in Drasil. These quantities are of type `UnitalChunk`, which represents concepts with quantities that require a unit definition. For example, **constAccel** is a physical concept with the definition "a one-dimensional acceleration that is constant", symbol a^c, and unit m/s.

The **sy** constructor creates an expression from a concept that contains a symbol. Additionally, it is clear that **$=**, **addRe**, and **mulRe** constructors are used for equating, adding, and multiplying two expressions, respectively.

Code 2.6: Code for Encoding rectVel

```
rectVel :: PExpr
rectVel = sy QP.speed $= sy QP.iSpeed `addRe`
(sy QP.constAccel `mulRe` sy QP.time)
```

Once the equation is defined, we can incorporate it into a `Sentence`. Code 2.7

shows an example of using an expression in a sentence, where **eS** lifts a `ModelExpr` to a `Sentence`. Equations can also be used in other content types that contain expressions, such as **DerivBlock**[2]. Alternatively, we can convert expressions directly to `Contents`, as shown in Code 2.8.

Code 2.7: Code for Converting rectVel to a Sentence

```haskell
equationsSent :: Sentence
equationsSent = S "From Equation" +:+ eS rectVel
```

Code 2.8: Source Code for Converting ModelExpr to Contents

```haskell
-- | Displays a given expression and attaches a '
  ↪ Reference' to it.
lbldExpr :: ModelExpr -> Reference -> LabelledContent
lbldExpr c lbl = llcc lbl $ EqnBlock c

-- | Same as 'lbldExpr' except content is unlabelled
-- (does not attach a 'Reference').
unlbldExpr :: ModelExpr -> Contents
unlbldExpr c = UlC $ ulcc $ EqnBlock c
```

After encoding the equation and creating the sentence, the printers take over and convert the expression to a printable **EqnBlock**, which can then be generated in a specific document language. In Code 2.9, we can see how an **EqnBlock** is converted from a `RawContent` to a printable `LayoutObj` and rendered in LaTeX.

For more details on how to create a project using Drasil and how information is encoded, please refer to Chapter 3 and the Drasil Wiki: Creating Your Project in Drasil.

[2]**DerivBlock** is a type of contents representing a derivation block.

Code 2.9: Source Code for Rendering EqnBlock to LaTeX

```
1     -- Line 2-15 is handled by Language.Drasil.Printing
2     -- | Helper that translates 'LabelledContent's to a
3     -- printable representation of 'T.LayoutObj'.
4     -- Called internally by 'lay'.
5     layLabelled :: PrintingInformation -> LabelledContent -> T.LayoutObj
6     layLabelled sm x@(LblC _ (EqnBlock c)) =
7       T.HDiv ["equation"] [T.EqnBlock
8       (P.E (modelExpr c sm))] (P.S $ getAdd $ getRefAdd x)
9
10    -- | Helper that translates 'RawContent's to a
11    -- printable representation of 'T.LayoutObj'.
12    -- Called internally by 'lay'.
13    layUnlabelled :: PrintingInformation -> RawContent -> T.LayoutObj
14    layUnlabelled sm (EqnBlock c) = T.HDiv ["equation"]
15     [T.EqnBlock  (P.E (modelExpr c sm))] P.EmptyS
16
17    -- Line 18-28 is handled by Language.Drasil.TeX
18    -- | Helper for rendering 'LayoutObj's into TeX.
19    lo :: LayoutObj -> PrintingInformation -> D
20    lo (EqnBlock contents) _ = makeEquation contents
21
22    -- | Prints an equation.
23    makeEquation :: Spec -> D
24    makeEquation contents = toEqn (spec contents)
25
26    -- | toEqn inserts an equation environment.
27    toEqn :: D -> D
28    toEqn (PL g) = equation $ PL (\_ -> g Math)
```

2.2 Notebook Printer

Since `LayoutObj` is the key to handling different types of contents, each document language's printer is responsible for rendering layout objects in that particular language and generating necessary information for the document. For example, CSS describes the style and presentation of an HTML page, so generating the necessary CSS selectors in HTML documents is handled by the HTML printer. In the case of a Jupyter Notebook document, metadata[3] is required. To implement a well-functioning notebook printer, our focus is on rendering contents in JSON format and generating necessary metadata.

2.2.1 Rendering LayoutObjs in notebook format

Code 2.10 is the primary function for rendering layout objects into a notebook. This function works similarly to the ones used by the HTML and TeX printers, and is responsible for generating content in the appropriate format. Each type of layout object is handled explicitly, taking into account how notebook users add content by hand in Jupyter Notebook, to ensure accurate reproduction and display of the contents. To help us properly render content in notebook format, we also created a few helper functions. For instance, `nbformat` (Code 2.11) helps create the necessary indentations for each line of content and encode them into JSON. We take advantage of the **encode** function from the Haskell package **Text.JSON**, which takes a Haskell value and converts it into a JSON string [22].

[3]Information about a book or its contents is known as metadata. It's often used to regulate how the notebook behaves and how its feature works [21].

Code 2.10: Source Code for Rendering LayoutObjs into JSON

```
-- | Helper for rendering LayoutObjects into JSON
printLO :: LayoutObj -> Doc
printLO (Header n contents l) = nbformat empty $$ nbformat
(h (n + 1) <> pSpec contents) $$ refID (pSpec l)
printLO (Cell layObs) = markdownCell $ vcat (map printLO
   ↪ layObs)
printLO (HDiv _ layObs _) = vcat (map printLO layObs)
printLO (Paragraph contents) = nbformat empty $$
nbformat (stripnewLine (show(pSpec contents)))
printLO (EqnBlock contents)  = nbformat mathEqn
where
toMathHelper (PL g) = PL (\_ -> g Math)
mjDelimDisp d = text "$$" <> stripnewLine (show d) <> text
   ↪  "$$"
mathEqn = mjDelimDisp $ printMath $ toMathHelper $
TeX.spec contents
printLO (Table _ rows r _ _) = nbformat empty $$
makeTable rows (pSpec r)
printLO (Definition dt ssPs l) = nbformat (text "<br>") $$
makeDefn dt ssPs (pSpec l)
printLO (List t) = nbformat empty $$ makeList t False
printLO (Figure r c f wp) = makeFigure (pSpec r) (pSpec c)
   ↪  (text f) wp
printLO (Bib bib) = makeBib bib
printLO Graph{} = empty
```

In addition, because non-code contents in Jupyter Notebook are built in Markdown, some types of contents require special treatment for Markdown generation, such as tables. Although Jupyter Notebook supports HTML tables (where we would be able to reuse the function from the HTML printer), we want to make the generated documents more “human-like” and reflect how people create contents in Jupyter. Therefore, instead of generating HTML tables, we create tables in Markdown format. The function `makeTable` from Code 2.12 generates a table in Markdown and converts it to the notebook format.

Code 2.11: Source Code for Converting Contents into JSON

```haskell
import qualified Text.JSON as J (encode)

-- | Helper for converting a Doc in JSON format
nbformat :: Doc -> Doc
nbformat s = text ("    " ++ J.encode (show s ++ "\n")
  ↪  ++ ",")
```

Code 2.12: Source Code for Rendering a Markdown Table

```haskell
-- | Renders Markdown table, called by 'printLO'
makeTable :: [[Spec]] -> Doc -> Doc
makeTable [] _      = error "No table to print"
makeTable (l:lls) r = refID r $$ nbformat empty $$
  (makeHeaderCols l $$ makeRows lls) $$ nbformat empty

-- | Helper for creating table rows
makeRows :: [[Spec]] -> Doc
makeRows = foldr (($$) . makeColumns) empty

-- | makeHeaderCols: Helper for creating table header
-- (each of the column header cells)
-- | makeColumns: Helper for creating table columns
makeHeaderCols, makeColumns :: [Spec] -> Doc
makeHeaderCols l = nbformat (text header) $$
  nbformat (text $ genMDtable ++ "|")
  where
    header = show(text "|" <> hcat(punctuate
      (text "|") (map pSpec l)) <> text "|")
    c = count '|' header
    genMDtable = concat (replicate (c-1) "|:--- ")

makeColumns ls = nbformat (text "|" <> hcat(punctuate
  (text "|") (map pSpec ls)) <> text "|")
```

To handle the various types of contents, we break them down into different types and handle each type individually in our code. When we encounter a more complex case, we create a specific **make** function to deal with it to reduce confusion in the main `printLO` function. For instance, we have `makeTable`, which handles table generation, and `makeList`, which generates a list of items. These functions are then called by `printLO`. We carefully consider how contents are created in the notebook and render each type of layout object in notebook format to ensure that the generated document is a valid Jupyter Notebook.

2.2.2 Metadata Generation

There are two types of metadata in a Jupyter Notebook: the first type is for the notebook environment setup (line 9-30 in Code A.1 in Appendix A), while the second type (line 3-7 in Code A.1 in Appendix A) is used to control the behavior of a notebook cell, where we define the type of cell (i.e, Code or Markdown). Generating the first type of metadata is straightforward since the metadata for setting up the environment is identical across all notebooks. We built a helper function called `makeMetadata` to generate the necessary metadata of a notebook document, as shown in Code 2.13. This function is called when a notebook document is being built, and the metadata is printed at the end of the document.

The second type of metadata is more complex. We need to break down our contents into units and differentiate them to generate the right type of cells. We will discuss this further in Chapter 4 after introducing a new case study in Chapter 3. For now, since there is no code in the SRS, all contents should be in Markdown. To generate the metadata for a Markdown cell, we use the helper function `markdownCell` function from Code 2.14. This function creates the necessary metadata and a cell for the given unit of content. An example implementation can be found in Code 2.15.

update the code when it's

Code 2.13: Source Code for Making Metadata

```
-- | Generate the necessary metadata for a notebook
   ↪ document.
makeMetadata :: Doc
makeMetadata = vcat [
  text " \"metadata\": {",
    vcat[
    text "  \"kernelspec\": {",
      text "   \"display_name\": \"Python 3\",",
      text "   \"language\": \"python\",",
      text "   \"name\": \"python3\"",
      text "  },"],
    vcat[
    text "  \"language_info\": {",
      text "   \"codemirror_mode\": {",
        text "    \"name\": \"ipython\",",
        text "    \"version\": 3",
        text "   },"],
      text "   \"file_extension\": \".py\",",
      text "   \"mimetype\": \"text/x-python\",",
      text "   \"name\": \"python\",",
      text "   \"nbconvert_exporter\": \"python\",",
      text "   \"pygments_lexer\": \"ipython3\",",
      text "   \"version\": \"3.9.1\"",
      text "  }",
    text " },",
  text " \"nbformat\": 4,",
  text " \"nbformat_minor\": 4"
]
```

The JSON printer implemented so far is not without flaws, there is always room for improvement. Nevertheless, the current implementation already enables Drasil to generate Jupyter Notebooks and expand the generated document to include SRS in JSON format. This makes it possible to edit and share Drasil-generated documents with Jupyter Notebook, thereby increasing their value.

Code 2.14: Source Code for markdownCell

```
-- | Helper for building markdown cells
markdownB', markdownE :: Doc
markdownB' = text "  {\n   \"cell_type\": \"markdown
  \",\n \"metadata\": {},\n   \"source\": ["
markdownE  = text "     \"\\n\"\n   ]\n  },"

-- | Helper for generate a Markdown cell
markdownCell :: Doc -> Doc
markdownCell c = markdownB' <> c <> markdownE
```

Code 2.15: Source Code for Calling markdownCell

```
printLO (Cell layoutObs) = markdownCell $ vcat (map
   ↪ printLO layoutObs)
```

For detailed implementation of the JSON printer, please refer to the Appendix . link

Chapter 3

Lesson Plans

With the addition of a JSON printer capable of generating Jupyter Notebooks, we are now looking to expand Drasil's application by generating educational documents. As discussed in Chapter 1, Jupyter Notebooks are commonly used in teaching engineering courses due to their characteristics and advantages. One of the educational practices to enhance education is conducting lesson plans [23, 24], which provide a guide for structuring daily activities in each class period. A lesson plan outlines the learning objectives, methods and procedures for achieving them, and the measurement of how student progress. Lesson plans are an ideal starting point for generating educational documents in Drasil because they are more accessible than academic papers. In addition, we are able to work with real examples in a lesson plan. This chapter will cover the structure of a lesson plan, how we define the language of lesson plans in Drasil, and a new case study on Projectile Lesson.

3.1 Language of Lesson Plans

To generate a new type of document, lesson plans, in Drasil, we must define its language first. Drasil's document language has SRS, and we are creating a language for lesson plans. As discussed in Chapter 2, a Drasil document has a title, authors, and sections, which hold the contents of the document. The definition of a document is defined in **drasil-lang** [1] as shown in Code 3.1 [2], where Document is the type for SRS document and Notebook is for Jupyter Notebook, specifically lesson plans at this moment. The reason why we define them separately is because we print the SRS and lesson plans differently. We are able to pattern match the way we print the document in the printer.

Code 3.1: Pseudocode for Definition of Document

```
data Document = Document Title Author ShowToC [Section]
              | Notebook Title Author [Section]
```

Before defining the language for lesson plans, we need to understand the components and categorize the knowledge to create a universal structure within Drasil. We analyzed the similarities and differences of elements in textbook chapters in Discussion of Projectile Lesson: What and Why using online resources. Based on our analysis, we narrowed down the elements and defined a structure that fits our lesson plans the most. It's worth noting that this structure may be subject to future modifications to better suit our needs. Following is the structure of our lesson plans:

- Introduction: an introduction of the lesson plan or the topic.

[1]**drasil-lang** holds the higher level language for Drasil.

[2]ShowToC is ShowTableOfContents in the source code, which is to determine whether to show the table of contents in the document.

- Learning Objectives: what students can do or will learn after the lesson.
- Review: a recap of what has been covered previously.
- A Case Problem: a case problem that link the topic to a real world problem.
- Example: an example of the case problem.
- Summary: a summary of the lesson plan.
- Bibliography: references that support the lesson plan.
- Appendix: additional resources or information of the lesson.

With the lesson plan structure in place, we can now define helper types and functions to create the document language for generating lesson plans. Our first step is to define the types and data for the lesson and its chapters in Drasil's document language, **drasil-docLang**. Code 3.2 is the core declaration of the lesson plan. A LsnDesc type represents a lesson description (line 1), which consists of lesson chapters (line 3), including an introduction, learning objectives, review, case problem, example, summary, bibliography, and appendix. The detail structure of each chapter is defined in line 12-31. At present, Contents is the only defined elements as the chapter structure has not yet been fully understood. We intend to further develop the chapter structure in the future.

link to future work

The LsnDecl type, as shown in Code 3.3, is used to declare all the necessary chapters for a lesson plan. It is similar in definition to LsnDesc, but in a more usable form. It is meant to be a semantic rendition of a lesson plan document, while LsnDesc is intended to be a general description and more suitable for printing [25]. They are

Code 3.2: Source Code for Notebook Core Language

```
type LsnDesc = [LsnChapter]

data LsnChapter = Intro Intro
                | LearnObj LearnObj
                | Review Review
                | CaseProb CaseProb
                | Example Example
                | Smmry Smmry
                | BibSec
                | Apndx Apndx

-- ** Introduction
newtype Intro = IntrodProg [Contents]

-- ** Learning Objectives
newtype LearnObj = LrnObjProg [Contents]

-- ** Review Chapter
newtype Review = ReviewProg [Contents]

-- ** A Case Problem
newtype CaseProb = CaseProbProg [Contents]

-- ** Examples of the lesson
newtype Example = ExampleProg [Contents]

-- ** Summary
newtype Smmry = SmmryProg [Contents]

-- ** Appendix
newtype Apndx = ApndxProg [Contents]
```

identical at this point because the chapter structure is not well understood, but they might evolve differently as we gain more understanding of our lesson plans.

Next, we need functions to generate chapters. We can use the Section type that

Code 3.3: Source Code for LsnDecl

```
type LsnDecl   = [LsnChapter]

data LsnChapter = Intro NB.Intro
                | LearnObj NB.LearnObj
                | Review NB.Review
                | CaseProb NB.CaseProb
                | Example NB.Example
                | Smmry NB.Smmry
                | BibSec
                | Apndx NB.Apndx
```

Code 3.4: Source Code for Section and the Constructor

```
data Section = Section
             { tle  :: Title
             , cons :: [SecCons]
             , _lab :: Reference
             }
makeLenses ''Section

-- | Constructor for creating 'Section's with a title
-- ('Sentence'), introductory contents, a list of
-- subsections, and a shortname ('Reference').
section :: Sentence -> [Contents] -> [Section] ->
   ↪ Reference -> Section
section title intro secs = Section title (map Con
   ↪ intro ++ map Sub secs)
```

is used for creating SRS sections, which consists of a title, a list of contents, and a short name, as shown in Code 3.4. We can also take advantage of the section smart constructor to build our own chapter constructors, as illustrates in Code 3.5. Once we have these constructors, we can use them to build each chapter (Code 3.6).

When building lesson plans, the document and chapters are encoded in the LsnDecl

Code 3.5: Source Code for Chapter Constructors

```haskell
learnObj, review, caseProb, example :: [Contents] ->
    [Section] -> Section
learnObj cs ss = section (titleize' Doc.learnObj) cs
  ↪ ss learnObjLabel
review   cs ss = section (titleize Doc.review)    cs
  ↪ ss reviewLabel
caseProb cs ss = section (titleize Doc.caseProb)  cs
  ↪ ss caseProbLabel
example  cs ss = section (titleize Doc.example)   cs
  ↪ ss exampleLabel
```

Code 3.6: Source Code for Making Chapters

```haskell
-- | Helper for making the 'Learning Objectives'.
mkLearnObj :: LearnObj -> Section
mkLearnObj (LrnObjProg cs) = Lsn.learnObj cs []

-- | Helper for making the 'Review'.
mkReview :: Review -> Section
mkReview (ReviewProg r) = Lsn.review r []

-- | Helper for making the 'Case Problem'.
mkCaseProb :: CaseProb -> Section
mkCaseProb (CaseProbProg cp) = Lsn.caseProb cp []

-- | Helper for making the 'Example'.
mkExample:: Example -> Section
mkExample (ExampleProg cs) = Lsn.example cs []
```

type, which is then converted to LsnDesc for printing. In Code 3.7, the mkNb function takes the user-encoded list of chapters (i.e., LsnDecl) and **System Information** [3] to form a lesson plan document.

All types and functions mentioned in this chapter are declared in **drasil-docLang**.

[3] System Information is a data structure designed to contain all the necessary information about a system for the purpose of generating artifacts.

Code 3.7: Source Code for mkNb

```haskell
mkNb :: LsnDecl -> (IdeaDict -> IdeaDict -> Sentence)
     -> SystemInformation -> Document
mkNb dd comb si@SI {_sys = s, _kind = k, _authors = a} =
  Notebook (nw k `comb` nw s) (foldlList Comma List $
  map (S . name) a) $ mkSections si l where
    l = mkLsnDesc si dd

-- | Helper for creating the lesson plan sections.
mkSections :: SystemInformation -> LsnDesc -> [Section]
mkSections si = map doit where
  doit :: LsnChapter -> Section
  doit (Intro i)    = mkIntro i
  doit (LearnObj l) = mkLearnObj l
  doit (Review r)   = mkReview r
  doit (CaseProb c) = mkCaseProb c
  doit (Example e)  = mkExample e
  doit (Smmry s)    = mkSmmry s
  doit BibSec       = mkBib (citeDB si)
  doit (Apndx a)    = mkAppndx a

mkLsnDesc :: SystemInformation -> LsnDecl -> NB.LsnDesc
mkLsnDesc _ = map sec where
sec :: LsnChapter -> NB.LsnChapter
sec (Intro i)     = NB.Intro i
sec (LearnObj l)  = NB.LearnObj l
sec (Review r)    = NB.Review r
sec (CaseProb c)  = NB.CaseProb c
sec (Example e)   = NB.Example e
sec (Smmry s)     = NB.Smmry s
sec BibSec        = NB.BibSec
sec (Apndx a)     = NB.Apndx a
```

Table 3.2 summarizes the responsibility of each module for the language of lesson plans.

Table 3.2: Summary of Notebook Modules

Module	**Responsibility**
Drasil.DocLang	
Notebook.hs	Contains constructors for building chapters.
Drasil.DocumentLanguage.Notebook	
Core.hs	Contains general description functions for lesson plans.
DocumentLanguage.hs	Holds functions to create chapters and form a lesson plan.
LsnDecl.hs	Contains declaration functions for generating lesson plans.

3.2 A Case Study: Projectile Motion

In Chapter 3.1, we discussed the language of lesson plans to introduce a new case study on projectile motion. We chose projectile motion as a starting point for our lesson plans for several reasons: i) it is often one of the initial concepts taught when students are introduced to the study of dynamics; ii) the developed model is considered relatively straightforward as it solely incorporates kinematics, which pertains to the geometric characteristics of motion [26]; iii) Drasil already captures the knowledge of projectile, allowing us to showcase the reuse of knowledge. In this chapter, we are going to reproduce the Projectile Motion Lesson, authored by Dr. Spencer Smith, and generate a Jupyter Notebook version with Drasil.

3.2.1 Knowledge Reusability

In Drasil, we store commonly used knowledge, such as physics concepts (e.g., acceleration) and mathematics ideas (e.g., Cartesian coordinates), in a package named

drasil-data. Additionally, each case study has its own package that contains concepts specific to that study. For example, "Projectile Motion" is an idea in the Projectile case study. Once these ideas and concepts are defined in Drasil, they can be utilized whenever needed. Since there is an overlap in knowledge between the Projectile SRS and Projectile Motion Lesson, we can reuse the information without the need to encode it again.

For example, the following equation is the position of a particle moving in straight line as a function of time, given that the object experiences a constant acceleration:

$$p = p^i + v^i t + \frac{a^c t^2}{2} \tag{3.2.1}$$

The left side p (**scalarPos**) is a physical quantity defined in **drasil-data** with units as a UnitalChunk [4], where the right side expression (**scalarPos'**) is a PExpr declared in the Projectile package in **drasil-example**, as shown in Code 3.8.

Code 3.8: Source Code for scalarPos

```
    scalarPos :: UnitalChunk
    scalarPos  = uc CP.scalarPos lP Real metre

    scalarPos' :: PExpr
    scalarPos' = sy iPos `addRe` (sy QP.iSpeed `mulRe`
      sy time `addRe` half (sy QP.constAccel `mulRe`
         ↪ square (sy time)))
```

The information of Equation 3.2.1 was already available prior to the development of Projectile Motion. By utilizing the definitions of both **scalarPos** and **scalarPos'** as a reference, we can incorporate this information into our own usage for the lesson

[4]UnitalChunks are concepts with quantities that require a definition of units. A 'UnitalChunk' contains a 'Concept', 'Symbol', and 'Unit'.

plan. The implementation of this can be seen in Code 3.9. The expression is defined in a LabelledContent because we are adding a label to it, allowing us to cross-reference it in the document.

Code 3.9: Source Code for lcrectPos

```
    lcrectPos :: LabelledContent
    lcrectPos = lbldExpr (sy scalarPos $= scalarPos') (
      ↪ makeEqnRef "rectPos")
```

Drasil offers a powerful way to store and reuse knowledge across different domains and aspects of the case study. By growing our knowledge database in this way, we believe that we can save time and effort while also ensuring consistency and accuracy in the use of concepts and ideas. This has the potential to greatly enhance the efficiency and effectiveness of engineering projects, and we are excited to continue exploring the possibilities of Drasil in the field of engineering.

3.2.2 Reproduce the Lesson

In accordance with the lesson plan structure discussed in Chapter 3.1, we divided the Projectile Motion Lesson into four chapters: learning objectives, review, a case problem, and examples. Each chapter is composed of a variety of content types, such as sentences, equations, or figures. To combine these contents into a chapter, we convert them to the Contents type and map them together. We provide smart constructors like lbldExpr [5] for transfering different kind of contents to Contents. In Code 3.10, we demonstrate how information and contents of the review chapter are encoded in Drasil.

[5] This converts a **ModelExpr** into a **Contents**.

Code 3.10: Source Code for Encoded Review Chapter

```
reviewContent :: [Contents]
reviewContent = [reviewHead, reviewContextP1,
  LlC E.lcrectVel, LlC E.lcrectPos, LlC E.lcrectNoTime,
  reviewEqns, reviewContextP2]

reviewHead, reviewContextP1,
  reviewEqns, reviewContextP2 :: Contents
reviewHead = foldlSP_ [headSent 2 (S "Rectilinear
  Kinematics: Continuous Motion")]
reviewContP1 = foldlSP_
  [S "As covered previously, the", plural equation, S
   "relating", phrase velocity, sParen (eS (sy QP.speed)
   ) `sC` phrase position, sParen (eS (sy QP.scalarPos))
   `S.and_` phrase time, sParen (eS (sy QP.time))
   `S.for` phrase motion `S.in_` S "one dimension with",
   phrase QP.constAccel, sParen (eS (sy QP.constAccel))
   +:+ S "are as follows:"]

reviewEqns = foldlSP [S "where", eS (sy QP.iSpeed)
  `S.and_` eS (sy QP.iPos), S "are the initial",
   phrase velocity `S.and_` phrase position,
   S ",respectively"]

reviewContP2 = foldlSP
  [S "Only two of these", plural equation, S "are
   independent, since the third" +:+ phrase equation, S
   "can always be derived from the other two"]
```

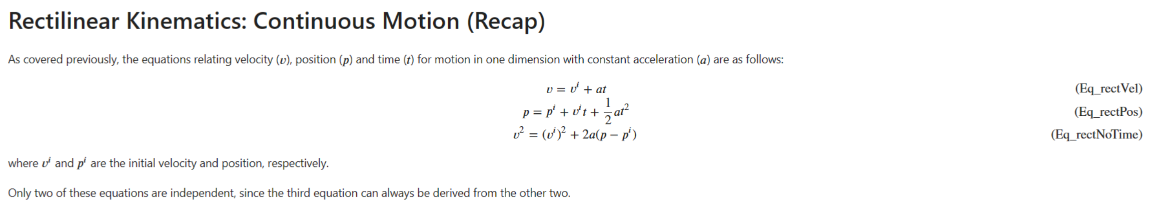
A lesson plan is represented in the LsnDecl type, which is a collection of chapters (see Code 3.11). We then use the mkNb function (presented in Code 3.7) to convert the lesson plan into a Drasil document. The resulting document can be printed and produced as a Jupyter Notebook with the Drasil printer, as discussed in Chapter 2.

Figures 3.2 and 3.3 display the review chapter of the lesson plan created manually and using Drasil, respectively.

Code 3.11: Source Code for Forming a Notebook

```haskell
mkNB :: LsnDecl
mkNB = [
  LearnObj $ LrnObjProg [learnObjContext],
  Review $ ReviewProg reviewContent,
  CaseProb $ CaseProbProg caseProbCont,
  Example $ ExampleProg exampleContent,
  BibSec
]
```

Figure 3.2: Review Chapter Created Manually



Rectilinear Kinematics: Continuous Motion (Recap)

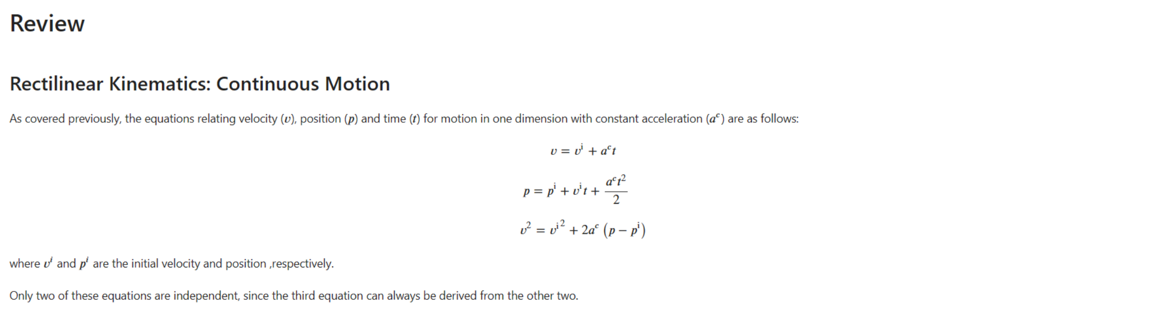
As covered previously, the equations relating velocity (v), position (p) and time (t) for motion in one dimension with constant acceleration (a) are as follows:

$$v = v^i + at \tag{Eq\_rectVel}$$
$$p = p^i + v^i t + \frac{1}{2}at^2 \tag{Eq\_rectPos}$$
$$v^2 = (v^i)^2 + 2a(p - p^i) \tag{Eq\_rectNoTime}$$

where v^i and p^i are the initial velocity and position, respectively.

Only two of these equations are independent, since the third equation can always be derived from the other two.

Figure 3.3: Review Chapter Generated using Drasil



Review

Rectilinear Kinematics: Continuous Motion

As covered previously, the equations relating velocity (v), position (p) and time (t) for motion in one dimension with constant acceleration (a^c) are as follows:

$$v = v^{\mathrm{i}} + a^c t$$

$$p = p^{\mathrm{i}} + v^{\mathrm{i}} t + \frac{a^c t^2}{2}$$

$$v^2 = {v^{\mathrm{i}}}^2 + 2a^c \left(p - p^{\mathrm{i}}\right)$$

where v^i and p^i are the initial velocity and position ,respectively.

Only two of these equations are independent, since the third equation can always be derived from the other two.

For a complete demonstration of how the lesson plan is created using Drasil, including the implementation and generated document, we recommend referring to the appendix. link

Chapter 4

Code Block Generation

Jupyter Notebooks are valued for their effectiveness in writing and revising code for data research. They allow code to be written in discrete blocks (or "cells"), which can be executed separately, as opposed to writing and running a whole program [27]. This allows for a mix of content types with code to better present information.

In Chapter 2, we cover two types of metadata in Jupyter Notebooks: one type is necessary for forming the notebook, while the other is required to create cells for the contents. We explain how to generate the metadata and create a Markdown cell. When generating SRS, we do not need to worry about generating code blocks since SRS does not include any code. However, when creating lesson plans, we may want to integrate real examples that involve code. As we are now combining text and code in a document, we need to address the following questions before generating the right type of cell: i) what type of cell should we use, Markdown or code? and ii) how do we know when to end a cell and start a new one? That is, how do we determine where to split the contents into cells?

To begin, we need to consider the conceptual definition of a cell in Jupyter Notebooks. A cell is essentially a standalone unit of information or code that can be executed independently. In other words, it is a unit of content within the notebook [28]. A cell can contain either text or code and can span multiple lines. Understanding the relationship between cells and their contents is crucial for implementing an effective splitting strategy. By identifying natural boundaries within the text or code and recognizing the unit of the contents, we can determine where to split the contents into cells.

In this chapter, we will discuss different approaches and implementations for splitting the contents and generating the appropriate type of cells.

4.1 Unit of Contents

4.1.1 Section-level

When considering what would be the appropriate unit of content for splitting, one might first think of paragraphs or sections. In the source language of Drasil, since a document is made up of sections (as seen in Code 3.1), it may appear reasonable to split these sections into individual cells. However, the nested structure of Drasil documents, where each Section is composed of a list of Contents and Sections (as demonstrated in Code 3.4), does not align well with the sequential flow of a Jupyter Notebook. To address this issue, we flatten the structure of the Drasil document by making each section and subsection an independent Section.

For example, Code 4.1 defines the Introduction section, where the original nested structure (lines 1-12) comprises a list of subsections, while in the flattened version (lines 13-21), each subsection is self-contained and has its own type. Code 4.2 further illustrates that each section is independent after the changes.

Code 4.1: Source code for Definition of Introduction

```haskell
1  -- Nested Structure
2  -- | Introduction section. Contents are top level
3  -- followed by a list of subsections.
4  data IntroSec = IntroProg Sentence Sentence [IntroSub]
5
6  -- | Introduction subsections.
7  data IntroSub where
8    IPurpose :: [Sentence] -> IntroSub
9    IScope   :: Sentence -> IntroSub
10   IChar    :: [Sentence] -> [Sentence] -> [Sentence] ->
         ↪ IntroSub
11   IOrgSec  :: Sentence -> CI -> Section -> Sentence ->
         ↪ IntroSub
12
13 -- Flatten Structure
14 -- | Introduction section.
15 data IntroSec = IntroProg Sentence Sentence
16
17 -- | Introduction subsections.
18 newtype IPurpose = IPurposeProg [Sentence]
19 newtype IScope = IScopeProg Sentence
20 data IChar = ICharProg [Sentence] [Sentence] [Sentence]
21 data IOrgSec = IOrgProg Sentence CI Section Sentence
```

While flattening the structure of a document can allow for it to be split into individual cells by sections, there are limitations to this approach. Splitting the contents at the section level might not always be the most effective approach. It's possible that certain sections might be too long to fit comfortably in a single cell. Moreover, when working with documents that combine text and code (such as lesson plans), section-level splitting may not be appropriate due to the different types of

Code 4.2: Pseudocode for Definition of DocSection

```
    -- Nested Structure
    data DocSection = TableOfContents
                    | RefSec RefSec
                    | IntroSec IntroSec
                    | StkhldrSec StkhldrSec
                    ...

    -- Flatten Structure
    data DocSection = TableOfContents TableOfContents
                    | RefSec RefSec
                    | TUnits TUnits
                    | TSymb TSymb
                    | TAandA TAandA
                    | IntroSec IntroSec
                    | IPurposeSub IPurposeSub
                    | IScopeSub IScopeSub
                    | ICharSub ICharSub
                    | IOrgSub IOrgSub
                    ...
```

cells needed for text and code. Therefore, a better approach is needed.

4.1.2 LayoutObj-level

In Jupyter Notebook, a cell can be seen as a self-contained unit of information, and it can contain multiple types of content, such as text, code, and figures. To determine the appropriate unit of content for splitting, we need to consider the content itself and what makes sense in terms of its structure and organization. Although a cell might not always be the most appropriate unit of conent for splitting, it is somehow the lowest level of "display content" that conveys a coherent piece of information [28]. Therefore, splitting the content based on logical units of information might be a more effective approach rather than using sections as the sole criterion.

In previous chapters, we discussed how Drasil handles different types of content through the use of the RawContent data type, which includes paragraphs, figures, equations, and other content types (Code 2.2). A Drasil Section can consist of a list of RawContent, allowing for the inclusion of different types of contents within a single section. Additionally, as we saw in Chapter 2, the document is printed in a specific document language using LayoutObj, which is derived from RawContent. Because each content type is handled explicitly by LayoutObj, we can take advantage of this and split each type of content into its own cell.

To implement this approach, we first need to ensure that each layout object is generated independently and is not nested with other layout objects. In Chapter 2, we discussed how RawContent is translated to a printable LayoutObj. The printLO function in Code 2.3 demonstrates how the printer renders each content type into a notebook format. However, it's worth noting that the current format of LayourObj is designed for SRS and may not be suitable for lesson plans. For instance, the **HDiv** type wraps sections and creates an HTML <div> tag, and even an equation block is translated into the **HDiv**, as seen in Code 2.9. Moreover, the **Definition** type is designed for the definition or model defined in SRS and may not be required for lesson plans. To better accommodate lesson plan content types, we may need to create a new LayoutObj in the future when we have a better understanding of the lesson plan structure.

Currently, we are using the existing LayoutObj to translate our lesson plan contents into printable layout object. Since these contents are not code and should be in Mardown, we print each required content type independently into a Markdown cell. To accomplish this, we use the markdownCell function from Code 2.14. This function generates the necessary metadata and creates the Markdown cell for each layout object, which is our unit of content.

Code 4.3 illustrates how each content type is rendered in notebook format in a Markdown cell. For layout objects that are not needed in lesson plans, we make them empty. We also separate equation blocks from **HDiv** with the equation tag to have more control over the structure.

Code 4.3: Source Code for printLO'

```haskell
-- printLO' is used for generating lesson plans
printLO' :: LayoutObj -> Doc
printLO' (HDiv ["equation"] layObs _) = markdownCell $
  vcat (map printLO' layObs)
printLO' (Header n contents l) = markdownCell $ nbformat
  (h (n + 1) <> pSpec contents) $$ refID (pSpec l)
printLO' (Cell layObs) = vcat (map printLO' layObs)
printLO' HDiv {} = empty
printLO' (Paragraph contents) = markdownCell $ nbformat
  (stripnewLine (show(pSpec contents)))
printLO' (EqnBlock contents) = nbformat mathEqn
  where
    toMathHelper (PL g) = PL (\_ -> g Math)
    mjDelimDisp d = text "$$" <> stripnewLine (show d) <>
       ↪ text "$$"
    mathEqn = mjDelimDisp $ printMath $ toMathHelper $
      TeX.spec contents
printLO' (Table _ rows r _ _) = markdownCell $
  makeTable rows (pSpec r)
printLO' (Definition dt ssPs l) = empty
printLO' (List t) = markdownCell $ makeList t False
printLO' (Figure r c f wp) = markdownCell $ makeFigure
  (pSpec r) (pSpec c) (text f) wp
printLO' (Bib bib) = markdownCell $ makeBib bib
printLO' Graph{} = empty
```

4.2 Code Block

We split contents with our content types for various reasons, including the need to differentiate between Markdown contents and code to generate the appropriate type of cell and to deal with each type of contents specifically. To better manage and generate code in Jupyter Notebook with Drasil, we introduce a new content type called **CodeBlock**. Like other content types, a **CodeBlock** is defined within RawContent, which requires a CodeExpr as shown in 4.4.

Code 4.4: Source Code for the New Definition of RawContent

```haskell
-- | Types of layout objects we deal with explicitly.
data RawContent =
Table [Sentence] [[Sentence]] Title Bool
| Paragraph Sentence
| EqnBlock ModelExpr
| DerivBlock Sentence [RawContent]
| Enumeration ListType
| Defini DType [(Identifier, [Contents])]
| Figure Lbl Filepath MaxWidthPercent
| Bib BibRef
| Graph [(Sentence, Sentence)] (Maybe Width) (Maybe
  ↪ Height) Lbl
| CodeBlock CodeExpr
```

CodeExpr is a language that allows us to define code expressions. It shares similarities with Expr functions, constructors, and operators, but is tailored specifically for generating code. We utilize this data type to define and encode the expressions for our Jupyter Notebook code.

The process of handling code blocks and printing the code within a code cell is similar to how we handle other Markdown contents, as discussed in the previous chapters. However, the conversion is unique to the **CodeBlock** type. For instance, to encode the code, we can use the unlbldCode (Code 4.5) function, which converts a CodeExpr to Contents.

Code 4.5: Source Code for Rendering CodeBlock to LayoutObj

```
-- | Unlabelled code expression
unlbldCode :: CodeExpr -> Contents
unlbldCode c = UlC $ ulcc $ CodeBlock c
```

After encoding the code expressions in Drasil, the printer then converts the code blocks into printable layout objects, as defined in Code 4.6, using the methods in Code 4.7. The resulting content, which is a RawContent, is translated into a printable object, a LayoutObj, before being processed by the document language printer.

Code 4.6: Source Code for the New Definition of LayoutObj

```
data LayoutObj =
    Table Tags [[Spec]] Label Bool Caption
  | Header Depth Title Label
  | Paragraph Contents
  | EqnBlock Contents
  | Definition DType [(String,[LayoutObj])] Label
  | List ListType
  | Figure Label Caption Filepath MaxWidthPercent
  | Graph [(Spec, Spec)] (Maybe Width) (Maybe Height)
     ↪ Caption Label
  | CodeBlock Contents
  | HDiv Tags [LayoutObj] Label
  | Cell [LayoutObj]
  | Bib BibRef
```

Generating a code cell in Jupyter Notebook requires metadata, similar to generating a markdown cell. However, since the metadata is identical across code cells, the codeCell function generates the required metadata and creates a code cell for the

Code 4.7: Source Code for Rendering CodeBlock to LayoutObj

```haskell
-- | Helper that translates 'LabelledContent's to a
-- printable representation of 'LayoutObj'.
layLabelled sm (LblC _ (CodeBlock c)) = T.CodeBlock
(P.E (codeExpr c sm))
-- | Helper that translates 'RawContent's to a
-- printable representation of 'LayoutObj'.
layUnlabelled sm (CodeBlock c) = T.CodeBlock (P.E (
  ↪ codeExpr c sm))
```

given code, which is the 'contents' in Code 4.9. This constructor eliminates the need for redundant metadata specification and provides a convenient way to generate code cells in Jupyter Notebook.

Code 4.8: Source Code for Generating a CodeBlock

```haskell
-- | Helper for generate a Code cell
codeCell :: Doc -> Doc
codeCell c = codeB <> c <> codeE

codeB, codeE :: Doc
codeB = text "  {\n   \"cell_type\": \"code\",\n
  \"execution_count\": null,\n   \"metadata\":
  {},\n   \"outputs\": [],\n   \"source\": ["
codeE  = text "\n   ]\n  },"
```

Finally, the JSON printer takes the printable layout object of the code block, prints the code, converts it to the notebook format, and generates a code cell, as demonstrated in Code 4.9.

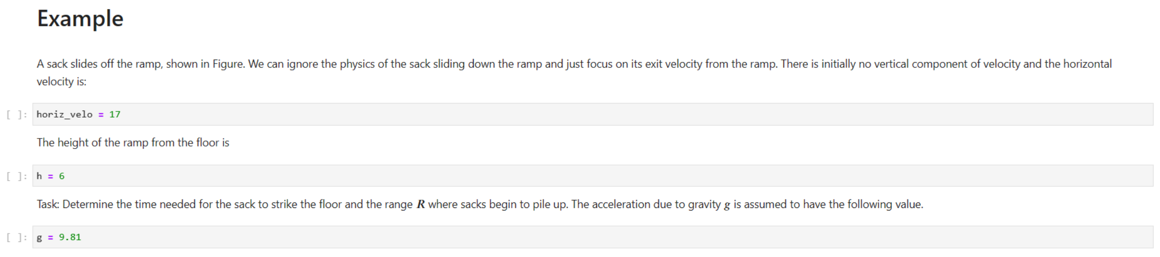
To conclude this chapter, we can say that the benefits of using Jupyter Notebook lie in its ability to allow users to write a portion of code and combine it with text.

Code 4.9: Source Code for Rendering CodeBlock into JSON

```
    -- | Helper for rendering CodeBlock into JSON
    printLO' (CodeBlock contents) = codeCell $ nbformat $
      ↪ cSpec contents
```

We have discussed various approaches to split the contents and generate the appropriate types of cell. The Projectile Motion Lesson generated by Drasil, as shown in Figure 4.4, demonstrates that we are able to mix text and code and generate the appropriate cell types in Jupyter Notebook with Drasil. The source code for encoding this example can be found in Code 4.10. In comparison, Figure 4.5 shows the same part created manually.

Figure 4.4: Snapshot of Example Chapter Generated using Drasil



Example

A sack slides off the ramp, shown in Figure. We can ignore the physics of the sack sliding down the ramp and just focus on its exit velocity from the ramp. There is initially no vertical component of velocity and the horizontal velocity is:

```
[ ]: horiz_velo = 17
```

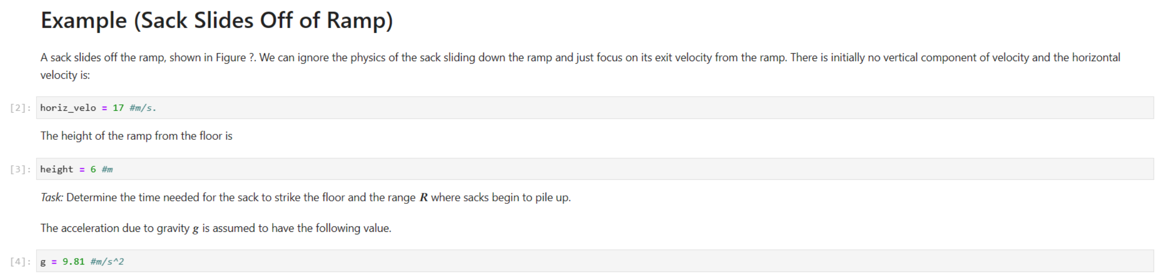
The height of the ramp from the floor is

```
[ ]: h = 6
```

Task: Determine the time needed for the sack to strike the floor and the range R where sacks begin to pile up. The acceleration due to gravity g is assumed to have the following value.

```
[ ]: g = 9.81
```

Figure 4.5: Snapshot of Example Chapter Created Manually



Example (Sack Slides Off of Ramp)

A sack slides off the ramp, shown in Figure ?. We can ignore the physics of the sack sliding down the ramp and just focus on its exit velocity from the ramp. There is initially no vertical component of velocity and the horizontal velocity is:

```
[2]: horiz_velo = 17 #m/s.
```

The height of the ramp from the floor is

```
[3]: height = 6 #m
```

Task: Determine the time needed for the sack to strike the floor and the range R where sacks begin to pile up.

The acceleration due to gravity g is assumed to have the following value.

```
[4]: g = 9.81 #m/s^2
```

Code 4.10: Source Code for Encoding Example Chapter

```
exampleContent :: [Contents]
exampleContent = [exampleContextP1, codeC1,
  exampleContextP2, codeC2, exampleContextP3, codeC3]

exampleContextP1, exampleContextP2, exampleContextP3 :: Contents
exampleContextP1 = foldlSP_ [S "A sack slides off the
  ramp, shown in Figure.", S "We can ignore the physics
  of the sack sliding down the ramp and just focus on
  its exit", phrase velocity +:+. S "from the ramp",
  S "There is initially no vertical component of",
  phrase velocity `S.andThe` S "horizontal",
  phrase velocity, S "is:"]
exampleContextP2 = foldlSP_ [S "The", phrase height
  `S.ofThe` S "ramp from the floor is"]
exampleContextP3 = foldlSP_ [S "Task: Determine the",
  phrase time, S "needed for the sack to strike the
  floor and the range", P cR +:+. S "where sacks begin
  to pile up", S "The", phrase acceleration, S "due to",
  phrase gravity, P lG +:+. S "is assumed to have the
  following value"]

codeC1, codeC2, codeC3 :: Contents
codeC1 = unlbldCode (sy horiz_velo $= exactDbl 17)
codeC2 = unlbldCode (sy QP.height $= exactDbl 6)
codeC3 = unlbldCode (sy QP.gravitationalAccel $= dbl 9.81)
```

Chapter 5

Conclusion

5.1 Future Work

5.1.1 JSON Printer Improvement

While the current JSON printer is capable of generating Jupyter Notebook documents, there are several issues that need to be addressed. For example, the JSON printer currently relies on the TeX printer function for generating mathematical equations. However, this approach has some limitations, and some equations may not be displayed correctly in Jupyter Notebook, such as the use of the **symbf** command for math equations in LaTeX, which is not valid in Jupyter Notebook. To ensure mathematical symbols and expressions are displayed correctly, it is crucial to understand how Jupyter Notebook works with these elements. It may be necessary to modify the JSON printer and use different methods or consider using specialized libraries or tools designed for generating mathematical equations in Jupyter Notebook.

5.1.2 Design Lesson Plan Content Type

In Chapter 4, we discussed the potential limitations of the current LayoutObj for the structure of lesson plans. Most of the existing layout objects are designed for SRS data types such as **Definition**. To better accommodate the content types found in lesson plans, we could define a new set of LayoutObjs that are specific to these types of contents, such as a model that includes step-by-step instructions, since many lessons include these instructions. By doing so, we could ensure that each content type is handled explicitly by the appropriate LayoutObj, and we could create a separate cell for each type of content as discussed earlier. This approach would make it easier to split the content into logical units of information, and it would also make the resulting notebook more modular and easier to navigate.

5.1.3 Develop the Structure of Lesson Plans

The current structure of lesson plans includes several chapters such as learning objectives, case problems, and examples, and each chapter is made up of a list of contents. However, this structure needs improvement to better fit the architecture of each chapter. By gaining a better understanding of our lesson plans and the structure of each chapter, we can incorporate the newly designed specific content types (as discussed in 5.1.2) into each chapter. For example, the Case Problem chapter should include the model of procedure analysis, which includes step-by-step instructions. Having a more detailed and adaptable structure of lesson plans would enable greater consistency and efficiency in creating and delivering content. Furthermore, it would make it easier to capture the key elements and knowledge of each lesson.

5.1.4 Develop the Language of Code Block

As we have discussed in earlier chapters, Drasil has the capability of generate source code as a part of software artifacts. To generate code content, we can use the available code expression, known as CodeExpr. However, generating code in a 'text' document is different from generating it as a program. While we can generate code and code blocks in the Jupyter Notebook, the current language is not yet mature and requires further improvement. For example, to encode the code variable, we need to define it as a **UnitalChunk** (Code 5.1) before we can use it in an expression. However, **UnitalChunk**s are concepts with quantities that require unit definition, which does not align with the concept of code variables. We can introduce a new data type that better fits code variables or create smart constructors. In addition, we still need to explore how to make the most of our **CodeBlock**, and generate code flawlessly. These are interesting areas to investigate.

Code 5.1: Source Code for horiz_velo

```
    horiz_velo :: UnitalChunk
    horiz_velo = uc horizontalMotion (variable "horiz_velo
      ↪ ") Real velU
```

5.2 Conclusion

This paper demonstrates the potential of Jupyter Notebook as a versatile tool for creating and sharing scientific documents and for enhancing the teaching and learning efficiency in engineering education. To extend the capabilities of Jupyter Notebook to Drasil, we present the implementation of a JSON printer that is capable of generating Drasil software artifacts, such as SRS, in the notebook format. We discuss the necessary functions and data types for working with notebook generation, as well as the process of encoding information in Drasil and generating and printing Jupyter Notebook documents using the printer.

The addition of the JSON printer expands the application of Drasil, making it possible to generate educational documents and develop lesson planes. We analyze the similarities and differences of elements in textbook chapters to create a universal structure that fits our lesson plans the most and provide insights into the design and implementation of the structure in the Drasil language. With the lesson plan structure in place, we demonstrate how the knowledge can be manipulated and reused in Drasil through the creation of a new case study on Projectile Motion Lesson.

Furthermore, we highlight the benefits of using Jupyter Notebooks for data research and how they enable users to seamlessly combine different content types with code. When creating lesson plans that involve code, we need to address questions such as which type of cell to use and how to determine where to split the contents into cells. By understanding the conceptual definition of a cell and identifying natural boundaries within the text or code, we can effectively divide the contents and generate appropriate cell types. We cover the implementation of Markdown and code cell generation, which are essential components for creating a Jupyter Notebook document.

In conclusion, this research addresses three main problems and provides a starting point for generating Jupyter Notebook in Drasil. With further refinement and development of the JSON printer and the language of lesson plans, generating Jupyter

Notebook documents in Drasil can open up more possibilities.

Appendix A

Your Appendix

Figure A.6 shows the dependency between modules in **drasil-printers**. The arrow points to the module that is being relied on.

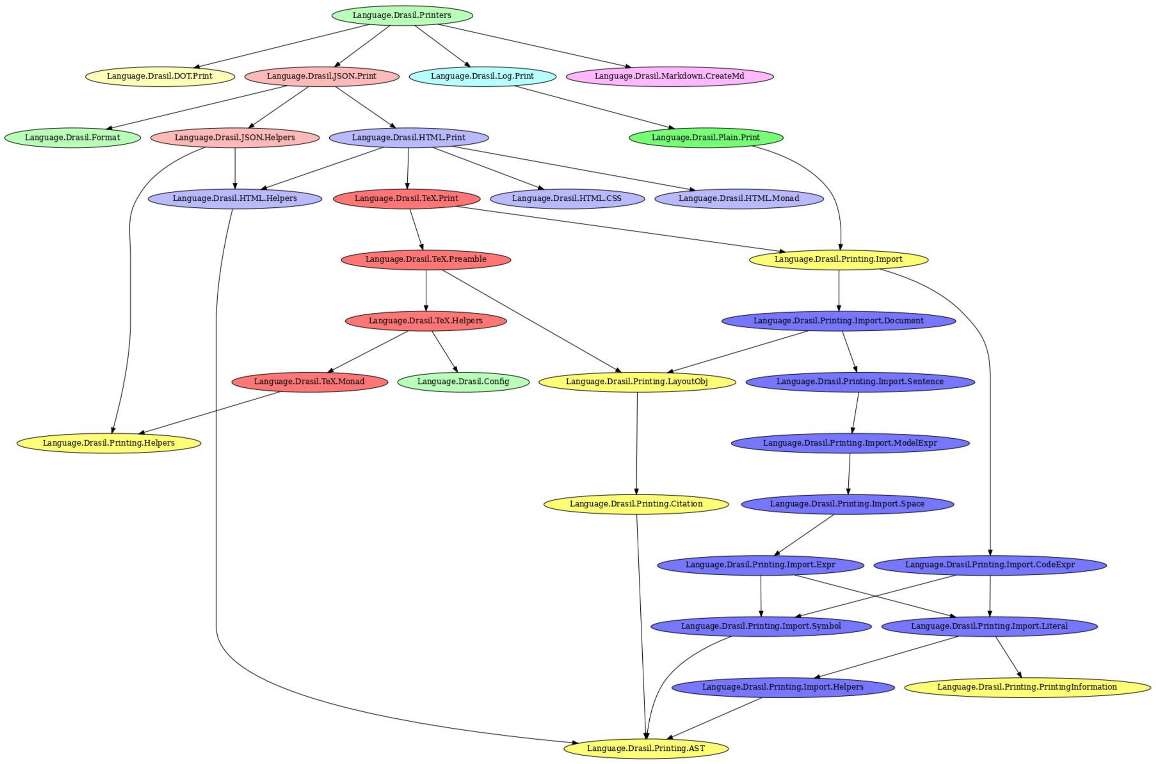

Language.Drasil.Printers
Language.Drasil.DOT.Print
Language.Drasil.JSON.Print
Language.Drasil.Log.Print
Language.Drasil.Markdown.CreateMd
Language.Drasil.Format
Language.Drasil.JSON.Helpers
Language.Drasil.HTML.Print
Language.Drasil.Plain.Print
Language.Drasil.HTML.Helpers
Language.Drasil.TeX.Print
Language.Drasil.HTML.CSS
Language.Drasil.HTML.Monad
Language.Drasil.TeX.Preamble
Language.Drasil.Printing.Import
Language.Drasil.TeX.Helpers
Language.Drasil.Printing.Import.Document
Language.Drasil.TeX.Monad
Language.Drasil.Config
Language.Drasil.Printing.LayoutObj
Language.Drasil.Printing.Import.Sentence
Language.Drasil.Printing.Helpers
Language.Drasil.Printing.Import.ModelExpr
Language.Drasil.Printing.Citation
Language.Drasil.Printing.Import.Space
Language.Drasil.Printing.Import.Expr
Language.Drasil.Printing.Import.CodeExpr
Language.Drasil.Printing.Import.Symbol
Language.Drasil.Printing.Import.Literal
Language.Drasil.Printing.Import.Helpers
Language.Drasil.Printing.PrintingInformation
Language.Drasil.Printing.AST

Figure A.6: drasil-printer Dependency Graph

Code A.1: JSON Code of A Notebook Document

```
{
  "cells": [
  {
    "cell_type": "markdown",
    "metadata": {},
    "source": []
  }
  ],
  "metadata": {
    "kernelspec": {
      "display_name": "Python 3",
      "language": "python",
      "name": "python3"
    },
    "language_info": {
      "codemirror_mode": {
        "name": "ipython",
        "version": 3
      },
      "file_extension": ".py",
      "mimetype": "text/x-python",
      "name": "python",
      "nbconvert_exporter": "python",
      "pygments_lexer": "ipython3",
      "version": "3.9.1"
    }
  },
  "nbformat": 4,
  "nbformat_minor": 4
}
```

Bibliography

[1] Andrew Forward. *Software documentation: Building and maintaining artefacts of communication.* University of Ottawa (Canada), 2002 (cit. on p. 1).

[2] David Lorge Parnas. “Precise documentation: The key to better software”. In: *The Future of Software Engineering* (2011), pp. 125–148 (cit. on p. 1).

[3] Vikas S Chomal and Jatinderkumar R Saini. “Significance of software documentation in software development process”. In: *International Journal of Engineering Innovations and Research* 3.4 (2014), p. 410 (cit. on p. 1).

[4] Noela Jemutai Kipyegen and William PK Korir. “Importance of software documentation”. In: *International Journal of Computer Science Issues (IJCSI)* 10.5 (2013), p. 223 (cit. on p. 1).

[5] Koothoor Nirmitha and Smith Spencer. *Developing Scientific Computing Software: Current Processes and Future Directions.* 2016. URL: http://hdl.handle.net/11375/13266 (cit. on p. 1).

[6] Yu Wen and Smith Spencer. *A Document Driven Methodology for Improving the Quality of a Parallel Mesh Generation Toolbox.* 2007. URL: http://hdl.handle.net/11375/21299 (cit. on p. 1).

[7] Jeffrey M. Perkel. “Why Jupyter is data scientists’ computational notebook of choice”. In: *Nature* 563 (2018), pp. 145–146 (cit. on p. 2).

[8] Alberto Cardoso, Joaquim Leitão, and César Teixeira. “Using the Jupyter notebook as a tool to support the teaching and learning processes in engineering courses”. In: *The Challenges of the Digital Transformation in Education: Proceedings of the 21st International Conference on Interactive Collaborative Learning (ICL2018)-Volume 2*. Springer. 2019, pp. 227–236 (cit. on p. 2).

[9] Pengfei Zhao and Junwei Xia. “Use JupyterHub to Enhance the Teaching and Learning Efficiency of Programming Related Courses”. In: (2019) (cit. on p. 2).

[10] Sibylle Hermann and Jörg Fehr. “Documenting research software in engineering science”. In: *Scientific Reports* 12.1 (2022), p. 6567 (cit. on p. 2).

[11] Jiyoo Chang and Christine Custis. “Understanding Implementation Challenges in Machine Learning Documentation”. In: *Equity and Access in Algorithms, Mechanisms, and Optimization*. 2022, pp. 1–8 (cit. on p. 2).

[12] Rebecca Sanders and Diane Kelly. “Dealing with risk in scientific software development”. In: *IEEE software* 25.4 (2008), pp. 21–28 (cit. on p. 2).

[13] Spencer Smith, Thulasi Jegatheesan, and Diane Kelly. “Advantages, disadvantages and misunderstandings about document driven design for scientific software”. In: *2016 Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE)*. IEEE. 2016, pp. 41–48 (cit. on p. 2).

[14] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. "Understanding and improving the quality and reproducibility of Jupyter notebooks". In: *Empirical Software Engineering* 26.4 (2021), p. 65 (cit. on p. 2).

[15] Jiawei Wang, Li Li, and Andreas Zeller. "Better code, better sharing: on the need of analyzing jupyter notebooks". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. 2020, pp. 53–56 (cit. on p. 2).

[16] The Drasil Team. *Drasil - Generate All the Things!* URL: https://jacquescarette.github.io/Drasil/ (cit. on p. 3).

[17] W Spencer Smith and Lei Lai. "A new requirements template for scientific computing". In: *Proceedings of the First International Workshop on Situational Requirements Engineering Processes–Methods, Techniques and Tools to Support Situation-Specific Requirements Engineering Processes, SREP*. Vol. 5. Citeseer. 2005, pp. 107–121 (cit. on p. 4).

[18] GitHub contributors. *The Jupyter Notebook*. URL: https://jupyter-notebook.readthedocs.io/en/stable/notebook.html (cit. on p. 5).

[19] GitHub contributors. *Jupyter Notebook*. URL: https://github.com/jupyter/notebook (cit. on p. 5).

[20] Marijan Beg, Juliette Taka, Thomas Kluyver, Alexander Konovalov, Min Ragan-Kelley, Nicolas M Thiéry, and Hans Fangohr. "Using Jupyter for reproducible scientific workflows". In: *Computing in Science & Engineering* 23.2 (2021), pp. 36–46 (cit. on p. 6).

[21] Jupyter Notebook contributors. *Add metadata to your book pages.* URL: https://jupyterbook.org/en/stable/content/metadata.html (cit. on p. 16).

[22] The Haskell Team. *Text.JSON.* URL: https://hackage.haskell.org/package/json-0.10/docs/Text-JSON.html (cit. on p. 16).

[23] Volkan Cicek and Tok Hidayet. "Effective use of lesson plans to enhance education". In: *International Journal of Economy, Management and Social Sciences* 2.6 (2013), pp. 334–341 (cit. on p. 22).

[24] Harry K Wong and Rosemary Tripi Wong. *The first days of school: How to be an effective teacher.* Harry K. Wong Publications Mountain View, CA, 2018 (cit. on p. 22).

[25] The Drasil Team. *Do we need both LsnDecl and LsnDesc for lesson plan?* URL: https://github.com/JacquesCarette/Drasil/issues/3308 (cit. on p. 24).

[26] Spencer Smith. *Discussion of Projectile Lesson: What and Why.* URL: https://github.com/smiths/caseStudies/blob/master/CaseStudies/projectile/projectileLesson/AboutProjectileLesson.pdf (cit. on p. 29).

[27] The Codecademy Team. *How To Use Jupyter Notebooks.* URL: https://www.codecademy.com/article/how-to-use-jupyter-notebooks-py3 (cit. on p. 34).

[28] The Drasil Team. *Separating cells of SRS.* URL: https://github.com/JacquesCarette/Drasil/issues/2346 (cit. on pp. 35, 37).