

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

© Copyright by Jane Doe, MONTH YEAR
All Rights Reserved

Masters of Engineering (YYYY)
(Department You Belong To)

McMaster University
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: xiii, 37

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

Abstract here (no more than 300 words)

Your Dedication
Optional second line

Acknowledgements

Acknowledgements go here.

Contents

Lay Abstract	iii
Abstract	iv
Acknowledgements	vi
Notation, Definitions, and Abbreviations	xi
Declaration of Academic Achievement	xiii
1 Introduction	1
1.1 Background	2
1.2 Problem Statement	5
1.3 Thesis Outline	6
2 Drasil Printer	7
2.1 How documents are printed in Drasil?	8
2.2 Notebook Printer	11
3 Lesson Plans	19
3.1 Language of Lesson Plans	20

3.2 A Case Study: Projectile Motion	26
4 Conclusion	31
A Your Appendix	32

List of Figures

3.1	Review Chapter Created Manually	30
3.2	Review Chapter Generated using Drasil	30
A.1	drasil-printer Dependency Graph	33

List of Tables

2.1	Summary of Packages and Modules in drasil-printers	8
3.2	Summary of Notebook Modules	26

Notation, Definitions, and Abbreviations

Notation

$A \leq B$ A is less than or equal to B

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

SRS

CSS

HTML

SCS

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 an intersection of computer science, mathematics, and science. It is a field that solves complex scientific problems by using computing techniques and tools. Writing documentation is a 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 instil knowledge of the system it describes” [1]. The significance of software documentation has been presented in many papers by previous researchers [2], [3], [4]. It is further shown by Smith et al. [5], [6] that developing scientific computing software (SCS) in a document-driven methodology improves the quality of the software .

Jupyter Notebook is a system for creating and sharing data science and scientific computing documentation. It is a nonprofit, open-source application born out in 2014, providing 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, the notebook 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 documented in SCS 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].

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.

1.1 Background

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 [14]. Drasil captures the knowledge through our hand-made case studies. We currently have 10 case studies that cover different physics problems, such as Projectile and Pendulum. Recipes for scientific problems are encoded in Drasil, and it 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. More information and an example of how knowledge is reused can be found in Chapter 3.2.

Drasil is currently capable of generating SRS, a template for designing and documenting scientific computing software requirement decisions created by Smith et al [15], in document languages HTML and LaTeX. We are looking to extend the capability of Drasil by generating Jupyter Notebook 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 sequence of cells: code cells, markdown cells, and raw cells. A cell is a multiline text input field. The notebook works by users entering a piece of information (text or programming code) in cells from the web page user interface. That information is then passed to the back-end kernels which execute the code and return the results [16].

The Value of Jupyter Notebook

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. Secondly, it combines all aspects of data in one single document, making the document easy to visualize, maintain and modify. In addition, Jupyter Notebook provides 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.

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 achieve 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 only 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 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

Thesis outline here.

Chapter 2

Drasil Printer

The first step to generate Jupyter Notebook in Drasil is building a printer to deal with notebook generation. As mentioned in Chapter 1, a notebook is a JSON document composed of code and Markdown contexts, such as text and images. Drasil is capable of generating SRS documents in HTML and LaTeX, which are handle by our HTML and TeX printer, respectively. We are adding a JSON printer to Drasil for generating SRS documents in notebook format.

When we have the user-encoded document (i.e., recipes of their scientific problem), contents are passed to Drasil’s printers for printing. The printer is located in **drasil-printers**, which holds all the modules and functions needed for printing the software artifacts. The responsibility of **drasil-printers** is 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. Note that the majority of **drasil-printers** already existed before this research, and we only added a JSON printer and made a few changes to it for better notebook printing.

In this chapter, we discuss 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 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?

A “printable” document contains a title, authors, and contents, where the contents are different types of layout objects (Code 2.1). In Drasil’s document source language, contents are categorized into different types and dealt with explicitly. Code 2.2 shows the definition of contents we defined in source code. For example, a type **Paragraph** is

made up of sentences, and an **EqnBlock** holds an expression (of type `ModelExpr`¹). These contents are then converted to printable layout objects (as defined in Code 2.3) in **Language.Drasil.Printing**, with similar types but are more suitable for printing. After that, we can target these layout objects to generate them in a specific format in different document languages in language printers.

Code 2.1: Pseudocode for Definition of a Printable Document

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

Code 2.2: Source Code for Definition of RawContent

```
1  -- | Types of layout objects we deal with explicitly.
2  data RawContent =
3      Table [Sentence] [[Sentence]] Title Bool
4      | Paragraph Sentence
5      | EqnBlock ModelExpr
6      | DerivBlock Sentence [RawContent]
7      | Enumeration ListType
8      | Defini DType [(Identifier, [Contents])]
9      | Figure Lbl Filepath MaxWidthPercent
10     | Bib BibRef
11     | Graph [(Sentence, Sentence)] (Maybe Width) (Maybe
        ↳ Height) Lbl
```

Here is an example workflow of how an expression is encoded and printed. Equation 2.1.1 is an expression - velocity integrating constant acceleration with respect to time in one dimension, which is used in the case study **Projectile**:

$$v = v^i + ac^t \quad (2.1.1)$$

¹Modelling expression is a mathematical expression language.

Code 2.3: Source Code for Definition of LayoutObj

```

1      -- | Defines types similar to content types in
2      -- "Language.Drasil" but better suited for printing.
3      data LayoutObj =
4          Table Tags [[Spec]] Label Bool Caption
5      | Header Depth Title Label
6      | Paragraph Contents
7      | EqnBlock Contents
8      | Definition DType [(String,[LayoutObj])] Label
9      | List ListType
10     | Figure Label Caption Filepath MaxWidthPercent
11     | Graph [(Spec, Spec)] (Maybe Width) (Maybe Height)
12         ↪ Caption Label
13     | HDiv Tags [LayoutObj] Label
14     | Cell [LayoutObj]
15     | Bib BibRef

```

To encode this equation (**rectVel**), we might write it as shown in Code 2.4, where the type **PExpr** is a synonyms used for **ModelExpr**. After defining the equation, it can be used in a **Sentence**² (Code 2.5) or other content types that contain expressions, such as **DerivBlock**³. Expressions can also be converted directly to **Contents** (Code 2.6).

Code 2.4: Pseudocode for Encoding rectVel

```

1      speed' :: PExpr
2      speed' = sy QP.iSpeed `addRe` (sy QP.constAccel `mulRe
3          ↪ ` sy time)
4
5      rectVel :: PExpr
6      rectVel = sy speed $= speed'

```

Next, the printers transfer the expression to a printable **EqnBlock** and generate

²In Drasil, some content types are manipulated into a **Sentence** to form printable **Contents**.

³DerivBlock is a type of contents representing a derivation block.

Code 2.5: Pseudocode for Converting rectVel to Sentence

```

1  equationsSent :: Sentence
2  equationsSent = S "From Equation" +:+ eS rectVel
3
4  -- | Lifts an expression into a Sentence.
5  E :: ModelExpr -> Sentence
6
7  eS :: ModelExpr -> Sentence
8  eS = E

```

Code 2.6: Source Code for Converting ModelExpr to Contents

```

1  -- | Displays a given expression and attaches a '
    ↳ Reference' to it.
2  lblExpr :: ModelExpr -> Reference -> LabelledContent
3  lblExpr c lbl = llcc lbl $ EqnBlock c
4
5  -- | Same as 'lblExpr' except content is unlabelled
6  -- (does not attach a 'Reference').
7  unlblExpr :: ModelExpr -> Contents
8  unlblExpr c = UlC $ ulcc $ EqnBlock c

```

it in a specific document language. Code 2.7 shows how a [RawContent](#): **EqnBlock** is converted to a [LayoutObj](#): **EqnBlock** and rendered in LaTeX.

More information on how to create a project with Drasil and how information is encoded can be found in Chapter 3 and on the [Drasil Wiki: Creating Your Project in Drasil](#).

2.2 Notebook Printer

As we know that [LayoutObj](#) is the key of handling different types of contents, the responsibility of each document language's printer is rendering layout objects in that

Code 2.7: 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
6  ⇔ -> T.LayoutObj
7  layLabelled sm x@(LblC _ (EqnBlock c)) =
8    T.HDiv ["equation"]
9    [T.EqnBlock (P.E (modelExpr c sm))]
10   (P.S $ getAdd $ getRefAdd x)
11
12 -- | Helper that translates 'RawContent's to a
13 -- printable representation of 'T.LayoutObj'.
14 -- Called internally by 'lay'.
15 layUnlabelled :: PrintingInformation -> RawContent ->
16 ⇔ T.LayoutObj
17 layUnlabelled sm (EqnBlock c) = T.HDiv ["equation"]
18 [T.EqnBlock (P.E (modelExpr c sm))] P.EmptyS
19
20 -- Line 18-28 is handled by Language.Drasil.TeX
21 -- | Helper for rendering 'LayoutObj's into TeX.
22 lo :: LayoutObj -> PrintingInformation -> D
23 lo (EqnBlock contents) _ = makeEquation contents
24
25 -- | Prints an equation.
26 makeEquation :: Spec -> D
27 makeEquation contents = toEqn (spec contents)
28
29 -- | toEqn inserts an equation environment.
30 toEqn :: D -> D
31 toEqn (PL g) = equation $ PL (\_ -> g Math)

```

particular language, in addition, generating necessary information of the document. For example, CSS describes the style and presentation of a HTML page, therefore, generating the necessary CSS selectors in HTML documents is taken care of by the HTML printer. Similarly, metadata⁴ is required for a Jupyter Notebook document. To implement a well-functioning notebook printer, we are focusing on rendering contents in JSON format and generating necessary metadata.

2.2.1 Rendering LayoutObjs in notebook format

Code 2.10 is the main function for rendering layout objects into a notebook. This function is similar to the one in HTML and TeX printers. It is the core of generating the type of contents in that format. We deal with each type of layout object explicitly, considering how notebook users add contents by hand in Jupyter Notebook, and try to reproduce them. To properly render contents in notebook format, we also created a few helper functions. For example, `nbformat` from Code 2.8 helps create necessary indentations for each line of contents and encodes 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 [18].

Code 2.8: Source Code for Converting contents into JSON

```

1  import qualified Text.JSON as J (encode)
2
3  -- | Helper for converting a Doc in JSON format
4  nbformat :: Doc -> Doc
5  nbformat s = text ("      " ++ J.encode (show s ++ "\n")
    ↪      ++ ",")

```

⁴Information 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 [17].

In addition, since non-code contents are built in Markdown in Jupyter Notebook, some types of contents need to be taken care of particularly for Markdown generation, such as tables. Although Jupyter Notebook allows HTML tables, where we would be able to reuse the function from HTML printer, as mentioned previously, we care about how people would actually create contents in Jupyter to make the generated documents more “human-like”. Hence, instead of generating HTML tables, we are making tables in Markdown format. `makeTable` from Code 2.9 renders a table in Markdown and converts it to notebook format.

Code 2.9: Source Code for Rendering a Markdown Table

```

1  -- | Renders Markdown table, called by 'printL0'
2  makeTable :: [[Spec]] -> Doc -> Doc
3  makeTable [] _ = error "No table to print"
4  makeTable (l:lls) r = refID r $$ nbformat empty $$
5    (makeHeaderCols l $$ makeRows lls) $$ nbformat empty
6
7  -- | Helper for creating table rows
8  makeRows :: [[Spec]] -> Doc
9  makeRows = foldr (($) . makeColumns) empty
10
11 -- | makeHeaderCols: Helper for creating table header
12 -- (each of the column header cells)
13 -- | makeColumns: Helper for creating table columns
14 makeHeaderCols, makeColumns :: [Spec] -> Doc
15 makeHeaderCols l = nbformat (text header) $$
16   nbformat (text $ genMDtable ++ "|")
17   where
18     header = show(text "|" <> hcat(punctuate
19       (text "|") (map pSpec l)) <> text "|")
20     c = count '|' header
21     genMDtable = concat (replicate (c-1) "|:--- ")
22
23 makeColumns ls = nbformat (text "|" <> hcat(punctuate
24   (text "|") (map pSpec ls)) <> text "|")

```

We break down contents into different types and handle them type by type. If the case is more complicated, we build a **make** function to deal with it specifically to reduce confusion in the main `printLO` function. For example, `makeTable` takes care of table generation and `makeList` generates a list of items; they are called by `printLO`. We carefully consider how contents are created in the notebook and render each type of layout object in notebook format to make sure that the generated document is a valid Jupyter Notebook document.

Code 2.10: Source Code for Rendering LayoutObjs into JSON

```

1  -- | Helper for rendering LayoutObjects into JSON
2  printLO :: LayoutObj -> Doc
3  printLO (Header n contents l) = nbformat empty $$
4    nbformat (h (n + 1) <> pSpec contents) $$ refID (pSpec l)
5  printLO (Cell layoutObs) = markdownCell $ vcat (map printLO
6    ↪ layoutObs)
7  printLO (HDiv _ layoutObs _) = vcat (map printLO layoutObs)
8  printLO (Paragraph contents) = nbformat empty $$
9    nbformat (stripnewLine (show(pSpec contents)))
10 printLO (EqnBlock contents) = nbformat mathEqn
11   where
12     toMathHelper (PL g) = PL (\_ -> g Math)
13     mjDelimDisp d = text "$$" <> stripnewLine (show d) <>
14       ↪ text "$$"
15     mathEqn = mjDelimDisp $ printMath $ toMathHelper $ TeX.
16       ↪ spec contents
17 printLO (Table _ rows r _ _) = nbformat empty $$
18   makeTable rows (pSpec r)
19 printLO (Definition dt ssPs l) = nbformat (text "<br>") $$
20   makeDefn dt ssPs (pSpec l)
21 printLO (List t) = nbformat empty $$ makeList t False
22 printLO (Figure r c f wp) = makeFigure (pSpec r) (pSpec c) (
23   ↪ text f) wp
24 printLO (Bib bib) = makeBib bib
25 printLO Graph{} = empty

```

2.2.2 Metadata Generation

There are two kinds of metadata in the notebook: the first type is for the notebook environment setup (line 9-30 in Code A.1), and the other (line 3-7 in Code A.1) is used to control the behavior of a notebook cell, where we define the types of a cell (i.e., Code or Markdown). Generating the first type is straightforward since the metadata for setting up the environment is identical across all notebooks. We built a helper function `makeMetadata` in Code 2.11, to generate necessary metadata of a notebook document. This function is called when a Jupyter Notebook document is being built, and the metadata is printed at the end of the document.

The second type of metadata is trickier. We need to break down our contents in units and differentiate them to generate the right type of cells. We are going to discuss this further in Chapter 4 after a new case study is introduced in Chapter 3. For now, since there is no code in SRS, all contents should be in Markdown. The `markdownCell` function in Code 2.12 is the helper function for making the metadata of a Markdown cell; it generates the needed metadata and creates a cell for the passed-in unit of content. An example implementation can be found in Code 2.13.

The current JSON printer is not perfect; there is still room for improvement. However, with the printer, Drasil is capable of generating Jupyter Notebooks and expanding the generated document to include SRS in JSON format. This enables us to edit and share Drasil-generated notebooks with Jupyter Notebook, providing additional value.

The complete implementation of the JSON printer can be found in Appendix .

3,4

link

update

the

code

when

it's

final

link

to fu-

ture

work

link

Code 2.11: Source Code for Making Metadata

```
1  -- | Generate the metadata necessary for a notebook
    ↪ document.
2  makeMetadata :: Doc
3  makeMetadata = vcat [
4  text " \"metadata\": {",
5    vcat[
6    text "  \"kernel_spec\": {",
7      text "    \"display_name\": \"Python 3\",",
8      text "    \"language\": \"python\",",
9      text "    \"name\": \"python3\"",
10     text "  },",
11    vcat[
12    text "    \"language_info\": {",
13      text "      \"codemirror_mode\": {",
14        text "        \"name\": \"ipython\",",
15        text "        \"version\": 3",
16      text "      },",
17      text "      \"file_extension\": \".py\",",
18      text "      \"mimetype\": \"text/x-python\",",
19      text "      \"name\": \"python\",",
20      text "      \"nbconvert_exporter\": \"python\",",
21      text "      \"pygments_lexer\": \"ipython3\",",
22      text "      \"version\": \"3.9.1\"",
23    text "    },",
24    text "  },",
25  text " \"nbformat\": 4,",
26  text " \"nbformat_minor\": 4"
27  ]
```

Code 2.12: Source Code for markdownCell

```

1  -- | Helper for building markdown cells
2  markdownB', markdownE :: Doc
3  markdownB' = text "    {\n    \"cell_type\": \"markdown
4    \",\n \"metadata\": {},\n    \"source\": [\"
5  markdownE  = text "    \"\\n\"\\n    ]\n    },\"
6
7  -- | Helper for generate a Markdown cell
8  markdownCell :: Doc -> Doc
9  markdownCell c = markdownB' <> c <> markdownE

```

Code 2.13: Source Code for Calling markdownCell

```

1  printL0 (Cell layoutObs) = markdownCell $ vcat (map
    ↪ printL0 layoutObs)

```

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 [19, 20], 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 3.1², 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

```

1  data Document = Document Title Author ShowToC [Section]
2                  | 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.

²ShowToC 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 3), which consists of lesson chapters (line 5), including an introduction, learning objectives, review, case problem, example, summary, bibliography, and appendix. The detail structure of each chapter is defined in line 14-33. 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.

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 [21]. They are

link
to fu-
ture
work

Code 3.2: Source Code for Notebook Core Language

```

1  type LsnDesc = [LsnChapter]
2
3  data LsnChapter = Intro Intro
4                      | LearnObj LearnObj
5                      | Review Review
6                      | CaseProb CaseProb
7                      | Example Example
8                      | Smmry Smmry
9                      | BibSec
10                     | Apndx Apndx
11
12  -- ** Introduction
13  newtype Intro = IntrodProg [Contents]
14
15  -- ** Learning Objectives
16  newtype LearnObj = LrnObjProg [Contents]
17
18  -- ** Review Chapter
19  newtype Review = ReviewProg [Contents]
20
21  -- ** A Case Problem
22  newtype CaseProb = CaseProbProg [Contents]
23
24  -- ** Examples of the lesson
25  newtype Example = ExampleProg [Contents]
26
27  -- ** Summary
28  newtype Smmry = SmmryProg [Contents]
29
30  -- ** Appendix
31  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, as

Code 3.3: Source Code for LsnDecl

```

1  type LsnDecl  = [LsnChapter]
2
3  data LsnChapter = Intro NB.Intro
4                  | LearnObj NB.LearnObj
5                  | Review NB.Review
6                  | CaseProb NB.CaseProb
7                  | Example NB.Example
8                  | Smmry NB.Smmry
9                  | BibSec
10                 | Apndx NB.Apndx

```

Code 3.4: Source Code for Section and the Constructor

```

1  data Section = Section
2  { tle  :: Title
3    , cons :: [SecCons]
4    , _lab :: Reference
5  }
6  makeLenses ''Section
7
8  -- | Constructor for creating 'Section's with a
9  -- title ('Sentence'), introductory contents,
10 -- a list of subsections, and a shortname ('Reference
11 --   ↪ ').
12 section :: Sentence -> [Contents] -> [Section] ->
13         ↪ Reference -> Section
14 section title intro secs = Section title (map Con
15         ↪ intro ++ map Sub secs)

```

shown in Code 3.4, which consists of a title, a list of contents, and a short name that is used for creating SRS sections. 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

```

1   learnObj, review, caseProb, example :: [Contents] ->
2       [Section] -> Section
3   learnObj cs ss = section (titleize ' Doc.learnObj) cs
        ↪ ss learnObjLabel
4   review    cs ss = section (titleize Doc.review)    cs
        ↪ ss reviewLabel
5   caseProb  cs ss = section (titleize Doc.caseProb)  cs
        ↪ ss caseProbLabel
6   example   cs ss = section (titleize Doc.example)   cs
        ↪ ss exampleLabel

```

Code 3.6: Source Code for Making Chapters

```

1   -- | Helper for making the 'Learning Objectives'.
2   mkLearnObj :: LearnObj -> Section
3   mkLearnObj (LrnObjProg cs) = Lsn.learnObj cs []
4
5   -- | Helper for making the 'Review'.
6   mkReview :: Review -> Section
7   mkReview (ReviewProg r) = Lsn.review r []
8
9   -- | Helper for making the 'Case Problem'.
10  mkCaseProb :: CaseProb -> Section
11  mkCaseProb (CaseProbProg cp) = Lsn.caseProb cp []
12
13  -- | Helper for making the 'Example'.
14  mkExample :: Example -> Section
15  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**.

³System 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

```

1  mkNb :: LsnDecl -> (IdeaDict -> IdeaDict -> Sentence)
2      -> SystemInformation -> Document
3  mkNb dd comb si@SI {_sys = s, _kind = k, _authors = a} =
4      Notebook (nw k `comb` nw s) (foldlList Comma List $
5          map (S . name) a) $ mkSections si l where
6          l = mkLsnDesc si dd
7
8  -- | Helper for creating the lesson plan sections.
9  mkSections :: SystemInformation -> LsnDesc -> [Section]
10 mkSections si = map doit where
11     doit :: LsnChapter -> Section
12     doit (Intro i)      = mkIntro i
13     doit (LearnObj lo)  = mkLearnObj lo
14     doit (Review r)     = mkReview r
15     doit (CaseProb cp)  = mkCaseProb cp
16     doit (Example e)    = mkExample e
17     doit (Smmry s)      = mkSmmry s
18     doit BibSec         = mkBib (citeDB si)
19     doit (Apndx a)      = mkAppndx a
20
21 mkLsnDesc :: SystemInformation -> LsnDecl -> NB.LsnDesc
22 mkLsnDesc _ = map sec where
23     sec :: LsnChapter -> NB.LsnChapter
24     sec (Intro i)      = NB.Intro i
25     sec (LearnObj l)   = NB.LearnObj l
26     sec (Review r)     = NB.Review r
27     sec (CaseProb c)   = NB.CaseProb c
28     sec (Example e)    = NB.Example e
29     sec (Smmry s)      = NB.Smmry s
30     sec BibSec         = NB.BibSec
31     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 [22]; 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:

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

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

```

1      scalarPos  :: UnitalChunk
2      scalarPos  = uc CP.scalarPos 1P Real metre
3
4      scalarPos'  :: PExpr
5      scalarPos' = sy iPos `addRe` (sy QP.iSpeed `mulRe`
6      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

⁴UnitalChunks 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 `lcreatPos`

```

1   lcreatPos :: LabelledContent
2   lcreatPos = lbldExpr (sy scalarPos $= scalarPos') (
      ↪ makeEqnRef "creatPos")

```

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 transferring different kind of contents to `Contents`. In Code 3.10, we demonstrate how information and contents of the review chapter are encoded in Drasil.

⁵This converts a `ModelExpr` into a `Contents`.

Code 3.10: Source Code for Encoded Review Chapter

```

1  reviewContent :: [Contents]
2  reviewContent = [reviewHead, reviewContextP1,
3    LlC E.lcrectVel, LlC E.lcrectPos, LlC E.lcrectNoTime,
4    reviewEqns, reviewContextP2]
5
6  reviewHead, reviewContextP1,
7    reviewEqns, reviewContextP2 :: Contents
8  reviewHead = foldlSP_ [headSent 2 (S "Rectilinear
9    Kinematics: Continuous Motion")]
10 reviewContP1 = foldlSP_
11   [S "As covered previously, the", plural equation, S
12     "relating", phrase velocity, sParen (eS (sy QP.speed)
13     ) `sC` phrase position, sParen (eS (sy QP.scalarPos))
14     `S.and_` phrase time, sParen (eS (sy QP.time))
15     `S.for` phrase motion `S.in_` S "one dimension with",
16     phrase QP.constAccel, sParen (eS (sy QP.constAccel))
17     +:+ S "are as follows:"]
18
19 reviewEqns = foldlSP [S "where", eS (sy QP.iSpeed)
20   `S.and_` eS (sy QP.iPos), S "are the initial",
21   phrase velocity `S.and_` phrase position,
22   S ",respectively"]
23
24 reviewContP2 = foldlSP
25   [S "Only two of these", plural equation, S "are
26     independent, since the third" +:+ phrase equation, S
27     "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 3.1 and 3.2 display the review chapter of the lesson plan created manually and using Drasil, respectively.

Code 3.11: Source Code for Forming a Notebook

```

1  mkNB  :: LsnDecl
2  mkNB  = [
3      LearnObj $ LrnObjProg [learnObjContext],
4      Review $ ReviewProg reviewContent,
5      CaseProb $ CaseProbProg caseProbCont,
6      Example $ ExampleProg exampleContent,
7      BibSec
8  ]

```

Figure 3.1: 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:

$$\begin{aligned}
 v &= v' + at & (\text{Eq_rectVel}) \\
 p &= p' + v't + \frac{1}{2}at^2 & (\text{Eq_rectPos}) \\
 v^2 &= (v')^2 + 2a(p - p') & (\text{Eq_rectNoTime})
 \end{aligned}$$

where v' and p' 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 3.2: 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) are as follows:

$$\begin{aligned}
 v &= v' + a't \\
 p &= p' + v't + \frac{a't^2}{2} \\
 v^2 &= v'^2 + 2a'(p - p')
 \end{aligned}$$

where v' and p' 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 to create a lesson plan using Drasil, including the implementation and generated document, we recommend referring to the appendix.

[link](#)

Chapter 4

Conclusion

Every thesis also needs a concluding chapter

Appendix A

Your Appendix

Figure A.1 shows the dependency between modules in **drasil-printers**. The arrow points to the module that is being relied on.

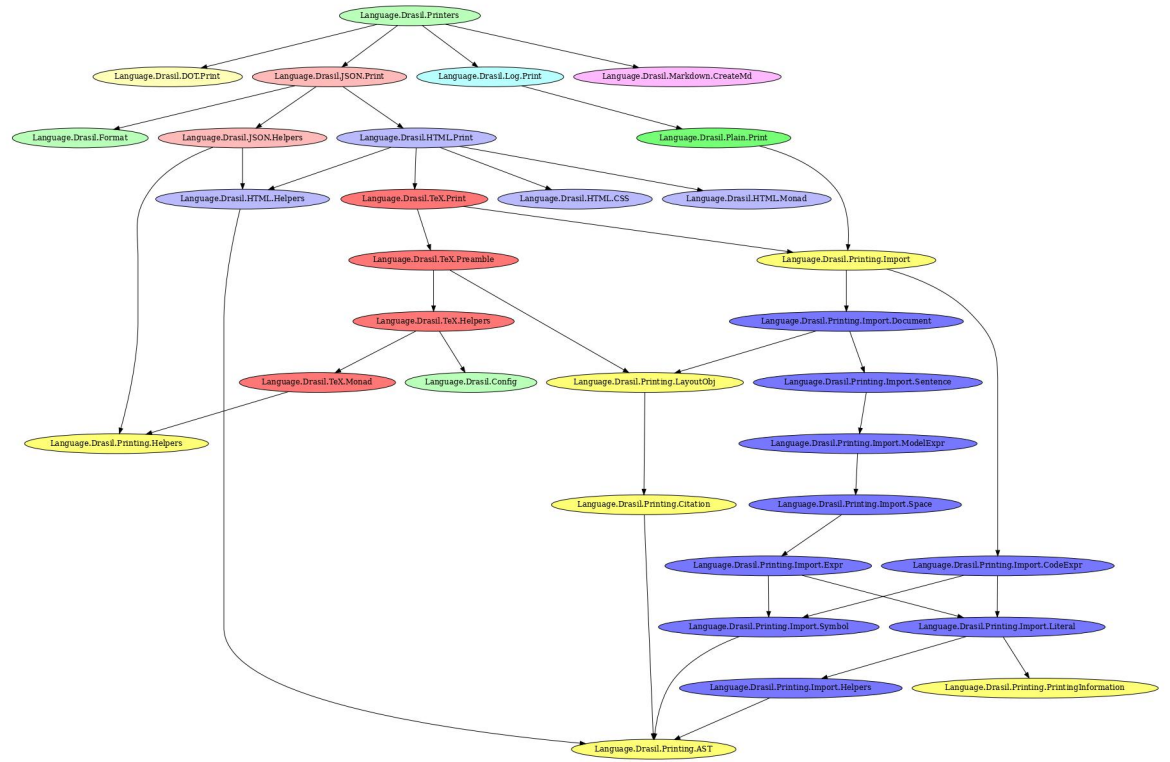


Figure A.1: drasil-printer Dependency Graph

Code A.1: JSON Code of A Notebook Document

```
1  {
2    "cells": [
3      {
4        "cell_type": "markdown",
5        "metadata": {},
6        "source": []
7      }
8    ],
9    "metadata": {
10     "kernelspec": {
11       "display_name": "Python 3",
12       "language": "python",
13       "name": "python3"
14     },
15     "language_info": {
16       "codemirror_mode": {
17         "name": "ipython",
18         "version": 3
19       },
20       "file_extension": ".py",
21       "mimetype": "text/x-python",
22       "name": "python",
23       "nbconvert_exporter": "python",
24       "pygments_lexer": "ipython3",
25       "version": "3.9.1"
26     }
27   },
28   "nbformat": 4,
29   "nbformat_minor": 4
30 }
```


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

- [15] 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. 3).
- [16] GitHub contributors. *The Jupyter Notebook*. URL: <https://jupyter-notebook.readthedocs.io/en/stable/notebook.html> (cit. on p. 4).
- [17] Jupyter Notebook contributors. *Add metadata to your book pages*. URL: <https://jupyterbook.org/en/stable/content/metadata.html> (cit. on p. 13).
- [18] The Haskell Team. *Text.JSON*. URL: <https://hackage.haskell.org/package/json-0.10/docs/Text-JSON.html> (cit. on p. 13).
- [19] 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. 19).
- [20] 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. 19).
- [21] The Drasil Team. *Do we need both LsnDecl and LsnDesc for lesson plan?* URL: <https://github.com/JacquesCarette/Drasil/issues/3308> (cit. on p. 21).
- [22] 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. 26).