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:	xiii, 32

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

List of Figures

List of Tables

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. SRS is a template for designing and documenting scientific computing software requirement decisions created by Smith et al [15]. Drasil is capable of generating SRS in document languages HTML and LaTeX. We are looking to extend the capability of Drasil by generating Jupyter Notebook in Drasil.

add an example

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 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 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

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

Code 2.2: Source Code for Definition of RawContent

```
-- | 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
```

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:

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

[1]Modelling 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)
        ↪ Caption Label
12    | HDiv Tags [LayoutObj] Label
13    | Cell [LayoutObj]
14    | 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: Source Code for Encoding rectVel

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

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

[2]In Drasil, some content types are manipulated into a **Sentence** to form printable **Contents**.
[3]DerivBlock is a type of contents representing a derivation block.

Code 2.5: Pseudocode for Converting rectVel to Sentence

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

-- | Lifts an expression into a Sentence.
E :: ModelExpr -> Sentence

eS :: ModelExpr -> Sentence
eS = E
```

Code 2.6: Source Code for Converting ModelExpr to Contents

```
-- | 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
```

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

[4] Information about a book or its contents is known as metedata. 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 'printLO'
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
     ↪ layoutObs)
6  printLO (HDiv _ layoutObs _) = vcat (map printLO layoutObs)
7  printLO (Paragraph contents) = nbformat empty $$
8    nbformat (stripnewLine (show(pSpec contents)))
9  printLO (EqnBlock contents)  = nbformat mathEqn
10   where
11     toMathHelper (PL g) = PL (\_ -> g Math)
12     mjDelimDisp d = text "$$" <> stripnewLine (show d) <>
         ↪ text "$$"
13     mathEqn = mjDelimDisp $ printMath $ toMathHelper $ TeX.
         ↪ spec contents
14 printLO (Table _ rows r _ _) = nbformat empty $$
15   makeTable rows (pSpec r)
16 printLO (Definition dt ssPs l) = nbformat (text "<br>") $$
17   makeDefn dt ssPs (pSpec l)
18 printLO (List t) = nbformat empty $$ makeList t False
19 printLO (Figure r c f wp) = makeFigure (pSpec r) (pSpec c) (
     ↪ text f) wp
20 printLO (Bib bib) = makeBib bib
21 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.

3,4 link

update the code when it's final

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 .

link to future work

link

Code 2.11: Source Code for Making Metadata

```
    -- | Generate the metadata necessary 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"
    ]
```

Code 2.12: 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 2.13: Source Code for Calling markdownCell

```
    printLO (Cell layoutObs) = markdownCell $ vcat (map
      ↪ printLO 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 3.1 [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

```
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.

[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 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.

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 [21]. 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, as

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)
```

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

```
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

```
-- | 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

```
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
```

where the languages of SRS and lesson plans are located. Table 3.2 summaries the responsibility for each module related to 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 kinematic, 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 discuss how we reproduce the Projectile Motion Lesson created by Dr. Spencer Smith and let Drasil generate the notebook document.

what information is reused from projectile.

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.

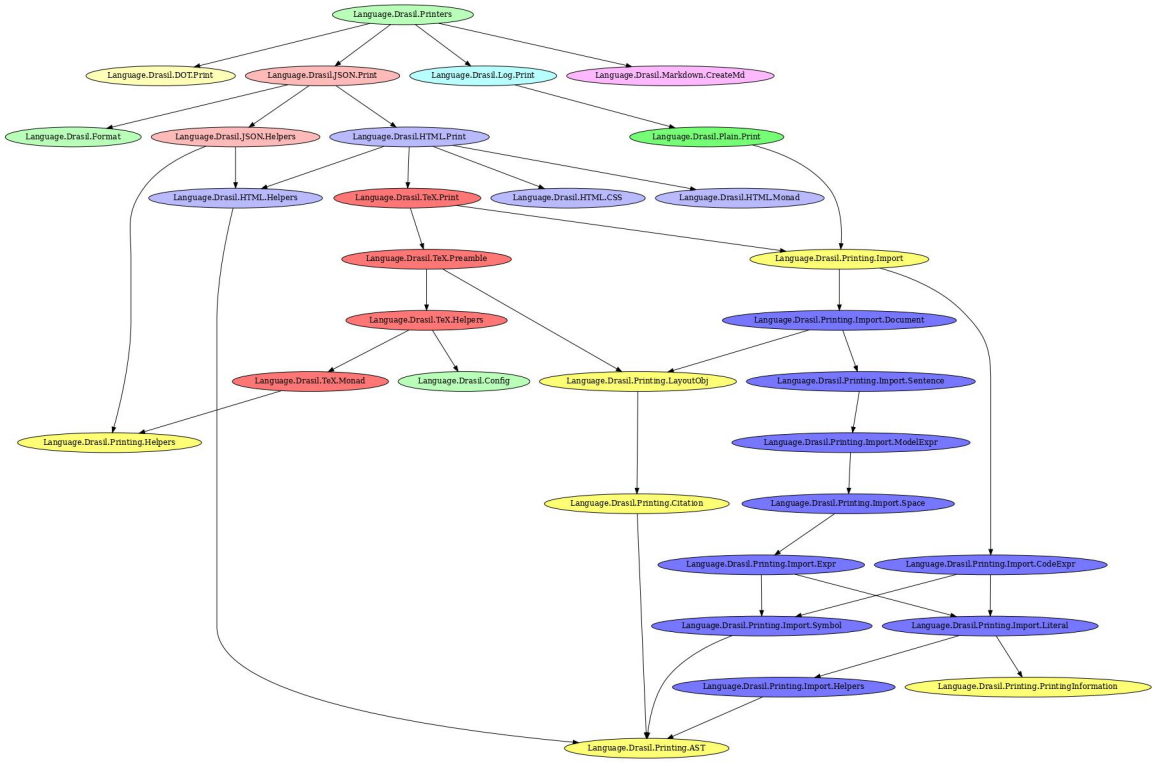

Figure A.1: drasil-printer Dependency Graph

Code A.1: JSON Code of A Notebook Document

```
{
  "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] 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. 3).

[17] Jupyter Notebook contributors. *Add metadata to your book pages*. URL: https://jupyterbook.org/en/stable/content/metadata.html (cit. on p. 12).

[18] The Haskell Team. *Text.JSON*. URL: https://hackage.haskell.org/package/json-0.10/docs/Text-JSON.html (cit. on p. 12).

[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. 18).

[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. 18).

[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. 20).

[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. 25).