Markdown to Question & Test Interoperability

Su Kim

San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Graphics and Human Computer Interfaces Commons, and the Programming Languages and Compilers Commons

Recommended Citation
DOI: https://doi.org/10.31979/etd.3y54-evg5
https://scholarworks.sjsu.edu/etd_projects/1038

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.
Markdown to Question & Test Interoperability

A Project
Presented to
The Faculty of the Department of Computer Science
San José State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Su Kim
December 2021
The Designated Project Committee Approves the Project Titled

Markdown to Question & Test Interoperability

by

Su Kim

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

December 2021

Ben Reed  Department of Computer Science
Thomas Austin  Department of Computer Science
Cassandra Paul  Department of Physics & Astronomy
ABSTRACT

MARKDOWN TO QUESTION & TEST INTEROPERABILITY

by Su Kim

As the classroom setting shifted to a virtual one as a result of Covid-19, numerous software are readily available to accommodate for the change, including Canvas, the online course management system. Canvas has a core feature that allows teachers to generate and administer quizzes for students through their interface, but it does not fully utilize the potential with online exams. The first step to exploring this potential is this project, known as Markdown to Question & Test Interoperability (M2QTI). Based on the QTI specifications, this tool lets users to plan and write quizzes in Markdown format. Combined with Canvas’s ability to import files, M2QTI allows for convenient and organized quiz generation. This project explores the design and implementation of the software, and paves the way for future work.
ACKNOWLEDGMENTS

First, I would like to thank my advisor, Ben Reed, for being an amazing advisor. He did a great job keeping me on task and always gave the right advice when I was stuck.

Next, I would like to thank the members of my team, Archit Jain and Mingyun Kim. This project was a collaborative effort, and without them the overall process would have been very difficult.

Last, but not least, I would like to thank my friends and family. Thanks to them, I managed to get through the stress of grad school.
# TABLE OF CONTENTS

List of Tables ........................................................................................................ viii

List of Figures ........................................................................................................ x

1 Introduction....................................................................................................... 1

2 Background........................................................................................................ 2
  2.1 Canvas ................................................................. 2
  2.2 Markdown ................................................................. 4
  2.3 QTI ........................................................................ 5
  2.4 Text2QTI ................................................................. 7
  2.5 Motivation ................................................................. 7

3 Design ............................................................................................................. 9
  3.1 Compiler Design .......................................................... 9
  3.2 Input ....................................................................... 9
  3.3 The Tree Data Structure .................................................. 10
  3.4 Question Types ............................................................ 12
  3.5 Output ..................................................................... 13

4 Markdown Format .......................................................................................... 15
  4.1 Multiple Choice ........................................................... 15
  4.2 True/False ................................................................. 16
  4.3 Multiple Answers ........................................................... 16
  4.4 Fill in the Blank ........................................................... 17
  4.5 Fill in Multiple Blanks ................................................... 17
  4.6 Multiple Dropdowns ...................................................... 18
  4.7 Matching .................................................................. 18
  4.8 Numerical ................................................................. 19
  4.9 Formula.................................................................... 19
  4.10 File Upload................................................................ 20
  4.11 Essay ...................................................................... 20
  4.12 Group ...................................................................... 21
  4.13 Code Blocks............................................................... 22

5 Parser ............................................................................................................. 23
  5.1 mistletoe ................................................................... 23
  5.2 ParsedQuiz ................................................................ 24
  5.3 ParsedGroup .............................................................. 24
  5.4 ParsedQuestion ........................................................... 25
6 Generator ........................................................................................................... 27
  6.1 Elementree ................................................................................................. 27
  6.2 IMSGenerator() ......................................................................................... 27
  6.3 MetaGenerator() ......................................................................................... 28
  6.4 QuizGenerator() ......................................................................................... 28

7 QTI Profile for Questions...................................................................................... 30
  7.1 Item Overview .............................................................................................. 30
      7.1.1 Item Metadata ....................................................................................... 31
      7.1.2 Presentation .......................................................................................... 32
      7.1.3 Reprocessing ......................................................................................... 33
      7.1.4 Item Feedback ....................................................................................... 34
  7.2 QTI per Question Type ................................................................................. 34
      7.2.1 LID based Question Types ................................................................... 34
      7.2.2 FIB based Question Types ................................................................... 35
      7.2.3 Number based Question Types ............................................................ 36
      7.2.4 FRQ based Questions ............................................................................ 36

8 Future Work ....................................................................................................... 37

9 Conclusion ......................................................................................................... 38

Literature Cited ..................................................................................................... 39
LIST OF TABLES

Table 1. Supported question types. ................................................ 8
LIST OF FIGURES

Fig. 1. Importing Course Content feature. ........................................... 3
Fig. 2. Top: Successfully imported zip. Bottom: Corresponding generated quiz. 3
Fig. 3. Example of Markdown format. ................................................ 4
Fig. 4. Example of a multiple choice question in XML. ........................ 6
Fig. 5. An invalid QTI zip import. .................................................. 7
Fig. 6. The MD2QTI process. ....................................................... 9
Fig. 7. Markdown Quiz as input. .................................................... 10
Fig. 8. UML Class Diagram for Quiz. ............................................. 12
Fig. 9. IMS Content Packaging scope. ........................................... 14
Fig. 10. Quiz title and description. .................................................. 15
Fig. 11. Multiple Choice question with (optional) feedback. ................... 15
Fig. 12. True/False question. ....................................................... 16
Fig. 13. Multiple Answers question with (optional) choice feedback. ........ 16
Fig. 14. Fill in the Blank question. .................................................. 17
Fig. 15. Fill in Multiple Blanks question. ........................................ 17
Fig. 16. Multiple Dropdowns question. .......................................... 18
Fig. 17. Matching question. ....................................................... 18
Fig. 18. Numerical question. ....................................................... 19
Fig. 19. Formula question. ........................................................ 20
Fig. 20. File upload question. ...................................................... 20
Fig. 21. Essay question. ............................................................ 20
Fig. 22. Group of questions. ....................................................... 21
Fig. 23. Questions generated by a Code Block. ................................. 22
Fig. 24. ParsedQuiz inheriting from the Quiz superclass. ...................... 24
Fig. 25. ParsedGroup inheriting from the Group superclass. .................. 25
Fig. 26. ParsedQuestion inheriting from the Question superclass. ............ 26
Fig. 27. Settings for a Canvas quiz. ................................................ 29
Fig. 28. A Multiple Choice question seen in editing mode. ..................... 31
1 INTRODUCTION

As a consequence of the global pandemic and quarantine, the classroom setting has shifted from in-person learning to a virtual classroom setting to maintain social distancing. Furthermore, several aspects of formal education have adapted as a result. In particular, online exams have become the primary form of online assessments.

One important tool for online teaching is Canvas. Canvas is an online course management system, primarily used to support university professors and K-12 teachers in managing their classes [1]. This platform offers a variety of features, including class material, submit assignments, host discussions, and create quizzes.

Canvas has become an integral part of online education, which means online exams play a role as well. However, there is a lot of potential from online quizzes that Canvas doesn’t explore. For example, generating questions via scripts, analyzing data from exam results, and understanding quiz metadata in relation to effectiveness of online education.

The initial step to bridging the gap between Canvas and online exams is called Markdown to Question & Test Interoperability, or MD2QTI for short. Canvas has the ability to import content in multiple formats, which can be used to import settings, files, and quizzes [2]. With this in mind, the Python-based software MD2QTI will allow users, primarily teachers, to easily create quizzes by simply writing questions in Markdown format; the software will automatically convert the Markdown file into a QTI zip file that can be imported to Canvas to automatically generate quizzes.
2 BACKGROUND

2.1 Canvas

Canvas is a web-based learning management system, or LMS for short, developed by Instructure, Inc [1]. Originally named Instructure, the software was founded in 2008 before it was renamed into Canvas. It is used by educational institutions worldwide to support online courses; teachers can create and facilitate online learning materials, while students can utilize said materials and engage in learning. Canvas is also open source [3], which means the source code is publicly available for modification; integration of MD2QTI will be discussed in the Future Works section.

Canvas has become a key tool for education. In 2018, Canvas became the most popular LMS in North America [4]. In 2020, more American states adopted Canvas as their primary LMS in order to address the challenges of the COVID-19 pandemic [5]. Furthermore, a study by Flores et al. shows the impact of the pandemic on the LMS platform [6] by monitoring increased Canvas usage over the past three years. The study also shows initially low assignment submission rates that grew significantly over the years, demonstrating the students ability to adapt to online learning. It concludes by discussing some limitations with the study since the Canvas API is unable to provide the necessary data. To address this, MD2QTI will be the initial step to future works that can enable more insightful research.

Canvas follows the Common Cartridge specifications [7] and offers a variety of features that support course creation and management [1]. Administrative users, such as admins, designers, and instructors, can create courses and integrate assignments, modules, quizzes, discussions, and pages. They can also share files, rubrics, and assignments. Students can enroll in those courses, submit assignments, take quizzes, and participate in forum-like discussions.
This project will make use of Canvas’s ability to import content, which is available for users with administrative roles like teachers. As seen in Fig. 1, teachers can navigate to the Settings page of a course and choose the Import Course Content option on the right pane. From there, teachers can import a variety of relevant content. This includes: previously existing course content, individual assignments and files, and content used in other LMS platforms. This project will focus on the QTI zip file option in order to import quizzes.

Fig. 1. Importing Course Content feature.

Upon a successfully completed QTI zip file import, a quiz with the submitted contents will automatically be created and available in the quiz tab, as seen in Fig. 2. A QTI import with invalid contents, such as unrecognizable XML bindings, will cause an error and no quiz will be generated.

Fig. 2. Top: Successfully imported zip. Bottom: Corresponding generated quiz.
2.2 Markdown

Markdown is a plain text formatting syntax. Developed by John Gruber in 2004, the Markdown syntax was designed to make it as readable as possible. It allows users to write plain text using an easy-to-read, easy-to-write format that can be converted to HTML [8]. Fig. 3 shows an example of Markdown format displayed on a browser window. Furthermore, any users with access to a computer also has access to write plain text, using Notepad for Windows or TextEdit for Mac. With this in mind, Markdown is designated as the primary format for MD2QTI’s input due to its easability, usability, and accessibility.

While there is no official universal specification to implement Markdown, CommonMark has become the unofficial standard to eliminate ambiguity [9]. MD2QTI closely follows the syntax specifications outlined by CommonMark to define the input format. It uses the syntax for lists, block quotes, code fences, and more. The structure for each type of question is discussed in section 4.

Fig. 3. Example of Markdown format.
2.3 QTI

QTI is short for IMS Question and Test Interoperability specification. Developed by IMS Global Learning Consortium (IMS GLC), QTI is a standardized format that represents test content and results that can be used between systems such as LMS, repositories, and scoring/analytic systems [10]. It can be delivered and exchanged among said systems, allowing for interoperability.

The initial draft of the specification (v0.5) was developed in 1999, and it was not until the year 2000 that the first version of QTI (v1.0) was released. The developers, IMS GLC, focused their requirements around quiz providers, quiz takers, and virtual learning environments (VLE). The requirements included the following: ability to provide quiz content to users regardless of VLE, ability to use quiz content from various sources in a single VLE, ability to develop new quiz content consistently, and ability to get quiz results consistently [11]. It is also important that the specifications are extensible and flexible for future needs.

QTI consists of a data model to structure the various types of tests, questions, and answers. All content information is represented by XML data bindings. Fig. 4 shows an example of a multiple choice question represented in XML format. Every part of a question, such as the question, the answer, and the choices, has a corresponding XML binding, along with any necessary metadata such as the 'rcardinality' tag that indicates the number of choices that can be picked.

Each version of QTI is built on top of the previous versions, taking the best practices and improving their interoperability and consistency. One example is QTI v2.2, released in 2015, supporting W3C specifications and ensuring compatibility with technologies such as HTML5 [10]. The most recent version is v3.0, released in 2020.

While Canvas supports versions 1.2 and 2.1 [12], the LMS platform is more compatible with version 1.2 [13]. Further testing supports this; more recent QTI versions
Fig. 4. Example of a multiple choice question in XML.

use updated XML data bindings that Canvas does not recognize, even for types of questions that were previously supported. When this occurs, the Canvas import page will show a yellow completed status (seen in Fig. 5), indicating an error with the imported QTI zip file, and no quiz will be automatically generated. While more recent QTI versions cover more types of questions, such as interactive images, it is unclear whether Canvas support these types or their corresponding XML bindings. MD2QTI will primarily focus on QTI version 1.2 to ensure complete compatibility with Canvas.
Fig. 5. An invalid QTI zip import.

2.4 Text2QTI

Text2QTI is a Python-based software by Geoffrey M. Poore [14], and it serves as inspiration for this project. Similar to MD2QTI, it is used to convert Markdown text files into QTI zip files for various LMS platforms including Canvas. It covers several types of questions, including multiple choice, short answers, and more. Installation and usage is done primarily on the command line.

Despite its usefulness, there are some flaws that MD2QTI covers. The first problem is the lack of support for some types of questions. Table 1 illustrates the difference in the available question types that are supported from both programs. Another problem is poor error handling. While there are some proper error messages, some of them are not quite clear as to why the error occurs. Finally, there is a lack of consistency. Markdown is not used for the quiz metadata (such as the question title, points, and metadata), whereas the rest of the question format is in Markdown.

2.5 Motivation

On paper, this project is simply creating a document with questions on it, but this allows us to explore the potential with online exams. Some topics come to mind: data, convenience, and programmatically generated quizzes.

Compared to in class exams, online exams allow for gathering data with ease. This is possible because the Markdown language provides a distinct format for quizzes. This format can be used to extract information in an organized manner. Further analysis can be
performed with the quiz data structures that are specifically designed for this project. Furthermore, the available data can cover a wide range of topics such as individual student performance, difficulty of quizzes, and how well the class material is covered.

Online exams also allow for convenience. M2QTI is simple and requires less clicking and navigating through the Canvas interface to create a quiz. The user only needs to type out the questions in a specified format on an editor that can be universally found in most personal computers.

Furthermore, our tool can programmatically generate questions, accomplished in two distinct ways. The first way is to create a Group of questions; a specified number of questions from the group can be randomly selected to be generated. The benefit of groups is to be able to create quizzes targeted for different students. The second way is to write code to generate questions that will be appended to the list of questions. For example, a simple for loop can turn 10 lines of code into 100 questions. It also ensures correct answers only. Both of these methods introduce an efficient way of generating a sizable number of questions.
3 DESIGN

3.1 Compiler Design

At a high level, MD2QTI’s design closely matches to that of a compiler. A compiler is essentially a translator; it takes in some code written in one programming language and outputs an equivalent code written in a different language [15]. Fundamentally, a compiler has two distinct phases to accomplish this task. The first part is the front-end, or the Analysis. This consists of a parser to parse the input code, analyzing its syntax and semantics. It creates tokens out the language to create an abstract syntax tree (AST). The second part is the back-end, or the Synthesis. This consists of a code generator, which takes the AST and generate the output code in a target language. This concept is illustrated in Fig. 6 using MD2QTI as an example.

![Fig. 6. The MD2QTI process.](image)

3.2 Input

The first part of the MD2QTI process is the input. The input has to be a text file, that is, a file ending with the TXT extension. This file is to be populated with the quiz content the user wants on Canvas. This includes the quiz title, description, and a list of questions. There is no limit to the number of questions as long as they follow the syntax format defined in section 4. Groups of questions and code blocks to programmatically create
questions are supported as well. An example of the input is shown in Fig. 7 using Notepad.

![Example Quiz Input](example.png)

Fig. 7. Markdown Quiz as input.

The parser is responsible for taking the input and converting it. This is done in the Parser.py file where it employs an open source Markdown parser Python package known as mistletoe [16]. The parser takes the Markdown input and internally generates tokens to render the AST to create an HTML document. This document is then read line by line to allocate the proper values to the attributes of the Quiz and Question data structures.

### 3.3 The Tree Data Structure

An Abstract Syntax Tree is a data structure and a tree-like representation of code [17]. The idea is that, as each line of text is read, the internal parser breaks each part down to tokens. For example, the line `@quiz title: Overview of Question Types` can be broken down into:
MD2QTI will focus on customized block-level tokens to account for the Markdown format. These tokens are turned into the AST seen below. This structure is used to render the HTML like a Document Object Model (DOM), simplifying the parsing process.

```json
{
    "type": "Program",
    "body": [
        {
            "type": "ExpressionStatement",
            "declarations": [
                {
                    "type": "ExpressionStatement",
                    "id": {
                        "type": "Identifier",
                        "name": "quiz title"
                    },
                    "init": {
                        "type": "Literal",
                        "value": "Overview of Question Types"
                    }
                }
            ],
            "kind": "@"
        }
    ],
    "sourceType": "script"
}
```

The Quiz class, along with the Question class, was carefully designed to accurately reflect the major parts of a Canvas quiz. It can also be represented as a tree-like structure. The Quiz class has attributes such as 'title' and 'description', and its children nodes are either Question objects or Group objects, which also has Question objects as children. The Question class also contains similar attributes, and its children nodes, which are also
the leaf nodes, are the Choice objects. A class diagram, illustrated in Fig. 8, depicts the use relationship between Quiz to Question or Group, and Question to Choice. The use relationship can be described as unidirectional association because an object stores another object in a field, but not the other way around [18]. In this case, a single Quiz object stores multiple Question objects in an array, denoted by the 1 and * respectively. Furthermore, a Group object stores multiple Question objects, and each Question Object stores multiple Choice objects.

**Fig. 8. UML Class Diagram for Quiz.**

### 3.4 Question Types

Currently, users are able to create eleven types of questions that are supported by Canvas [19]. During the design phase, common characteristics among various question types were grouped for modularity, promoting code efficiency. Combined with the classifications defined in the QTI section of the Common Cartridge specs [7], this results in four distinct groups in no particular order.

The first group of questions can be classified as Logical Identifier (LID) based questions. This includes: Multiple Choice, True/False, and Multiple Answers. These questions are questions where the user must select one or multiple answers from a
pre-defined set of choices. The format for the question text is not strict as long as it prompts the user to select a radio button. For multiple choice and true/false questions, only one answer can be the correct choice. For multiple answers, one or more answers can be correct.

The next group of questions is classified as Fill in Blank (FIB) based questions. This includes: Fill in the Blank, Fill in Multiple Blank, Multiple Dropdowns, and Matching. The idea behind this group is that the evaluation of the correct answer is based on equality or pattern matching. To expand on this, Fill in the Blank and Fill in Multiple Blank questions require the user to provide one or more response(s) that is/are then checked for equality with the question’s correct answer(s). If the equality holds true, then the submitted response(s) is/are correct. Furthermore, Multiple Dropdown and Matching questions’ evaluation can be described as selecting the appropriate value from a dropdown list for each key, and there are multiple keys. A response is correct when all the keys match with their respective values.

The third group of questions is dubbed as Number based questions. This includes: Numerical and Formula. This group is very straightforward; it involves answering with just numbers, often with calculated precision or specified decimal places. These types of questions are often associated with Mathematics or Physics courses.

The final group of questions is classified as Free Response Questions (FRQ). This includes: File Upload and Essay. For these questions, the user must manually upload a file or type out a response. As a result, there can be varying responses so it is up to the issuing proctor to determine if the submitted response is correct or not.

3.5 Output

The output of MD2QTI is a QTI file, which is quiz related content as a package incorporated into a single file known as a Package Interchange File (PIF). In this case, it is a 'ZIP' file, a way of transporting structured content in a concise delivery format. The
PIF contains two major parts, illustrated in Fig. 9. The first part is the Manifest, an XML document that describes the contents of the package, and it is required for every IMS Content. The manifest file provides details regarding metadata, organizations, resources, and any sub-manifests if needed. The second part is the content. Course materials, assessments, media, and other files go in here.

Fig. 9. IMS Content Packaging scope.

MD2QTI generates the appropriate files and zip them. The zip file includes the Manifest file and a folder that contains the assessment and its metadata as two separate files, for a total of three files. MD2QTI is designed to have a separate generator class for each of these files to promote organization. Furthermore, these are the only required files for a successful Canvas import, as determined by vigilant testing during the design phase. Another form of validation is using the ’export course content’ on Canvas to generate a QTI file for an existing quiz and comparing its contents. One thing to note is that the exported file contains a folder called ’non_cc_assessment’ which is for content that does not adhere to the Common Cartridge specifications; MD2QTI currently does not support such content.
4 MARKDOWN FORMAT

Every setting of a quiz or question must start with the ’@’ symbol, denoting that the proceeding keyword is used to select a setting. A colon must follow the keyword, and any text after determines the value for that keyword/setting. Fig. 10 shows an example of setting the quiz title and quiz description. The following keywords are required for every question: title, points, type, question, and answer. The ‘feedback’ keyword is optional.

Fig. 10. Quiz title and description.

4.1 Multiple Choice

Fig. 11. Multiple Choice question with (optional) feedback.
4.2 True/False

For most question types, lists of choices are denoted with the * symbol, followed by some text. The correct choice(s) is/are marked with a * > symbol. Also note that true/false questions are, by design, multiple choice questions with only two choices.

![Fig. 12. True/False question.](image)

4.3 Multiple Answers

Furthermore, each choice can have its own feedback as seen in Fig. 13. Please note that an indent in Markdown is 4 spaces.

![Fig. 13. Multiple Answers question with (optional) choice feedback.](image)
4.4 Fill in the Blank

For Fill in the Blank questions, users can write any number of correct answers to the prompt, seen in Fig. 14.

![Image of Fill in the Blank question](image)

Fig. 14. Fill in the Blank question.

4.5 Fill in Multiple Blanks

For Fill in Multiple Blanks, users can define a number of “blanks” or variables in the question body, enclosed in square brackets as seen in Fig. 15. In the answers, the user must list each variable with the + symbol, then list possible correct answers as normal with an indent (4 spaces).

![Image of Fill in Multiple Blanks question](image)

Fig. 15. Fill in Multiple Blanks question.
4.6 Multiple Dropdowns

Multiple Dropdowns questions follow a format similar to Fill in Multiple Blanks. The main difference is that there is only one correct choice for each defined variable.

Fig. 16. Multiple Dropdowns question.

4.7 Matching

For Matching questions, the format is similar to Multiple Dropdowns question. Users can add Incorrect Match Options by adding the DISTRACTOR keyword to the list of keys, along with a list of values.

Fig. 17. Matching question.
4.8 Numerical

Numerical questions only require one correct answer. The answer must be composed of the correct digit and a margin of error, separated by a comma.

Fig. 18. Numerical question.

4.9 Formula

Formula questions can be tricky. The Markdown format for this question type is designed in a way to set the values that Canvas requires for this question. Similar to FID based questions, the user must set variables in the question body. In the answers list, the first number of rows must set the Min, Max, and Decimal Places for each variable from the question body. The following row defines the formula used to generate the solution. Finally, the last row sets how many solutions to generate and the margin of error, separated by a comma.
4.10 **File Upload**

Fig. 20. File upload question.

4.11 **Essay**

Fig. 21. Essay question.
4.12 Group

In order to create a group of questions, the user can encompass a list of questions with the 'group: start' keyword at the beginning of the list and 'group: end' at the end of the list. The user can specify how many questions from the list can be selected for generated quiz. The selection is random. The user can also decide the how many points each question is worth from the group, overwriting its previous value. Fig. 22 illustrates three questions in one group, and only two of them is generated on Canvas.

![Question 1 and Question 2](image)

Fig. 22. Group of questions.
4.13 Code Blocks

MD2QTI supports code blocks. For example, users can use a For Loop to generate a question per iteration as seen in Fig. 23.

Fig. 23. Questions generated by a Code Block.
5 PARSER

The role of the parser is to take the Markdown text file as input and convert it to a Python class object that holds all the information to be used by the generator. It contains three classes: ParsedQuestion, ParsedGroup, and ParsedQuiz. As the parser reads the input, the text file is decomposed and all the vital information is contained in their respective classes. Ultimately, the end product of the parser is a structured set of related objects that can be utilized by the generator. This is all done in Parser.py, implemented by SJSU alumni Mingyun Kim.

5.1 mistletoe

mistletoe is a Python based Markdown parser, developed by Mi Yu [16]. It is designated as MD2QTI’s internal Markdown parser for several reasons. First, mistletoe performs incredibly fast. There are benchmarks between other Markdown parsers such as mistune, CommonMark-py, and Python-Markdown, and mistletoe performs nearly at the top. Mistune is marginally faster, but upon careful consideration, mistune does not offer the extensibility that mistletoe boasts, so the performance speed is negligible. Second, mistletoe is CommonMark compliant. mistletoe closely follows these specs so parsing is straightforward, resulting in predictable and well-defined outputs. Finally, mistletoe is extensible. This means that renderers can be customized and new ones can be easily implemented.

MD2QTI uses mistletoe’s HTMLRenderer to read the input and instantiate ParsedQuiz and ParsedQuestion objects. Internally, the HTMLRenderer breaks down each line of the Markdown input into tokens that are used to create an AST. It then traverses through the tree and generates HTML Document. This greatly simplifies the overall process, as it is much easier to read the document in order to allocate variables. Furthermore, mistletoe has been customized to match the Markdown format input defined for MD2QTI. The main change is adding the ‘Setting’ class. As the renderer reads
through the block tokens of the Markdown input, it will return a 'Setting' token when it
detects the @ syntax. This is used to set various attribute values.

5.2 ParsedQuiz

The ParsedQuiz class is designed to hold all related information of the quiz. It is a
subclass of the Quiz superclass, illustrated in Fig. 24. When a ParsedQuiz object is
initialized, mistletoe’s HTMLRenderer processes the input data and sets the object’s
variables. It contains the following attributes:

- **title (string)** - The title of the quiz.
- **description (string)** - The description of the quiz. This is optional.
- **ident (string)** - A unique 32-character hexadecimal string used as the quiz identifier.
  It is randomly generated using UUID [20].
- **assessmentIdent (string)** - A unique 32-character hexadecimal string used to identify
  the assessment’s metadata file. It is randomly generated using UUID. [20].
- **questions (list)** - A list containing a number of ParsedQuestion objects and/or
  ParsedGroup objects.

![Fig. 24. ParsedQuiz inheriting from the Quiz superclass.](image)

5.3 ParsedGroup

A ParsedQuiz object can also contain a ParsedGroup object. While both of these
objects contains a list of ParsedQuestions, the distinction is that questions are populated
in a group and a number of them are randomly selected to be generated. The ParsedGroup
object has an attribute ‘qtype’ hard coded to ’0’. This is an intentional decision in order to
assist the generator process. As the generator iterates through a ParsedQuiz’s questions list, an item is identified by its ‘qtype’ value. When it detects a ‘0’ qtype item, it determines the item is a Group object. It contains the following attributes:

- `self.pick = -1`
- `self.ppq = -1`
- `self.questions = []`
- `self.ident = uuid.uuid4().hex`
- `self.qtype = 0`

- **pick (int)** - The number of randomly selected questions to be generated.
- **ppq (int)** - Acronym for Points Per Question. It overwrites a question’s points value.
- **questions (list)** - A list containing a number of ParsedQuestion objects. It cannot contain another Group.
- **ident (string)** - A unique 32-character hexadecimal string used to identify the ParsedGroup. It is randomly generated using UUID.
- **qtype (int)** - Hard coded to 0 to differentiate from ParsedQuestion objects.

![Diagram of Group and ParsedGroup classes](image)

**Fig. 25.** ParsedGroup inheriting from the Group superclass.

### 5.4 ParsedQuestion

A ParsedQuiz object or a ParsedGroup object contains an array of ParsedQuestion objects. The ParsedQuestion object holds all the necessary information for a question item. Like the ParsedQuiz object, it uses mistletoe’s HTMLRenderer’s parsed Markdown data to set its attributes. However, a ParsedQuestion object may not use all of its attribute values. For example, the ’keys’ list is only used for FIB based questions and not LID based questions. During implementation, the smallest possible number of attributes was considered to promote readability. It contains the following attributes:

- **title (string)** - The title composed of ’md2qti_question’ prefix and a UUID string.
- points (int) - The number of points a question is worth.
- question (string) - The question prompt.
- answers (list) - A list of possible answers.
- keys (list) - A list of key variables found in FIB based questions.
- matchingKey (list) - A list of indices used in conjunction with the list of keys.
- matchingVal (list) - A list of IDs for each value in FID based questions.
- qtype (int) - A number used to retrieve the type of question from an enum.
- feedback (string) - Text that is displayed after a question is answered.
- in_question (Boolean) - Flag that is used to set the proceeding text as the prompt.
- in_answer (Boolean) - Flag that is used to set the proceeding text as answers.

Fig. 26. ParsedQuestion inheriting from the Question superclass.
6 GENERATOR

The output of MD2QTI is a zip file that contains three required files: imsmanifest XML, assessment meta XML, and the assessment content XML. Each file is created from their respective class generator functions in no particular order. Furthermore, each function is fully implemented to produce all the necessary lines. Using a ParsedQuiz object in its parameter, the functions pulls the appropriate data in order to populate some XML bindings and/or decide to omit some. Some XML values are hard coded due to the value being constant for every QTI file.

6.1 Elementree

Elementree is a Python based module that allows for a simple and efficient way of writing XML data [21]. It gets its name from the idea that XML can be represented like a tree due to its hierarchical data format. Elementree is incredibly easy to use in order to build a document. To use it, one can call the `ET.Element()` method to create a root element, and then call `ET.SubElement()` to create all subsequent elements, building a tree-like structure. Parameters of the function calls can be used to define parent elements, namespaces, tags, and attributes. At the end, the function returns the root element.

During implementation, MD2QTI utilized Elementree to write all the XML code to create the necessary documents. An example QTI file was used as reference in order to understand all the XML elements that make up a document. For code modularity, parts of code were refactored to prevent repeat code, but the majority of the elements were embedded directly.

6.2 IMSGenerator()

This function returns the XML data to generate the `imsmanifest.xml` file.

The manifest is a required XML document that describes the package [22]. It contains three sections. The first section is the Metadata section, which describes the manifest as a
whole. It is made up of elements such as 'metadata', 'schema', and various 'imsmd' elements. Implementation of this section involves directly embedded string values, except 'imsmddatetime' which is set when the function is called. The second section is the Organization section, which describes any organizations that were involved in creating the manifest. The value is left blank since no organizations were involved. The third section is the Resources section, which contains references to all of the resources needed for the manifest. It describes the assessment filename and location, along with the assessment’s metadata filename and location. These fields are set based on the ParsedQuiz object’s attribute values.

6.3 MetaGenerator()

This function returns the XML data to generate the assessment\_meta.xml file.

The assessment metadata file is an XML document that describes the various settings for a quiz. Several metadata pertains to conditions about access. For example, the assessment may have a time limit, or lockdown browser must be used to access the quiz. Other metadata can affect quiz interaction. For example, the assessment may not allow for backtracking, or may show feedback for a question. Fig. 27 illustrates the Canvas interface equivalent for such settings.

For the implementation, most of the inhibiting settings are false by default and they are unchanged. The only information that is pulled from the ParsedQuiz object is the quiz title, description, identifiers, and the total number of points possible.

6.4 QuizGenerator()

This function returns the XML data to generate the assessment file. The filename is a concatenation with the string 'm2qti\_assessment_' and the quiz identifier value.

The assessment file is an XML document that describes all the questions of the quiz. Each type of question has its own specific function to return the XML bindings as discussed in section 7. The main idea of this function is to iterate through a ParsedQuiz
object’s list of ParsedQuestion objects, identifies the question type, and calls the respective function while passing in a ParsedQuestion object. If it detects a group of questions, it enters an inner for loop to iterate through the group’s list of questions and performs the same operations.
7 QTI PROFILE FOR QUESTIONS

The assessment file is the primary file that contains the bulk of the assessment. It contains a set of questions in descending order. The root element is `<questestinterop>` and directly inside it is the `<assessment>` element. The `<assessment>` element has two attributes: the 'title' attribute and the 'ident' attribute, and they are set by the ParsedQuiz’s 'title' and 'ident' variable values respectively. This element contains the `<qtimetadata>` element and the `<section ident='root_section'>` element.

The `<qtimetadata>` section contains all the QTI specific metadata that are present in the following form:

```xml
<qtimetadatafield>
  <fieldlabel></fieldlabel>
  <fieldentry></fieldentry>
</qtimetadatafield>
```

The `<fieldlabel>` denotes a quiz setting key, while the `<fieldentry>` denotes the value for the corresponding fieldlabel. For example, a fieldlabel can be 'cc_maxattempts' and the fieldentry can be '1', indicating that the quiz can only be taken at max 1 time. Other fieldlabels include 'timelimit', 'allow late submission', 'scoretype', and more. Currently, MD2QTI can only set one fieldlabel (cc_maxattempt = 1).

7.1 Item Overview

The `<section ident='root_section'>` contains all the quiz questions. Each individual question is represented as an `<item>` element. It has the 'title' attribute and the 'ident' attribute, and they are set by a ParsedQuestion’s 'title' and 'ident' variables respectively. Every `<item>` element follow the same structure:

```xml
<item>
  <itemmetadata> ... </itemmetadata>
</item>
```
This structure is accurately reflected in Fig. 28.

Fig. 28. A Multiple Choice question seen in editing mode.

7.1.1 Item Metadata

The <itemmetadata> element describes the question’s qti metadata. The structure for this element is the same for every question. It has the following structure:

<itemmetadata>
  <qtimetadata>
    <qtimetadatafield>
      <fieldlabel>question_type</fieldlabel>
      <fieldentry>multiple_choice_question</fieldentry>
    </qtimetadatafield>
    <qtimetadatafield>
      <fieldlabel>points_possible</fieldlabel>
      <fieldentry>4</fieldentry>
    </qtimetadatafield>
  </qtimetadata>
</itemmetadata>
The 'question_type' fieldlabel represents the type of question. There are twelve possible entries, set by the ParsedQuestions’s qtype value. The 'points_possible' fieldlabel represents how many points a question is worth, set by the ParsedQuestion’s points value. The 'original_answer_id' represents the identifiers for each possible answer to the question. The respective fieldentry is a list composed of identifiers found in each Choice in the ParsedQuestion’s answers array. This fieldentry may also be empty for some non-LID based questions. Finally, the 'assessment_question_identifier' represents the question’s unique identifier, set by the ParsedQuestion’s ident value.

7.1.2 Presentation

The <presentation> element contains the question’s text and possible answers. The question text is represented with the <material> <mattext> elements. MD2QTI currently supports question text in either plain text or html format. The rest of the <presentation> element varies among question types, but it follows a similar structure:

```xml
<presentation>
  <material>
    <mattext>6-2(2+1) = ?</mattext>
  </material>
  <response_str ident="question_id_1">
    <render_fib>
      <response_label ident="choice1id"/>
      <response_label ident="choice2id"/>
      <response_label ident="choice3id"/>
    </render_fib>
  </response_str>
</presentation>
```
7.1.3 Reprocessing

The scoring and/or the response processing to its question’s answers are decided in the \texttt{<reprocessing>} element. Most questions follow a similar structure:

\begin{verbatim}
<reprocessing>
  <outcomes>
    <decvar minvalue="0" maxvalue="100" varname="SCORE"/>
  </outcomes>
  <respcondition continue="No">
    <conditionvar>
      <varequal respident="response1">choice2id</varequal>
    </conditionvar>
    <setvar action="Set" varname="SCORE">100</setvar>
  </respcondition>
</reprocessing>
\end{verbatim}

The scoring variable can be found in the \texttt{<outcomes>} element, where the value is either 0 for incorrect or 100 for correct. MD2QTI also accounts for partial scores for questions where multiple answers must be submitted, in which case the ’action’ attribute is set to ’ADD’ instead of ’Set’. Determining if a chosen/submitted answer is correct is done in with the \texttt{<conditionvar>} element, which varies among question types. It functions like an if-statement, where if the condition evaluates to true, the score is ”set” to the score.
Furthermore, the `<reprocessing>` element can determine when a feedback, if set, can present itself. It has the following structure:

```xml
<respcondition continue="Yes">
  <conditionvar>Show when a choice is picked</conditionvar>
  <displayfeedback linkrefid="general_fb" />
</respcondition>
```

### 7.1.4 Item Feedback

The `<itemfeedback>` element has the same structure regardless of the question type. However, this element can show up a number of times depending on how much feedback the user decides to incorporate. There is 1 `<itemfeedback>` element per feedback. MD2QTI supports general feedback, and individual choice feedback. It has the following structure:

```xml
<itemfeedback ident="general_fb">
  <material>
    <mattext>Don’t forget PEMDAS!</mattext>
  </material>
</itemfeedback>
```

### 7.2 QTI per Question Type

Other than the metadata structure, every item differs from another based on their question type. As mentioned earlier, some of these questions are grouped based on common characteristics that they share. This continues to hold true for common XML bindings as well.

#### 7.2.1 LID based Question Types

For the presentation section, these questions use the `<response_lid>` element to contain individual choices. The `rcardinality` attribute of this element determines is set to 'Single' for multiple and true/false questions, or 'multiple' for multiple answers questions.
Furthermore, the `<render_choice>` element contains one or more `<response_label>` elements that represent the individual choices.

For the response processing section, the `<conditionvar>` determines the outcome of the question. For multiple choice and true/false questions, it is made of a single `<varequal>` element that checks if the correct identifier was selected. For multiple answers questions, the `<conditionvar>` element checks if one or more `<varequal>` elements are correctly selected, and also checks if the incorrect choice is not selected; both conditions must hold true.

### 7.2.2 FIB based Question Types

For the presentation section, there is a `<response_label>` element for each blank in found in Fill in the Blank or Fill in Multiple Blanks questions. For Multiple Dropdowns, there is a `<response_lid>` element for each set of choices, and each choice is represented by a `<response_label>` element. Similarly, each key in Matching questions is `<response_lid>` element. However, since the keys all share the same pool of possible values, each `<response_lid>` element contains a `<response_label>` element for every value including the Distractor values.

For the response processing section, the `<conditionvar>` element checks for mainly equality. If the submitted response for Fill in Blank and Fill in Multiple Blank questions are equivalent to any of the `<varequal>` element values, then the submitted response is deemed correct. For Multiple Dropdowns and Matching questions, multiple `<conditionvar>` elements can be found, one for each key. Each of these elements check if the selected value matches the `<varequal>` element value to determine the result. Furthermore, these responses were designed to support partial credit for points by splitting 100% by the number of `<conditionvar>` elements. For example, if there are four blanks for a Fill in Multiple Blanks question, then each blank is worth 25% of the question’s point value.
7.2.3 *Number based Question Types*

Since these question types generally require a single number input, the presentation consists of a single `<response_str>` element with an 'rcardinality = Single' attribute. Like a Fill in the Blank question, the response is checked for equality with the `<varequal>` element in the `<conditionvar>` element. If the range or margin is provided, then another condition is that the response must be greater than the `<vargt>` element’s value and less than or equal to the `<varlte>` element’s value.

7.2.4 *FRQ based Questions*

For the presentation section, there’s only a single `<response_str>` element for the essay question. For both essay and file upload questions, the `<conditionvar>` element is empty because the submitted response must be manually checked.
MD2QTI has a lot of potential for future work to be done, both internally and externally. In other words, this project has a lot of room for improvement in terms of implementation and optimization, and also enables new projects or research.

Internally, MD2QTI is not perfect. First of all, the parser and the generator were worked on separately by two different people over the course of two semesters. Although there was plenty of communication in the team, it was inevitable to have some inconsistencies. One example is the naming convention: what the Parser developer calls as "Choices," the Generator developer calls them "Answers." Both of these variable names were used interchangeably, but a consistent name should be decided in the future upon to avoid ambiguity. Furthermore, more optimizations in the code can be made. For example, some attributes were added on a whim due to hasty design decisions regarding complicated question types. Finally, more features can be added to support more functionality. Whether it’s being able to enable more settings or recognize more Markdown syntax, MD2QTI’s structure was designed to be flexible. Regardless, any improvement that can happen that will directly increase the quality of MD2QTI.

Externally, there are plans to integrate MD2QTI into future Canvas-related projects. One project idea is to develop a website for MD2QTI. With this possibility, users will not need to download the software and instead access it directly on their browser. Another project idea is to develop a reverse MD2QTI where it takes a QTI file as input and outputs a Markdown file. The goal behind this is to break down a quiz into a Markdown format, where it is easy to edit and structure. Elementree has the ability to parse XML data [21] as well so this is very possible. In the end, the possibilities are endless.
9 CONCLUSION

The year 2020 and 2021 has proved to be difficult years and the world was forced to adapt to a virtual learning environment. Online education has become prevalent, and online proctoring has to be considered as well. As Canvas usage was increasing across the country, the idea of creating MD2QTi was born in order to make Canvas quiz creation easier. It turns out that there are other great reasons behind this idea too.

MD2QTi is essentially a compiler. For its input, it takes in a Markdown text file, parses it into a tree-like data structure. This data structure is converted to XML data bindings to create a QTI file as output. The output is to be uploaded onto Canvas for automatic quiz generation.

Designing and implementing this project was an amazing collaborative experience. While the design phase was straight forward, the biggest challenge was making sure everything was accounted for. This was further emphasized during implementation, where it was treated more like an experiment to explore the potential of Canvas, Markdown, and QTI. The studies and research done proved to be extremely helpful as well.

Hopefully in the future, Canvas will add more quiz features and question types. MD2QTI is designed so that future work can easily be implemented, and there is no limit to how much improvement can happen.
Literature Cited

[1] Instructure, “What is Canvas?.”

[2] Instructure, “What is the Course Import Tool?.”


[12] Instructure, “How do I import quizzes from QTI packages?.”


[17] D. KUNDEL, “ASTs - What are they and how to use them.”


[19] Instructure, “How do I create a quiz with individual questions?.”


[22] I. Global, “IMS Content Packaging Information Model.”