Exam DSL Project

H. Conrad Cunningham

7 November 2018 (after class)

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this document requires use of a browser that supports the display of MathML. A good choice as of November 2018 is a recent version of Firefox from Mozilla.

Exam DSL Project

Introduction

Few computer science graduates will design and implement a general-purpose programming language during their careers. However, many graduates will design and implement—and all likely will use—special-purpose languages in their work.

These special-purpose languages are often called *domain-specific languages* (or DSLs). For more discussion of DSL concepts and terminology, see the accompanying notes on Domain-Specific Languages.

In this project, we design and implement a simple *internal DSL*. This DSL describes simple "programs" using a set of Haskell algebraic data types. We express a program as an *abstract syntax tree* (AST) using the DSL's data types.

The package first builds a set of functions for creating and manipulating the abstract syntax trees for the exams. It then extends the package to translate the abstract syntax trees to HTML.

Note: A similar project is the SandwichDSL case study.

Building Exam DSL

Suppose Professor Harold Pedantic decides to create a DSL to encode his (allegedly vicious) multiple choice examinations. Since his course uses Haskell to teach programming language organization, he wishes to implement the language processor in Haskell. Professor Pedantic is too busy to do the task himself. He is also cheap, so he assigns us, the students in his class, the task of developing a prototype.

In the initial prototype, we do not concern ourselves with the concrete syntax of the Exam DSl. We focus on design of the AST as a Haskell algebraic data type. We seek to design a few useful functions to manipulate the AST and output an exam as HTML.

First, let's focus on multiple-choice questions. For this prototype, we can assume a question has the following components:

- the text of the question
- a group of several choices for the answer to the question, exactly one of which should be be a correct answer to the question
- a group of tags identifying topics covered by the question

We can state a question using the Haskell data type `Question`, which has a single constructor `Ask`. It has three components—a list of applicable topic tags, the text of the question, and a list of possible answers to the question.

```
type QText    = String
type Tag      = String
data Question = Ask [Tag] QText [Choice] deriving Show
```

We use the type `QText` to describe the text of a question. We also use the type `Tag` to describe the topic tags we can associate with a question.

We can then state a possible answer to the question using the data type `Choice`, which has a single constructor `Answer`. It has two components—the text of the answer and a Boolean value that indicates whether this is a correct answer to the question (i.e. `True`) or not.

```
type AText    = String
data Choice   = Answer AText Bool deriving (Eq, Show)
```

As above, we use the type `AText` to describe the text of an answer.

For example, we could encode the question "Which of the following is a required course?" as follows.

```
Ask ["curriculum"]
    "Which of the following is a required course?"
    [ Answer "CSci 323" False,
```

```
        Answer "CSci 450" True,
        Answer "CSci 525" False ]
```

The example has a single topic tag `"curriculum"` and three possible answers, the second of which is correct.

We can develop various useful functions on these data types. Most of these are left as exercises.

For example, we can define a function `correctChoice` that takes a `Choice` and determines whether it is marked as a correct answer or not.

```
correctChoice :: Choice -> Bool
```

We can also define function `lenQuestion` that takes a question and returns the number of possible answers are given. This function has the following signature.

```
lenQuestion :: Question -> Int
```

We can then define a function to check whether a question is valid. That is, the question must have:

- a non-nil text
- at least 2 and no more than 10 possible answers
- exactly one correct answer

It has the type signature.

```
validQuestion :: Question -> Bool
```

We can also define a function to determine whether or not a question has a particular topic tag.

```
hasTag :: Question -> Tag -> Bool
```

To work with our lists of answers (and other lists in our program), let's define function `eqBag` with the following signature.

```
eqBag :: Eq a => [a] -> [a] -> Bool
```

This is a "bag equality" function for two polymorphic lists. That is, the lists are collections of elements that can be compared for equality and inequality, but not necessarily using ordered comparisons. There may be elements repeated in the list.

Now, what does it mean for two questions to be equal?

For our prototype, we require that the two questions have the same question text, the same collection of tags, and the same collection of possible answers with the same answer marked correct. However, we do not require that the tags or possible answers appear in the same order.

We note that type `Choice` has a derived instance of class `Eq`. Thus we can give an `instance` definition to make `Question` an instance of class `Eq`.

```
instance Eq Question where
-- fill in the details
```

Now, let's consider the examination as a whole. It consists of a title and a list of questions. We thus define the data type `Exam` as follows.

```
type Title = String
data Exam = Quiz Title [Question] deriving Show
```

We can encode an exam with two questions as follows.

```
Quiz "Curriculum Test" [
     Ask ["curriculum"]
         "Which one of the following is a required course?"
         [ Answer "CSci 323" False,
           Answer "CSci 450" True,
           Answer "CSci 525" False ],
     Ask ["language","course"]
         "What one of the following languages is used in CSci 450?"
         [ Answer "Lua" False,
           Answer "Elm" False,
           Answer "Haskell" True ]
     ]
```

We can define function `selectByTags` selects questions from an exam based on the occurrence of the specified topic tags.

```
selectByTags :: [Tag] -> Exam -> Exam
```

The function application `selectByTags tags exam` takes a list of zero or more `tags` and an `exam` and returns an exam with only those questions in which at least one of the given `tags` occur in a `Question`'s tag list.

We can define function `validExam` that takes an exam and determines whether or not it is valid. It is valid if and only if all questions are valid. The function has the following signature.

```
validExam :: Exam -> Bool
```

To assist in grading an exam, we can also define a function `makeKey` that takes an exam and creates a list of `(number,letter)` pairs for all its questions. In a pair, `number` is the problem number, a value that starts with `1` and increases for each problem in order. Similarly, `letter` is the answer identifier, an uppercase alphabetic character that starts with `A` and increases for each choice in order. The function returns the tuples arranged by increasing problem number.

The function has the following signature.

```
makeKey :: Exam -> [(Int,Char)]
```

For the example exam above, `makeKey` should return `[(1,'B'),(2,'C')]`.

Exercise Set A

Define the following functions in a module named `ExamDSL` (in a file named `ExamDSL.hs`).

1. Develop function `correctChoice :: Choice -> Bool` as defined above.
2. Develop function `lenQuestion :: Question -> Int` as defined above.
3. Develop function `validQuestion :: Question -> Bool` as defined above.
4. Develop function `hasTag :: Question -> Tag -> Bool` as defined above.
5. Develop function `eqBag :: Eq a => [a] -> [a] -> Bool` as defined above.
6. Give an `instance` declaration to make data type `Question` an instance of class `Eq`.
7. Develop function `selectByTags :: [Tag] -> Exam -> Exam` as defined above.
8. Develop function `validExam :: Exam -> Bool` as defined above.
9. Develop function `makeKey :: Exam -> [(Int,Char)]` as defined above.

Outputting the Exam as HTML

Professor Pedantic wants to take an examination expressed with the Exam DSL, as described above, and output it as HTML.

Again, consider the following `Exam` value.

```
Quiz "Curriculum Test" [
     Ask ["curriculum"]
         "Which one of the following courses is required?"
         [ Answer "CSci 323" False,
           Answer "CSci 450" True,
           Answer "CSci 525" False ],
     Ask ["language","course"]
         "What one of the following is used in CSci 450?"
         [ Answer "Lua" False,
           Answer "Elm" False,
           Answer "Haskell" True ]
     ]
```

We want to convert the above to the following HTML.

```
<html lang="en">
<body>
<h1>Curriculum Test</h1>
<ol type="1">
<li>Which one of the following courses is required?
<ol type="A">
<li>CSci 323</li>
<li>CSci 450</li>
<li>CSci 525</li>
</ol>
</li>
<li>What one of the following is used in CSci 450?
<ol type="A">
<li>Lua</li>
<li>Elm</li>
<li>Haskell</li>
</ol>
</li>
</ol>
</body>
</html>
```

This would render in a browser something like the following.

Curriculum Test

1. Which one of the following courses is required?
   A. CSci 323
   B. CSci 450
   C. CSci 525
2. What one of the following is used in CSci 450?
   A. Lua
   B. Elm
   C. Haskell

Professor Pedantic developed a module of HTML template functions named `SimpleHTML` to assist us in this process. (See file `SimpleHTML.hs`.)

A function application `to_html lang content` wraps the `content` (HTML in a string) inside a pair of HTML tags `<html>` and `</html>` with `lang` attribute set to `langtype`, defaulting to `English` (i.e. `"en"`). This function and the data types are defined in the following.

```
type HTML     = String
data LangType = English | Spanish | Portuguese | French
                deriving (Eq, Show)
langmap = [ (English,"en"), (Spanish,"es"), (Portuguese,"pt"),
            (French,"fr") ]
```

```
to_html :: LangType -> HTML -> HTML
to_html langtype content =
    "<html lang=\"" ++ lang ++ "\">" ++ content ++ "</html>"
        where lang = case lookup langtype langmap of
                        Just l  -> l
                        Nothing -> "en"
```

For the above example, the `to_html` function generates the the outer layer:

```
<html lang="en"> ... </html>
```

Function application `to_body content` wraps the `content` inside a pair of HTML tags `<body>` and `</body>`.

```
to_body :: HTML -> HTML
to_body content = "<body>" ++ content ++ "</body>"
```

Function application `to_heading level title` wraps string `title` inside a pair of HTML tags `<hN>` and `</hN>` where `N` is in the range 1 to 6. If `level` is outside this range, it defaults to the nearest valid value.

```
to_heading:: Int -> String -> HTML
to_heading level title = open ++ title ++ close
    where lev   = show (min (max level 1) 6)
          open  = "<h"  ++ lev ++ ">"
          close = "</h" ++ lev ++ ">"
```

Function application `to_list listtype content` wraps the `content` inside a pair of HTML tags `<ul>` and `</ul>` or `<ol>` and `</ol>`. For `<ol>` tags, it sets the `type` attribute based on the value of the `listtype` argument.

```
data ListType = Decimal | UpRoman | LowRoman
              | UpLettered | LowLettered | Bulleted
                deriving (Eq, Show)

to_list :: ListType -> HTML -> HTML
to_list listtype content = open ++ content ++ close
    where
        (open,close) =
            case listtype of
                Decimal     -> ("<ol type=\"1\">", "</ol>")
                UpRoman     -> ("<ol type=\"I\">", "</ol>")
                LowRoman    -> ("<ol type=\"i\">", "</ol>")
                UpLettered  -> ("<ol type=\"A\">", "</ol>")
                LowLettered -> ("<ol type=\"a\">", "</ol>")
                Bulleted    -> ("<ul>",            "</ul>")
```

Finally, function application `to_li content` wraps the `content` inside a pair of HTML tags `<li>` and `</li>`.

```
to_li :: HTML -> HTML
to_li content = "<li>" ++ content ++ "</li>"
```

By importing the `SimpleHTML` module, we can now develop functions to output an `Exam` as HTML.

If we start at the leaves of the `Exam` AST (i.e. from the `Choice` data type), we can define a function `choice2html` function as follows in terms of `to_li`.

```
choice2html :: Choice -> HTML
choice2html (Answer text _) = to_li text
```

Using `choice2html` and the `SimpleHTML` module, we can define `question2html` with the following signature.

```
question2html :: Question -> HTML
```

Then using `question2html` and the `SimpleHTML` module, we can define `question2html` with the following signature.

```
exam2html :: Exam -> HTML
```

Note: These two functions should add newline characters to the HTML output so that they look like the examples at the beginning of the "Outputting" section. Similarly, it should not output extra spaces. This both makes the string output more readable and makes it possible to grade the assignment using automated testing.

For example, the output of `question2html` for the first `Question` in the example above should appear as the following when printed with the `putStr` input-output command.

```
<li>Which one of the following courses is required?
<ol type="A">
<li>CSci 323</li>
<li>CSci 450</li>
<li>CSci 525</li>
</ol>
```

In addition, you may want to output the result of `exam2html` to a file to see how it displays in a browser a particular `exam`.

```
writeFile "output.html" $ exam2html exam
```

Exercise Set B

Add the following functions to the module `ExamDSL` developed in Exercise Set A.

1. Develop function `question2html :: Question -> HTML` as defined above.
2. Develop function `exam2html :: Exam -> HTML` as defined above.

Source Code

- `ExamDSL_base.hs` is the skeleton to flesh out for a solution to this project.
- `SimpleHTML.hs` is the module of HTML string templates.

Acknowledgements

I developed this project description in Fall 2018 motivated by the Sandwich DSL project and a set of questions I gave on an exam in the past.

I maintain this document as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed. The HTML version of this document may require use of a browser that supports the display of MathML.

References

TODO

Concepts

TODO