Sandwich DSL Project (Scala)

16 April 2022

Contents

Sandwich DSL Project (Scala) 1
  Project Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
  Developing the Sandwich DSL . . . . . . . . . . . . . . . . . . . . . . 2
  Sandwich DSL exercise set A . . . . . . . . . . . . . . . . . . . . . . . 4
  Compiling the Program for the SueChef Controller . . . . . . . . . . . 6
  Sandwich DSL Exercise Set B . . . . . . . . . . . . . . . . . . . . . . . 7
  Sandwich DSL Source Code . . . . . . . . . . . . . . . . . . . . . . . . . 7
  Acknowledgements . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
  Concepts . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
  References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

Copyright (C) 2014, 2016, 2017, 2018, 2019, 2022 H. Conrad Cunningham
Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser
that supports the display of MathML. A good choice as of April 2022 is a recent
version of Firefox from Mozilla.

Sandwich DSL Project (Scala)

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
In this case study, we design and implement a simple internal DSL [1]. This DSL describes simple “programs” using a set of Scala algebraic data types. We express a program as an abstract syntax tree [1] using the DSL’s data types.

In this project, we first build a package of functions for creating and manipulating the abstract syntax trees. We then extend the package to translate the abstract syntax trees to a sequence of instructions for a simple “machine”.

**Developing the Sandwich DSL**

Suppose Emerald de Gassy, the owner of the Oxford-based catering business Deli-Gate, hires us to design a domain-specific language (DSL) for describing sandwich platters. The DSL scripts will direct Deli-Gate’s robotic kitchen appliance SueChef (Sandwich and Utility Electronic Chef) to assemble platters of sandwiches.

In discussing the problem with Emerald and the Deli-Gate staff, we discover the following:

- A sandwich platter consists of zero or more sandwiches. (Zero? Why not! Although a platter with no sandwiches may not be a useful, or profitable, case, there does not seem to be any harm in allowing this degenerate case. It may simplify some of the coding and representation.)
- Each sandwich consists of layers of ingredients.
- The categories of ingredients are breads, meats, cheeses, vegetables, and condiments.
- Available breads are white, wheat, and rye.
- Available meats are turkey, chicken, ham, roast beef, and tofu. (Okay, tofu is not a meat, but it is a good protein source for those who do not wish to eat meat. This is a college town after all.)
- Available cheeses are American, Swiss, jack, and cheddar.
- Available vegetables are tomato, lettuce, onion, and bell pepper.
- Available condiments are mayo, mustard, relish, and Tabasco. (Of course, this being the South, the mayo is Blue Plate Mayonnaise and the mustard is a Creole mustard.)

Let’s define this as an internal DSL—in particular, by using a relatively deep embedding [1].

What is a sandwich? . . . Basically, it is a stack of ingredients.

Should we require the sandwich to have a bread on the bottom? . . . Probably. . . . On the top? Maybe not, to allow “open-faced” sandwiches. . . . What can
the SueChef build? ... We don’t know at this point, but let’s assume it can
stack up any ingredients without restriction.

For simplicity and flexibility, let’s define a Scala data type **Sandwich** to model
sandwiches. It wraps a possibly empty list of ingredient layers. *We assume the
head of the list to be the layer at the top of the sandwich.*

```scala
case class Sandwich(sandwich: List[Layer])
```

Note: In this case study, we implement Scala algebraic data type constructors
(i.e., product types) as *case class* or *case object* entities. We implement
union types using a *sealed trait* with subtypes for the variants.

Data type **Sandwich** gives the specification for a sandwich. When “executed”
by the SueChef, it results in the assembly of a sandwich that satisfies the
specification.

As defined, the **Sandwich** data type does not require there to be a bread in
the stack of ingredients. However, we add function **newSandwich** that starts a
sandwich with a bread at the bottom and a function **addLayer** that adds a new
ingredient to the top of the sandwich. We leave the implementation of these
functions as exercises.

```scala
def newSandwich(b: Bread): Sandwich
def addLayer(s: Sandwich)(x: Layer): Sandwich
```

Ingredients are in one of five categories: breads, meats, cheeses, vegetables, and
condiments.

Because both the categories and the specific type of ingredient are important, we
choose to represent both in the type structures and define the following types. A
value of type **Layer** represents a single ingredient. Type **Layer** has five variants
(subtypes) **Bread**, **Meat**, **Cheese**, **Vegetable**, and **Condiment**. Each of these
variants itself has several variants. For example, **Bread** has variants (subtypes)
**White**, **Wheat**, and **Rye**.

```scala
sealed trait Layer

sealed trait Bread extends Layer
  case object White extends Bread
  case object Wheat extends Bread
  case object Rye extends Bread

sealed trait Meat extends Layer
  case object Turkey extends Meat
  case object Chicken extends Meat
  case object Ham extends Meat
  case object RoastBeef extends Meat
  case object Tofu extends Meat
```
sealed trait Cheese extends Layer
  case object American extends Cheese
  case object Swiss extends Cheese
  case object Jack extends Cheese
  case object Cheddar extends Cheese

sealed trait Vegetable extends Layer
  case object Tomato extends Vegetable
  case object Onion extends Vegetable
  case object Lettuce extends Vegetable
  case object BellPepper extends Vegetable

sealed trait Condiment extends Layer
  case object Mayo extends Condiment
  case object Mustard extends Condiment
  case object Ketchup extends Condiment
  case object Relish extends Condiment
  case object Tabasco extends Condiment

We need to be able to compare ingredients for equality and convert them to strings. Because the automatically generated definitions are appropriate, we do not need to do anything further.

We will need to provide an appropriate definition of equality for Sandwich because the default element-by-element equality of lists does not seem to be the appropriate equality comparison for sandwiches.

To complete the model, we define type Platter to wrap a list of sandwiches.

    case class Platter(platter: List[Sandwich])

We also define functions newPlatter to create a new Platter and addSandwich to add a sandwich to the Platter. We leave the implementation of these functions as exercises.

    def newPlatter: Platter
    def addSandwich(p: Platter)(s: Sandwich): Platter

Sandwich DSL exercise set A

Please put these functions in a Scala module SandwichDSL. You may use functions defined earlier in the exercises to implement those later in the exercises.

1. Define and implement the Scala functions newSandwich, addLayer, newPlatter, and addSandwich described above.

2. Define and implement the Scala query functions below that take an ingredient (i.e. Layer) and return true if and only if the ingredient is in the specified category.
def isBread(x: Layer): Boolean
def isMeat(x: Layer): Boolean
def isCheese(x: Layer): Boolean
def isVegetable(x: Layer): Boolean
def isCondiment(x: Layer): Boolean

3. Define and implement a Scala function noMeat that takes a sandwich and returns true if and only if the sandwich contains no meats.

def noMeat(x: Sandwich): Boolean

4. According to a proposed City of Oxford ordinance, in the future it may be necessary to assemble all sandwiches in Oxford Standard Order (OSO): a slice of bread on the bottom, then zero or more meats layered above that, then zero or more cheeses, then zero or more vegetables, then zero or more condiments, and then a slice of bread on top. The top and bottom slices of bread must be of the same type.

Define and implement a Scala function inOSO that takes a sandwich and determines whether it is in OSO and another function intoOSO that takes a sandwich and a default bread and returns the sandwich with the same ingredients ordered in OSO.

def inOSO(s: Sandwich): Boolean
def intoOSO(s: Sandwich)(defaultbread: Bread): Sandwich

Hint: Remember library functions like dropWhile.

Note: It is impossible to rearrange the layers into OSO if the sandwich does not include exactly two breads of the same type. If the sandwich does not include any breads, then the default bread type (second argument) should be specified for both. If there is at least one bread, then the bread type nearest the bottom can be chosen for both top and bottom.

5. Assuming that the price for a sandwich is the base price plus the sum of the prices of the individual ingredients, define and implement a Scala function priceSandwich that takes a price list, a base price, and a sandwich and returns the price of the sandwich.

def priceSandwich(pl: List[(Layer,Int)], base: Int)(s: Sandwich): Int

Use the following price list as a part of your testing:

val prices = List(
  (White,20),(Wheat,30),(Rye,30),
  (Turkey,100),(Chicken,80),(Ham,120),(RoastBeef,140),(Tofu,50),
  (American,50),(Swiss,60),(Jack,60),(Cheddar,60),
  (Tomato,25),(Onion,20),(Lettuce,20),(BellPepper,25),
  (Mayo,5),(Mustard,4),(Ketchup,4),(Relish,10),(Tabasco,5)
)
6. Define and implement a Scala function `eqSandwich` that compares two sandwiches for equality.

What does equality mean for sandwiches? Although the definition of equality could differ, you can use “bag equality”. That is, two sandwiches are equal if they have the same number of layers (zero or more) of each ingredient, regardless of the order of the layers.

```scala
def eqSandwich(sl: Sandwich)(sr: Sandwich): Boolean```

Compiling the Program for the SueChef Controller

In this section, we look at compiling the `Platter` and `Sandwich` descriptions to issue a sequence of commands for the SueChef’s controller.

The SueChef supports the special instructions that can be issued in sequence to its controller. The algebraic data type `SandwichOp` below represents the instructions.

```scala
sealed trait SandwichOp
case object StartSandwich extends SandwichOp
case object FinishSandwich extends SandwichOp
case class AddBread(bread: Bread) extends SandwichOp
case class AddMeat(meat: Meat) extends SandwichOp
case class AddCheese(cheese: Cheese) extends SandwichOp
case class AddVegetable(vegetable: Vegetable) extends SandwichOp
case class AddCondiment(condiment: Condiment) extends SandwichOp
case object StartPlatter extends SandwichOp
case object MoveToPlatter extends SandwichOp
case object FinishPlatter extends SandwichOp
```

Note: You may find the builtin Scala methods `isInstanceOf` and `asInstanceOf` helpful for use of the above.

We also define the type `Program` to represent the sequence of commands resulting from compilation of a `Sandwich` or `Platter` specification.

```scala
case class Program(program: List[SandwichOp])```

The flow of a program is given by the following pseudocode:

```
StartPlatter
for each sandwich needed
  StartSandwich
  for each ingredient needed
    Add ingredient on top
    FinishSandwich
  MoveToPlatter
FinishPlatter```

Consider a sandwich defined as follows:
Sandwich(List(Rye, Mayo, Swiss, Ham, Rye))

The corresponding sequence of SueChef commands would be the following.
List(StartSandwich, AddBread(Rye), AddMeat(Ham), AddCheese(Swiss),
AddCondiment(Mayo), AddBread(Rye), FinishSandwich, MoveToPlatter)

Sandwich DSL Exercise Set B

1. Define and implement a Scala function `compileSandwich` to convert a
sandwich specification into the sequence of SueChef commands to assemble
the sandwich.

   ```scala
def compileSandwich(s: Sandwich): List[SandwichOp]
   ```

2. Define and implement a Scala function `compile` to convert a platter specification into the sequence of SueChef commands to assemble the sandwiches on the platter.

   ```scala
def compile(p: Platter): Program
   ```

Sandwich DSL Source Code

The Scala source code for this case study is in the file:

- SandwichDSL_base.scala

Acknowledgements

I devised the first version of the Sandwich DSL problem for a question on a
take-exam in the Lua-based, Fall 2013 offering of CSci 658 (Software Language
Engineering). I subsequently developed a full Haskell-based case study for the
Fall 2014 offering of CSci 450 (Organization of Programming Languages). I then
converted the case study to use Scala for the Spring 2016 offering of CSci 555
(Functional Programming).

In Spring and Fall 2017, I converted case study document from HTML to Pandoc
Markdown and updated it for use in the Haskell-based, Fall 2017 offering of CSci
450. In Fall 2018, I updated the Haskell case study to be more compatible with
the ELIFP textbook materials.

In Spring 2018, I recreated this separate Scala-based version of the case study
by combining aspects of Haskell-based version with the Scala-based version from
Spring 2016. In Spring 2019, I further updated the materials to use the current
approach to formatting.

I retired from the full-time faculty in May 2019. As one of my post-retirement
projects, I am continuing work on possible textbooks based on the course
materials I had developed during my three decades as a faculty member. In
January 2022, I began refining the existing content, integrating separately
developed materials together, reformatting the documents, constructing a unified bibliography (e.g., using citeproc), and improving my build workflow and use of Pandoc.

I maintain this chapter as text in Pandoc’s dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

**Concepts**

**TODO**

**References**