University of Groningen

Bachelor Project

Corel: A DSL for Cooking Recipes

Authors:
Auke Roorda *(s2973782)*

Supervisors:
prof. dr. T. van der Storm
dr. V. Andrikopoulos

August 17, 2021

Abstract

A domain-specific language (DSL) provides an accesible way to write domain knowledge and procedures. However, none exists that can accurately describe a cooking recipes content. We aim to develop this DSL. We analysed the structure of cooking recipes using the feature-oriented domain analysis (FODA) method. This yields a feature diagram, which is used in the language design process. The DSL, named Corel, is implemented in Rascal. It enables understanding of and computation with ingredients, and can construct a nutrition label for the recipe. We found that the DSL is able to express each the features of a recipe we focussed on.

Contents

1 Introduction

Domain-specific languages (DSLs) are widely used across all programming domains, and are studied in the area of Software Language Engineering. An example of a prevalent DSL is HyperText Markup Language (HTML), which is used to describe the structure of webpages. Unlike a General-purpose Language (GPL), a DSL is aimed at a specific domain, trading generality for expressiveness [14].

1.1 Goals

The aim of this project is to design a DSL for cooking recipes. This requires an analysis of the features of cooking recipes, which answers the question *What are the common aspects of a recipe, and in which aspects do they vary?*. With these features, a design will be created and implemented in Rascal [9], resulting in an environment in which the cooking recipe DSL can be written. Another goal is to analyse the *well-formedness properties of recipes*, and explore *which computations can be done with a Cooking Recipe DSL.*

1.2 Methodology

First, the domain of cooking recipes is analysed using the feature-oriented domain analysis (FODA) method, as described by Kyo C. Kang et al. in their paper *Feature-Oriented Domain Analysis (FODA) Feasibility Study* [12]. This method is applied on a sample of the Food.com cooking recipe dataset [6]. From there, we continued by implementing the DSL in the metaprogramming language Rascal. After this, a proof-of-concept compiler was constructed, to showcase the computations that can be done with the DSL. The expressiveness of the DSL was evaluated by converting a control group of recipes.

2 Domain Analysis

To aid the design process, we analysed the domain of textual cooking recipes. A subset of twenty recipes is sampled from a dataset of `Food.com` recipes [6]. They are then analysed using the FODA method.

2.1 FODA Method

The FODA method is a framework for the identification of prominent and distinctive features of software systems in a domain. It is evaluated by Kyo C. Kang et al. in their paper *Feature-Oriented Domain Analysis (FODA) Feasibility Study* [12]. Features are the attributes of a system that directly affect end-users, and can be used by end-users to choose between different applications within a domain. Some features are more apparent, such as the different *capabilities* of an application, and others less so, such as the under the hood *implementation techniques* used by an application. These features can be organized in a feature diagram, to illustrate how something is decomposed. An example from Kyo C. Kang et al. can be seen in figure 1.

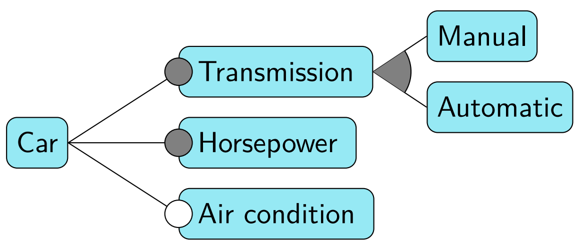

Figure 1: Feature diagram of a car, simplified example from *Feature-Oriented Domain Analysis (FODA) Feasibility Study* [12]

Figure 1 legend: A gray dot indicates a mandatory feature, and a white dot indicates an optional feature. A circular sector between edges mark the child nodes as disjunct.

In figure 1 we can see the features of a car: two of them are mandatory (the transmission and horsepower), and one is optional (air condition). The transmission itself is either a manual transmission, or an automatic transmission. When buying a car, this graph can be used to see the features on which you can base your choice.

2.1.1 Applying FODA on recipes

Domain analysis is usually applied to analyze a set of related software systems, to investigate in which way they vary, and which aspects they have in common. In this project it is used to analyse textual recipes instead of software systems. The aim is to construct a feature diagram for the features of a textual recipe. With this different application, we adjusted the focus as well, to the expressive capabilities of recipes, e.g. the different semantic elements that appear in a recipe, such as a phrase stating an *action* or a *target state*.

About the *operating environment*: with the FODA method, this is aimed at analysing the different hardware and interfaces of each system on which the software is ran. Between the recipes in the dataset, there is no difference in operating environment; they are all displayed on an equal webpage, without any interactivity. We will however analyse the interfacing capabilities of the DSL later.

An analysis of the *application domain technology* that occurs within cooking recipes is skipped, and considered outside the scope of this project. It would provide insight into different ways of executing the same instruction in a recipe. An instruction like *"Cook the rice"* can be executed in different ways, yielding slightly different results, but can also require different ingredients. Understanding this would allow for substituting these methods in a recipe, and updating the required ingredients with it. These *domain technologies* are usually not listed in recipes however, but can be found as separate guides.

2.2 Feature analysis

The FODA method uses different modelling primitives to capture the *abstractions* of functionalities and architecture designs within a system. Aggregation (abstracting a collection of units into a new unit) and generalization are used to capture the commonalities and differences between instances of the domain applications [12]. In this project we analyse these variations between the recipes in the dataset. As a starting point for this analysis, we looked at the type of data that was provided in the dataset, using the descriptions of the columns. These could be categorized in three groups:

1. Metadata, describing either `Food.com` metadata (`RecipeId`, `AuthorId`, etc.), or metadata about the recipe itself (`Name`, `CookTime`, `Description`, `Yield`, etc.).

2. Ingredients, describing the quantities, ingredient name and possible preparations.

3. Instructions, which state the actions for the recipe.

Problems with the dataset Note that during this analysis, it was discovered that the dataset is broken, in multiple ways:

1. The dataset has the columns `RecipeIngredientQuantities` and `RecipeIngredientParts` stored separately. These both contain an array of values, which are supposed to pairwise describe the quantity and ingredient name of each ingredient. However, there are values missing in both lists. If an ingredient is listed without a quantity, such as *A pinch of salt*, then from that point on, the lists do not align anymore.

2. Quite a lot of the ingredient specification is missing in the `RecipeIngredientParts` column. A recipe stating *1 1/2 cups thinly sliced leeks (about 2 medium leeks)* is split up incorrectly: the `RecipeIngredientParts` column just stores the value `leeks`, missing the surrounding text.

3. Only the ingredient names for which `Food.com` has a webpage are stored in the `RecipeIngredientParts` column. This caused the ingredient description *cups reduced-sodium fat-free chicken broth* to not appear in the dataset at all, but its quantity `4` was still stored in the quantities column, again causing a mis-alignment of the data. This also goes wrong when an ingredient states an alternative, as in *1 cup uncooked arborio rice or 1 cup other medium grain rice*. There are pages for both *arborio rice* and *medium grain rice*, causing both rices to be stored in the `RecipeIngredientParts` column, again causing a mis-alignment.

To solve this issue, we repaired each of the sampled recipes, by using the recipes text as found on `Food.com`, and continued our analysis with the repaired dataset.

2.2.1 Analysing the Metadata

The metadata provides extra data about the recipe, that is not required when following the recipe. These are the key units we found in the dataset:

1. `Name`: Each recipe has a name, usually describing the outcome of the recipe, such as *Ham Ristotto With Sugar Snap Peas*. A name is used as an identifier for a recipe, and is mandatory for each one.

2. `Servings` and `Yield`: stating what you will have when completing the recipe. `Servings` is followed by a number or range, indicating how many people can eat from this recipe, and `Yield` is accompanied by its own unit of measure, as in *Yield: 6 sandwiches*. These values are not always present, and even though they can be very informative when choosing a recipe, they are optional, and are not required when executing a recipe.

3. `Desciption`: Contains varied prose: suggestions for other recipes that go along well, a personal anecdote or just a placeholder text from `Food.com`.

4. `Cooking time`, `Preparation time` and `Total time`: Optional indications of how much time (the parts of) the cooking process takes.

5. `Nutrients`: This is the nutritional information about the recipe (calories, protein, etc.).

The decomposition of a recipes description is not part of this project; sometimes it contains alternatives for ingredients or notes for the instructions, but these should not appear in the description.

2.2.2 Analysing the Ingredients

Each recipe contains a list of ingredient specifications. Comparing the ingredients allowed us to distill the commonalities and differences between them. Here are some samples from the dataset and how they are decomposed:

1. *"1 lime, zest of, finely grated"*: This contains a value *1* without unit of measure (a count), an ingredient name *lime, zest of*, and a preparation *finely grated*.

2. *"2 lbs Polish sausage or 2 lbs smoked sausage"*: Here we see a value *2*, followed by a unit of measure *lbs*. Then an ingredient name *Polish Sausage*. New here is the indication of an alternative ingredient *or 2 lbs smoked sausage*.

3. *"black pepper (to taste)"*: Here we see just an ingredient name *black pepper* and a note *(to taste)*.

By combining related units, such as *Unit of measure* and *Value* into a generalized new concept *Quantity*, a hierarchy of *consists-of* relationships is created. These are considered the features of which an ingredient specification consists. Both their presence and content can be important for following the recipe to the end-user.

2.2.3 Analysing the Instructions

A similar approach is taken for analysing the instructions of the recipe. Let us have a look at some examples from the dataset:

1. *"Bring broth to a simmer in a medium saucepan (do not boil). Keep warm."* This instruction is disected as follows:

   (a) This states an action: *bring to a simmer*

   (b) for an ingredient: *broth*, referring to the ingredient *reduced-sodium fat-free chicken broth*

   (c) and specifies an appliance: *in a medium saucepan*

   (d) and a note: *(do not boil)*

   (e) Finally, the second sentence states another action: *Keep warm.* It is implied that this is meant for the most recently referred to object, the broth of the previous sentence.

2. *"Cook peas in boiling water 2 minutes or until crisp-tender. Drain and rinse with cold water; drain"*

   (a) Again, an action is specified *Cook in boiling water*

(b) and an ingredient is referred to *peas*, using the identifying part of the ingredients name, without any of the adjectives from the specification *sugar snap peas*.

(c) For this action, a target state *until crisp-tender* and a guideline *2 minutes* are specified.

(d) Then again, actions are specified with an implied object: *Drain and rinse with cold water; drain*

By decomposing the instructions into conceptual units (action, ingredient reference, target state) a framework for the contents of cooking recipes is created.

2.2.4 Constructing the Feature Diagram

A feature diagram contains the standard features of a system within the domain [12]. The relationship between nodes is a *consists-of* relationship, e.g. a *recipe consists-of metadata, ingredients and instructions*. The feature diagram in figure 2 is derived from analyzing twenty sampled recipes from the dataset.

Figure 2: Feature diagram of cooking recipes

Legend for figure 2: A gray dot indicates a mandatory feature, and a white dot indicates an optional feature. A circular sector between edges mark the child nodes as disjunct. Nodes in orange are features which are not implemented in the DSL (see section 3.1.2).

In figure 2 we see the structure of features in a recipe. In the metadata of a recipe we find a lot of optional elements, but a name is always required. There is also quite some variation in the way each ingredient is specified. Some ingredients have extensive names, such as *reduced-sodium fat-free chicken broth*, and are accompanied by a quantity, preparation and even alternatives, where others require very little information, such as *salt*. For each of the leaf nodes, and example is given in appendix A. The non-leaf nodes each consists of all of their mandatory children, and possibly their non-mandatory children. Note that to refer to nodes in text, we use a descendant notation, much like the breadcrumbs design pattern that denotes the hierarchy of directories. The notation **A > B** means that unit A consists of unit B, and that B can be found as a child of A in the feature diagram in figure 2. Note that we leave out the root element **Recipe** in most cases.

3 Language Design

Corel is the name of the DSL we developed during this project, and stands for COoking REcipe Language. The main purpose Corel is to allow effective expressions for the features of recipes. It aims to keep the prose of recipes intact, and, in a similar vein as Markdown [2], to keep the source readable as-is. It is implemented in the metaprogramming language Rascal, with Java bindings for two small libraries. We start this section with an example of a recipe written in Corel, which can be see in listing 1, in which the structure and syntax can be seen. We continue with more in-depth information about the language design. Finally, we will have a look at the webpage that can be compiled from a Corel recipe.

Listing 1: A Corel recipe

```
Recipe:
Pasta Bolognese

Yield: 2 plates

Ingredients:
- 8 [ounces] white fresh {pasta}
- 1 [floz] olive {oil}
- 1/4 [ounce] {garlic}; minced
- 4 [ounces] {onions}; chopped
- 4 [ounces] shallow fried {beef}; minced
- 1 - 1 1/2 [ounce] lean prepared {bacon}
- 1/3 [cup] red {wine}
- 150 [gram] raw {carrots}; thinly sliced
- 2/3 [ounce] concentrated {tomato puree}
- 4 [ounces] red {sweet pepper}; cut julienne
- 1 [ounce] {parmesan} cheese

Instructions:
Add the @oil@ to a large saucepan, heat to <300 F>,
    and saute the @onions@. After |2 minutes|, add
    the @garlic@. Keep on medium to high heat, and
    don't stir. After |2 minutes| more, add the
    @beef@.
Fry the @bacon@ in a separate pan, on high heat.
    Remove liquified fat when done.
Boil @pasta@ in a medium pan, until al dente (~|8
    minutes|). Drain when done.
Once the @beef@ is done, add the @carrots@, @sweet
    pepper@ and @tomato puree@. Slowly add the
    @wine@ as well, to not lower the temperature.
    Let it simmer (but not boil) for |5-10 minutes|.
Add the @bacon@ to the large saucepan.
Serve with grated @parmesan@  cheese.
```

Indicates the start of a recipe, and is followed by the name.

Each ingredient starts with a dash. An identifier is specified in curly braces.

A preparation can be specified after a semicolon.

Units are enclosed by brackets.

Temperatures between angled brackets.

Refer to an ingredient using at-signs.

Times are annotated with vertical pipes.

3.1 Writing a Corel recipe

A Corel recipe is written in the Eclipse editor, using a plugin created for Corel. This provides syntax highlighting in the editor, as well as warnings and errors for semantic inconsistencies within the recipe. Some elements are annotated with docs, to provide information about density used when converting from volume to mass.

3.1.1 Structure

A Corel recipe always contains the three segments that you would expect in any recipe:

1. `Recipe`, followed by the name of the recipe and possibly other metadata, such as the yield of the recipe.
2. `Ingredients`, with a list of ingredients.
3. `Instructions`, after which a list of instructions follow.

These segments are indicated by their equivalent keywords. The syntax of this language is trying to leave as much room for prose as possible. The grammar allows different syntactic elements to occur in the different segments, to match the required expressivity of each segment.

3.1.2 Syntax

A character set disjoint from those appearing in the dataset is computed, to prevent conflicts between the prose of the recipe and punctuation marks used for annotations. Table 1 shows the annotations and elements are used in Corel.

<table>
<tr><th>element</th><th>description</th></tr>
<tr><td><code>Recipe:</code>
<code>Ingredients:</code>
<code>Instructions:</code></td><td>Keywords to indicate a section</td></tr>
<tr><td><code>Yield:</code>
<code>Servings:</code></td><td>Keywords indicating metadata values</td></tr>
<tr><td><code>-</code></td><td>A dash at line-start in the ingredients section marks the start of an ingredient</td></tr>
<tr><td><code>3</code>

<code>1/2</code>
<code>4 - 6</code></td><td>A natural number can be specified at certain places, without markup.
Fractions can be written with a forward slash <code>/</code>
Range values are separated with a dash <code>-</code></td></tr>
<tr><td><code>[ounce]</code>
<code>[cup]</code></td><td>A unit of measure is annotated with brackets <code>[]</code>.</td></tr>
<tr><td><code>{milk}</code>
<code>{olive oil}</code></td><td>Ingredient definitions are wrapped between curly braces <code>{}</code></td></tr>
<tr><td><code>@milk@</code>
<code>@olive oil@</code></td><td>Ingredient references are surrounded with at-signs <code>@</code></td></tr>
<tr><td><code>; minced</code></td><td>A semicolon <code>;</code> delimits the start of an ingredients preparation text. This matches the rest of the line.</td></tr>
<tr><td><code><350 °F></code>
<code><180 C></code></td><td>Angled brackets <code><></code> define a temperature</td></tr>
<tr><td><code>|8 minutes|</code>
<code>|1 hour|</code></td><td>Vertical bars <code>|</code> are used to annotate time</td></tr>
</table>

Table 1: Syntax in Corel

3.1.3 Grammar

The grammar of Corel, with starting symbol ⟨*recipe*⟩:

⟨*recipe*⟩	::=	⟨*declaration*⟩ ⟨*yield*⟩ ⟨*servings*⟩ ⟨*ingredients*⟩ ⟨*instructions*⟩
⟨*declaration*⟩	::=	recipe : ⟨*recipe_name*⟩
⟨*yield*⟩	::=	yield : ⟨*number_or_range*⟩ ⟨*unit_of_measure*⟩
	|	⟨*empty*⟩
⟨*servings*⟩	::=	servings : ⟨*number_or_range*⟩
	|	⟨*empty*⟩
⟨*ingredients*⟩	::=	ingredients : ⟨*ingredient_list*⟩
	|	ingredients : ⟨*empty*⟩
⟨*ingredient_list*⟩	::=	⟨*ingredient*⟩ ⟨*ingredient_list*⟩
	|	⟨*ingredient*⟩
⟨*ingredient*⟩	::=	- ⟨*quantity*⟩ ⟨*description*⟩ ⟨*preparation*⟩
⟨*quantity*⟩	::=	⟨*number_or_range*⟩ ⟨*unit*⟩
	|	⟨*number_or_range*⟩
	|	⟨*empty*⟩
⟨*description*⟩	::=	⟨*text*⟩ ⟨*ingredient_definition*⟩ ⟨*text*⟩
⟨*preparation*⟩	::=	; ⟨*prep_word_list*⟩
	|	⟨*empty*⟩
⟨*instructions*⟩	::=	instructions : ⟨*instruction_list*⟩
	|	instructions : ⟨*empty*⟩
⟨*instruction_list*⟩	::=	⟨*instruction*⟩ ⟨*instruction_list*⟩
	|	⟨*instruction*⟩
⟨*instruction*⟩	::=	⟨*word_list*⟩ .

The elements ⟨*prep_word_list*⟩ and ⟨*word_list*⟩ can be populated with *prose*, a *temperature* and *time*. A ⟨*world_list*⟩ can also be contain an *ingredient_ref* and *ingredient_def*. The grammar matches the structure of recipes found earlier, and leaves space for proze in an ingredients preparation, and in the instructions of a recipe.

3.1.4 Limitations

Not all of the features of a recipe can be annotated, and thus understood by Corel.

1. Under *Metadata*: There is no syntax for a *Description*, *Preparation time* or *Cooking time* element. These features are not of interest for the computations that we implemented.
2. *Metadata* > *Nutrients*: There is no section to list the nutritional values of the recipe.
3. Under *Ingredients*: *Alternatives* and *Note* cannot be declared.
4. Under *Instructions:* There is no syntax to specify a *Note*
5. Under *Instructions* > *Action*: There is only syntax to annotate *Ingredient* references.

This can also be seen in the feature diagram in the section 2, showing an overview of the recipe features that are implemented in Corel. Another limitation is that decimal numbers are not allowed, and they should be replaced by mixed numbers or sole fractions.

3.2 Interactivity and checking

The editor can assist in writing a recipe, by providing feedback. Currently, the following is checked:

Listing 2: Corel usedef checks

```
Ingredients:
- 1/3 [cup] {sugar}
- 2/3 [cup] brown {sugar}
- 1 1 Duplicate definition: "sugar" }
- 3 [cups] fresh {mushrooms};
    sliced
- 3 [tablespoons] green
    {onions}; sliced
 Unused ingredient: "onions"

Instructions:
Slice the @mango@ in small
    piec Undefined ingredient: "mango"

```

An error is generated when defining two ingredients using the same identifier. This prevents ambiguous ingredient referencing later. When an ingredient is declared, but not used, a warning is given. Referencing an undeclared ingredient raises an error.

Using a database of units from Frink [4], feedback about the specified units is given as well.

Listing 3: Corel unit validation

```
Ingredients:
- 2 [cups] {milk}
- 1 [cap] {flour}
- Unkown unit: ”cap”

Instructions:
Preheat the oven to <400
   vahrenheit>
Unknown temperature: ”vahrenheit”
```

If the unit of an ingredient is not known, a warning is given. The same is true for temperatures: when the unit of measure of a temperature is not found, a warning is given.

There are also informative messages, in the form of docs, that display computed information, and which source data is used for the computation:

Listing 4: Corel feedback

```
Ingredients:
- 8 [ounces] white fresh {pasta}
- 1 [floz] olive {oil}
-  1 floz is equal to 29.6 ml or 27.1 gr (us-
-  ing density of Oil, olive (at 15.6C), 0.915
-  g/ml)
   {tomato puree}
- 150 [gram] raw {carrots};
   thinly sliced
- 4 [ounces] shallow fried
   {beef}; minced
 Best matches for "shallow fried beef":
 - Minced beef shallow fried,
 - Minced beef lean shallow fried,
 - Minced beef/pork shallow fried

Instructions:
Add the @oil@ to a large
   saucepan, heat to <300 F>,
   and s 300 F is equal to 148.9 Celsius
   After |2 minutes|, add the
   @garlic@. Keep on medium to
   high heat, and don't stir.
   After |2 minutes| more, add
   the @beef@.
```

When hovering the unit of a quantity as in `1 [floz]`, the docs for this node are displayed. These contain conversion information: for mass the equivalent in grams, for volume the equivalent in milliliters. If a quantity is specified in volume, Corel will try to find the best ingredient match in a density database, and use the density to convert the volume to mass. The found match and its density are displayed as well. The ingredient description is annotated with it the best matches found in the nutrient database; sometimes it is best to slightly tweak the name of the ingredient, to ensure the correct match in the nutrient database. These are used later for computing the nutritional values of the recipe. Each temperature is annotated with the equivalent degrees Celsius. This means that you can hover the source text `<300 °F>` and read out that this is equal to 148.9 degrees Celsius.

3.3 Compilation

A Corel recipe can be compiled to a webpage. On the next page we showcase the page that is compiled from the source in listing 1. The webpage can also be seen at the web archive [1]. This page contains all of the docs that are available in the editor, except the list of best matches. During compilation, a table with nutritional information is computed. This is done using a database of density values, to convert volume to mass, and a database of nutritional values from [7]. This table with nutritional information is added to the webpage, with information listed per ingredient. The webpage also contains interactive elements, such as *highlighting* the matching ingredient definitions and references, and timers for each specified time, which can be controlled individually, and run in parallel. There is also the possibility to compile a scaled version of the recipe, in which the quantity of each ingredient is adjusted, as well as the recipes servings and yield values.

Pasta Bolognese

2 plates

Ingredients

1. 8 ounces white fresh pasta
2. 1 floz olive oil
3. ¼ ounce garlic, minced
4. 4 ounces onions, chopped
5. 4 ounces shallow fried beef, minced
6. 1 - 1½ ounce lean prepared bacon
7. ⅓ cup red wine
8. 150 gram raw carrots, thinly sliced
9. ⅔ ounce concentrated tomato puree
10. 4 ounces red sweet pepper, cut julienne
11. 1 ounce parmesan cheese

1/3 cup is equal to 78.9 ml or 78.7 gr (using density of Wine, red, 0.998 g/ml)

Two way links between ingredient definition and reference.

Quantity conversions, including used density, displayed on focus.

Instructions

1. Add the oil to a large saucepan, heat to 300 F, and saute the onions
2. After 2 minutes, add the garlic
3. Keep on medium to high heat, and don't stir
4. After 2 minutes more, add the beef
5. Fry the bacon in a separate pan, on high heat
6. Remove liquified fat when done
7. Boil pasta in a medium pan, until al dente (~7:52 minutes)
8. Drain when done
9. Once the beef is done, add the carrots, sweet pepper and tomato puree
10. Slowly add the wine as well, to not lower the temperature
11. Let it simmer (but not boil) for 5 - 10 minutes
12. Add the bacon to the large saucepan
13. Serve with grated parmesan cheese

Temperatures are shown in degrees Celsius when focused.

Times are converted to timers. Click to play or pause. They will ring when reaching zero.

Nutritional Values

Name	Quantity	Energy (kCal)	Protein (g)	Sugar (g)	Fat (g)
Pasta white fresh boiled	226.8 g	292.6	11.1	0.5	2.9
Oil olive	27.1 g	243.8	0.0	0.0	27.1
Garlic raw	7.1 g	11.2	0.5	0.1	0.0
Onions boiled	113.4 g	39.7	1.1	3.4	0.2
Minced beef shallow fried	113.4 g	375.4	34.5	0.3	26.2
Bacon lean prepared	35.4 g	150.1	9.1	0.0	12.6
Wine red	78.7 g	64.6	0.0	2.4	0.0
Carrot winter raw	150.0 g	51.0	0.9	4.7	0.5
Tomato puree concentrated tinned	18.9 g	16.4	0.9	2.4	0.0
Sweet pepper red boiled	113.4 g	31.8	1.0	4.8	0.1
Cheese 30+ average	28.3 g	81.8	8.6	0.0	5.2
Total	912.5 g	1358.2	67.6	18.5	74.9

Nutritional values are displayed in a table.

Computed using data from *NEVO online version 2019/6.0, RIVM, Bilthoven*

Note that only ingredients with a specified unit are present here, and that the ingredients might not exactly match with those listed in the recipe.

4 Implementation

4.1 Rascal

Rascal is a language and environment for metaprogramming, which has evolved significantly since the informal presentation of the first version in 2009 [13]. Rascal joined the small set of technologies that incorporates both Source Code Analysis and Manipulation (SCAM) elements, and since has been called a ”one-stop shop for meta programming” [13]. In this project Rascal is used as a language prototyping tool. Everything that defines the language is written in Rascal, from the syntax definition and algebraic datatypes, the checker and transformations, to the compiler.

4.2 Syntax to AST

The syntax and grammar, as seen in Chapter 3.1, are implemented in Rascal. These are the first building blocks of the programming language. Writing the syntax specification in Rascal is much like writing a context-free grammar. There are production rules which can be defined in terms of other non-terminals.

Listing 5: Syntax for numeric values in Rascal

```
syntax NumberOrRange
    = @category="Constant" number: ExactValue val
    | @category="Constant" range: ExactValue lower
      "-" ExactValue upper
    ;

syntax ExactValue
    = sole_integral: NaturalNumber nat
    | mixed: NaturalNumber nat Fraction frac
    | sole_fraction: Fraction frac
    ;

lexical Fraction
    = NaturalNumber num "/" NaturalNumber den;

lexical NaturalNumber
    = [0-9] !<< ([1-9][0-9]*) val !>> [0-9];
```

Syntax highlighting category is specified.

Assigned field names. Alternatives are labeled to aid imploding.

A terminal that matches natural numbers; a building block for other symbols.

Listing 5 shows how the syntax for numeric values in Corel is constructed. The natural number is the most elementary building block, on which a fraction and exact values are build. These exact values are then used in the definition of a range. The syntax definition is used by Rascal to construct a scannerless parser. Having a concrete syntax tree (CST), the next step is converting it to an abstract syntax tree (AST). Rascal provides an `implode` function to help with this. It traverses an algebraic data type (ADT) together with the CST, constructing an AST in the process [10].

4.2.1 Benefits of using an AST

The AST is the main representation of the source we use, upon which we apply transformations and check certain properties, and eventually compile. The benefit compared to a CST is that the whitespace, comments or structure defining elements are removed, making the tree smalller, and making it easier to do pattern matching.Rascal offers different types of pattern matching, making it easy to modify and analyze the nodes accross the tree.

4.3 Frink binding

Frink is a calculating tool and programming language, designed for physical calculations [4]. These are some example inputs for Frink and the output:

1. `38 feet -> meters` results in `11.5824`
2. `cup conforms volume` results in `true`
3. `1.5 cups -> ml` results in `354.88235475`

In this project, Frink is used to validate units (i.e. ensure a unit of measure is a volume, or a mass), and for unit conversion, as in the last example, where the unit `cup` is converted to `ml`.

Frink is available as a jar file, which allows programs to bind to it. Bringing the functionality of Frink into Rascal is done in two parts. First we declare a Rascal module, and declare a java function and the class it belongs to, as seen in listing 6.

Listing 6: Frink binding

```
module Frink

import IO;
import String;

@javaClass{rascalJava.FrinkBinding}
public java str frink_parse(str s);
```

Next is actual Java code, a bit more complex, as it has to convert Java types to Rascal types. Also, in case Frink throws an exception, this exception has to be converted. This can be seen in listing 7.

Listing 7: Binding Frink

```java
    public IString frink_parse(IString text, IEvaluatorContext ctx)
    {
        String unsafe_user_input = text.getValue();
        String result = "";

        try
        {
            result = interp.parseString(unsafe_user_input);
        }
        catch (FrinkEvaluationException e)
        {
            // Rethrow as Rascal exception
            throw RuntimeExceptionFactory.illegalArgument(text, "Input cannot be converted by Frink");
        }

        return vf.string(result);
    }
```

The type conversion is done using ValLang, a collection of datatypes for the Java Virtual Machine [11]. With this in place, any of Frink's functionality can be used in the Rascal project.

4.4 Checker implementation

A checker is used to provide diagnostics about the recipe. In Corel it ensures no references to undefined ingredients are made, and that the unit of measure of each ingredient is valid.

4.4.1 Definitions and uses

First, the relation between the ingredient definitions and references is constructed from the source file. This is then used to validate that each ingredient definition is unique, and that each ingredient reference refers to a declared ingredient. These checks are implemented with relative ease in Rascal, as can be seen in listing 8.

Listing 8: Usedef validation

```
// IngredientRefs referring to non-existing ingredients
for (/AIngredientRef ref := r)
{
    if (!(ref.src in usedef<0>))
```

```
    {
        msgdocs.messages += {error("Reference to undeclared
           ingredient: <ref.name>", ref.src)};
    }
}
```

4.4.2 Unit validation

Unit validation is done using a list of units from the Frink programming language/tool. Frink contains a file `units.txt`, in which it lists all the units it understands. The first check we do for each ingredient with a unit of measure specified, is validating this unit.

With the Frink binding shown earlier, we can check whether a unit has a specific base unit, i.e. check whether `ounce` is a unit of mass (see listing 9).

Listing 9: Unit validation

```
bool unit_conforms(str unit, str base_unit)
{
    str result = "";

    try
        result = frink_parse("<unit> conforms <base_unit>");
    catch _:
        println("Caught an exception from unit_conforms");

    return result == "true";
}

bool unit_is_mass(str unit)
{
    return unit_conforms(unit, "mass");
}
```

Note that there is something peculiar about the plural units of measure in Frink. Frink's `units.txt` contains definitions for irregular plurals, but not for regular ones. We assume Frink checks whether the final character is an "s", and in case it is, it compares the unit without the "s" suffix as well with the units.txt. An example: Frink's specification contains both "century" and the irregular plural "centuries": *century* and *centuries*. Now if we let Frink parse the "double plural" of century, "centuriess", Frink parses this without problems: `frink_parse("1 centuriess -> year")` returns `str: "99.999999999999999999"`. We mimic this behaviour when validating units.

4.4.3 Converting units with Frink

For further calculations, it is convenient to have each unit of measure in a standard format. We chose the units millilitres and grams, as they are used in the databases we use as well. First the input for Frink is constructed in a string. There are helper functions, which accept an ADT, and format them for Frink, to match the required notation for fractions, intervals, etc. This is then evaluated by Frink. The conversion process can be seen in 10.

Listing 10: Unit standardisation

```
tuple[real quantity, str unit] convert_to_si(AQuantity q)
{
    str conversion_target_unit =
       get_conversion_base_unit(q.unit.name);
    str frink_conversion_input = "<frink_print(q.val)>
       <q.unit.name>";

    str quantity_conversion_result =
       frink_parse("round[<frink_conversion_input>, 0.1
       <conversion_target_unit>] -\> <conversion_target_unit>");

    real quantity;

    if (q.val is range)
    {
        // Take the center of the range as quantity value
        str stripped =
           replaceAll(replaceAll(quantity_conversion_result,
           "[", ""), "]", "");
        list[str] values = split(", ", stripped);
        quantity = (toReal(values[0]) + toReal(values[1]))/2;
    }
    else
    {
        quantity = toReal(quantity_conversion_result);
    }
    return <quantity, conversion_target_unit>;
}
```

4.4.4 Converting volume to mass

Some ingredient quantities are specified by volume. However, weight specifications are more exact than those in volume, as the density of powder-like substances can vary quite a bit, depending on the grain size. Therefore we decided to converted all volumes to masses, using a density table from FAO INFOODS [3]. To find the best match for each ingredient, we take the ingredient description, including the ingredient definition. From the ingredient description `- 4 [ounces] shallow fried {beef}; minced`, the text `shallow fried beef` is distilled and used when finding the best match. Since the volumes are already converted to `milliliters`, and the densities are stored in `g/ml`, the conversion becomes a simple multiplication.

4.5 IDE plugin

Messages generated during the checking phase (warnings, errors, docs) are available in the Corel editor. This is done by registering the language and providing certain contributions. One of those contributions is a context menu, from which two functions are available (compiling the recipe at a scale, and without scaling). Rascal has easy to use implementations for this, as can be seen in listing 11.

Listing 11: Context menu in the IDE

```
    popup(
        menu("Recipe",[
            action("Compile", compile_unscaled),
            action("Compile scaled", compile_scaled)
        ])
    )
```

4.6 Compilation to a webpage

Compiling the AST to a webpage takes multiple steps. In short, they are:

1. Optionally scaling the recipe
2. Collecting the docs that are computed during checking
3. Computing nutrient data for the recipe
4. Construction an HTML page from the AST.

This HTML page is then padded with some CSS for styling, and JavaScript to enable interactive elements. It takes less than a second to compile the webpage.

4.6.1 Scaling a recipe

Scaling a recipe means to scale the ingredient quantities, the recipes servings and the recipes yield. This is done by first converting the mixed numbers and sole fractions to their real equivalent. This is then scaled, and converted back to a fraction, a natural number or a mixed number, whichever fits best. The scaled value replaces the old value in the AST, and is used for subsequent computations (such as the nutritional values).

4.6.2 Computing nutritinal values

This computation is based on the listed ingredients in the recipe and their quantities. If an ingredient is specified without a unit of measure, it is not included in this calculation. For each ingredient we find the best match in the RIVM NEVO Nutrient Database [7]. This database contains data for over one hundred nutrients for two thousand foods. This row is then scaled to match the mass of the ingredient in the recipe, scaling all the nutrient quantities with it. These rows are summed together to compute a row to represent the total nutrient values of the recipe.

4.6.3 Ingredient name matching

To find matches for each ingredient in the density database and the nutrient database, we use a Java binding of the FuzzyWuzzy Python algorithm, as found on [5]. This does not always yield the correct results, and thus we added feedback to the editor, stating the current best match. This allows users to update their ingredient specification to get their intended match.

4.6.4 Constructing HTML in Rascal

Rascal has an AST model for HTML5, including a pretty printer [8]. This is used to construct the HTML of the recipe page as well. The main idea is to add nodes within nodes to define the structure (see listing 12).

Listing 12: Constructing HTML

```
body(
    header("..."),
    ...
    h2("Ingredients"),
    ol(
        ([] | it + li(ast2html(ing, msgdocs)) | ing <-
           r.ingredients)
    ),
    h2("Instructions"),
    ol(
        ([] | it + li(ast2html(ins, msgdocs)) | ins <-
           r.instructions)
    )
```

```
)
```

This is accompanied by a set of functions that convert an ADT to an HTML node for the more complex elements. An example of a timer being constructed from a `ATime` node is shown in listing 13.

Listing 13: Converting to HTML

```
HTML5Node time2html(ATime t)
{
    int seconds = convert_to_seconds(t);
    str original_time_text = numberorrange2str(t.val);

    return span(
        span(original_time_text,
             class("time_value"),
             html5attr("data-original-text",
                 original_time_text)),
        " <t.unit>",
        class("timer"),
        html5attr("tabindex", 0),
        html5attr("data-original-time", seconds),
        html5attr("data-current-time", seconds)
    );
}
```

4.6.5 Interactivity on the webpage with JavaScript

Two-way highlighting between ingredients is implemented using JavaScript and HTML5 data-attributes. This allows people to click on an ingredient in the ingredient list, and see the ingredient highlighted in each instruction it is used it. A simple timer is constructed for each element with the required `data-original-time` attribute, using `setInterval` and adding `click` event listeners. A beeping sound is made using the AudioContext web API, to prevent having to supply a separate audio file.

4.7 Rascal source size

To give an indication of the size of the project, and each of the individual modules, we computed the linecount. Lines of code are counted using `cloc --force-lang="Java" *.rsc --by-file`. Rascal source files use a similar style for comments as Java. The counts can be seen in table 2.

Table 2: Lines of Code in Corel

LOC	*File*	*Description*
197	./CST2AST.rsc	Converting parse tree to AST
107	./AST.rsc	ADT definitions
40	./DensityDb.rsc	Interfacing density database
18	./Resolve.rsc	Linking definitions and uses
69	./IDE.rsc	Corel IDE plugin functions
547	./NutrientDb.rsc	Interfacing nutrient database
87	./Frink.rsc	Bindings for Frink language
7	./FuzzyWuzzy.rsc	Bindings for FuzzyWuzzy
82	./Syntax.rsc	Corel concrete syntax definition
264	./Check.rsc	Recipe content validation
502	./Compile.rsc	Compilation to webpage
129	./Transform.rsc	Scaling a recipe
2049	total	

Notes Note that the high linecount in NutrientDb, the module to interface with the *RIVM NEVO database*, is due to some functions working on the 100+ column-wide dataset. The compilation code has become quite large as well, due to inlining of the CSS styles and javascript.

Two small Java bindings were created, to convert between Java and Rascal types. Their sizes are show in table 3

Table 3: Java binding sizes

LOC	*File*	*Description*
29	./FuzzyWuzzyBinding.java	String distances
36	./FrinkBinding.java	Unit conversions
65	total	

These are both quite small, as their main task is to forward strings to their respective libraries.

5 Results

5.1 Evaluating expressivity

To measure the expressivity of Corel, we converted a control group of twenty recipes from plaintext to Corel. Here, we annotated each of the features that is implemented in Corel, and highlighted those that we could not annotate correctly, as seen in listing 14.

Listing 14: Annotating the control group

```
- 1 1/2 [cups] {tomatoes}; peeled chopped or 1 1/2 cups canned
    tomatoes
- 1/2 [cup] white {wine}
- 1/2 [cup] {feta cheese}; crumbled

Instructions:
In a pot of boiling water, add @shrimp@ and cook for |1 minute|.
    Drain well. Place @shrimp@ on bottom of greased baking dish
    in single layer. Set aside.
```

The features that are implemented can all be annotated without problems: the units `[ cups ]`, the ingredient definition `{ tomatoes }`, etc.. The process of annotating the control group recipes is described in appendix B. To give an insight in the frequency of each of the un-implemented features, we constructed table 4, containing the total amount of occurences of each un-implemented feature in the control group. As earlier, the notation **A > B** means that unit A consists of unit B, and that B can be found as a child of A in the feature diagram in figure 2. Note that we leave out the root element **Recipe** here as well.

Unimplemented feature	*Count*
Ingredients > Alternatives	7
Ingredients > Note	20
Instructions > Note	17
Instructions > Action > Condition	2
Instructions > Action > Objects > Appliance	48
Instructions > Action > Objects > Utensils	7
Instructions > Action > Target state	32
Instructions > Action > Guideline	15

Table 4: Un-implemented feature count over 20 recipes

Note that these elements still can occur in a Corel source text, but they will not be recognized as the feature their text represents.

6 Conclusion

We learned that there is a lot of structure in recipes that are written in natural language. We found mandatory elements, such as the presence of actions and objects in instructions, and constructed a feature diagram from this. We were able to decompose every recipe from the control group into these features, and describe almost all of these features in a Corel recipe. We experienced that it is straightforward to do computations with recipes in this structure, and that this can yield informative results.

Interactivity Syntax highlighting communicates how elements are interpreted by Corel. We use multiple methods (warnings, annotated docs) to communicate the results of the checker to the user in the editor. These provide insight in the current understanding by Corel of the recipe, and display which database matches are used for volume-to-mass conversion and computing the nutritional values. We did not study how the editor interactivity is experienced by users.

Compiled webpage Compiling a Corel recipe to a webpage allowed us to showcase the computations that can be done with a recipe: the added nutritional information, the unit conversions and creating interactive timers, as seen in section 3.3.

6.1 Evaluating expressivity

In the process of annotating the conrol group, we experienced the following:

1. Annotating the sections of a recipe takes little effort, and never fails. Note that we do not have room for prose inbetween the segments, and thus we did not convert the metadata features that are not implemented in Corel.

2. Adding the right markup to each of the ingredients raised some issues. Ocassionally an alternative for an ingredient was specified, or a note regarding the ingredient. These could not be annotated, and as a result they were categorized as part of the ingredients preparation. However, each of the features that are implemented in Corel could be annotated without problems.

3. The instructions are less structured than the ingredient specification, and since fewer of these features are defined in Corel for the instructions, we came accross more elements that we could not annotate.

6.2 Discussion

The FODA method is designed to evaluate families of software systems. We applied it on a different domain, and with a different intention. Certain elements that are analysed in the method are not present in recipes, and thus we had to adapt the analysis. We focused on the different elements a textual recipe can express. Whether this analysis holds its ground when used on this different domain is unexplored, and no evaluation of this is done either.

A second difficult part is distilling the right features from the phrases in recipes. Evaluating whether a feature affects an end-user is dependend on the context in which the language is used. A textual recipe is now always read by a human, and the instructions are followed to produce a dish. With a DSL, the end-user is not as clearly defined, and it can be argued that the compiler for this DSL is one too.

The dataset that is used for the domain analysis possibly brings biases with it:

1. The form in which recipes can be input could enforce a certain style. Unfortunately we cannot access this ourselves, as the submission page is not available in our country (the Netherlands). This would cause recipes in the dataset to follow these requirements, and possibly skew the results of our domain analysis.

2. There can be a bias originating in the userbase of Food.com. A mostly north-european user base could write mostly north-european recipes, which causes the domain analysis to be one of mostly north-european recipes.

6.3 Future work

6.3.1 Analysis of application domain technology

Analysing the domain technologies in cooking recipes would allow a better understanding of some of the methods described in recipes. These could then be linked up to explanatory material, or substituted for different methods, depending on the end-user's wishes. See also the paragraph on application domain technology 2.1.1.

6.3.2 Implementing remaining features

There are quite some features in recipes that cannot be expressed in Corel yet. For these features, new annotation syntax has to be designed. This would enable the DSL to better understand the phrases that occur in the instructions, or recognize an alternative ingredient.

6.3.3 Improving ingredient matching

Text-based ingredient matching is difficult, and our method does not always yield the correct match. Our approach uses no understanding of the meaning of words; only string based distances are used to find the best match. This is very error-prone in situations where an ingredient is described with few characters, such as *oil*. Even more so, since the databases we use have over-specific descriptions for some ingredients. An example of this problem was when a match for *ham* was looked for, and the found match was *jam*, instead of *ham schoulder medium fat boiled*.

6.3.4 Disjunct specification for density and nutrient match

Right now, the same part of the ingredient specification is used for finding matches in both the density database and the nutrient database. This makes it difficult, or sometimes impossible, to write the specification in such a way that the correct match is found in both databases. Possibly a new annotation has to be designed, to separate the textual ingredient description from those that are used to search the databases.

6.3.5 Expanding the horizon

In this project a compilation to a simple recipe webpage is implemented. Integration into different areas could still be explored, such as generating a visual animation of the recipe, making it accessible for people with a lower reading comprehension, or using the DSL in automated kitchens. Scoping this domain would provide insight into the development direction Corel could take.

Glossary

ADT Algebraic Data Type. 22, 25, 28, 29

aggregation Abstracting a collection of units into a new unit, e.g. school is an aggregation of students, teachers, etc. [12].. 5

AST tree representation of abstract structure of source code. 22, 26, 27

Corel Name of the COoking REcipe Language. 11–14, 16–18, 21, 23, 26, 29–33, 37

CST tree representation of syntactic structure of source code. 22

DSL Domain-specific Language. 3, 5, 9, 11, 32, 33

Feature Diagram displays the standard features of a family of systems in the domain [12]. 16

FODA Feature-oriented Domain Analysis. 3–5, 31

Frink A tool and programming language for physical calculations. Used in this project for unit conversions.. 16, 22–25

generalization Abstracting the commonalities among a collection of units into a new conceptual unit, suppressing detailed differences. An example is generalizing secretaries, managers and technical staff into the conceptual entity employee [12].. 5

GPL General-purpose Language. 3

HTML HyperText Markup Language. 3, 26

Rascal A metaprogramming language. 3, 11, 21–23, 26–29

SCAM Source Code Analysis and Manipulation. 21

Appendices

A Recipe Features

These are examples that display each of the features found in the feature diagram. Note that these are values from the dataset, and that some features are displayed with additional markup or surrounding text, such as being prefixed with "Servings:" or occur in the middle of an ingredient description or in a table. An indication of what this context could look like is added in *italics*. Note that **A > B** means that unit A consists of unit B, and that B can be found as a child of A in the feature diagram in figure 2.

Recipe > Metadata > Name
Recipe: Shrimp Stuffed Twice-Baked Potato

Recipe > Metadata > Yield
Yield: 15-17 pancakes

Recipe > Metadata > Servings
Servings: 4

Recipe > Metadata > Description
About this recipe: This is delicious so creamy and tender. The sauce is really tasty. It takes a little time but that's all simmering time. Serve with potato. You'll enjoy it I'm sure.

Recipe > Metadata > Preparation time
Preparation time: 10 minutes

Recipe > Metadata > Cooking time
Cooking time: 25 minutes

Recipe > Metadata > Nutrition

Nutrient	*Quantity*
Fats	10.1 *gr*
Sodium	213 *mg*
Protein	15.5 *gr*

Recipe > Ingredients > Quantity > Value > Range
12 - 16 *white corn tortillas*

Recipe > Ingredients > Quantity > Value > Exact value
2 *garlic cloves, crushed*

Recipe > Ingredients > Quantity > Unit of measure
2 tablespoons *soy sauce*

Recipe > Ingredients > Name
3/4 cup plain soy yogurt

Recipe > Ingredients > Preparation
1 carrot, finely shredded

Recipe > Ingredients > Alternatives
3 tablespoons butter (or margarine)

Recipe > Ingredients > Note
1 pound sandwich buns, (about 4 large sandwich buns, *halved* (we use sourdough)

Recipe > Instructions > Action > Conditional
If the mixture is too thick, *add milk, a little at a time, until pancake batter consistency.*

Recipe > Instructions > Action > Objects > Ingredient
Whisk in flour

Recipe > Instructions > Action > Objects > Appliance
Melt butter in small saucepan

Recipe > Instructions > Action > Objects > Utensil
Prick pie shell with fork

Recipe > Instructions > Action > Target state
Saute mushrooms and onions until tender and mushroom liquid has evaporated

Recipe > Instruction > Action > Guideline
Bake until filling is puffed, about 11/3 hour

Recipe > Instruction > Note
This is good after being frozen, but loses some crunch.

B Annotating recipes

These are the general changes we made to each plaintext recipe, such that they would adhere to the syntax of Corel.

B.1 General steps

For each recipe, the process looks as follows (starting from a plaintext recipe that matches the segments as in 3.1.1):

1. Add brackets `[ ]` around the unit of measure of an ingredient, if it is present
2. Add curly braces `{ }` around the defining ingredient name
3. Separate description from preparation with a semicolon `;`
4. Add at-symbols `@ @` to ingredient references in the instructions
5. Wrap temperatures in angled brackets `< >`
6. Wrap time in vertical bars `| |`.

B.2 Encountered problems

While converting the recipes from the control group to Corel, we encountered certain difficulties; elements that were expressed in natural language, that we are unable to annotate correctly with the Corel syntax. In listing 15 is a recipe from the control group, in which we highlighted the elements that we could not annotate.

Listing 15: Control group recipe 1

```
Recipe:
Greek Tomato, Shrimp and Feta

Servings: 4

Ingredients:
- 1 1/2 [lbs] {shrimp}; peeled and deveined
- 1/4 [cup] olive {oil}
- 2 {garlic} cloves; minced
- 3/4 [cup] {onion}; chopped
- 1/4 [teaspoon] red {pepper flakes}
- 1 [teaspoon] {oregano}
- 1/2 [teaspoon] {basil}
- 1 [tablespoon] {parsley}
```

```
- 1 1/2 [cups] {tomatoes}; peeled chopped or 1 1/2 cups canned tomatoes
- 1/2 [cup] white {wine}
- 1/2 [cup] {feta cheese}; crumbled

Instructions:
In a pot of boiling water, add @shrimp@ and cook for |1 minute|. Drain well. Place @shrimp@ on bottom of greased baking dish in single layer. Set aside.
In a skillet heat @oil@. Add @garlic@, @onion@ and red @pepper flakes@. Cook until veggies are soft.
Add @oregano@, @basil@, @salt@ and @pepper@. Stir and cook |1 minute|.
Add @wine@ and bring to boil. Cook for |2 minutes|.
Add @tomatoes@ and stir well. Reduce heat to low and simmer |8 minutes|. Most of liquid should evaporate.
Pour mixture over @shrimp@ in baking dish. Top with @feta cheese@.
Bake, uncovered, at <325 degrees F> for |15 minutes|.
Serve.
```

Observe in listing 15 that there is an alternative ingredient specified: `or 1 1/2 cups canned tomatoes`. This is something we cannot annotate yet. In the instructions we see multiple appliances being referred to. These are left as prose, and not annotated either. For this recipe, we thus count 1 occurence of the un-implemented feature `Ingredients > Alternatives`, and 4 occurences of `Instruction > Action > Objects > Appliance`. A target state `until veggies are soft` and a note `Most of the liquid should evaporate` are also highlighted and counted. This process is done for each of the recipes in the control group. Here we have listed each of the features we could not annotate per recipe, followed by the amount of times they occured in a recipe:

Gf Easy Delicious Chili

Unimplemented feature	*Count*
Ingredients > Alternatives	1
Ingredients > Note	2
Instructions > Action > Condition	1
Instructions > Action > Objects > Appliance	1
Instructions > Action > Guideline	2

Chile Relish

Unimplemented feature	*Count*
Instructions > Note	1
Instructions > Action > Objects > Appliance	2
Instructions > Action > Target state	1

Greek Tomato, Shrimp and Feta

Unimplemented feature	*Count*
Ingredients > Alternatives	1
Instructions > Note	1
Instructions > Action > Objects > Appliance	4
Instructions > Action > Target state	1

Best Soy Stuffed Bell Peppers

Unimplemented feature	*Count*
Ingredients > Note	3
Instructions > Note	1
Instructions > Action > Objects > Appliance	1
Instructions > Action > Guideline	1

Adobo De Chile

Unimplemented feature	*Count*
Ingredients > Note	4
Instructions > Note	4
Instructions > Action > Condition	1
Instructions > Action > Objects > Appliance	6
Instructions > Action > Objects > Utensils	1
Instructions > Action > Target state	4

Peanut Butter Chocolate Pretzel Candy

Unimplemented feature	*Count*
Instructions > Action > Objects > Appliance	4

Whole Wheat Chocolate Chip Banana Bread

Unimplemented feature	*Count*
Ingredients > Alternatives	1
Instructions > Action > Objects > Appliance	6
Instructions > Action > Objects > Utensils	2
Instructions > Action > Target state	5
Instructions > Action > Guideline	1

Pan Roasted Chicken With Artichokes and Lemon

Unimplemented feature	*Count*
Ingredients > Alternatives	1
Instructions > Action > Objects > Appliance	2
Instructions > Action > Target state	1
Instructions > Action > Guideline	1

Instant Triple Coffee Ice Cream

Unimplemented feature	*Count*
Ingredients > Alternatives	1
Ingredients > Note	1
Instructions > Action > Objects > Appliance	3
Instructions > Action > Target state	2
Instructions > Action > Guideline	1

Chicken Breasts With Spicy Honey-Orange Glaze

Unimplemented feature	*Count*
Ingredients > Note	1
Instructions > Action > Objects > Appliance	2
Instructions > Action > Target state	2

Spaghetti Alla Norma

Unimplemented feature	*Count*
Instructions > Action > Objects > Appliance	1
Instructions > Action > Target state	3
Instructions > Action > Guideline	3

My Mum’s Christmas Cake

Unimplemented feature	*Count*
Ingredients > Note	1
Instructions > Note	3
Instructions > Action > Objects > Appliance	1

Grilled Herb-Coated Chicken Breasts

Unimplemented feature	*Count*
Instructions > Note	2
Instructions > Action > Objects > Appliance	3
Instructions > Action > Objects > Utensils	1
Instructions > Action > Target state	2
Instructions > Action > Guideline	1

Bacon Wrapped Shrimp

Unimplemented feature	*Count*
Instructions > Action > Objects > Utensils	1
Instructions > Action > Target state	1
Instructions > Action > Guideline	1

California Burgers

Unimplemented feature	*Count*
Instructions > Action > Objects > Appliance	1

Sloppy Joe Style Pizza Burger

Unimplemented feature	*Count*

Grilled Pineapple With Key Lime and Agave Nectar

Unimplemented feature	*Count*
Ingredients > Alternatives	1
Ingredients > Note	4
Instructions > Note	2
Instructions > Action > Objects > Appliance	3
Instructions > Action > Target state	1
Instructions > Action > Guideline	1

Black Bean Brownies

Unimplemented feature	*Count*
Ingredients > Alternatives	1
Ingredients > Note	1
Instructions > Note	3
Instructions > Action > Objects > Appliance	6
Instructions > Action > Objects > Utensils	2
Instructions > Action > Target state	6
Instructions > Action > Guideline	2

White Bean and Roasted Eggplant Hummus (Baba Ghanoush)

Unimplemented feature	*Count*
Ingredients > Note	3
Instructions > Action > Objects > Appliance	2
Instructions > Action > Target state	3
Instructions > Action > Guideline	1

References

[1] Corel recipe page website. `https://web.archive.org/web/20210721122556/https://roorda.dev/recipes/0`. Accessed: 2021-07-21.

[2] Daring Fireball: Markdown website. `https://daringfireball.net/projects/markdown/`. Accessed: 2021-07-15.

[3] FAO INFOODS Density database v2 website. `http://www.fao.org/infoods/infoods/tables-and-databases/faoinfoods-databases/en/`. Accessed: 2021-08-04.

[4] Frink Language website. `https://frinklang.org/`. Accessed: 2021-07-19.

[5] FuzzyWuzzy Java binding on github. `https://github.com/xdrop/fuzzywuzzy`. Accessed: 2021-08-09.

[6] Kaggle Food.com recipe dataset. `https://www.kaggle.com/irkaal/foodcom-recipes-and-reviews`. Accessed: 2021-03-26.

[7] NEVO-online versie 2019/6.0, RIVM, Bilthoven website. `https://web.archive.org/web/20210305021057/https://www.rivm.nl/nederlands-voedingsstoffenbestand/toegang-nevo-gegevens/nevo-online/copyright-en-disclaimer`. Accessed: 2021-07-21.

[8] Rascal html5 dom. `https://github.com/usethesource/rascal/blob/master/src/org/rascalmpl/library/lang/html5/DOM.rsc`. Accessed: 2021-08-04.

[9] Rascal metaprogramming language. `https://www.rascal-mpl.org/`. Accessed: 2021-02-17.

[10] Rascal, ParseTree, Implode website. `https://docs.rascal-mpl.org/unstable/Libraries/#ParseTree-implode`. Accessed: 2021-07-19.

[11] ValLang usethesource github page. `https://github.com/usethesource/vallang`. Accessed: 2021-08-17.

[12] K. C. Kang, S. Cohen, J. A. Hess, William E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. 1990.

[13] Paul Klint, Tijs van der Storm, and Jurgen Vinju. Rascal, 10 years later. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 139–139, 2019.

[14] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.