Louis Mahon / Carl Vogel

The Proof is in the Pudding: Using Automated Theorem Proving to Generate Cooking Recipes

Abstract

This paper presents FASTFOOD, a rule-based natural language generation (NLG) program for cooking recipes. We consider the representation of cooking recipes as discourse representation, because the meaning of each sentence needs to consider the context of the others. Our discourse representation system is based on states of affairs and transtions between states of affairs, and does not use discourse referents. Recipes are generated by using an automated theorem-proving procedure to select the ingredients and instructions, with ingredients corresponding to axioms and instructions to implications. FASTFOOD also contains a temporal optimization module which can rearrange the recipe to make it more time efficient for the user, e.g. the recipe specifies to chop the vegetables while the rice is boiling. The system is described in detail, including the decision to forgo discourse referents and how plausible representations of nouns and verbs emerge purely as a by-product of the practical requirements of efficiently representing recipe content. A comparison is then made with existing recipe generation systems, NLG systems more generally, and automated theorem provers.

1 Introduction

In recent decades, cooking recipes have received a degree of attention in the field of rule-based Natural Language Processing. They are highly uniform in their general structure, with an ingredients list followed by a series of instructions, and can be easily accessed in large numbers on cooking websites and recipe search engines. However, the majority of work has focused either on searching and annotating large databases of online recipes, or on methods for processing and representing individual recipes, in either case, the approach has been one of Natural Language Understanding. The present system, named FASTFOOD, instead approaches cooking recipes from the direction of Natural Language Generation (NLG): rather than trying to extract a representation of the discourse of a recipe from a natural language description, it first derives the discourse representation, and then realizes this representation in natural language.

We consider cooking recipes as a discourse, because their constituent sentences contribute to a shared meaning, and should not be analyzed independently. Recall that the discourse representation theory (DRT) literature in particular (see Kamp and Reyle (1993)) emerged as a framework that captured conditions on the accessibility of discourse referents to anaphora and cataphora within sentences and in sequences of sentences as uttered by individual speakers, even outside dialogue contexts. DRT also attended to questions of fiction (see Kamp (2021)). Fundamental questions are about the nature of the objects that serve as antecedents to pronouns. We feel that recipes are merely another variety of discourse in this same spirit, and an interesting one because of the nature of change to the starting point objects. While it is certainly interesting to dwell on the inaccessibility of “a bicycle” as an antecedent to “it” in “Leslie does not own a bicycle. It is red” as has been addressed within DRT, it is also interesting to see that “Leslie pureed a potato. It is large” is an incoherent discourse because of the change that the underlying referent undergoes.

The purpose of DRT is to specify what referents are named in discourse, what expressions co-refer to those same entities (e.g. pronouns pointing to objects introduced via indefinite descriptions), and what relations hold true among those entities. Our system uses a related but alternative approach to representing discourse, in which the first-class citizens are the processes that change the relevant entities—some to the extent that the entities are obliterated, some merely changing the state, other entities brought into existence–rather than the entities themselves (see Section 3).

The task of NLG has been analysed in numerous ways. Some early examples (K. R. McKeown, 1985)) make a binary distinction between a strategic component that decides what to say, and a tactical component that decides how to say it. Following work has continued the idea of a spectrum from strategic to tactical but increased the number of levels to 3 (Bateman, 2002; Panaget, 1994) or even 6 (Dale & Haddock, 1991; Reiter, Sripada, Hunter, Yu, & Davy, 2005). The most fruitful framework in which to view the present work is that of Robin (1993), which uses 4: content production, content selection, content organization, and content realization. Using this framework, the operation of FASTFOOD can be summarized as follows.

Content Production The first phase is to produce the content that may possibly appear in discourse. FASTFOOD represents discourse content with two types of structure: descriptive strings, intuitively a natural language description of a state of affairs; and processes, intuitively transitions from one state of affairs to another. Internally these are produced from more fundamental structures such as cooking actions, intuitively verbs such as chop or mash, and food classes, intuitively a type of food such as carrot. The result of the content production phase is the creation of a database consisting of a set containing all recognized descriptive strings and a set containing all recognized processes.

Content Selection Once the possible content has been produced, the system must select the appropriate content for a given discourse (recipe) in response to a user input. This is done by an algorithm akin to a backwards chaining Automated Theorem Prover (ATP): the desired dish is treated as a formula, the possible ingredients as axioms and the processes as rules of inference. The algorithm proceeds backwards from the desired dish, attempting to reach a set of possible ingredients. The output of the content selection phase is a set of descriptive strings corresponding to an ingredients list and a set of processes corresponding to a list of cooking instructions; or else an indication that the dish cannot be made from the set of foods the user has access to (the formula is unprovable).

Content Organization FASTFOOD is concerned not only with generating a recipe, but also with ensuring the recipe is time efficient. Improvements in time efficiency can be achieved by performing multiple tasks at once: it is quicker to chop the carrots while waiting for the rice to boil than to wait for the rice to boil and then chop the carrots. The information on how to achieve efficiency is expressed by the third of the NLG subtasks: content organization. FASTFOOD must determine the order in which instructions are to be performed, and provide an indication of which are to be performed simultaneously, i.e. when to print successive instructions together in the form ‘while A, B’. The output of the content organization phase is an ordered list of processes, some of which have been marked as warranting concurrent execution. There are two advantages of including this module. Firstly, it demonstrates that the discourse representation system is robust enough to allow meaningful inference of the sort that might be made by a human cook. Secondly, it is of practical benefit to a user, and the growing area of smart kitchen appliances has specifically called for a representation of the cooking process that facilitates temporal optimization (Hamada et al., 2005; Reichel et al., 2011).

Content Realization The final phase is to create text from the organized content. Each process contains an imperative formed by an, often simple, combination of the action and food class from which the process was formed in the content production phase. The realized text is a successive printing of each imperative in the organized list of processes generated in the previous phase; along with the ingredients list, the total cooking time, and the times during the cooking at which the cook will be passive.

Figure 2 shows an example of a possible input-output pair for FASTFOOD. The corresponding fully worked example can be found in the appendix.

Section 2 relates FASTFOOD to the existing literature on NLP of cooking recipes. Section 2.1 conducts a comparison with an early example of an NLG system for cooking recipes, (Dale, 1990). Section 2.2 discusses an instance of the more common NLU approach, specifically a technique for depicting recipes as workflows. A potential problem with this technique is demonstrated, and it is shown that workflow representation is easily accomplished in FASTFOOD.

Section 3 describes the representation system in FASTFOOD. Section 3.1 addresses some problems with using discourse referents in the domain of cooking recipes. Section 3.2 introduces FASTFOOD’s representational structures: descriptive strings and processes. Section 3.3 briefly relates this representational system to a long standing philosophical debate on the nature of names and descriptions.

Section 4 describes the content production module. Sections 4.1 and 4.2 introduce two types of structure, cooking actions and food classes, that interact to produce a number of descriptive strings and processes. Section 4.3 describes the elements of the system created to deal with the existence of multiple descriptions for the same dish e.g. ‘chips’ vs. ‘fried sliced potatoes’.

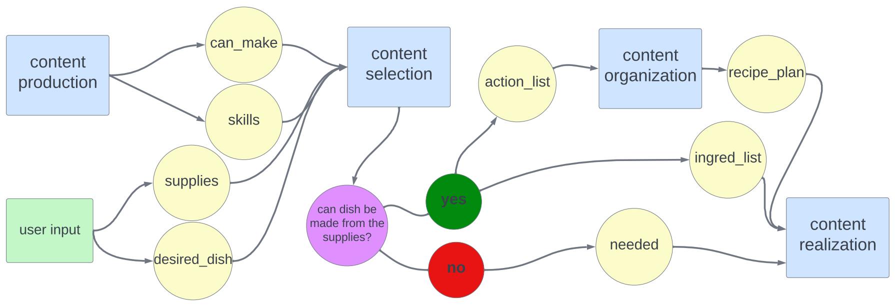


Figure 1: Graphical overview of our recipe generation system. Yellow circles indicate data, in various structures, as it flows through the system is *can_make* is the set of all makeable foods; *skills* is the set of all performable processes; supplies is the set of ingredients on hand; *desired_dish* is the descriptive string of the dish to make; *action_list* is partially ordered set of instructions; *needed* is set of ingredients that are missing from *supplies*; *ingred_list* is subset of *supplies*; *recipe_plan* is parallelized, topological sort of the instructions from *action_list*.

Section 5 is an exposition of the content selection module. Section 5.1 directly describes the algorithm that selects the appropriate descriptive strings and processes from FASTFOOD's internal database in response to a user input. Section 5.2 compares this algorithm with automated methods of parsing and, in more detail, with Automated Theorem Proving.

Section 6 describes the content organization module, which arranges the selected instructions into a coherent, temporally optimized recipe.

For algorithm description, a basic style pseudocode is used. Syntactic definitions are, where appropriate, given in Backus-Naur form. A Python implementation is available at https://github.com/Lou1sM/FastFood.

2 Related Work

The majority of recent computational approaches to cooking recipes have focussed either on searching and annotating large online databases, or on deriving a graphical depiction of a given recipe in terms of a workflow. These are efforts of Natural Language Understanding (NLU), which is a fundamentally different approach to the recipe generation attempted by FASTFOOD. On the side of generation, there exists an early and detailed system which also takes the generative approach in Dale's EPICURE (1992) (2.1). More recently, there has been some work applying neural language models and rule-based systems to the problem of recipe generation, discussed in Section 2.3 and 2.2, respectively.

```
INPUT:
vegetable dahl

OUTPUT:
Time: 53mins
Ingredients:
coconut milk
lentils
olive oil
raw peppers
raw broccoli

Instructions:
0mins: while placing water in pot and heating on cooker; chop the peppers, chop the broccoli
5mins: while adding the lentils to the boiling water and cooking for 45mins, stirring occasionally; heat olive
 oil in pan and add chopped vegetables, fry until soft
50mins: strain the lentils
51mins: add the sauteed vegetables and coconut milk to the lentils and stir

Passive:
from 4mins30secs to 5mins while while placing water in pot and heating on cooker; choping the peppers,
choping the broccoli
from 7mins30secs to 50mins while while adding the lentils to the boiling water and cooking for 45mins, stir
ring occasionally; heating olive oil in pan and adding chopped vegetables, frying until soft
```

Figure 2: An example of an input-output pair, the user inputs the title of a dish and receives a recipe for that dish along with the total cooking time and the passive intervals that occur during the recipe.

2.1 EPICURE

EPICURE is a program for recipe generation proposed by Dale (1989). It contains a representation system for recipe discourse and a method of generating a text describing this recipe. It is similar to FASTFOOD in that its focus is on natural language generation (NLG) as opposed to natural language understanding (NLU), and the difficulties identified by Dale (1992) in semantically representing cooking discourse are similar to those identified in Section 3.1 of the present paper. Thus, EPICURE and FASTFOOD both aim to achieve the same goal—a method of text generation in a domain with mutable entities, see Section 3.1 and Dale and Haddock (1991), Chapter 2.

However, there are considerable differences in the structure and performance of the two programs. As discussed in the Section 1, NLG systems can generally be thought of as first performing strategic choices of 'what to say', and then moving towards a tactical specification of 'how to say it' K. McKeown (1992); K. R. McKeown (1985). This framework can provide a fruitful comparison between FASTFOOD and EPICURE. Within such a comparison FASTFOOD focuses primarily on the 'what to say' component of NLG, employing an ATP style algorithm to determine the steps necessary in a cooking recipe (Section 5), and to derive a time-efficient way of performing these steps (Section 6). There is no scope for verb phrase coordination—it may for example contain a fragment such as ‘peel the carrot, chop the carrot, roast the carrot’, instead of ‘peel chop and roast the carrot’—and in certain contexts verbs do not receive the correct inflexion—‘while boil the lentils..’ instead of ‘while boiling the lentils. . . ’. EPICURE, focuses more on tactical realization than strategic planning, such as correct use of anaphora and level of conjunction application: “add the salt and pepper” vs “add the salt and add the pepper”.

This difference is also reflected in the use of discourse referents. As discussed in Section 3.1, these are not used by FASTFOOD at all, whereas they are employed in a detailed manner by EPICURE in order to assist in generating referring expressions.

However, there are also disadvantages to the representation system of EPICURE as compared with FASTFOOD. Firstly, it is less efficient in its content generation, deriving its discourse content, ‘what to say’, from a plan library that, for a given dish, explicitly contains the steps necessary to make it. For example, the fact that beans can be prepared by first soaking, then draining and then rinsing is hard-coded into the program. FASTFOOD on the other hand, creates its discourse content by selecting elements from a database of descriptive strings and processes, (Section 5) with this database in turn being produced by more fundamental structures corresponding to foods such as carrot, and cooking actions such as chopping (Section 4).

There are strengths and weaknesses to both systems. EPICURE’s more detailed discourse representation, in terms of discourse referents and states holding at a sequence of time points, produces a smoother and more natural output text but requires pre-existing specification of discourse content. FASTFOOD on the other hand, uses a lighter representation of discourse as a set of processes and descriptive strings. This sometimes produces unnatural output text but means that discourse content can be formed in response to user input The advantages of each could possibly be achieved by describing the processes in FASTFOOD as transitions between the state descriptions in EPICURE. The state descriptions in EPICURE are formed from EPICURE objects and so encode grammatical properties such as ‘count noun’ and ‘plural’ as well as cooking-related properties such as ‘soaked’ and ‘whole’. This may allow the recipe to be generated and optimized by an architecture like that of FASTFOOD, with the resulting discourse being represented by the discourse referents and states of EPICURE.

2.2 Rule-based Systems

FASTFOOD, similar to EPICURE (Dale, 1989) discussed in Section 2.1, is a program for Natural Language Generation. Most existing rule-based work on cooking recipes however, approaches the problem from the side of Natural Language Understanding (NLU). There, the goal is to analyse a natural language recipe and extract a more abstract representation. The purpose of such extraction can be quite varied: aligning actions across recipes (Donatelli et al., 2021), recipe search (Yamakata et al., 2017), recipe recommendation and alteration (Cordier et al., 2014; Gaillard, Nauer, Lefevre, & Cordier, 2012; Gunamgari, Dandapat, & Choudhury, 2014); recipe comparison (Mori, Sasada, Yamakata, & Yoshino, 2012); instruction of robotic cooking assistants; or translation recipes from one natural language to another (Mori, Maeta, Yamakata, & Sasada, 2014). However, the representation systems themselves are more uniform. While some technical systems exist for annotating recipes with logical or computer language notation (Cazzulino, Aprea, Greenwood, & Hart, 2004), the majority of recent work has used the intuitive method of representing recipes as workflows. The technical structure for such a representation is a directed acyclic graph (DAG) where an edge from step u to step v indicates that v depends on, and so must be performed before, u. An example of a workflow representation can be seen in Figure 5 in the appendix, which represents FASTFOOD's recipe for vegetable dahl.

Expressing a cooking recipe in the form of a workflow requires identifying the steps in the recipe and the dependencies that exist between them. The typical method for extracting workflows from procedural texts has been described in general by (Abend, Cohen, & Steedman, 2015), and in the specific domain of cooking recipes by (Dufour-Lussier, Le Ber, Lieber, Meilender, & Nauer, 2012; Hamada, Ide, Sakai, & Tanaka, 2000; Mori et al., 2014). It uses a combination of named entity recognition (NER) and predicate-argument analysis (PAA). Firstly, NER is performed on each instruction to extract the name[1] of the food being operated on. Next, a predicate-argument structure is formed for each verb, essentially by pairing it with the nearest noun that was identified in NER. Formally, these extracted entities correspond to the vertex set of the DAG. Finally the dependencies between these structures — formally the edges of the DAG — are determined using the order in which they appeared in the text — a predicate -argument structure A depends on another B if they share the same argument and A appears later in the text than B. The recipe fragment 'peel the carrot, chop the carrot' would be represented as 'peel(carrot), chop(carrot)' and so the latter would be said to depend on the former, i.e. they would be joined with a directed edge. However, this technique faces a problem in certain circumstances which comes back to the transitory quality of the ontology of a cooking discourse (see Section 3.1). Specifically, it would seem unable to identify dependencies in cases where new entities have been introduced. Consider the following online recipe for sweet potato burgers.

1. chop cilantro and leaves
2. bake patties for 35 min
3. place burgers on burger bun

The entity 'burgers' has never been mentioned before, so there would be no guide as to which nodes it should be joined, no reason for 3 to be joined to 2. If it were specified that every node must be joined to at least one preceding node (formally that the DAG be connected) then there would be nothing in the above technique to indicate that 3 should be joined to 2 rather than 1.

Of course, one could supplement this technique in an attempt to address such problems. For example, (Hamada et al., 2000) also employ a dictionary of cooking items. However, even a dictionary may fail to resolve this problem: the entry for 'burgers' does

[1] See Section 3.3 of the present work for a brief philosophical discussion of the ontological properties of names in cooking recipes.

not contain the word ‘patties’ in any of 4 large online dictionaries (Oxford Dictionary of English 2022; Collins Dictionary 2022; Cambridge Dictionary, 2022; Merriam-Webster 2022), so it is not clear how this could directly provide the missing link between 3 and 2.

In FASTFOOD on the other hand, workflow representation is readily attained by the methods described in Sections 4 and 5. Indeed this representation is generated first, and then used as the basis for generating the text. In the appendix, Figure 5 is a graphical depiction of this representation in the case of the worked example. This can be seen as a more general advantage of exploring the NLG of cooking recipes: when the representational elements are created by the system itself, they can be created in a way that facilitates the extraction of whatever information is most pertinent. In this case, representing cooking instructions in terms of input and output states (Section 3) allows dependencies to be readily established by matching the input of one with the output of another (Section 6.1).

These dependency relations were used in FASTFOOD’s temporal optimization module (Section 6). Such optimization required the consideration of multiple orders for the steps in the cooking recipe, and accurate identification of dependency relations was necessary to establish which orders produce coherent recipes (see Section 6.1). Further to this point, the only account of recipe optimization in the surveyed literature (Hamada et al., 2005), observed that automated DAG extraction was insufficiently accurate for their purposes, and that a manual annotation was required instead.[2] Thus it can also be said that the NLG approach to cooking recipes is helpful to their temporal optimization: generating (as opposed to extracting) discourse representations aids in the accurate identification of dependency relations, and this in turn is necessary for any optimization process that considers reorderings.

Another potential approach to cooking recipe generation is via controlled languages (Kuhn, 2014), which have been applied as knowledge representation languages for other applications of rule-based reasoning systems (Pulman, 1996). No work, to our knowledge, has attempted this, but the similarity between cooking recipes and controlled languages has been noted by at least one author (Mogensen, 2004).

2.3 Neural Approaches

Much of the natural language generation research in recent years has been dominated by neural language models. These are, generally very large, over-parametrized (more parameters than datapoints) neural networks trained on very large datasets to perform next-token prediction (Li, Tang, Zhao, & Wen, 2021). Currently, top-performing models have several billion parameters, (He, Gao, & Chen, 2022; Patra et al., 2022; W. Wang et al., 2020), with some reaching hundreds of billions (Brown et al., 2020). These have several advantages, such as obviating the need for feature engineering and showing, so far, continuing improvement with each increase in model and dataset size (Brown

[2]The optimization attempted by Hamada et al. (2005) is different to that of FASTFOOD as it was not solely concerned with temporal optimization, but also the optimization of other factors such as utensil use.

et al., 2020; Kaplan et al., 2020). However, a significant disadvantage is the lack of guarantees and consistency on the output. For example, a question-answering system using a large language model is highly sensitive to how the question is asked (Shin, Razeghi, Logan IV, Wallace, & Singh, 2020), and can give completely different answers when asked the same question twice.

This instability makes large language models less suitable for structured text generation, where it is very important that the output be globally coherent, rather than free-form text, where the output can drift from topic to topic without needing to make reference to content from many tokens back. One example of such structured text is code generation. Several works have applied large language models to code generation. The results show some promise and can often generate correct code, but the average accuracy is almost never above 30% (Xu, Alon, Neubig, & Hellendoorn, 2022), with models especially struggling when code requires longer sequences of steps (Chen et al., 2021). It has also been shown that, even when generated code is correct, the model is unable to predict the output of basic programs (Chen et al., 2021).

Cooking recipes are another example of structured text generation. A recipe needs to be a meaningful sequence of steps that a user can follow, where the ingredients for each step are all in the ingredients list or produced by previous steps. If even one ingredient is missing or one instruction step does not make sense, the entire recipe can be ruined. For these reasons, existing recipe generation by large language models has focused on completing partially given recipes (H. Lee et al., 2020) or measuring perplexity on existing recipes (Parvez, Chakraborty, Ray, & Chang, 2018). Perhaps the most successful results were found by conditioning the recipe generation on an image of the food to be cooked (Salvador, Drozdzal, Giró-i-Nieto, & Romero, 2019; H. Wang, Lin, Hoi, & Miao, 2020). Thus, although deep learning dominates the current research on natural language generation, it is not currently able to generate meaningful cooking recipes from scratch. Moreover, as it struggles to identify the purpose or output of generated code, it is very unlikely to be able to perform temporal reasoning about a cooking recipe, even if it were able to generate such a recipe.

Our proposed method, in contrast, can reliably generate cooking recipes from scratch, perform temporal reasoning to determine the passive times, and perform temporal optimization to rearrange the recipe to make it more time efficient. Additionally, in comparison with deep learning approaches, it carries the obvious advantage of not requiring large datasets or compute, and not requiring any time to train. Most importantly, deep learning does not give any insight into the cognitive nature of foods, cooking actions and abstract food classes, but our model does, via the structures that emerge naturally from the representation system, as described in Section 3.

3 Representation System

This section describes the format in which discourse content is represented in FAST-FOOD. Section 3.1 discusses problems arising from the use of discourse referents in the domain of cooking recipes, and explains why they are not used here. This discussion is included because discourse referents are a common modelling framework for discourse, and so our choice not to use them here may be thought to require some justification. Section 3.2 exposits the structures termed descriptive strings and processes, which together form the representation structure. Section 3.3 sketches a brief connection between this representation choice and a long-standing debate in philosophy on the nature of names, objects and descriptions. This section is not needed to understand the behaviour of our system itself, but may be of interest to one concerned with the general philosophical issue of reference vs description.

3.1 Problems with Discourse Referents

Discourse referents constitute a popular and often useful method for semantically representing a discourse. Each discourse referent corresponds to an entity discussed in the discourse, and the semantics of the discourse can then be described as various changes in the properties of these referents. Discourse referents have been used with success for a number of different purposes: e.g. modelling of anaphoric reference (Kamp, Genabith, & Reyle, 2011; Kamp & Reyle, 1993), topic segmentation/document summarization (Webber, Egg, & Kordoni, 2012), and opinion mining (Indurkhya & Damerau, 2010; Mooney, 1996; Sarawagi, 2008). However, in the domain of cooking recipes, their use may be problematic. Cooking is essentially an activity which makes new things out of old things, so we would expect entities to come into and out of existence over the course of a recipe; and for a representational system based on these entities, this irregularity in the ontology presents some difficulties. Firstly, how could it be determined whether a new noun phrase calls for a new referent? It does not seem that chopped carrot should indicate something entirely new, rather it should be a modification of the already existent, (raw) carrot; but perhaps the same cannot be said of carrot puree or carrot soup. Some actions transform one thing into another whereas some simply modify the original, and deriving a definite rule to distinguish is non-trivial. A second, related, problem is in accounting for the disappearance of individuals. After an instruction like ‘add the salt to the boiling water’ there is no longer (at least in the immediate sense) an individual called ‘the salt’; whereas after ‘add the quinoa to the boiling water’ the entity corresponding to the ‘the quinoa’ is extant. The latter might be followed by ‘strain the quinoa’, but the former could not be followed by ‘strain the salt’. Some additions constitute combinations in which the parts are no longer easily distinguishable or separable, and some do not; and again there is no obvious rule to capture this difference. Indeed, even among artificial intelligence programs specifically designed for chemical reasoning it is not clear that there would be anything capable of a cook’s intuitive understanding of transformation vs. modification, or of reversible vs. irreversible combinations (McCudden, 2017). Despite being a fruitful approach elsewhere, the utilization of discourse referents encounters difficulties in the domain of cooking recipes. As humans can easily intuitively understand the meaning of cooking recipes, these difficulties also suggests that our semantic representation of cooking recipes does not involve discourse referents, that they do not form a part of human understanding, and so they are not employed in the current representation.

3.2 Descriptive Strings and Processes

The fundamental representational unit used in FASTFOOD is a structure called a descriptive string. Intuitively descriptive strings can be thought of as a partial description of the state of affairs at some stage during the carrying out of a recipe. Formally they are written as strings in natural language — ‘there is a mixture of fried vegetables and boiled lentils’, ‘there is chopped carrot in boiling water’. However, for the sake of simplification, the initial ‘there is’ is omitted. Descriptive strings are then used to define processes, defined by an input and output set of description strings. Intuitively, processes correspond to the acts performed by the cook, e.g. chopping carrots, which contains ‘raw carrot’ in the input and ‘chopped carrot’ in the output, i.e. chopping carrot is an action that begins with there being carrot and ends with there being chopped carrot. As will be seen in Section 6, it is necessary to keep track of both the total time taken by the action, and the passive time that exists during the action.

The discourse content for a recipe comprises an ingredients list, i.e., a set of descriptive strings, and a list of cooking instructions, i.e. a list of processes. The only constraint on descriptive strings is a semantic one, that they describe a state of affairs. Here we use strings in natural language, but other alternatives are possible, as briefly discussed in Section 2.

3.3 Things vs Descriptions

A representation system in terms of states of affairs can be seen to be related to a long-standing philosophical debate on the logical and semantic properties of names, objects and descriptions. Introduced in its current form by Frege et al. (1892), this debate is now well over 100 years old (French, Uehling, & Wettstein, 1979; Quine, 1948; Russell, 1905, 1910; Searle, 1958). The central point has been to question what is indicated by using a name, whether names are essential and, if not, what they might be replaced with. It should be noted that typically the discussion was of named objects vs. proper definite descriptions, and upon reflection cooking recipes cannot contain either proper definite descriptions or uniquely identifying names[3]. If they did they would become chronicles of one particular instance of cooking, for example describing what took place in John’s kitchen yesterday, rather than the blueprint for a potentially infinite number of such chronicles that we expect them to be. The idea of a name can be employed in a looser sense to provide a reference to an entity, rather than attaching an enduring label to it—‘the carrot’ as opposed to ‘John’.[4] For example, it is this looser sense at play in the technique of Named Entity Recognition (discussed in Section 2.2).

[3]Brand names may occur, but, we contend, this would be as noun-modifiers, ‘the Cadbury chocolate’, rather than names ‘Cadbury was acquired by Mondelez’.

[4]See Evans (1982)) for a comprehensive discussion of different sorts of names and references.

However, to avoid confusion remainder of this paper equates the term ‘name’ with proper names, such as ‘John’.

With an acknowledgement of the difference between named entities and discourse referents in mind, a parallel can be drawn between names vs. definite descriptions on one hand, and discourse referents vs. states of affairs on the other. A particular connection might be seen with Russell’s [1905] well-known analysis of ‘the present king of France’ as being a description rather than a reference, anything that fits the description of being the present king of France, or Searle’s 1958 claim that names are merely ‘pegs on which to hang descriptions’ (p172). These both suggest that descriptions are more fundamental than named objects. Even Wittgenstein’s famously cryptic remark that ‘the world is the totality of facts, not of things’ [1922; 1961] can be broadly interpreted as stating that everyday objects, such as tables and chairs, supervene on states of affairs (see Copi and Beard (2013); Georgallides (2015); Miller (2017) for discussion of a more detailed interpretation). The key point of similarity is that entities (whether named or not) are not essential, but rather to be used only when convenient. In many circumstances it is indeed efficacious to be able to group multiple properties together. For example, entities can be helpful for discourse segmentation methods: if entities are in fact pegs holding together multiple properties or ideas then it would be expected that identifying a change in the entities under consideration would be an effective way of sectioning a text (more effective than, for example, identifying a change in the verbs being used) (Bayomi, Levacher, Ghorab, & Lawless, 2015; Fragkou, 2017). However, in the cooking domain, whose central purpose is to manipulate and recombine to produce a desired outcome, the same properties do not remain together for very long. Consequently, entity-based understanding, which often offers a simplification, was felt to be less appropriate than a representation system in terms of states of affairs.

4 Content Production

The algorithm of this section determines the set of all possible descriptive strings, termed *can_make*, and the set of all processes that the cook can perform, termed *skills*. These sets are then used by the content selection and organization modules (Section 5 and 6, respectively). In order to efficiently generate the sets *can_make* and *skills*, we can exploit the similarity in the input-output structure of many different elements. For example, one process would take “(there is) raw broccoli” as input and return “(there is) chopped broccoli”, another would take “(there is) raw carrot” as input and return “(there is) chopped carrot”. To this end, we use two intermediary structures, cooking actions and food classes, which intuitively correspond to verbs and nouns, respectively. Cooking actions (verbs) generalize the change between input and output across a set of processes, food classes (nouns) generalize the similarity between input and output across a set of processes. The current version of FASTFOOD contains 8 cooking actions, including chop, boil, mash and soak; and a number[5] of food classes including broccoli,

[5]This number is different at different points in the production phase, see Algorithm 2.

coconut milk, potato and pasta. The formation of *skills* requires a specification of how these interact to obtain meaningful processes. The term ‘meaningful’ is used to describe a process that it is pertinent for FASTFOOD to consider; this excludes those that may be called physically impossible, such as chopping coconut milk along with those that may be possible but are highly impractical, such as soaking pasta. The complete action of the content production module is given in Algorithm 2.

4.1 Cooking Actions

Syntactically, each cooking action is a set of processes of a similar form. For example, the above processes would be generalized under the cooking action ‘chop’. Part of the form for ‘chop’ is that the output is ‘chopped’ concatenated with the input. These sets are formed separately and then their elements are all added to *skills*. The general form for the cooking action ‘chop’, that is, the form of all processes that fall under the ‘chop’ cooking action, is given in Grammar 1. For this, and all following syntactic specifications, are given in Backus-Naur form.

⟨*chop-process*⟩ ::= ⟨*chop-input-pair*⟩ ⟨*chop-output-pair*⟩ ⟨*chop-time-pair*⟩ ⟨*chop-ftime-pair*⟩ ⟨*chop-direction-pair*⟩

⟨*chop-input-pair*⟩ ::= ‘input’ ⟨*choppable-food*⟩

⟨*chop-output-pair*⟩ ::= ‘output’ ‘chopped’ ⟨*choppable-food*⟩

⟨*chop-time*⟩ ::= ‘time’ float

⟨*chop-ftime*⟩ ::= ‘f time’ 0

⟨*chop-direction*⟩ ::= ‘direction’ ‘chop the’ ⟨*choppable-food*⟩

Grammar 1: Grammar for the chop process.

Certain values of the descriptive string ‘choppable-food’ could produce non-meaningful processes, such as chopping hummus. Thus, it must be specified which descriptive strings are compatible with which actions. Directly defining a separate set of suitable descriptive strings for each cooking action would require many specifications, e.g., ‘chop’ getting ‘raw carrot’, ‘peeled carrot’, ‘boiled carrot’, but not ‘chopped carrot’. Indeed, this would be just as tedious as specifying each allowable process directly. The problem is instead solved using an intermediary structure termed food classes, which correspond closely to nouns.

4.2 Food Classes

They are abstract entities that indicate compatibility or incompatibility with each of the cooking actions. They capture that, for example, the descriptive string ’raw carrot’ is compatible with chop and peel, but not with mash. This information is encoded by a class Carrot, which has a root string ‘carrot’, a state string, e.g., ‘peeled’, and an indication as to which cooking actions it is compatible with. In order to also capture the time that can be taken for the action to apply to the food class, e.g., boiling spinach is considerably quicker than boiling lentils, this indication is given by a float specifying the ‘time’ value of the process (in seconds) that will be produced, or -1 if the action and food class are incompatible. This element is referred to as an indicator. The full syntax for food classes is shown in grammar 2. The eight indicators specify compatibility/incompatibility with the eight cooking actions. For example, if the first indicator is false, it means the foodclass cannot be chopped.[6]

$$\begin{aligned}\langle \textit{foodclass} \rangle \quad &::= \langle \textit{root} \rangle\ \langle \textit{state} \rangle\ \langle \textit{Indicator} \rangle\ \langle \textit{Indicator} \rangle\ \langle \textit{Indicator} \rangle\ \langle \textit{Indicator} \rangle \\ &\quad\ \ \langle \textit{Indicator} \rangle\ \langle \textit{Indicator} \rangle\ \langle \textit{Indicator} \rangle\ \langle \textit{Indicator} \rangle \\ \langle \textit{indicator} \rangle \quad &::= \text{False} \mid \text{float}\end{aligned}$$

Grammar 2: Grammar for foodclass.

Having defined these food classes, it becomes apparent that some sets of food classes are similar, both in their descriptive string and in their indicators, and it is therefore efficient to group them under a common abstraction. Carrot is grouped with parsnip, potato, sweet potato and onion under the abstract class RootVegetable, which specifies that they can be peeled and chopped and boiled, but not soaked, and can be fried only if their state included ‘chopped’ etc. This can be accomplished naturally in our object-oriented programming description, where RootVegetable is an abstract class from which Carrot inherits. Exceptions to the properties of the parent class, such as onion only being able to be fried if it has been peeled, can be simply overridden in the child class.

The sets *can_make* and *skills* can then be defined by instantiating an object for each food class, the iteratively applying the processes corresponding to each of the actions. So, beginning with a Carrot object in the ‘raw’ state, the cooking action Peel could be applied to produce the new process that converts “raw carrot” to “peeled raw carrot”, and the new descriptive string “peeled raw carrot”. These are then added to *skills* and *can_make*, respectively. The peel indicator on the carrot object is also changed to False, capturing the fact that you cannot peel what has already been peeled. An example, for the *chop* action, is described formally in Algorithm 1. Other actions are defined similarly, but with some slight differences in the changes to the indicators, e.g. the fry function also sets the ‘boilable’ indicator to false, as we do not wish to consider boiling a food after frying it. The action of Algorithm 1 is recursive, applying to the modified food objects from previous applications, but the recursion depth is limited to the number of cooking actions, because of the restriction that no action can be applied to the same food object twice. As an example, Algorithm 1 would operate

[6] An alternative solution, in a functional paradigm, could be to use dependent types to restrict the arguments to the chop function, see Ranta (2011), p.130.

on the food class *raw_carrot* as follows (using python-style syntax):

$$\begin{aligned}
&\textbf{input}\\
raw\_carrot \;&=\; \{\text{`root:'}, \text{`carrot:'}\\
&\quad \text{`state:'}, \text{`raw:'},\\
&\quad \text{`choppable:'}, \text{True},\\
&\quad \text{`boilable:'}, \text{True},\\
&\quad \text{`fryable:'}, \text{True},\\
&\quad \text{`soakable:'}, \text{False},\\
&\quad \vdots\\
&\textbf{output}\\
chopped\_carrot \;&=\; \{\text{`root:'}, \text{`carrot:'})\\
&\quad \text{`state:'}, \text{`chopped:'},\\
&\quad \text{`choppable:'}, \text{False},\\
&\quad \text{`boilable:'}, \text{True},\\
&\quad \text{`fryable:'}, \text{True},\\
&\quad \text{`soakable:'}, \text{False},\\
&\quad \vdots
\end{aligned}$$

Similarly, Algorithm 2 produces all possible processes. For example, the processes corresponding to the above input-output pair is

$$\begin{aligned}
chop\_carrot \;=\; \{&(\text{input}, \{\text{`raw carrot'}\}),\\
&\text{output}, \{\text{`chopped carrot'}\}),\\
&\text{time}, 120),\\
&\text{ftime}, 0),\\
&\text{direction}, \text{`chop the carrot'})\}\,.
\end{aligned}$$

At this point in the production phase, *skills* includes all processes formed from the interaction of the appropriate food classes and cooking actions, and *can_make* contains all descriptive strings that are inputs or outputs to these processes. These are then added to by the structures discussed in Section 4.3.

Algorithm 1 Formal description of a function used to create new food classes, f[choppable], f[boilable] and f[fryable] refer respectively to f's indicators for chopping, boiling and frying.

```
function CHOP($x$)
    $x \leftarrow$ a food class
    $chopped\_x \leftarrow x$ will be updated to show effects of chopping
    $chopped\_x$[state] $\leftarrow$ 'chopped'
    $chopped\_x$[choppable] $\leftarrow False$
    Return chopped_x
```

4.3 Synonyms and Ghosts

The production method described in Sections 4.1 and 4.2 produces only those descriptive strings constructed from combinations of the given cooking actions and food classes, e.g. 'chopped carrot', 'mashed chickpeas'; and this excludes the likes of 'hummus', 'ratatouille' or 'chips'. These sorts of descriptions, unlike 'chopped carrot' have nothing in their linguistic structure to indicate their physical structure. The production methods of Sections 4.1 and 4.2 would apply the cooking action for frying to the descriptive string 'sliced potato' to produce 'fried sliced potato'; but a human would instead use the description 'chips'. Therefore, we encode these anomalous descriptions, like 'chips', along with a synonymous one that could be understood by FASTFOOD, 'fried sliced potato'. Such a relation between two strings, one of which is a descriptive string produced by Algorithm 2, is defined syntactically in Grammar 3.

$\langle synonym \rangle \quad ::= \langle descriptive\text{-}string \rangle \ \langle descriptive\text{-}string \rangle$

Grammar 3: Grammar for synonym.

For each synonym (x, y) a process is produced which has x as input, y as output, zero time, zero f time and empty direction. These processes are termed ghost processes and are defined syntactically in Grammar 4. In the case of 'chips', this results in

$$
\begin{aligned}
S &= (\text{'fried sliced potatoes'}, \text{'chips'}) \\
G &= \{(\text{input}, \{\text{'fried sliced potatoes'}\}), \\
&\quad (\text{output}, \{\text{'chips'}\}), \\
&\quad (\text{time}, 0), \\
&\quad (\text{ftime}, 0), \\
&\quad (\text{direction}, \text{''})\}
\end{aligned}
$$

Algorithm 2 Formal description of the algorithm used to produce many descriptive strings and processes, which are then stored in can_make and skills. The fields of each process are defined as in Algorithm 1.

```
function PRODUCE_CONTENT(foods)
    $foods \leftarrow$ predefined set of food classes
    $chop \leftarrow \emptyset$ , will be updated to contain all chopping-related processes
    $boil \leftarrow \emptyset$, will be updated to contain all boiling-related processes
    $fry \leftarrow \emptyset$, will be updated to contain all frying-related processes
    $skills \leftarrow \emptyset$, will be updated to contain all chopping- boiling- and frying-related processes
    $can\_make \leftarrow \emptyset$, will be updated to contain recognized descriptive strings
    for each $f \in foods$ do
        add $f$[description] to $can\_make$
    for each $f \in foods$ do
        if $f$[choppable] is a float then
            $chopped\_f \leftarrow chop(f)$
            $chop\_f \leftarrow$ (input, $f$[description]), (output, $chopped\_f$[description]), (time, $f$[choppable]), (f time, 0), (direction, 'chop the $f$[root]')
            add $chopped\_f$ to $foods$
            add $chop\_f$ to $chop$
        if $f$[boilable] is a float then
            $boiled\_f \leftarrow boil(f)$
            $boil\_f \leftarrow$ (input, $f$[description], 'boiling water'), (output, $boiled\_f$[description]), (time, $f$[boilable]), (f time, $f$[boilable]$-30$), (direction, 'boil the $f$[root] for $f$[boilable]')
            add $boiled\_f$ to $foods$
            add $boil\_f$ to $boil$
        if $f$[fryable] is an float then
            $fried\_f \leftarrow fry(f)$
            $boil\_f \leftarrow$ (input, $f$[description]), (output, $fried\_f$[description]), (time, $f$[fryable]), (f time, $f$[fryable] $- 120$), (direction, 'fry the $f$[root] for $f$[fryable]')
            add $fried\_f$ to $foods$
            add $fry\_f$ to $fry$
    $skills \leftarrow chop \cup boil \cup fry$
    Return $skills$
```

$\langle$*ghost-process*$\rangle$ ::= $\langle$*input-pair*$\rangle\langle$*output-pair*$\rangle$ $\langle$*ghost direction*$\rangle$ $\langle$*ghost-time*$\rangle$ $\langle$*ghost-f-time*$\rangle$

$\langle$*input-pair*$\rangle$ ::= 'input' $\langle$*descriptive-string*$\rangle$

$\langle$*output-pair*$\rangle$ ::= 'output' $\langle$*descriptive-string*$\rangle$

$\langle$*ghost-direction*$\rangle$ ::= 'direction' ''

$\langle$*ghost-time*$\rangle$ ::= 'time' 0

$\langle$*ghost-f-time*$\rangle$ ::= 'f time' 0

Grammar 4: Grammar for ghost process.

A ghost process with input x and output y, allows FASTFOOD to simplify the search for a recipe for y to the search for a recipe for x. On a request for a recipe for chips FASTFOOD would decide that chips can be made from 'fried sliced potatoes', and then begin searching for a recipe for the latter.[7] Ghost processes are removed after the content selection stage (see Section 5) and would make no appearance in the final generated output. Ghosts also provide a way of allowing for multiple methods of making the same dish. Chips may also be made by baking sliced potatoes, and so we could define a new synonym between 'chips' and 'baked sliced potatoes', along with the corresponding ghost process. In the current version, when there are multiple ways of making the given dish, a selection is made at random.[8] This calls into question whether 'chips' and 'fried sliced potato' are in fact synonymous after all. If synonymy is an equivalence relation, and both S and S' both describe such relations, then this would seem to imply the spurious conclusion that 'fried sliced potato' is a synonym of 'baked sliced potato'. One interpretation is that 'chips' has multiple senses, one corresponding to 'fried sliced potato' and another to 'baked sliced potato'. Another is that, rather than synonymy, the relation is really one of hyponymy-hypernymy, or near-synonymy Edmonds and Hirst (2002), with synonym reserved for pairs such as 'coriander' and 'cilantro'.[9]

Another feasible interpretation is to view ghosts as paraphrases: if asked for a recipe for 'chips', G would allow FASTFOOD to replace 'chips' with 'fried sliced potato', and instead search for a recipe for the latter. By employing various paraphrase extraction techniques to a number of online corpora, Ganitkevitch, Van Durme, and Callison-Burch

[7]A more precise function of these ghosts can be understood in the context of the content selection algorithm (Section 5) and the worked example in the appendix.

[8]While synonyms and ghosts are mostly used to allow multiple ways to make the same dish, or to relate names for processed foods such as hummus, to a longer descriptive string that reveals the contents of that food, they can also be used to model synonymous terms for basic ingredients, such as cilantro and coriander.

[9]This concern is purely semantic. Synonyms are ordered pairs—ghost processes do not exist in both directions—and so it would not be possible to use 'chips' as an intermediary and employ both S and S' to conclude that one can make 'fried sliced potato' from 'baked sliced potato' or vice versa.

(2013) have constructed a large database of paraphrases. This database has allowed a practical investigation of synonymous pairs based on the assumption that synonyms are words that can act as paraphrases for each other (Koeva (2015); Pavlick and Nenkova (2015); Preotiuc-Pietro, Xu, and Ungar (2016). However, as noted by Koeva (2015) synonymy understood in terms of paraphrasing may not formally be an equivalence relation—for example, 'brain' could likely be a substitute for 'encephalon' in the medical literature, but the reverse substitution could be incongruous in less formal discourse, and so the relation is not symmetrical.

Certain synonyms are built in to FASTFOOD from the beginning—'hummus', 'chips','guacamole','pancakes' etc., but often such a synonym may be required of the user, for example if a recipe for 'butter bean soup' were requested, then the user would be required to add a synonym such as

$$\begin{aligned} R \quad = \quad & (\text{`butterbean soup'}, \\ & \text{`liquidized mixture of butterbeans, milk and pureed vegetables'}) . \end{aligned}$$

It may seem a shortcoming of the system that a user would have to specify much of the essential information, but note that the question 'what exactly do you mean by butter bean soup?' is exactly the sort of question a menu reader, or personal chef, might ask. The description 'butter bean soup', like 'chips' is ambiguous in natural language, and the hesitation of FASTFOOD is an appropriate response to this ambiguity.

5 Content Selection

The content selection algorithm in this section requires the following for operation:

- the content produced by the methods of Section 4, consisting of *can_make* and *skills*
- a user description of the desired dish in a form understood by FASTFOOD, i.e. a descriptive string that is an element of *can_make*
- a user-specified subset of *can_make*, called *supplies*, corresponding to all foods the user has at home

The algorithm can then select which elements of *can_make* are required as ingredients, and which elements of *skills* are required to transform the ingredients into the desired dish. If the ingredients are not all in *supplies*, then the program terminates with output 'insufficient ingredients, you need: $A1, A2, ..., An$' where $\{A1, A2, ..., An\} = \mathit{ingred\_list} \setminus \mathit{supplies}$. Section 5.1 describes the algorithm for selecting both the subset of *supplies* that will form the ingredients list, and the subset of *skills* that will collectively transform the ingredients list into the desired dish. Section 5.2 relates this algorithm to parsing and, in more detail, to well established work in the field of automated theorem proving.

5.1 Selection Algorithm

Algorithm 3 Algorithm that selects the appropriate ingredients (descriptive strings) and steps (processes) to form a recipe for a dish requested by the user.

```
function SELECT_CONTENT(dish)
    $ingred\_list \leftarrow \emptyset$, will contain ingredients that are in $supplies$
    $needed \leftarrow \emptyset$, will contain ingredients that are not in $supplies$
    $action\_list \leftarrow \emptyset$, will contain all processes required to make the dish
    $looking\_for \leftarrow$list containing only the dish, items will be removed from this list
  by (a) finding them in $ingred\_list$ (b) placing them in needed or (c) finding a process
  for which they are the output and replacing them with the corresponding inputs
    while $looking\_for \neq \emptyset$ do
        for each $item \in looking\_for$ do
            if $item \in supplies$ then
                remove $item$ from $looking\_for$
                add $item$ to $ingred\_list$
            else if $item \notin can\_make$ then
                remove $item$ from $looking\_for$
                add $item$ to $needed$
            else
                for each $p \in skills$ do
                    if item is an output of $p$ then
                        remove $item$ from $looking\_for$
                        add the inputs of $p$ to $looking\_for$
                        add $p$ to $action\_list$
                        break
        if $needed == \emptyset$ then
            Return $ingred\_list$, $action\_list$
        else
            Print 'Insufficient ingredients, you need:'
            for each $x \in needed$ do
                Print $x$
```

Given the sets of recognizable processes and descriptive strings, *can_make* and *skills*, as computed in Section 4, along with a user description of the desired dish and a user-specified subset of *can_make*, called *supplies*, corresponding to all foods the user has at home, the content selection algorithm proceeds as follows. First, the descriptive string for the desired dish is set as the sole element of the list *needed.* The following is then performed until this list is empty: if the first element of the list is in *supplies* then it is removed and added to *ingred_list*; otherwise, if the first element of the list is not in *can_make* then it is removed and added to *needed*; otherwise, a process is found with the first element of the list as output (a failure of the previous condition guarantees that such a process exists), the element is removed and the inputs of the process are added to the end of the list, the process is added to *action_list*. If, after this loop, needed is non-empty then the dish cannot be made with the current supplies, and so the program terminates and the elements of *needed* are printed, preceded by the directive 'Insufficient ingredients, you need:'. Otherwise, *ingred_list* and *action_list* are returned to the content organization phase (Section 6).

5.2 Comparison with Parsing and Automated Theorem Proving

This algorithm has similarities to both parsing and automated theorem proving. Parsing is the general term for any technique that analyses a string of symbols in terms of its role in a formal grammar. A formal grammar, as it is considered here, is a set of rewrite rules, which describe how certain symbols can be rewritten as others, along with a privileged start symbol from which the rewriting process may begin. A subset of these symbols are defined to be terminal, and the set of all generated strings composed purely of terminal symbols is defined as the language of the formal grammar.[10] A prototypical instance of parsing is to determine whether a given string is part of a given language, and if so to identify the production rules from which it can be generated. For example, treating natural language as a formal grammar, a sentence can be parsed to determine whether it is grammatical and, if so, to produce a diagram, sometimes called a parse tree, detailing its grammatical structure. The similarity with the algorithm Algorithm 3 can be seen by equating production rules to processes, terminal symbols to the descriptive strings in *supplies*, and non-terminal symbols to all other descriptive strings. Both techniques aim to establish a connection between a given element (dish/start symbol) and a given set of elements (subset of *supplies*/string of terminal symbols) by applying certain transformations (processes/rewrite rules).

The start symbol in formal grammars is unique, but we could simply include, for each dish a user might specify, a rule transforming the start symbol into that dish, which would allow the parsing of more than one input dish. A difference with parsing is that the set of symbols in a string is ordered, whereas the descriptive strings in *ingred_list* are not—the parse of a natural language sentence would change if the order of the words was changed, but the same is not true with respect to the ingredients list of a recipe. There is existing work on applying parsing techniques to unordered (or only partially ordered) domains such as image recognition (Elliott & Keller, 2013; Hongxia & Li, 2008; Zhao et al., 2010) and narrative comprehension of stories ((Zhang, 1994) and the references therein). One solution is to use a formal grammar that does not discriminate based on order: a formal grammar such that for every permutation P and rewrite rule

$$a_1, \ldots, a_n \to b_1, \ldots, b_n \,,$$

[10]For discussion of formal grammars see Chomsky (1956); Meduna (2014).

there also exists a rule

$$a_1, \ldots, a_n \to P(b_1, \ldots, b_n)\,.$$

Robust frameworks for representation of natural language with a separation of immediate dominance and linear precedence constraints are readily available – for example, Generalized Phrase Structure Grammar (Gazdar et al, 1985) – and decidable parsing regimes are available to them (Seiffert, 1987), with polynomial time complexity in some cases (Nederhof, Satta, & Shieber, 2003).

Another fruitful comparison is that with automated theorem provers (ATP). An ATP is an algorithm that seeks a method of proving a formula P from a known list of axioms and implicational rules. In the case of a backwards chaining ATP, the basic method for proving P is to search for an implication R with consequent P, thus reducing the search for P to a search for the antecedents of R, and continue to apply this method to these antecedents until the formulae to be proved are all in the list of axioms. If this can be done then P can be established as a theorem provable by the axioms reached and the implicational rules used to reach them.[11] When compared with the SELECT_CONTENT algorithm, the set of axioms corresponds to *supplies*, the implications to *skills*, and P to the desired dish. Deriving a proof of the formula P corresponds to generating a recipe for the dish. One important difference with ATP is that the latter contains explicit implications, 'raw carrot' $\to$ 'chopped carrot', and the former implicational rules e.g. $(P \to Q)$. Explicit implications can only be backward-chained if the consequent (output) exactly matches a specific proposition to be proved, whereas implicational rules can describe an arbitrary number of implications. The consequent of modus ponens for example, can be matched to any formula and so the search could proceed indefinitely: proving P_1 could be reduced to proving P_2 and $(P_2 \to P_1)$, then proving P_2 could be reduced to proving P_3 and $(P_3 \to P_2)$.[12] Here, however, if a formula (descriptive string) doesn't exactly match the consequent of any rule ($\notin$ *can_make*) then there are no options left and the formula can be declared unprovable (the dish unmakeable). This is an advantage. Formally, it means that given a set of foods and cooking actions, the set of makeable dishes is recursive, whereas it has long been known that the set of provable formulae is merely recursively enumerable (Gödel, 1929). Practically, it means that it is immune to the significant danger faced by ATP of being led down blind alleys and failing to halt. Heuristics are necessary to guide ATP search in an 'intelligent' direction, e.g. not continue to apply modus ponens as above. After initial interest in ATP in the 1950s and 1960s, the necessity of effective heuristics soon became clear (Bledsoe, 1984; Loveland, 2016). Automated theorem proving remains an open problem (Avigad & Harrison, 2014). Most work is still devoted to intelligent proof guidance (Sutcliffe & Suttner, 2001; Urban, Hoder, & Voronkov, 2010), with a growing trend to derive such guidance from machine learning ((Bansal, Loos, Rabe, Szegedy, & Wilcox, 2019; Bridge, Holden, & Paulson, 2014; Loos,

[11] For a full exposition of the various approaches to automated theorem proving, see (Geffner, 2013)

[12] Such a possibility recalls Lewis Carrol's famous discussion of the validity of Modus Ponens in a conversation between Achilles and the Tortoise [1895].

Irving, Szegedy, & Kaliszyk, 2017; Schulz & Sutcliffe, 2015). In contrast, Algorithm 3 does not require any search heuristics.

Another problem faced by automated theorem provers is that the resulting proofs, are often extremely long and difficult for human readers to understand, and so require postprocessing such as removing superfluous or repeated inferences (Fontaine, Merz, & Woltzenlogel Paleo, 2011) or graphically representing parts of the reasoning process (Flower, Masthoff, & Stapleton, 2004; Stapleton, Masthoff, Flower, Fish, & Southern, 2007). The inferences made by mathematicians are of a much higher order than the fundamental implicational rules such as modus ponens, a single step in the head of a mathematician might correspond to thousands or even tens of thousands of logical inferences (Fontaine et al., 2011). A cooking recipe, on the other hand, is rarely more than a couple of pages: the average recipe has less than ten ingredients (Kinouchi, Diez-Garcia, Holanda, Zambianchi, & Roque, 2008) and a brief search through 3 commercial recipe books (Culleton & Higgs, 2016; Flynn & Dave, 2016; Gavin, 2016) encounters nothing longer than 50 steps, with the vast majority being under 20.[13] SELECT_CONTENT is therefore unlikely to struggle with overly long output.

One potential issue with the treatment of recipe generation as theorem proving concerns resource sensitivity: applying a process results in the deletion of the input, whereas, in classical logic, applying a rule of inference does not delete the antecedent. So if the idea of chopping carrot were to be expressed as 'raw carrot' implies 'chopped carrot', then we could use the inference $A \wedge (A \implies B)) \implies (A \wedge B)$ to incorrectly conclude that if there is raw carrot and we can chop carrots, then we could produce a state of affairs in which there is both raw carrot and chopped carrot. However, this problem could be circumvented by a switch to a resource-sensitive logic. Linear logic, for example, includes the notion of linear implication to describe cases in which the antecedent must be deleted in order to produce the consequent (Galmiche, 2000; Galmiche & Marion, 1995; Galmiche & Perrier, 1994).

A final point is that the two objects of comparison in this section, parsing and ATP, can themselves be constructively compared. This comparison is not explored in any depth here, but the characterization of parsing as deduction has been conducted in detail by numerous authors (Shieber, Schabes, & Pereira, 1995) since its formulation by Pereira and Warren in 1983. The same idea is the basis of the significant work employing logic programming to parse natural language, (Matsumoto, Tanaka, Hirakawa, Miyoshi, & Yasukawa, 1983; Mooney, 1996; Stabler Jr, 1983), logic programming being essentially a systematization of simple ATP methods (Lloyd, 2012).

6 Content Organization

The algorithm in the previous section selects the required ingredients and cooking instructions, respectively *ingredlist* and *actionlist*, to produce the user-specified input dish. The next phase is to organize this content.

[13]The longest encountered was a recipe for Mushroom and Leek Pie with Roast Potatoes and Cabbage (Culleton & Higgs, 2016) p93., which was analysed as containing 32 steps.

6.1 Overview of the Optimization Problem

The principles used to guide this the organization of content are that the final recipe (a) be coherent, (b) be as time efficient as possible. Fulfilling (a) requires a sensitivity to the fact that not all orders form coherent recipes. Some processes can only be performed after the completion of others, e.g. one can only strain the lentils after boiling the lentils. Such constraints occur when one process takes as input a descriptive string that is in the output of another.

Definition 6.1.1 For any two processes P_1, P_2, $requires(P_1, P_2)$ iff the input of P_1 and the output of P_2 have non-empty intersection.

Definition 6.1.2 Two processes $P_1, P2$ are independent iff neither $requires(P_1, P_2)$ nor $requires(P_2, P_1)$.

Definition 6.1.3 A list of processes L is permissible iff for any P_1, P_2 in L, $requires(P_1, P_2)$ implies P_2 precedes P_1 in L.

Thus part of the requirement in (a) is that the list into which the processes in *actionlist* are organized is permissible. There are two reasons why the temporal optimization in (b) is desirable. Firstly it demonstrates that the representation system is robust enough to facilitate inference of the sort that might be made by a human chef; and secondly, optimization has been identified as a desirable feature of representations used by smart kitchen appliances (Hamada et al., 2005; Reichel et al., 2011). The basis for temporal optimization is that some processes have times during their execution which allows parallelization, e.g. peeling potatoes while the casserole is in the oven. Choosing to perform some processes $P_1, \ldots, P_n$ during the passive time of some other process P will be referred to as inserting $P_1, \ldots, P_n$ into P or combining $P, P_1, \ldots, P_n$, and the result will be referred to as a combination. The result of ordering and combining the processes in *actionlist* is referred to as a recipe plan. This terminology is formalized in Grammar 5.

⟨*organized-content*⟩ ::= ⟨*ingredients-list*⟩ | ⟨*recipe-plan*⟩

⟨*ingredients-list*⟩ ::= ⟨*descriptive-string*⟩ | ⟨*descriptive-string*⟩ ⟨*ingredients-list*⟩

⟨*recipe-plan*⟩ ::= ⟨*process*⟩| ⟨*combination*⟩ | ⟨*process*⟩ ⟨*recipe plan*⟩ | ⟨*combination*⟩ ⟨*recipe-plan*⟩

⟨*combination*⟩ ::= ‘while’ ⟨*process*⟩ ‘,’ ⟨*process-list*⟩

⟨*process-list*⟩ ::= ⟨*process*⟩ | ⟨*process*⟩ ‘and’ ⟨*process_list*⟩

Grammar 5: Grammar for organized-content

Just as there are restrictions on which orders are coherent, so too are there restrictions on which processes can be coherently combined. One condition for inserting P_1 into P_2

is that the cook be free for long enough during P_2 for P_1 to be performed. A process is encoded as a 5 element associative array and two key-value pairs of this array have form (time, M) and (f time, N) where M and N are non-negative floats. These key-value pairs respectively express the total time the process will take, and the amount of time during the process at which the cook is free. So by definition, combinations are only possible when the total time of the inserted processes is less than or equal to the free time of the process into which they are to be inserted. A second restriction is imposed by the requires relation. $requires(P_1, P_2)$ means P_1 must occur after the completion of P_2, so they cannot occur at the same time. One cannot chop the broccoli while boiling the chopped broccoli. The two conditions for the insertion of P_1 into P_2 are shown in Condition 6.1.4.

Condition 6.1.4 $P_1[\text{time}] \leq P_2[\text{f time}]$, and P_1 and P_2 are independent

Therefore another requirement of (a) is that no combinations are performed that violate these two conditions. Temporal optimization is performed by considering multiple organized discourses, calculating the total time of each organized discourse by adding together the times of each process that hasn't been inserted into another, and then selecting the organized discourse with the least overall time. It is contended that the precise way this is done is guaranteed to achieve an optimal solution, i.e. there will exist no faster coherent organized discourse with shorter overall time; however, a proof of this contention is outside the scope of the present work. The content organization algorithm contains two main components. The first generates all permissible lists of the processes in *actionlist*, the second acts on each list to combine certain processes. These two components are the respective subjects of the following two subsections.

6.2 FIND_ALL_LISTS

Before the application of the algorithm to generate all permissible lists of the processes in *actionlist*, any ghost processes (see Section 4.3) are removed. After doing this, the requires relations in *actionlist* must be updated to ensure no impermissible lists are generated. If A, B, G are 3 processes and G is a ghost; and if $requires(A, G)$ and $requires(G, B)$ then when G is removed, the relation $requires(A, B)$ must be added. Before the temporal optimization, ghost processes are removed and the *requires* relations updated accordingly, as decribed in Algorithm 4.

The input set of processes, in this case *actionlist*, is iterated through. For each ghost G found, the set of all processes that G requires are placed in new_requirements; then all processes that require G are updated to require everything in new_requirements; then G is removed.[14] After this removal of ghosts, the algorithm for finding all permissible

[14]The necessity of such a specification can be seen as follows. Before the removal of G, if permissibility requires that B precede G and G precede A, then B would have to precede A. If G were to be removed and the requires relations not updated, there would be nothing to prevent A preceding B, and this would result in an incoherent recipe. As discussed in Section 4.3, A semantically requires the input of G, and the fact that $requires(G, B)$ holds means the input of G is in the output of B. Therefore A semantically requires something in the output of B and so any order in

Algorithm 4 Algorithm for removing ghost processes from *actionlist* before application of FIND_ALL_LISTS.

```
function REMOVEGHOSTS(L)
    $L \leftarrow$ set of processes, some of which may be ghosts
    for each $item \in L$ do
        if $item$[time] == 0 then
            $newrequirements \leftarrow \emptyset$, will contain all elements that $item$ requires
            for each $item' \in L$ do
                if $requires(item, item')$ then
                    $newrequirements \leftarrow newrequirements \cup \{item'\}$
            for each $item'' \in L$ do
                if $requires(item'', item)$ then
                    for each $req \in newrequirements$ do
                        $requires(item'', req) \leftarrow True$
            $L \leftarrow L \setminus \{item\}$
```

lists, called FIND_ALL_LISTS, is run. A necessary precursor to this algorithm is Definition 6.2.1.

Definition 6.2.1 $norequirements(x, S)$ iff X is a process and S is a set of processes such that requires(x,s) does not hold for any element s of S.

If x is an element of the given set S then $norequirements(x, S)$ is equivalent to saying that x can be the first element in a permissible order of S. Algorithm 5 applies this idea iteratively to compute all permissible orders of a given set of processes. It begins by adding to paths all elements that have no requirements in S and so could come first in a permissible order. Then it iterates through paths, replacing each element x, with all lists $[x, y]$ where y is a process that doesn't require any other except for possibly x; then it again iterates through paths, replacing each list $[x, y]$ with all lists $[x, y, z]$, where z is a process that doesn't require any other except for possibly x or y. It iterates this operation a number of times equal to the cardinality of the input set. At this point paths contains all permissible orders as desired.[15]

The second component of the content selection algorithm is to temporally optimize and maintain coherence of a given order of instructions. Algorithm 6 receives as input

which A precedes B must be marked as impermissible. This is ensured precisely by specifying that $requires(A, B)$.

[15] It is possible to represent the steps in a recipe using a Directed Acyclic Graph. Figure 5 below is an example of such a representation, and techniques for extracting DAG representations from natural language recipes are discussed in Section 2.2. A DAG, in turn, is equivalent to a partial order, see Kozen (1992) for a proof of this equivalence and a general discussion of partial orders. Thus the above search for all permissible orders is equivalent to the search for all total orders consistent with a partial order, and so FIND_ALL_LISTS is similar to the Kahn algorithm for topological sorting [1962]. The latter determines a single total order consistent with a given partial order, the algorithm here determines all total orders consistent with a given partial order.

Algorithm 5 Algorithm that determines all permissible lists of a given set of processes. $|S|$ refers to the cardinality of the set S

```
function POSSIBLENEXT(S,L)
    $S \leftarrow$a set of processes
    $L \leftarrow$a list of processes in S
    $possibles \leftarrow \emptyset$ will contain all permissible lists obtained by adding a further element of $S$ to $L$
    $S \leftarrow S \setminus L$
    for each $x$ of S: do
        if norequirements(x,S) then
            $possibles \leftarrow possibles \cup \{x\}$
    replace each $y \in possibles$ with the list $L + [y]$
    $S \leftarrow S \cup L$
    Return $possibles$
function FINDALLLISTS(S)
    $paths \leftarrow \emptyset$ that will contain all permissible lists
    for each $x \in S$ do
        if $norequirements(x, S)$ then
            $paths \leftarrow paths \cup \{[x]\}$
    for $i$ in range($|S| - 1$) do
        for each $z \in paths$: do
            for each $p \in possiblenext(S, z)$ do
                $paths \leftarrow paths \cup \{p\}$
            $paths \leftarrow paths \setminus \{z\}$
    Return $paths$
```

a list L of processes (note, L is already coherent) and inserts some processes into others so as to reduce the overall cooking time as much as possible without violating 6.1.4. In Section 6.1 an organized discourse was said to be coherent iff

i there are no two processes P_1 preceding P_2 such that $requires(P_1, P_2)$

ii there are no two processes P_1 and P_2 such that P_1 and P_2 are combined and $requires(P_1, P_2)$

iii it does not contain a process P and a set of processes S such that every element of S has been inserted into P, and the sum of the total time of every element of S is greater than the free time of P

The resulting output is guaranteed to be coherent in this sense. The algorithm accomplishes (i) by inserting neighbouring processes one at a time and ending the insertions when an incompatible process is encountered. Thus only contiguous sublists

Algorithm 6 Algorithm that iterates through a given list of processes and performs combinations under certain criteria.

```
function CONCURRENTCOMPRESSION(L)
    $L \leftarrow$ list of processes
    for each process $p \in L$, beginning from the end of the list do
        $concurrents \leftarrow$ empty list that will contain inserted processes
        for each process $p'$ to the right of $p$, moving right do
            if $p'$[time] $\leq p$[f time] and $p$ and $p'$ are independent then
                add $p'$ to $concurrents$
                $p$[f time] $\leftarrow p$[f time] $- p'$[time] $+ p'$[f time]
        combine($concurrents, P, L$)
function COMBINE(concurrents,p,L)
    $concurrents \leftarrow$ a list of processes
    $p \leftarrow$ a process into which the processes in $concurrents$ will be inserted
    $L \leftarrow$ concatenation of $p$ and $concurrents$
    $futureinsertee \leftarrow \emptyset$
    $requiredf\_time \leftarrow p$[time]
    for each $p' \in L$ preceding $p$, moving left do
        if $requires(p, p')$ or $requires(p', p)$ then
            break
        else if $p'$[f time] $\geq requiredf\_time$ then
            $futureinsertee \leftarrow p'$
            break
        else
            increase $requiredf\_time$ by $p'$[time]
    if $futureinsertee \neq \emptyset$ then
        for each $q \in concurrents$ do
            if $q$ and $futureinsertee$ are not independent then
                remove $q$ from $concurrents$
    for each $item \in concurrents$ do
        add the input of $item$ to the input of $p$
        add the output of $item$ to the output of $p$
        $x$ stands for all known processes
        if $requires(item, x)$ then
            $requires(p, x) \leftarrow True$
        if $requires(x, item)$ then
            $requires(p, item) \leftarrow True$
        remove $item$ from $L$
    $concurrentinstruction \leftarrow$ concatenation of the 'direction' entry of all processes
in $concurrents$, with 'and' occurring between each
    $p[direction] \leftarrow$ 'while' + $cont\_form(p$[direction]$)$ + $concurrentinstruction$
```

of processes are combined and so the order is not changed, i.e. there are no two processes P_1 and P_2 such that P_1 preceded P_2 in the input and P_2 precedes P_1 in the output. The input list satisfies (i), and therefore so does the output. It accomplishes (ii) by making it an explicit condition for the combination of two processes that neither requires the other. It accomplishes (iii) by making it an explicit condition for the insertion of P_1 into P_2 that $P_1[\text{time}] \leq P_2[\text{f time}]$; and then inserting one at a time and, whenever P_1 is inserted into P_2, reducing the free time of P_2 by an amount equal to the non-passive time of P_1.[16]

Algorithm 6 is run on each permissible list of the processes in *actionlist* a coherent recipe plan. FASTFOOD then determines the total time of each of these recipe plans by summing the times of all processes that have not been inserted into another, and selects that with shortest total time. This temporally optimized recipe plan constitutes the final discourse representation. It is then realized by printing the total cooking time, the ingredients list, the cooking instructions and the times during the recipe at which the cook is passive.

7 Conclusion

This paper introduced FASTFOOD, an NLG system that uses a backwards-chaining automated theorem prover (ATP) to select the correct steps and ingredients required to make a given dish, and then modifies the recipe to optimize time efficiency in the generated recipes. We now conclude and summarize by considering the theoretical and practical value of our work.

7.1 Pratical Use

The representation system and NLG methods discussed in this paper are capable of generating a cooking discourse which is understandable, albeit slightly unnatural (see Figure 2 and appendix for full example). The temporal optimisation procedure (Section 6) is effective, clearly there are time savings by performing the instructions concurrently where directed, rather than one at a time. We claim that the temporal optimisation algorithm will always obtain an optimal solution, but a proof of this is outside the scope of the present paper. In comparison with those from a cookbook, there are some clear advantages and some disadvantages to FASTFOOD's generated recipes.

Advantages

- The language is uniform and concise. Something which has been identified as desirable in cooking recipes (Mori et al., 2012) and advantageous to users in other practical domains (Reiter et al., 2005).

[16]It is important to subtract the non-passive time, and not the total time, because the passive time in $P1$ should not reduce the passive time of $P2$. For example, if there is fives minutes of free time while boiling the pasta, and if toasting bread takes two minutes with one minute of passive time, then toasting the bread while the pasta is boiling still leaves four minutes of passive time, not three.

- The user can input the foods on hand and then receive a yes or no as to whether the dish was makeable (yes in this case). This is perhaps an easier task than the corresponding task of the user of a cookbook: for every item in the ingredient list, checking whether it is on hand.
- It gives a precise indication of the time taken, although this is based on assumed times for each step and so may vary depending on the cook.
- It saves the cook the effort of deciding which steps to perform simultaneously, as this is indicated explicitly in every case.
- It displays the times at which the cook will be passive. Two recipes may have the same total time, but differ significantly in how much free time they include and consequently in how demanding they are.

Disadvantages

- There is no notion of quantity, no specification of the amount of each ingredient required. It is thought that an extension to include quantity would not be overly difficult within the existing framework, though the function of FASTFOOD is currently impoverished in the absence of such an extension.
- The language is less natural than would be desired. A particular problem is the lack of inflection for progressive aspect in the combined instructions: 'while boil the lentils..' instead of 'while boiling the lentils..'.
- A more complete version would have considered alternative ways of making the dish, e.g. by using different vegetables.

The performance of FASTFOOD is partially of academic interest, as it constitutes support for the NLG techniques discussed in this paper. However it also presents some practical advantages over cookbook recipes. These advantages may become more significant if the disadvantages were mitigated through various extensions and improvements to the program, most notably the inclusion of the notion of quantity and the correct inflection of certain verbs in the output text.

7.2 Theoretical Interest

NLG in FASTFOOD was analysed as operating over four distinct phases: content production, content selection, content organisation which were described in detail in Sections 4, 5, and 6, and content realisation, which is given only minimal treatment. The use of a backwards-chaining ATP algorithm is of theoretical interest, as it draws a connection between the two ostensibly disparate areas of ATP and cooking recipes; and also of practical use, as it allows the NLG process to be modularized so that a set of representational elements is produced and stored, and then elements from this database are selected in response to various user inputs.

A number of theoretical points of interest were also encountered in the design choices of FASTFOOD. Choosing a suitable representation system (Section 3) meant omitting discourse referents; a decision which connected to an established philosophical debate on the nature of names and descriptions, and the method for handling different descriptions of the same dish touched on various semantic interpretations of the synonymy relation (Section 4.3). The content selection algorithm, while initially inspired by ATP, also exhibited a similarity to automated parsing. This led to a three-way comparison between ATP, automated parsing and FASTFOOD's content selection algorithm. The content organization algorithm, i.e. the temporal optimization module, validated the representation system by showing that it was capable of supporting reasoning about recipe content.

This work can be compared to existing literature on Natural Language Processing of cooking recipes (Section 2, especially EPICURE (Dale & Haddock, 1991). The two were shown to have different strengths and weaknesses, and a way to combine the benefits of both was briefly explored. A comparison was also made with a common NLU technique of converting natural language recipes to directed acyclic graphs, where it was shown that, due in part to its representation system (Section 3), FASTFOOD avoids a common problem encountered by such techniques. In comparison to neural recipe generation, FASTFOOD has the advantage of reliably producing coherent recipes and being able to reason recipe content. Future work includes introducing a notion of quantity, allowing the user to choose between multiple recipes for the same dish, and linking with the field of image processing.

References

Abend, O., Cohen, S. B., & Steedman, M. (2015). Lexical event ordering with an edge-factored model. In *Proceedings of the 2015 conference of the north american chapter of the association for computational linguistics: Human language technologies* (pp. 1161–1171).

Avigad, J., & Harrison, J. (2014). Formally verified mathematics. *Communications of the ACM*, *57*(4), 66–75.

Bansal, K., Loos, S., Rabe, M., Szegedy, C., & Wilcox, S. (2019). HOList: An environment for machine learning of higher order logic theorem proving. In *International conference on machine learning* (pp. 454–463).

Bateman, J. A. (2002). *Natural language generation: an introduction and open-ended review of the state of the art.* `http://www.fb10.uni-bremen.de/anglistik/langpro/webspace/jb/info-pages/nlg/ATG01/ATG01.html`. (Accessed: 2023-10-17)

Bayomi, M., Levacher, K., Ghorab, M. R., & Lawless, S. (2015). Ontoseg: a novel approach to text segmentation using ontological similarity. In *2015 ieee international conference on data mining workshop (icdmw)* (pp. 1274–1283).

Bledsoe, W. W. (1984). *Automated theorem proving: After 25 years: After 25 years* (Vol. 89). American Mathematical Soc.

Bridge, J. P., Holden, S. B., & Paulson, L. C. (2014). Machine learning for first-order theorem proving. *Journal of automated reasoning*, *53*(2), 141–172.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., ... others (2020). Language models are few-shot learners. *Advances in neural information processing systems*, *33*, 1877–1901.

Cazzulino, D., Aprea, V. G., Greenwood, J., & Hart, C. (2004). *Beginning visual web programming in c#: From novice to professional.* Apress.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., ... others (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on information theory*, *2*(3), 113–124.

Copi, I. M., & Beard, R. W. (2013). *Essays on wittgenstein's tractatus.* Routledge.

Cordier, A., Dufour-Lussier, V., Lieber, J., Nauer, E., Badra, F., Cojan, J., ... others (2014). Taaable: a case-based system for personalized cooking. In *Successful case-based reasoning applications-2* (pp. 121–162). Springer.

Culleton, T., & Higgs, D. (2016). *Simply vegetarian.* Merlin.

Dale, R. (1989). Cooking up referring expressions. In *27th annual meeting of the association for computational linguistics* (pp. 68–75).

Dale, R. (1990). Generating recipes: An overview of epicure. *Current research in natural language generation*, 229–255.

Dale, R. (1992). *Generating referring expressions: Constructing descriptions in a domain of objects and processes.* The MIT Press.

Dale, R., & Haddock, N. (1991). Content determination in the generation of referring expressions. *Computational Intelligence*, *7*(4), 252–265.

Donatelli, L., Schmidt, T., Biswas, D., Köhn, A., Zhai, F., & Koller, A. (2021). Aligning actions across recipe graphs. In *Proceedings of the 2021 conference on empirical methods in natural language processing* (pp. 6930–6942).

Dufour-Lussier, V., Le Ber, F., Lieber, J., Meilender, T., & Nauer, E. (2012). Semi-automatic annotation process for procedural texts: An application on cooking recipes. In *Computers workshop (cwc)* (p. 35).

Edmonds, P., & Hirst, G. (2002). Near-synonymy and lexical choice. *Computational linguistics*, *28*(2), 105–144.

Elliott, D., & Keller, F. (2013). Image description using visual dependency representations. In *Proceedings of the 2013 conference on empirical methods in natural language processing* (pp. 1292–1302).

Evans, J. (1982). *The varieties of reference.* Oxford University Press.

Flower, J., Masthoff, J., & Stapleton, G. (2004). Generating readable proofs: A heuristic approach to theorem proving with spider diagrams. In *International conference on theory and application of diagrams* (pp. 166–181).

Flynn, S., & Dave, F. (2016). *The world of the happy pear.* Penguin Random House.

Fontaine, P., Merz, S., & Woltzenlogel Paleo, B. (2011). Compression of propositional resolution proofs via partial regularization. In *International conference on automated deduction* (pp. 237–251).

Fragkou, P. (2017). Applying named entity recognition and co-reference resolution for segmenting english texts. *Progress in Artificial Intelligence*, *6*(4), 325–346.

Frege, G., et al. (1892). Über Sinn und Bedeutung. *Zeitschrift für Philosophie und philosophische Kritik*, *100*(1), 25–50.

French, P. A., Uehling, T. E., & Wettstein, H. K. (1979). *Contemporary perspectives in the philosophy of language* (Vol. 1). U of Minnesota Press.

Gaillard, E., Nauer, E., Lefevre, M., & Cordier, A. (2012). Extracting generic cooking adaptation knowledge for the taaable case-based reasoning system. In *Cooking with computers workshop@ ecai 2012.*

Galmiche, D. (2000). Connection methods in linear logic and proof nets construction. *Theoretical Computer Science*, *232*(1-2), 231–272.

Galmiche, D., & Marion, J.-Y. (1995). Semantic proof search methods for ALL-a rst approach. In *4th workshop on theorem proving with analytic tableaux and related methods, st goar am rhein, germany.*

Galmiche, D., & Perrier, G. (1994). Foundations of proof search strategies design in linear logic. In *International symposium on logical foundations of computer science* (pp. 101–113).

Ganitkevitch, J., Van Durme, B., & Callison-Burch, C. (2013). Ppdb: The paraphrase database. In *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies* (pp. 758–764).

Gavin, P. (2016). *Italian vegetarian cookery.* Macdonald Optima.

Gazdar, G., Klein, E., Pullum, G., & Sag, I. (1985). *Generalized phrase structure grammar.* Blackwell, London, England.

Geffner, H. (2013). Computational models of planning. *Wiley Interdisciplinary Reviews: Cognitive Science*, *4*(4), 341–356.

Georgallides, A. (2015). *The unclarity of the notion 'object' in the tractatus* (Unpublished doctoral dissertation). University of Sussex.

Gödel, K. (1929). *Über die vollständigkeit des logikkalküls* (Unpublished doctoral dissertation). University of Vienna.

Gunamgari, S. R., Dandapat, S., & Choudhury, M. (2014). Hierarchical recursive tagset for annotating cooking recipes. In *Proceedings of the 11th international conference on natural language processing* (pp. 353–361).

Hamada, R., Ide, I., Sakai, S., & Tanaka, H. (2000). Structural analysis of cooking preparation steps in japanese. In *Proceedings of the fifth international workshop on on information retrieval with asian languages* (pp. 157–164).

Hamada, R., Okabe, J., Ide, I., Satoh, S., Sakai, S., & Tanaka, H. (2005). Cooking navi: assistant for daily cooking in kitchen. In *Proceedings of the 13th annual acm international conference on multimedia* (pp. 371–374).

He, P., Gao, J., & Chen, W. (2022). Debertav3: Improving deberta using electrastyle pre-training with gradient-disentangled embedding sharing. In *The eleventh international conference on learning representations.*

H. Lee, H., Shu, K., Achananuparp, P., Prasetyo, P. K., Liu, Y., Lim, E.-P., & Varshney, L. R. (2020). Recipegpt: Generative pre-training based cooking recipe generation and evaluation system. In *Companion proceedings of the web conference 2020* (pp. 181–184).

Hongxia, X., & Li, Z. (2008). An efficient extension of earley's algorithm for parsing multidimensional structures. In *2008 international conference on computer science and software engineering* (Vol. 2, pp. 780–783).

Indurkhya, N., & Damerau, F. J. (2010). *Handbook of natural language processing.* Chapman and Hall/CRC.

Kahn, A. B. (1962). Topological sorting of large networks. *Communications of the ACM*, *5*(11), 558–562.

Kamp, H. (2021). Sharing real and fictional reference. *The Language of Fiction*, 37-87.

Kamp, H., Genabith, J. v., & Reyle, U. (2011). Discourse representation theory. In *Handbook of philosophical logic* (pp. 125–394). Springer.

Kamp, H., & Reyle, U. (1993). *From discourse to logic kluwer academic publishers.* Dordrecht.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., ... Amodei, D. (2020). Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Kinouchi, O., Diez-Garcia, R. W., Holanda, A. J., Zambianchi, P., & Roque, A. C. (2008). The non-equilibrium nature of culinary evolution. *New Journal of Physics*, *10*(7), 073020.

Koeva, S. (2015). Paraphrasing of synonyms for a fine-grained data representation. In *Semantics (posters & demos)* (pp. 79–83).

Kozen, D. C. (1992). *The design and analysis of algorithms.* Springer Science & Business Media.

Kuhn, T. (2014). A survey and classification of controlled natural languages. *Computational linguistics*, *40*(1), 121–170.

Li, J., Tang, T., Zhao, W. X., & Wen, J.-R. (2021). Pretrained language models for text generation: A survey. In *Proceedings of the thirtieth international joint conference on artificial intelligence* (p. 4492-4499).

Lloyd, J. W. (2012). *Foundations of logic programming.* Springer Science & Business Media.

Loos, S., Irving, G., Szegedy, C., & Kaliszyk, C. (2017). Deep network guided proof search. In *Proceedings of the 21st international conference on logic for programming, artificial intelligence and reasoning (lpar-21), epic series in computing* (p. 85-105).

Loveland, D. W. (2016). *Automated theorem proving: A logical basis.* Elsevier.

Matsumoto, Y., Tanaka, H., Hirakawa, H., Miyoshi, H., & Yasukawa, H. (1983). Bup: a bottom-up parser embedded in prolog. *New Generation Computing*, *1*(2), 145–158.

McCudden, C. (2017). The future of artificial intelligence and interpretative specialization in clinical biochemistry. *Clinical biochemistry*, *50*(6), 253–254.

McKeown, K. (1992). *Text generation.* Cambridge University Press.

McKeown, K. R. (1985). Discourse strategies for generating natural-language text. *Artificial intelligence*, *27*(1), 1–41.

Meduna, A. (2014). *Formal languages and computation: models and their applications.* CRC Press.

Miller, A. (2017). *Representation and reality in wittgenstein's tractatus.* Oxford University Press.

Mogensen, E. (2004). Controlled language. *Perspectives*, *12*(4), 243–255.

Mooney, R. J. (1996). Inductive logic programming for natural language processing. In *International conference on inductive logic programming* (pp. 1–22).

Mori, S., Maeta, H., Yamakata, Y., & Sasada, T. (2014). Flow graph corpus from recipe texts. In *Proceedings of the ninth international conference on language resources and evaluation (lrec'14)* (pp. 2370–2377).

Mori, S., Sasada, T., Yamakata, Y., & Yoshino, K. (2012). A machine learning approach to recipe text processing. In *Proceedings of the 1st cooking with computer workshop* (pp. 29–34).

Nederhof, M.-J., Satta, G., & Shieber, S. (2003, April). Partially ordered multiset context-free grammars and free-word-order parsing. In *Proceedings of the eighth international conference on parsing technologies* (pp. 171–182). Nancy, France. Retrieved from `https://aclanthology.org/W03-3020`

Panaget, F. (1994). Using a textual representational level component in the context of discourse or dialogue generation. In *Proceedings of the seventh international workshop on natural language generation* (pp. 127–136).

Parvez, M. R., Chakraborty, S., Ray, B., & Chang, K.-W. (2018, jul). Building language models for text with named entities. In *Proceedings of the 56th annual meeting of the association for computational linguistics (volume 1: Long papers)* (pp. 2373–2383). Melbourne, Australia: Association for Computational Linguistics. Retrieved from `https://aclanthology.org/P18-1221` doi: 10.18653/v1/P18-1221

Patra, B., Singhal, S., Huang, S., Chi, Z., Dong, L., Wei, F., ... Song, X. (2022). Beyond english-centric bitexts for better multilingual language representation learning. *arXiv preprint arXiv:2210.14867*.

Pavlick, E., & Nenkova, A. (2015). Inducing lexical style properties for paraphrase and genre differentiation. In *Proceedings of the 2015 conference of the north american chapter of the association for computational linguistics: Human language technologies* (pp. 218–224).

Pereira, F. C., & Warren, D. H. (1983). Parsing as deduction. In *21st annual meeting of the association for computational linguistics* (pp. 137–144).

Preotiuc-Pietro, D., Xu, W., & Ungar, L. (2016). Discovering user attribute stylistic differences via paraphrasing. In *Proceedings of the aaai conference on artificial intelligence* (Vol. 30, pp. 3030–3037).

Pulman, S. (1996). Controlled language for knowledge representation. In *Proceedings of* *the first international workshop on controlled language applications* (p. 233-242).

Quine, W. V. (1948). On what there is. *The review of metaphysics*, *2*(1), 21–38.

Ranta, A. (2011). *Grammatical framework: Programming with multilingual grammars* (Vol. 173). CSLI Publications, Center for the Study of Language and Information Stanford.

Reichel, S., Muller, T., Stamm, O., Groh, F., Wiedersheim, B., & Weber, M. (2011). Mampf: An intelligent cooking agent for zoneless stoves. In *2011 seventh international conference on intelligent environments* (pp. 171–178).

Reiter, E., Sripada, S., Hunter, J., Yu, J., & Davy, I. (2005). Choosing words in computer-generated weather forecasts. *Artificial Intelligence*, *167*(1-2), 137–169.

Robin, J. (1993). A revision-based generation architecture for reporting facts in their historical context. *New concepts in natural language generation: Planning, realization, and systems*, 238–268.

Russell, B. (1905). On denoting. *Mind*, *14*(56), 479–493.

Russell, B. (1910). Knowledge by acquaintance and knowledge by description. In *Proceedings of the aristotelian society* (Vol. 11, pp. 108–128).

Salvador, A., Drozdzal, M., Giró-i-Nieto, X., & Romero, A. (2019). Inverse cooking: Recipe generation from food images. In *Proceedings of the ieee/cvf conference on computer vision and pattern recognition* (pp. 10453–10462).

Sarawagi, S. (2008). *Information extraction.* Now Publishers Inc.

Schulz, S., & Sutcliffe, G. (2015). Proof generation for saturating first-order theorem provers. *All about Proofs, Proofs for All, Mathematical Logic and Foundations*, *55*.

Searle, J. R. (1958). Proper names. *Mind*, *67*(266), 166–173.

Seiffert, R. (1987). *Chart-parsing of unification-based grammars with ID/LP-rules* (Tech. Rep. No. LILOG-REPORT 22). Stuttgart, West Germany: IBM Deutschland GmbH.

Shieber, S. M., Schabes, Y., & Pereira, F. C. (1995). Principles and implementation of deductive parsing. *The Journal of logic programming*, *24*(1-2), 3–36.

Shin, T., Razeghi, Y., Logan IV, R. L., Wallace, E., & Singh, S. (2020). Autoprompt: Eliciting knowledge from language models with automatically generated prompts. In *Proceedings of the 2020 conference on empirical methods in natural language processing (emnlp)* (pp. 4222–4235).

Stabler Jr, E. P. (1983). Deterministic and bottom-up parsing in prolog. In *Aaai* (pp. 383–386).

Stapleton, G., Masthoff, J., Flower, J., Fish, A., & Southern, J. (2007). Automated theorem proving in euler diagram systems. *Journal of Automated Reasoning*, *39*(4), 431–470.

Sutcliffe, G., & Suttner, C. (2001). Evaluating general purpose automated theorem proving systems. *Artificial intelligence*, *131*(1-2), 39–54.

Urban, J., Hoder, K., & Voronkov, A. (2010). Evaluation of automated theorem proving on the mizar mathematical library. In *International congress on mathematical software* (pp. 155–166).

Wang, H., Lin, G., Hoi, S. C., & Miao, C. (2020). Structure-aware generation network for recipe generation from images. In *European conference on computer vision* (pp. 359–374).

Wang, W., Bi, B., Yan, M., Wu, C., Xia, J., Bao, Z., ... Si, L. (2020). Structbert: Incorporating language structures into pre-training for deep language understanding. In *International conference on learning representations.*

Webber, B., Egg, M., & Kordoni, V. (2012). Discourse structure and language technology. *Natural Language Engineering*, *18*(4), 437–490.

Wittgenstein, L. (1922). Tractatus Logico-Philosophicus, translated by C.K. Ogden with an introduction by Bertrand Russell. *London: Kegan Paul*.

Wittgenstein, L. (1961). *Tractatus logico-philosophicus (trans. pears and mcguinness)*. Springer.

Xu, F. F., Alon, U., Neubig, G., & Hellendoorn, V. J. (2022). A systematic evaluation of large language models of code. In *Proceedings of the 6th acm sigplan international symposium on machine programming* (pp. 1–10).

Yamakata, Y., Maeta, H., Kadowaki, T., Sasada, T., Imahori, S., & Mori, S. (2017). Cooking recipe search by pairs of ingredient and action — word sequence vs. flow-graph representation —. *Transactions of the Japanese Society for Artificial Intelligence*, *32*(1), 1–9.

Zhang, S. (1994). Story parsing grammar. *Journal of Computer Science and Technology*, *9*(3), 215–228.

Zhao, P., Fang, T., Xiao, J., Zhang, H., Zhao, Q., & Quan, L. (2010). Rectilinear parsing of architecture in urban environment. In *2010 ieee computer society conference on computer vision and pattern recognition* (pp. 342–349).

Correspondence

Louis Mahon
School of Informatics
University of Edinburgh
oneillml@tcd.ie

Carl Vogel
School of Computer Science and Statistics
Trinity College, Dublin

Worked Example: Generating Recipe for Vegetable Dahl What follows is a worked example of how FASTFOOD would generate the recipe for the dish described by the string 'vegetable dahl'. This is an application of the algorithms discussed in Sections 4, 5 and 6, and how they combine to produce a time-efficient recipe. Before 'vegetable dahl' can be used to generate a recipe, one must also input a specification of what it is composed of in terms of descriptive strings already understood by FASTFOOD. The specification amounts to defining what will be the final process in the recipe, the process that produces the string 'vegetable dahl' itself, and in this case is given by

$$\begin{aligned} mix\_dahl \;=\; &\{(\text{input}, \{\text{`fried vegetables'}, \text{`coconut milk'}, \text{`boiled lentils'}\}),\\ &\text{output}, \{\text{`vegetable dahl'}\}),\\ &\text{time}, 120),\\ &\text{ftime}, 0),\\ &\text{direction}, \text{`mix together the vegetables, coconut milk and boiled lentils'})\}\,. \end{aligned}$$

The user would also specify what foods are at hand, by defining the descriptive strings in *supplies* (see Section 5.1. For the purposes of this example it will be assumed that *supplies* contains the descriptive strings 'coconut milk', 'raw pepper', 'raw broccoli', 'lentils' and 'water'.

$$supplies \;=\; \{\text{`coconut milk'}, \text{`raw pepper'}, \text{`raw broccoli'}, \text{`lentils'}, \text{`water'}, \dots\}$$

With the addition of *mix_dahl* to the set *skills*[17], 'vegetable dahl' can be fed into SELECT_CONTENT (Algorithm 3) which will perform the following steps.

- check if 'vegetable dahl' is in *supplies* and determine that it is not
- search *skills* for a process which has 'vegetable dahl' in the output, and find *mix_dahl*, add *mix_dahl* to *action_list*
- check if the inputs to *mix_dahl*, 'fried vegetables', 'coconut milk' and 'boiled lentils' are in *supplies* and determine that 'coconut milk' is, add 'coconut milk' to *ingred_list*
- search for processes with 'fried vegetables' and 'boiled lentils' as outputs and find *fry_vegetables* and *strain_lentils* respectively, add *fry_vegetables* and *strain_lentils* to *action_list*
- check if the inputs to *fry_vegetables* and *strain_lentils*, 'chopped vegetables' and 'boiled lentils in boiling water' are in *supplies* and determine that they are not

[17] *skills* is the set of all known processes, see Section 4.

- search for processes with ‘chopped vegetables’ and ‘boiled lentils in boiling water’ as outputs and find the ghost process describing the synonym between ‘chopped broccoli’ and ‘chopped pepper’ and ‘chopped vegetables’, and *boil_lentils*
- add ghost and *boil_lentils* to *action_list*
- check if the inputs to *ghost*, ‘chopped broccoli’, ‘chopped pepper’, ‘lentils’ and ‘boiling water’ are in *supplies*, determine that ‘lentils’ is
- add ‘lentils’ to *ingred_list*
- search for processes with outputs ‘chopped broccoli’, ‘chopped pepper’, and ‘boiling water’, and find *chop_broccoli*, *chop_pepper* and *bring_to_boil* respectively
- add *chop_broccoli*, *chop_pepper* and *bring_to_boil* to *action_list*
- check if the inputs to *chop_broccoli*, *chop_pepper* and *bring_to_boil*, ‘raw broccoli’, ‘raw pepper’ and ‘water’, are in *supplies* and determine that they are
- add ‘raw broccoli’, ‘raw pepper’ and ‘water’ to *ingred_list*
- return *ingred_list* and *action_list*

Together *ingred_list* and *action_list* constitute the selected discourse content. This content is represented as follows:

$$\begin{aligned} ingred\_list &= \{\text{‘coconut milk’}, \text{‘raw pepper’}, \text{‘raw broccoli’}, \text{‘lentils’}, \text{‘water’}, \ldots\} \\ action\_list &= \{\text{‘chop\_broccoli’}, \text{‘chop\_pepper’}, \text{‘bring\_to\_boil’}, \\ &\quad \text{‘ghost’}, \text{‘boil\_lentils’}, \text{‘fry\_vegetables’}, \\ &\quad \text{‘strain\_lentils’}, \text{‘mix\_dahl’} \ldots\} \end{aligned}$$

This discourse content allows the appropriate requires relations to be determined: $requires(A, B)$ whenever A[input] contains an element from B[output].

Descriptive strings are brown circles and processes are blue boxes (and arrows). Processes take a set of descriptive strings as input and produce a set of descriptive strings as output. The figure is formed by matching the inputs of some processes with the outputs of others. The five initial descriptive strings constitute *ingred_list*, and the set of all processes constitutes *action_list*.

Next, the ghost is removed, giving Figure 4, and $requires(A, B)$ is added whenever $requires(A, ghost)$ and $requires(ghost, B)$ (see Section 6). This means adding

$$requires(fry\_vegetables, chop\_broccoli)$$
$$requires(fry\_vegetables, chop\_pepper)\,.$$

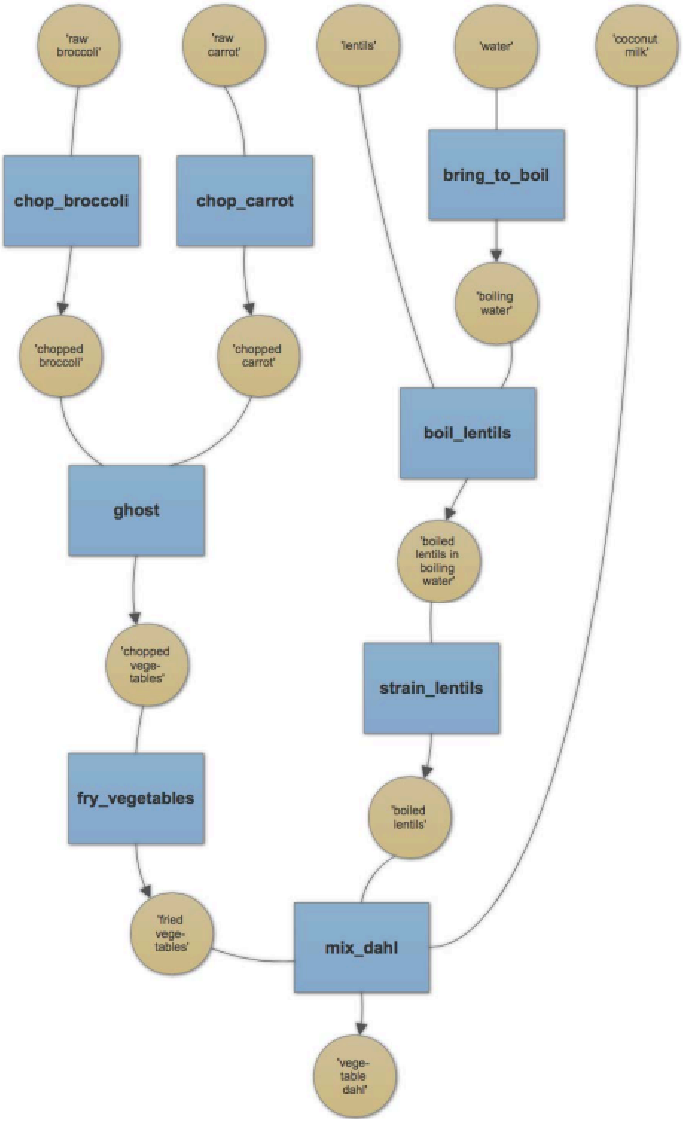


Figure 3: Full DAG representing content of the generated recipe.

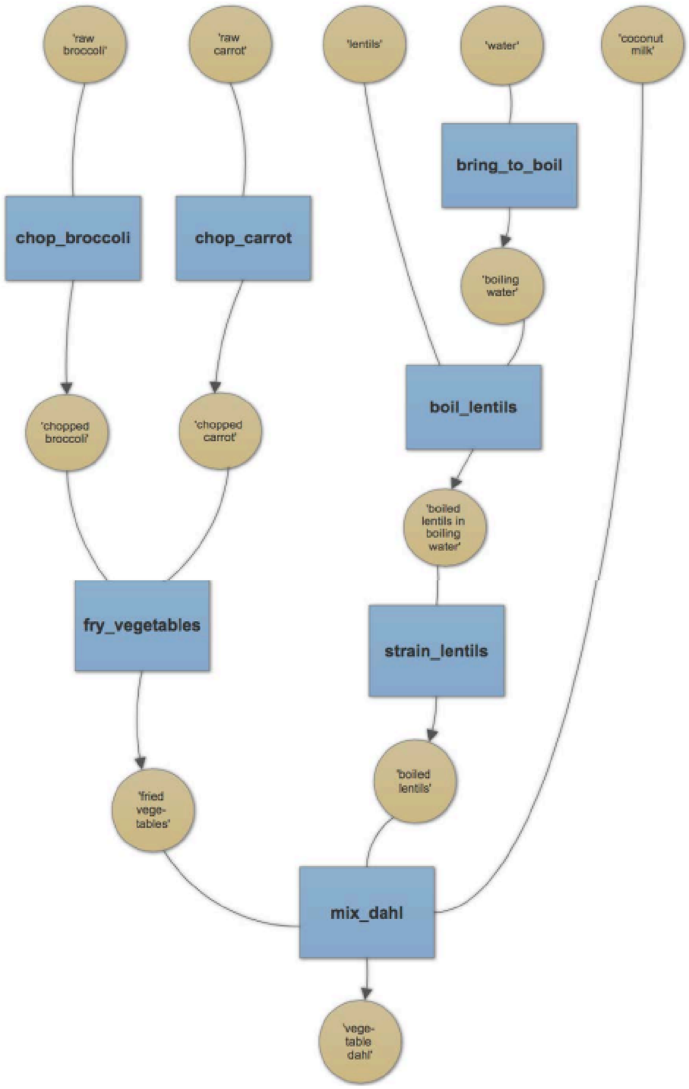


Figure 4: DAG representing recipe content after the removal of ghost processes. Its inputs are joined to $fry\_vegetables$. The requires relations are updated as per Section 6.

The final requires relations are as follows:

$$\begin{aligned}
&requires(mix\_dahl, fry\_vegetables)\\
&requires(mix\_dahl, boil\_lentils)\\
&requires(fry\_vegetables, ghost)\\
&requires(ghost, chop\_broccoli)\\
&requires(mix\_dahl, chop\_pepper)\\
&requires(strain\_lentils, boil\_lentils)\\
&requires(strain\_lentils, bring\_to\_boil)\\
&requires(boil\_lentils, bring\_to\_boil)
\end{aligned}$$

Then *action_list* is fed into FIND_ALL_LISTS, which determines all orderings of *action_list* such that no process precedes another which it requires. In this case 40 such orderings exist, so it is not feasible to describe the full action of the algorithm. Instead the first two steps are described and 3 of the output orders considered. The first two stages to FIND_ALL_LISTS are shown below.

- search for the processes in *action_list* that do not require any others, and determine these to be *chop_broccoli*, *chop_pepper* and *bring_to_boil*, add these to the set *possible_lists*

- iterate through the set *possible_lists*, for each element P obtain all elements that do not require any others except (possibly) P

- for each of these elements x, add the list $[P, x]$ to possible lists, remove P from *possible_lists*

- if $P = chop\_broccoli$, then $x = chop\_pepper$ or $bring\_to\_boil$; giving the following two element lists: [*chop_broccoli*, *chop_pepper*] [*chop_broccoli*, *bring_to_boil*]

- if $P = chop\_pepper$, then $x = chop\_broccoli$ or $bring\_to\_boil$; giving the following two element lists: [*chop_pepper*, *chop_broccoli*] [*chop_pepper*, *bring_to_boil*]

- if $P = bring\_to\_boil$, then $x = chop\_broccoli$ or $chop\_pepper$ or $boil\_lentils$; giving the following two element lists: [*bring_to_boil*, *chop_broccoli*] [*bring_to_boil*, *chop_pepper*] [*bring_to_boil*, *boil_lentils*]

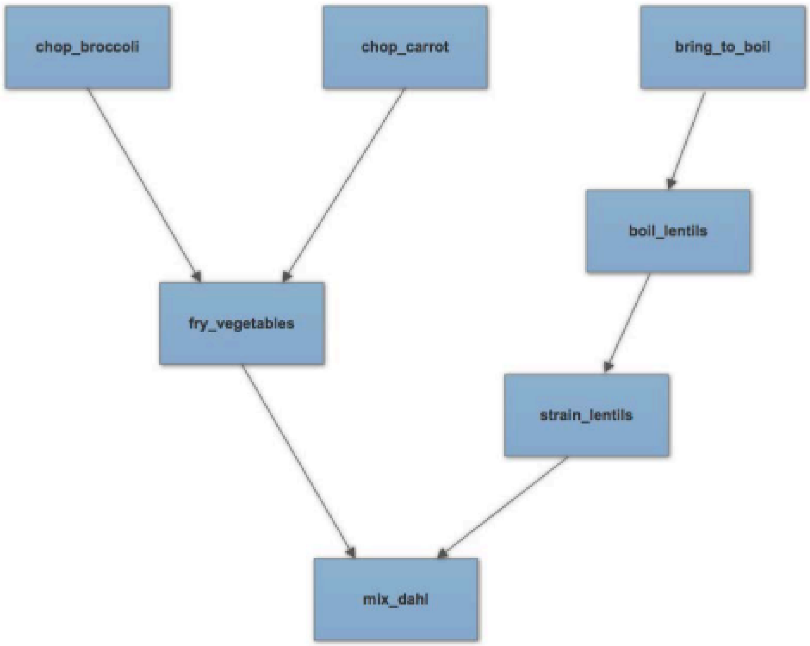


Figure 5: DAG showing recipe content with processes only, no descriptive strings. An arrow from P_1 to P_2 indicates that requires(P2,P1). This form of representation can be termed a workflow and is discussed briefly in Section 2.2.

At this point, *possible_lists* is composed of the following seven two-element lists:

$$\begin{aligned} possible\_lists = & [chop\_broccoli, chop\_pepper], \\ & [chop\_broccoli, bring\_to\_boil], \\ & [chop\_pepper, chop\_broccoli], \\ & [chop\_pepper, bring\_to\_boil], \\ & [bring\_to\_boil, chop\_broccoli], \\ & [bring\_to\_boil, chop\_pepper], \\ & [bring\_to\_boil, boil\_lentils]] \,. \end{aligned}$$

The next step would be to again iterate through *possible_lists* and, for each two-element list $[P, Q]$, determine all the processes that do not require any others except for (possibly) P and Q. For example, if $[P, Q]$ = [*chop_broccoli*, *chop_pepper*], then x = *bring_to_boil* or *fry_vegetables*; giving the following three element lists: [*chop_broccoli*,*chop_pepper*, *fry_vegetables*] [*chop_broccoli*, *chop_pepper*, *bring_to_boil*].

These three element lists would be added to *possible_lists*, and [*chop_pepper*, *chop_broccoli*] would be removed. Of the 40 valid lists that would eventually be obtained by FIND_-ALL_LISTS and hence fed into the CONCURRENT_COMPRESSION algorithm, the following three are considered in this worked example. The notation $P(x/y)$ refers to a process P with total time x and free time y.

$$
\begin{aligned}
A &= [chop\_pepper(120/0),\\
&\quad chop\_broccoli(120/0),\\
&\quad bring\_to\_boil(300/270),\\
&\quad fry\_vegetables(420/300),\\
&\quad boil\_lentils(2700/2670),\\
&\quad strain\_lentils(60/0),\\
&\quad mix\_dahl(120/0)]\\
B &= [bring\_to\_boil(300/270),\\
&\quad boil\_lentils(2700/2670),\\
&\quad chop\_broccoli(120/0),\\
&\quad chop\_pepper(120/0),\\
&\quad strain\_lentils(60/0),\\
&\quad fry\_vegetables(420/300),\\
&\quad mix\_dahl(120/0)]\\
C &= [bring\_to\_boil(300/270),\\
&\quad chop\_broccoli(120/0),\\
&\quad chop\_pepper(120/0),\\
&\quad boil\_lentils(2700/2670),\\
&\quad fry\_vegetables(420/300),\\
&\quad strain\_lentils(60/0),\\
&\quad mix\_dahl(120/0)]
\end{aligned}
$$

CONCURRENT_COMPRESSION iterates through a list of processes and combines neighbouring processes under certain conditions (see Algorithm 6 for a full description). It's action on A, B and C is described below. The notation X' indicates the output of CONCURRENT_COMPRESSION after input X.

Order A

1. *mix_dahl* has no elements following it and so is passed over

2. *strain_lentils* has f time = 0 and so is passed over

3. *boil_lentils* has f time > 0, but is followed by *strain_lentils*, and requires(*strain_lentils*, *boil_lentils*), so *boil_lentils* is passed over

4. *fry_vegetables* has f time > 0, but it is followed by *boil_lentils*, and *boil_lentils*[time] > *fry_vegetables*[f time], so *fry_vegetables* is passed over

5. *bring_to_boil* has f time > 0, but it has *fry_vegetables* to its right, and *fry_vegetables*[time] > *bring_to_boil*[f time], so *bring_to_boil* is passed over

6. *chop_broccoli* has f time = 0 and so is passed over

7. *chop_pepper* has f time = 0 and so is passed over

Thus, no processes are combined and the output from concurrent compression is the same as the input. A' = A

Order B

1. *mix_dahl* has no elements following it and so is passed over

2. *fry_vegetables* has f time > 0, but is followed by *mix_dahl* and *requires*(*mix_dahl*, *fry_vegetables*), so *fry_vegetables* is passed over

3. *strain_lentils*, *chop_pepper* and *chop_broccoli* have f time = 0 and so are passed over

4. *boil_lentils* has f time > 0, it is followed by *chop_broccoli*, *chop_broccoli*[time] < *boil_lentils*[f time] and neither *requires*(*chop_broccoli*, *boil_lentils*) nor *requires*(*boil_lentils*, *chop_broccoli*) hold; also, *boil_lentils* is not going to be inserted into anything else in the future as the only process preceding it is *bring_to_boil* and *requires*(*boil_lentils*, *bring_to_boil*); therefore *chop_broccoli* is inserted into *boil_lentils* and *boil_lentils*[f time] is then reduced by *chop_broccoli*[time] to 2670-120 = 2550

5. the next process after *boil_lentils* is *chop_pepper*, which for the same reasons is also inserted, *boil_lentils*[f time] is then reduced by *chop_pepper*[time] to 2550-120 = 2430

6. the next process following *boil_lentils* is *strain_lentils* and *requires*(*strain_lentils*, *boil_lentils*), so no more insertions into *boil_lentils* can take place

7. *bring_to_boil* is followed by *boil_lentils*, and *requires*(*boil_lentils*, *bring_to_boil*) so *bring_to_boil* is passed over

Thus, the only combination is the insertion of *chop_broccoli* and *chop_pepper* into *boil_lentils*, and the output is as follows:

$$\begin{aligned} B \;=\; [&bring\_to\_boil(300/270),\\ &while\ boil\_lentils;\,;chop\_broccoli\ and\ chop\_pepper(2700/2430),\\ &strain\_lentils(60/0),\\ &fry\_vegetables(420/300),\\ &mix\_dahl(120/0)] \end{aligned}$$

Order C

1. *mix_dahl* has no elements following it and so is passed over

2. *strain_lentils* has f time = 0 and so is passed over

3. *fry_vegetables* has f time > 0, it is followed by *strain_lentils*, *strain_lentils*[time] < *fry_vegetables*[f time] and *requires*(*strain_lentils*, *fry_vegetables*) does not hold; however, *fry_vegetables* is preceded by *boil_lentils* and the conditions for combination hold between *fry_vegetables* and *boil_lentils*, moreover the insertion of *strain_lentils* into *fry_vegetables* would prevent the insertion of *fry_vegetables* into *boil_lentils* as *requires*(*strain_lentils*, *boil_lentils*) and so it would be infelicitous to have while boil lentils, (while fry vegetables, strain lentils); therefore the insertion of *strain_lentils* into *fry_vegetables* is foregone in favour of the future insertion of *fry_vegetables* into *boil_lentils*

4. *boil_lentils* is followed by *fry_vegetables*, *boil_lentils*[*ftime*] > *fry_vegetables*[time] and *requires*(*boil_lentils*, *fry_vegetables*) does not hold, moreover *boil_lentils* is not going to be inserted into anything else in the future as *chop_broccoli*[f time] = *chop_pepper*[f time] = 0 and requires (*boil_lentils*, *bring_to_boil*); therefore *fry_vegetables* is inserted into *boil_lentils*, *boil_lentils*[f time] is then reduced by the non-passive time in *fry_vegetables* to give 2670-420+300 = 2550 the next element following *boil_lentils* is *strain_lentils* and requires(*strain_lentils*, *boil_lentils*), therefore no more insertions into *boil_lentils* can take place

5. *chop_broccoli* and *chop_pepper* have f time = 0 and so are passed over

6. *bring_to_boil* is followed by *chop_broccoli*, and *bring_to_boil*[f time] > *chop_broccoli*[time] and requires(*bring_to_boil*, *chop_broccoli*) does not hold, moreover *bring_to_boil* is not going to be combined with anything else in the future as it has no elements preceding it; therefore *chop_broccoli* is inserted into

$bring\_to\_boil$; $bring\_to\_boil$[f time] is then reduced by $chop\_broccoli$[time] to 270 – 120 = 150 the next process following $bring\_to\_boil$ is $chop\_pepper$, and for the same reasons it too is inserted into $bring\_to\_boil$, $bring\_to\_boil$[f time] is then reduced by $chop\_pepper$[time] to 150-120 = 30

7. the next process following $bring\_to\_boil$ is $boil\_lentils$ and $requires(boil\_lentils, bring\_to\_boil)$, therefore nothing else is inserted into $bring\_to\_boil$

Thus $fry\_vegetables$ was inserted into $boil\_lentils$, and $chop\_broccoli$ and $chop\_pepper$ were inserted into $bring\_to\_boil$, and the output is as follows:

$$\begin{aligned} C \;=\; & [while\, bring\_to\_boil; chop\_broccoli\; and\; chop\_pepper(300/30), \\ & while\, boil\_lentils; fry\_vegetables(2700/2550), \\ & strain\_lentils(60/0), \\ & mix\_dahl(120/0)] \end{aligned}$$

The total times for the compressed recipe plans A',B' and C' are

$$\begin{aligned} T_{A'} &= 120 + 120 + 300 + 420 + 2700 + 60 + 120 = \mathbf{3840} \\ T_{B'} &= 300 + 2700 + 60 + 420 + 120 = \mathbf{3600} \\ T_{C'} &= 300 + 2700 + 60 + 120 = \mathbf{3180}\,, \end{aligned}$$

where T_x denotes the total time for recipe plan x. Thus, C is the most efficient as it contains the largest amount of combinations. In fact of the 40 permissible orderings of these processes, no other, when fed through CONCURRENT_COMPRESSION, has total time less than that of C'. Thus, C' is chosen as the recipe plan. The final requirement is the temporal information: the total cooking time and the passive times. This can be obtained by iterating through each element in C' and noting those that still have strictly positive free time (remembering that free time will have been reduced in instances of insertion). The variable *clock* is used to keep track of when these free times will occur. The notation 'while P, $P_1, \ldots, P_n$' refers to the combination that results from inserting $P_1, \ldots, P_n$ into P.

- *clock* is set to 0
- *passives* is set to $\emptyset$
- *clock* is increased by the time value of while $bring\_to\_boil$, $chop\_broccoli$ and $chop\_pepper$, giving 0 + 300 = 300
- while $bring\_to\_boil$, $chop\_broccoli$ and $chop\_pepper$ has f time = 30 so the triple (270,300, 'while fill pot with water and bring to boil') is marked as passive;

the instruction of this process is also noted, so ((270,300),'fill pot with water and bring to boil') is added to *passives*

- *clock* is increased by the time value of while *boil_lentils*, *fry_vegetables* , giving 300 + 2700 = 3000
- while *boil_lentils*, *fry_vegetables* has f time = 2550 so the triple (450,3000, 'while boil the lentils')[18] is marked as passive; the instruction of this process is also noted, so the triple (750, 3000, 'while place lentils in boiling water and cook for 45min, stirring occasionally') is added to *passives*
- *clock* is increased by *strain_lentils*[time], giving 3000 + 60 = 3060
- *strain_lentils* has no passive time
- *clock* is increased by *mix_dahl*[time], giving 3060 + 120 = 3180
- *mix_dahl* has no passive time
- *clock* is saved as the total cooking time

Now the recipe can be generated from the set *ingred_list*, the recipe plan C' and the temporal information. The ingredients list is a printing of every element of *ingred_list*; the list of instructions is a printing of the instruction of each element in C' along with the time at which they are to be performed (this is obtained using a *clock* variable as in *passives*); and the description of passive times is a printing of the information in each triple x in *passives* in the format: 'from $x[0]$ to $x[1]$ while $x[2]$'. The total time for the recipe is also printed. The final generated recipe for 'vegetable dahl' is shown in Figure 6.

This example demonstrates that the representation system and NLG methods discussed in this paper are capable of generating a cooking discourse which is understandable, albeit slightly unnatural. It also shows that the temporal optimisation procedure (Section 6) is effective, clearly there are time savings by performing the instructions concurrently where directed, rather than one at a time. We claim that the temporal optimisation algorithm will always obtain an optimal solution, which here implies there is no faster way of performing the steps in this recipe than the method indicated; that, based on the time assumed for each step, the recipe cannot be executed in less than 53 minutes. However, a proof of the guarantee of optimality is outside the scope of the present paper.

Selected Examples In the absence of any existing metrics for quantitatively evaluating generated recipes, we show several example recipes here which can be inspected qualitatively.

[18]The original direction of *boil_lentils* is stored on insertion, so that it can be used in *passives*.

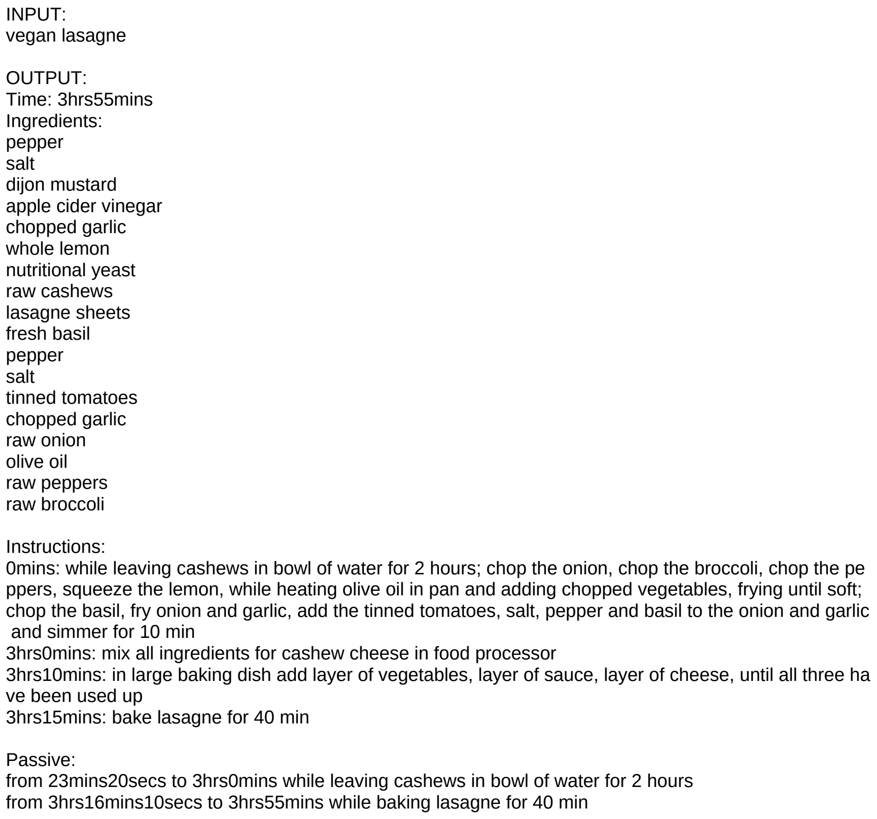
```
INPUT:
vegetable dahl

OUTPUT:
Time: 53mins
Ingredients:
coconut milk
lentils
olive oil
raw peppers
raw broccoli

Instructions:
0mins: while placing water in pot and heating on cooker; chop the peppers, chop the broccoli
5mins: while adding the lentils to the boiling water and cooking for 45mins, stirring occasionally; heat olive
 oil in pan and add chopped vegetables, fry until soft
50mins: strain the lentils
51mins: add the sauteed vegetables and coconut milk to the lentils and stir

Passive:
from 4mins30secs to 5mins while while placing water in pot and heating on cooker; choping the peppers,
choping the broccoli
from 7mins30secs to 50mins while while adding the lentils to the boiling water and cooking for 45mins, stir
ring occasionally; heating olive oil in pan and adding chopped vegetables, frying until soft
```

Figure 6: The recipe generated by FASTFOOD in response to a user input of ‘vegetable dahl’.



```
INPUT:
vegan lasagne

OUTPUT:
Time: 3hrs55mins
Ingredients:
pepper
salt
dijon mustard
apple cider vinegar
chopped garlic
whole lemon
nutritional yeast
raw cashews
lasagne sheets
fresh basil
pepper
salt
tinned tomatoes
chopped garlic
raw onion
olive oil
raw peppers
raw broccoli

Instructions:
0mins: while leaving cashews in bowl of water for 2 hours; chop the onion, chop the broccoli, chop the pe
ppers, squeeze the lemon, while heating olive oil in pan and adding chopped vegetables, frying until soft;
chop the basil, fry onion and garlic, add the tinned tomatoes, salt, pepper and basil to the onion and garlic
 and simmer for 10 min
3hrs0mins: mix all ingredients for cashew cheese in food processor
3hrs10mins: in large baking dish add layer of vegetables, layer of sauce, layer of cheese, until all three ha
ve been used up
3hrs15mins: bake lasagne for 40 min

Passive:
from 23mins20secs to 3hrs0mins while leaving cashews in bowl of water for 2 hours
from 3hrs16mins10secs to 3hrs55mins while baking lasagne for 40 min
```

Figure 7: The recipe generated by FASTFOOD in response to a user input of ‘vegan lasagne’.

```
INPUT:
green stir fry

OUTPUT:
Time: 18mins30secs
Ingredients:
pasta
pepper
salt
olive oil
raw tofu
raw green beans
raw asparagus
raw broccoli
soya sauce

Instructions:
0mins: chop the broccoli
2mins: while placing water in pot and heating on cooker; chop the tofu, chop the green beans
7mins: while adding the pasta to the boiling water and cooking for 10mins, stirring occasionally; chop the
asparagus, heat olive oil in pan and add chopped vegetables, fry until soft
17mins: strain the pasta
18mins: mix fried vegetables and tofu with pasta and soya sauce

Passive:
from 6mins30secs to 7mins while placing water in pot and heating on cooker
from 11mins30secs to 17mins while adding the pasta to the boiling water and cooking for 10mins, stirring
occasionally
```

Figure 8: The recipe generated by FASTFOOD in response to a user input of ‘green stir fry’.

```
INPUT:
pasta e fagioli

OUTPUT:
Time: 17mins
Ingredients:
happy pear tomato pesto
pine nuts
pasta
olive oil
raw peppers
raw broccoli

Instructions:
0mins: while placing water in pot and heating on cooker; chop the peppers, chop the broccoli
5mins: while adding the pasta to the boiling water and cooking for 10mins, stirring occasionally; heat olive
 oil in pan and add chopped vegetables, fry until soft
15mins: strain the pasta
16mins: add the vegetables, tomato pesto and pine nuts to the pot containing the strained pasta and stir

Passive:
from 270 to 300 while placing water in pot and heating on cooker
from 450 to 900 while adding the pasta to the boiling water and cooking for 10mins, stirring occasionally
```

Figure 9: The recipe generated by FASTFOOD in response to a user input of 'pasta e fagioli'.

```
INPUT:
butterbean curry

OUTPUT:
Time: 31mins
Ingredients:
tinned beans
black pepper
curry powder
raw mushrooms
coconut milk
raw tomato
chopped garlic
rice

Instructions:
0mins: while placing water in pot and heating on cooker; rinse the beans, chop the mushrooms
5mins: while adding the rice to the boiling water and cooking for 25mins, stirring occasionally; chop the to
mato, fry the mushrooms, ginger and cumin seeds for 2 mins, then add the garlic, chilli, soy sauce, chopp
ed tomato, coconut milk and curry powder
30mins: strain the rice

Passive:
from 3mins30secs to 5mins while placing water in pot and heating on cooker
from 11mins30secs to 30mins while adding the rice to the boiling water and cooking for 25mins, stirring o
ccasionally
```

Figure 10: The recipe generated by FASTFOOD in response to a user input of ‘butterbean curry’.

```
INPUT:
tofu pad thai

OUTPUT:
Time: 16mins
Ingredients:
coconut milk
raw peppers
chopped garlic
raw mushrooms
tinned beans
raw sugar snaps
fresh coriander
noodles

Instructions:
0mins: chop the mushrooms
2mins: while placing water in pot and heating on cooker; chop the peppers, chop the coriander
7mins: while adding the noodles to the boiling water and cooking for 8mins, stirring occasionally; rinse the
 beans, while frying the mushrooms, scallions and pepper for 2 mins, then adding the garlic, chilli, choppe
d tomato, coconut milk and curry powder; chop the sugar snaps
15mins: strain the noodles

Passive:
from 6mins30secs to 7mins while placing water in pot and heating on cooker
from 14mins30secs to 15mins while adding the noodles to the boiling water and cooking for 8mins, stirrin
g occasionally
```

Figure 11: The recipe generated by FASTFOOD in response to a user input of 'tofu pad thai'.

```
INPUT:
jackfruit bolognese

OUTPUT:
Time: 56mins
Ingredients:
tomato puree
raw tomato
chopped garlic
raw onion
raw carrot
raw jackfruit
raw mushrooms
lentils
pasta

Instructions:
0mins: place water in pot and heat on cooker
5mins: place water in pot and heat on cooker
10mins: while adding the lentils to the boiling water and cooking for 45mins, stirring occasionally; chop the
 tomato, chop the mushrooms, chop the jackfruit, while adding the pasta to the boiling water and cooking f
or 10mins, stirring occasionally; chop the carrot, chop the onion, while frying the mushrooms, scallions an
d pepper for 2 mins, then adding the garlic, chilli, chopped tomato, coconut milk and curry powder; strain t
he pasta
55mins: strain the lentils

Passive:
from 0mins30secs to 5mins while placing water in pot and heating on cooker
from 5mins30secs to 10mins while placing water in pot and heating on cooker
from 26mins to 55mins while adding the lentils to the boiling water and cooking for 45mins, stirring occasi
onally
```

Figure 12: The recipe generated by FASTFOOD in response to a user input of ‘jackfruit bolognese’.