# WHAT WE LEARN ABOUT PROCESS SPECIFICATION LANGUAGES, FROM STUDYING RECIPES

Isabelle M. Demeure
Leon J. Osterweil

CU-CS-373-87 August 1987

Department of Computer Science
Campus Box 430
University of Colorado,
Boulder, Colorado, 80309

# WHAT WE LEARN ABOUT PROCESS SPECIFICATION LANGUAGES, FROM STUDYING RECIPES

Isabelle M. Demeure
Leon J. Osterweil

August 1987

*ABSTRACT*

In this paper, we study recipes as examples of process programs. By doing so, we intend to learn about cooking as a process and recipes as specifications written in programming languages. We are looking for some insights into the principles which should be incorporated into computer programming languages, especially languages for the expression of computer software processes. We are also interested in studying how process descriptions are currently effectively communicated between humans in the specific problem domain of cooking recipes.

Our work is based upon close study of a modest number of recipes. From this study, we infer the features that a language supporting the expression of recipe programs must have. We suggest that a language that is basically algorithmic can suffice, but only if it also supports concurrency, real time constructs, and data type specification that is more complex than what is available in conventional languages. Such a language should probably also support rule-based function evaluation. This language therefore incorporates a mixture of different programming language paradigms (eg. algorithmic, real-time, and rule-based).

This suggests that it is necessary to create languages which effectively mix paradigms if we are to cleanly and satisfactorally support the programming of human processes. In that such a language would draw upon innate human capabilities, it would probably also have much wider applicability.

## 1. BACKGROUND.

It has been suggested [Ost87] that software activities such as development and maintenance should be viewed as systematic processes which can and should be described thoroughly and rigorously. Such processes are examples of what we call indirect problem solving. Indirect problem solving is a technique that humans seem use to solve a variety of very large classes of problems. This technique entails the creation of a description of a general procedure for solving all members of the class of problems. This procedure is then bound to the specific details of an instance of the class, so that the description can be applied to the solution of the problem instance. Thus, in indirect problem solving the solution of a range of many, and perhaps varied, instances of a large and important problem can all be effected through the use of a single solution created by a single problem solver.

We believe that indirect problem solving is a technique which is commonly used to good effect by humans in a wide range of everyday activities. We believe that direction giving, instruction writing, and procedure manual creation are all examples of indirect problem solving. Office, legal, and beaureaucratic procedure specification are also examples. In each case a human creates a detailed and (hopefully) complete and correct specification of how a particular sort of job is to be done. Other humans use this specification to guide their actions in particular specific situations. Thus one person may write a general description of how to drive to a particular location and other people use that description to guide themselves to that place. One human may describe how a kit is to be assembled and others use that description to carry out the job under a variety of circumstances.

In a similar way, humans also create computer programs as detailed and (hopefully) complete and correct specifications of how data manipulation jobs are to be carried out. Programs describe general processes for getting jobs done. When they are fed input data, the input data sets describe the specifics of getting a particular job done. The binding of this data to an execution on a computer effects the solution of the specific problem. Thus, a matrix inversion program describes how to invert matrices in general. The binding of the program to a specific matrix and subsequent execution of the program on a computer effects the solution of the specific problem of inverting that particular matrix.

In [Ost87] it is suggested that the problem of creating computer software can itself be viewed as a general class of data processing problems. It is suggested that every software development or maintenance activity involves the solution of a particular problem which can be solved by the application of a more general process. It is suggested that this more general development or maintenance process should be thought of as being programmable and that we should devote more time and attention to developing these software development and maintenance processes which can then be used to guide the solution of our specific everyday software development and maintenance problems. A major obstacle to being able to do this effectively is the lack of a language which is known to be a good medium for the expression of software processes.

We believe that we can come to understand the characteristics that a software process programming language ought to have by studying linguistic vehicles which have already proven themselves to be effective for the expression of processes between humans. We believe that cooking instructions (ie. recipes) are good examples of general process descriptions which humans already use effectively to solve specific problems (ie. creating dishes). Thus, the linguistic vehicles incorporated into recipes must be effective vehicles for communication of process information between human problem solvers. We believe it is reasonable to expect that these linguistic vehicles may well also be indicative of the constructs which should be used in programming computer applications, not just software processes. For this reason we think it is worthwhile and important to study recipes as examples of process programs. We think that by doing so we will be able to learn about cooking as a process and recipes as specifications written in a programming language. We are hoping to discover some basic principles of how process descriptions are currently effectively communicated between humans in the specific problem domain of cooking recipes. We hope that this might then shed light on the principles which should be incorporated into computer programming languages, especially languages for the expression of computer software processes.

## 2. APPROACH TO THIS WORK.

The approach we took was to examine some recipes closely in order to determine the structures and techniques which their authors used to convey to cooks the processes to be used in order to create dishes effectively. We decided to view these recipes as pieces of code and to study them from a base of our experience and intuition as programmers. Thus we studied, among other things, the data specification techniques which they exploit, the

algorithmic or flow-of-control constructs which they use and their use of parallelism.

As will be seen, we were impressed by the enormous amount of implicit information which is contained in recipes. We were struck by the fact that the process descriptions with which we are most familiar, namely computer code, are quite long and detailed, but that cooking process specifications are quite terse and omit considerable amounts of detail. We also noted that recipes and cookbooks are popular media for the communication of processes. It has been observed that the specification of software processes may not be popular among software practitioners because such process specifications are likely to be so detailed and specific that they will thwart programmer creativity. Thus, the terseness and lack of detail of recipes may indicate not only that such terseness is possible in process descriptions, but also that it is essential to acceptability. We became particularly interested in how this terseness is achieved in recipes.

These and other conclusions are best reserved until the end of this paper. For now, after a quick analysis of how recipes are composed, we shall describe the specific observations we made as a result of examining recipes as specifications of cooking processes, and as a result of attempting to materialize the implicit information which they contain.

We focused on the translation of two recipes "SUPREME DE VOLAILLE AU FOIE GRAS" and "CHEESE SOUFFLE". The first one comes from [Mil86] and is found in appendix 1; the second one is adapted from a recipe extracted from [RoR73], and is found in appendix 2.

## 3. A QUICK STUDY OF RECIPES.

### 3.1. Introduction.

There is a very abundant cooking literature, and each cook book has its own style and personality. Recipes can take different forms. They sometimes come with pictures or drawings of the dish in its final state, or of some of the main steps of the preparation. In other cases, only text is presented, following the pattern chosen by the author(s) of the book.

Although recipes are written following very different patterns, the following components can be identified in most of them:

- "extracted information" (title, number of servings, ...);
- list of ingredients;
- list of tools;
- the body of the recipe;
- comments about the recipe;

This list represents the principal sorts of semantic information which is explicitly found in virtually all recipes. It is interesting to note that virtually all of this information corresponds rather directly to analogous information stated explicitly in computer programs.

### 3.2. Extracted information.

Certain pieces of information are extracted from the body of the recipe, and listed, either at the beginning or the end, of the recipe. Among them are:

- the title of the recipe;
- the number of servings;
- the time required to cook;
- the time required to bake;
- the temperature of the oven;

- the level of difficulty of the recipe;
- the cost of the required ingredients;
- etc ...

This type of information gives the reader a quick notion of how many resources the execution of the recipe is likely to consume, as well as a notion of what the product will be like. Much of the information (eg.: cooking time) is redundant with information expressed in the body. This type of information is analogous to the sort documentation which is generally supplied externally with high quality programs. Often it is also embedded in comments at the beginning of program code.

### 3.3. Ingredients.

The ingredients are the "raw materials" of cooking. They are inputs to the cooking process. There are syntactic differences in how they are specified. Sometimes they are listed at the beginning of the recipe (as in recipe 1); sometimes they are nested in the body of the recipe (as in recipe 2). The required amount of each ingredient is given, as well as some specification of its type if required (eg.: "cake flour" instead of "flour"). They can be either basic ingredients (eg.: "unsalted butter"), or already processed ingredients (eg.: "unsalted butter, clarified"). In the first case, all the processing activity is left to the body of the recipe, in the second one, part is assumed to be done before fixing the recipe. In all cases, we believe that ingredient specifications in recipes are strongly analogous to data declarations in code. We will elaborate upon this in the next section.

### 3.4. Tools.

Some examples of tools used in cooking are pans, whips, spoons, and skillets. They are devices or agents required in the preparation of the recipe. In some cook books an entire special section is dedicated to specifying the range of tools required in all recipes in the book. The tools are then referred to from the bodies of the recipes. In most books, however, mention of tools is nested in the bodies of each of the recipes. We believe that the recipe toolset is analogous to virtual machine on which a program is to run. We note that the actual hardware machine is rarely specified explicitly within program code. On the other hand well-organized, "top-down" code usually rests heavily upon a substrate of support libraries and procedures, which constitute the virtual machine on which the high-level code "runs." These high-level procedures generally do include explicit specifications of their support procedures in external documentation and inline comments.

### 3.5. The body of the recipe.

The body of the recipe is the central part of it. It specifies the procedural steps to follow in order to prepare the dish. In most recipes, the main steps are clearly identified. They are sometimes numbered, sometimes not. Some of the steps have to be done sequentially, others can be done in parallel. Sometimes larger steps are clearly decomposable (often by explicit reference to other recipes) into smaller steps. We found that recipe procedures employ looping, branching, procedure invocation and parallelism constructs. Thus, virtually all of the control flow capabilities of a classical algorithmic language such as Pascal are used in the specification of recipes. Parallelism constructs are also used.

Most of the time, each step is a simple operation. It is described using:

- "everyday words";
- technical cooking terms;
- references to pictures or drawings;
- references to other recipes;
- references to a different section describing certain techniques;

These descriptive vehicles also help to define the virtual machine in terms of which the recipe is expressed. They correspond quite directly to the use of support libraries and procedures in structured program code.

### 3.6. Comments about the recipe.

Some aids and suggestions about the recipe are usually given before or after the body of the recipe. For example: references to specific sections of the cook book, or to other recipes;

- general indications about the the recipe and other existing versions of it;
- explanations about the ingredients (if they are exotic, or possibly difficult to find);
- tricks to preparing the recipe;
- how to prepare all or part of the dish in advance, freeze it, reheat it, etc.
- possible substitutions for one or several of the ingredients, either because they are not easy to find or as variants of the recipe;
- suggestion of what sauce, vegetables, etc., to serve with the dish;

Clearly, such comments, correspond to program documentation ranging from input specification to environmental constraints, to interface specifications.

This cursory comparison of cookbook recipes and computer program code suggested to us that strong analogs might exist, and encouraged us to examine them in more detail.

Recall that Appendices 1 and 2 of this paper consist of complete "encodings" of two recipes. We shall refer to such encodings as "recipe programs." The two appendices are used as sources of the illustrative examples cited in the following sections of the paper. The reader will notice that the syntax used in these appendices and imbedded examples is not rigorously defined. We chose to introduce the syntax of our "recipe language" by example. We felt that it was a good way to introduce various constructs for discussion while not distracting readers from the key concepts being presented. We believe the examples are sufficiently straightforward as to not require rigorous syntactic definition.

## 4. DATA DECLARATION AND MANIPULATION.

### 4.1. Introduction.

As noted in the previous section, ingredients specifications in recipes seem to correspond closely to data declarations in program code. In this section, we examine the semantic content of ingredients specifications and study the extent to which modern programming language technology is capable of capturing this semantic content accurately and effectively.

We view ingredients as the input data of the recipe, which, after a succession of transformations, become the final dish--the output result. In our recipe programs, we were able to model ingredients as "ingredient variables" that are instances of known "ingredient types". We declared them before using them, as we would do for the variables used in a computer program. Most recipes require that each ingredient receives a preprocessing treatment (eg. peel, chop) before being used. Thus, we found it helpful to define sets of accessing functions on the various ingredient types, where the accessing function sets differed for each different ingredient type (eg.: a tomato can be seeded; a carrot cannot). When studying the possible types of ingredients and their accessing functions, we discovered that there are families of ingredients that can be accessed through the same set of accessing functions (eg.: carrots, onions and shallots can be peeled, sliced, sauted in fat, etc.). Subfamilies of ingredients that can be handled in a similar way, can also be identified (eg.: onions and shallots). This led us to believe that it is useful to organize the ingredient domain into a hierarchy of types according to the accessing functions available on each type. In the remainder of this section we explain how ingredients can be declared, and how to use "ingredient variables". The hierarchy of ingredient types that we adopted is presented in an Appendix 3.

Finally, it should be noted that ingredients are also processed by applying a succession of operations to them. These operations will be described in more detail in a later section of this paper.

### 4.2. Declaration of the ingredients.

The list of the ingredients required for a given recipe can be made explicit in a *Declare INGREDIENTS* section of the recipe program. Each ingredient must be declared as an instance of a known type. An example syntax for the declaration of an ingredient might be:

```
ING INGREDIENT_NAME is TYPE_NAME amount_is AMOUNT unit_is UNIT
attributes_are(list of attribute values)
    access_is (list_of_accessing primitives);
```

where:

- INGREDIENT_NAME identifies the ingredient (eg.: BUTTER_TO_FRY_BREASTS);
- TYPE_NAME is the type of the required ingredient (eg.: T_BUTTER);
- UNIT is the unit in which the amount of required ingredient will be expressed (eg.: CUP);
- AMOUNT indicates the required amount of the ingredient (eg.: 0.75);
- "attributes_are" is followed by the list of the actual values of the attributes required by the recipe (eg.: (salt: TRUE) if the butter needs to be salted);
- "access_is" is followed by the list of the accessing primitives that are actually going to be used in the body of the recipe; (The accessing functions are listed for "documentation purposes". Listing them informs the reader of the kinds of "skills" required for the recipe) (eg.: clarify);

Thus, in Appendix 1, we find the following ingredient definition:

```
ING BUTTER_TO_FRY_BREASTS is T_BUTTER amount_is 0.75 unit_is CUP
        attributes_are(salt: TRUE) access_is (clarify);
```

This ingredient declaration mechanism proved to be entirely sufficient in writing our recipe programs.

### 4.3. Access, use and evolution of the ingredients.

In our recipe programs we always applied accessing functions to the ingredients in the body of the recipe, the first time we used them. For example in the procedure "BAKE_BREASTS" in Appendix 1, we have:

```
add_to(clarify(BUTTER_TO_SAUTE_BACKS), SKILLET),
```

indicating that the butter added to the skillet must be clarified. In this example, add_to is an operation, SKILLET is a tool, BUTTER_TO_SAUTE_BACKS is an ingredient of type T_BUTTER, and clarify an accessing function on type T_BUTTER.

Accessing functions can be composed using familiar function composition rules. If, for example, BREAST is the ingredient name of a chicken breast, and if the accessing functions used on BREAST are skin, bone and halve, at the time of the invocation of the ingredient BREAST, we will have "halve(bone(skin(BREASTS)))". The order in which the composition is given is the one in which the corresponding pre-treatments of the ingredient must be done. Other orders might also have been specified.

It is interesting to note that recipes often do not specify the exact order of such operations because the order is not important. Thus, recipes often show nondeterminism. This is not well supported by classical programming languages. Our investigations indicated that it is comfortable and useful to both writers and readers of recipes and that it would probably be useful to have it supported more commonly in contemporary programming languages.

Sometimes one needs to refer to only part of an ingredient. We found that this is conveniently modeled by the *get_part_ing* primitive. The syntax of the primitive is the following:

```
ingre := get_part_ing(ing, unit, amount)
```

where:

- ing is an ingredient;
- ingre identifies the part of the "ing" ingredient returned by get_part_ing;
- unit is the unit in which the amount of required ingredient will be expressed;
- amount indicates the required amount of the ingredient;

The ingredients evolve as the recipe program is executed. At first (when declared), they are unprocessed ingredients. Then, when they are first referenced, they evolve to preprocessed ingredients ready to be used by the recipe. After that, they are transformed by the operations described in the recipe. For the sake of clarity, we have chosen to use a single variable name to represent a single processed ingredient throughout the recipe. This choice is analogous to the computer programming stylistic directive to avoid the use of a single variable to hold a variety of conceptually different values, and to consistently use a single variable name to hold a single conceptual value.

e.g:

In the example in Appendix 1, we have the following succession of operations on BREAST_TO_STUFF[i] that transform it from a piece of chicken breast to a chicken breast stuffed with foie gras and seasoned:

```
remove_from(large_tendons, halve(bone(skin(BREAST_TO_STUFF[i]))));

detach_from(fillet_of(BREAST_TO_STUFF[i]), BREAST_TO_STUFF[i]);

slit_lengthwise(BREAST_TO_STUFF[i]);

put_into(foie_gras_slices, BREAST_TO_STUFF[i]);

season_with(BREAST_TO_STUFF[i], SALT);

season_with(BREAST_TO_STUFF[i], PEPPER);
```

### 4.4. A hierarchy of ingredient types for recipe-programs.

We found that a hierarchy of types can be quite useful as a model for the domain of the types of ingredients used in cooking. For each type we assigned a name, a collection of attributes, and a list of accessing functions for it. The attributes allow certain characteristics of the type to be defined. The accessing functions are the basic preprocessing activities that can be done on the corresponding ingredient type.

We found, further, that type inheritance can also be a useful concept. We experimented with the use of a strict type hierarchy such as used in Smalltalk (TM) [GoR83]. With this hierarchical discipline, when a type is defined as a subtype of another, in addition to its own attributes and accessing functions, the subtype also inherits all the accessing functions and attributes of its parent type. Further, a given type can have only one parent type.

By using this hierarchy of types to describe the ingredient domain, we captured the way in which ingredients can be handled. We will discuss the adequacy of this strict inheritance hierarchy later on in this section.

The example ingredient type hierarchy given in appendix 3 calls for several comments.

First, it is just a portion of the complete hierarchy we defined. Only some of the existing types are described, and even for these, none of the subtype sections are complete, and a lot of attributes are omitted. Note, however, that the entire hierarchy is not local to any recipe.

Second there are many different ways in which to establish this classification. We decided to create formal types for at least each of the main "families" of ingredients (eg.: carrots, butter). Then we tried to create "groupings" for types (therefore considered as subtypes) that have several attributes and accessing functions in common (eg.: T_CRUNCHY groups T_CARROT, T_TURNIP, T_ONION-FAMILY, T_CELERY-ROOT that can all be peeled, diced, sliced, etc; T_MUSHROOM is excluded from this category because it cannot be peeled). Some of these groupings are purely fictitious (eg.: T_EGG-FAMILY), but others map to a recognized category (T_VEGETABLE). Sometimes we decided to create a type for things such as an egg yolk, or a chicken breast that are parts of an existing type. The criterion was that a recipe can call for such parts, without calling for the whole ingredient (ie. the egg

or the chicken) from which it is extracted, and we can purchase a chicken breast separately, or have an isolated egg yolk left from another recipe calling for egg whites only.

Finally, a hierarchical decomposition allows for grouping the elements according to only one criterion. Our classification is done on the accessing functions of each type. It therefore puts the emphasis on the way ingredients can be handled through accessing functions. This will be useful in identifying what kinds of cooking skills are needed when scheduling the preparation of the recipe. Other hierarchical classifications could also be interesting, however. For example, a classification on the "chemical" properties of the different types would tell whether a specific ingredient is a thickening agent or not. This could be useful if we need to find a substitution for some ingredient.

Thus, we concluded that a strict type hierarchy is not a completely adequate data organization vehicle. Because it stresses only one type of criterion, a strict type hierarchy gives only a single restricted view of the ingredient domain. Having several type hierarchies, as suggested above, is a possible remedy to this inconvenience, but seems awkward. Another solution, is to define a set of attributes (list of accessing functions, taste characteristics, chemical properties) for the types. Each type would have specific attribute values. We could then get different views of the ingredient domain, depending on the attribute value(s) we are interested in (eg.: if we are interested in finding substitution for an ingredient, we can select the ingredients which have the same chemical properties and/or taste characteristics). This kind of typing mechanism does not currently seem to be popular among computer language designers. This example (and others we have found in more traditional computer science applications) indicates that it bears investigating, as it seems natural, intuitive and useful.

From a still broader perspective, we see that in creating a type structure we were really attempting to build a representation of the knowledge we have about cooking ingredients. It should not be surprising that a strict hierarchy is inadequate to completely represent this knowledge. A far more generalized semantic network of some sort would surely come much closer to doing the job. On the other hand, this made us reflect upon the fact that most complex computer programs must similarly rest upon a base of intricately organized datatypes. Languages which impose too strict a modelling regime upon this structure must often be similarly restrictive and inadequate.

## 5. ALGORITHMIC PROCEDURAL FLOW.

### 5.1. The use of a Pascal-like language.

Recipes seem to be almost universally expressed in a very algorithmic form. We found that a "Pascal-like" language was adequate to express most of the instructions given in the body of the recipes we studied. Our recipe programs use such basic Pascal constructs as conditional statements, (if-else) and looping constructs (for, while-do, until-do).

eg.:

In the example given in Appendix 1, we find the following code, that illustrates the "until-do" construct:

```
if (contents_of(recipient) too liquid)
begin
        add_to(CORNSTARCH_TO_THICKEN_SAUCE, recipient);
        until (contents_of(recipient) has light coating consistency) do
        begin
                let_simmer(contents_of(recipient));
        end;
end;
else -- contents_of(recipient) too sirupy --
begin
        add_to(WATER_TO_LIQUEFY_SAUCE, recipient);
        until (contents_of(recipient) has light coating consistency) do
        begin
                let_simmer(contents_of(recipient));
```

```
        end;
end;
```

The "for" construct is used in the following sequence, to show a repetition of the same sequence of actions:

```
for i=1 to 2
begin
        slit_lengthwise(BREAST_TO_STUFF[i]);
end;
```

We also noted the widespread use of procedural decomposition to emphasize the structure of the recipes we studied and the different tasks from which they are composed. Most of the recipes are written in a way that makes their structure (the hierarchy of processing activities, the succession of steps, the identification of the different tasks, the parallelism of some tasks, and so forth) obvious. In translating the recipes into recipe programs, we tried to follow this structure as much as possible.

eg.:

The preparation of the rich stock that serves as a basis for the sauce of the Supremes consists of four main steps. We make this clear by declaring a procedure corresponding to each of the steps. In the example given in appendix 1,we therefore have:

```
Procedure FIX_RICH_STOCK;
begin
        FIX_STOCK_INGREDIENTS;
        BAKE_STOCK_INGREDIENTS;
        FIX_STOCK;
        SIMMER_STOCK;
end;            -- end_of FIX_RICH_STOCK --
```

All recipe programs start with a "Program" statement. The syntax of this statement is:

Program *recipe_name*(*unit*, *amount_in_unit*)

where:

- *recipe_name* is an identifier for the recipe program,
- the pair (*unit*, *amount_in_unit*) designates the amount produced by the recipe, either in number of servings - in which case the unit is "serving", in number of cups - in which case the unit is "cup" -, or in any other unit.

eg.:
In the example given in appendix 2, we have:

Program BECHAMEL-SAUCE(CUP, 4)

meaning that given the amounts of ingredients required, the recipe will produce four cups of bechamel sauce, and:

Program CHEESE-SOUFFLE(SERVING, 4)

meaning that the recipe is good for four servings.

A lot of recipes use the product of another recipe as an ingredient. For example, the basis of a cheese souffle is bechamel sauce (see example in appendix 2). Thus, we introduced the "extern program" declaration. The extern program declarations are expected to be found, if needed, right after the "program" statement, and before the "Declare ingredients" section. The syntax of the extern program declarations is:

extern program *recipe_name(unit, amount_in_unit)*

where:

- *recipe_name* is the identifier of a recipe that has been declared somewhere else,
- the pair (*unit, amount_in_unit*) indicates the amount required as an ingredient by the current recipe.

eg.:
In the example given in appendix 2, we have:

```
Program CHEESE_SOUFFLE(SERVING, 4);

Extern program BECHAMEL_SAUCE(CUP, 1);

Declare ingredients
....
```

As shown in the following sequence of code, the BECHAMEL_SAUCE is then used as if it were an ingredient declared in the "Declare ingredients" section:

```
Procedure FIX_BECHAMEL_SAUCE;
begin
/* Prepare 1 cup of bechamel sauce */
        acqu_tool(CASS_FOR_MIXTURE);
        add_to(FIX(BECHAMEL_SAUCE(CUP, 1)), CASS_FOR_MIXTURE);
end;            -- end_of FIX_BECHAMEL_SAUCE --
```

Finally, we found it necessary to add a set of operations adapted to the cooking domain. We shall describe the set of "cooking operations" we adopted in a later section. We also noted the need to express parallelism, define critical sections, and make use of an error handling mechanism. These aspects are addressed in the next section.

## 6. NON-ALGORITHMIC CONTROL FLOW CONSTRUCTS.

### 6.1. Parallelism.

Some recipes give clear indications about tasks that can be executed in parallel (e.g: Simmer for 15 minutes. Meanwhile, ...). Others do not, but the cook naturally identifies the tasks that can be done in parallel.

We introduced a parbegin/parend statement to explicitly indicate which tasks, although part of the same recipe, can be executed in parallel. The syntax is:

```
parbegin
        [begin ... end] --task 1 --
        [begin ... end] --task 2 --
                .
                .
                .
        [begin ... end] --task n --
parend
```

The statements inside a begin/end pair correspond to a task, and have to be executed in sequence.

An example of the use of the parbegin/parend construct is found in the following sequence:

```
parbegin

        begin           -- begin task 1 --
                FIX_RICH_STOCK;
                FIX_SAUCE;
        end;            --end task 1 --

        begin           -- begin task 2 --
                STUFF_BREASTS;
                PRE_COOK_BREASTS;
                COAT_BREASTS;
        end;            -- end task 2 --

parend;
```

Task 1 includes the operations necessary to fix the sauce. Task 2 groups the successive operations required to fix the breasts before finally cooking them. These two tasks are totally independent and can be conducted in parallel. We found many ways in which recipes can be parallelized. We noted that cooks are usually quite adept at identifying places and ways in which such parallelization can be done, despite the fact that recipes often do not make such specifications explicit. We have also had experiences in which we have incorrectly supposed that parallelism was possible. The contrast between the casual way in which parallelism is specified in recipes and the rigorous and painstaking way in which it is specified in program code is striking. We were tempted to conclude that the rigor and precision used in programming languages might be unnecessary, but then realized that cooks often parallelize incorrectly (much to their sorrow) and generally draw upon considerable amounts of implicit information. Thus we concluded that parallelization is inherently tricky and probably requires the precision of a rigorous programming formalism. Moreover, we are tempted to conclude that this rigor would be useful to recipe writers, and other "programmers" of natural human processes.

### 6.2. Active versus stand alone sequences.

We named operations that must be performed by a cook such as chop, set_heat, ... (see section 7 for more details on these), "active operations". Operations that do not require the active participation of the cook such as bring-to-boil, let-bake, ... are called "stand alone operations".

Similarly, we called a sequence of consecutive active operations within a procedure an "active sequence" and a sequence of consecutive stand alone operations within a procedure is called a "stand alone sequence".

This distinction proved to be important in studying the parallel structure of recipe programs.

### 6.3. Critical sections.

We also found it useful to exploit the programming notion of a critical section. In recipe programs, a critical section is a sequence of operations that must be carried out without pauses (or waiting times) between them. The only time during a critical section when the cook can do something else, is during the execution of a "stand alone" sequence. Even in such cases, however, the cook must be available as soon as it is finished, to go on with the following active sequence.

In our encodings of recipes we found that it was important for all bottom level procedures to be considered to be critical sections. We delineated critical sections by placing them between "begin-critical-section" and "end-critical-section" statements.

An example of critical section is given by the following sequence:

```
begin_critical_section
-- Once the breasts are fried they must be served without waiting --

        FRY_BREASTS;
```

SERVE;

end_critical_section

In the future, we would like to devise algorithms that are able to automatically schedule the parallel execution of several recipes. The notion of a critical section seems essential to being able to do this.

### 6.4. Decomposition of the recipe into procedures.

In a previous section we have already given some guidelines which we used in the decomposition of a recipe program into procedures. We can now add some more rules that were found to be useful in facilitating the scheduling of several recipes in parallel.

(1) The structure of the recipe is generally best seen by using the upper level cooking procedures as a level of decomposition (a sequence of procedure calls bracketed, if needed, by parbegin/parend statements and/or begin-critical-section/end-critical-section statements). This leaves the specification of the actual operations to be performed to the bottom level procedures.

(2) Each bottom level procedure should correspond to a sequence of operations that must be accomplished without pauses between them (ie. a critical section).

(3) Each bottom level procedure should, as much as possible, include the acquisition and the release of its required tools. This allows the bottom level procedure to be considered to be an independent execution unit.

We believe that these rules are also useful as guidelines in organizing computer program code.

### 6.5. Timers.

In encoding recipes we found it useful to create a timer construct which models commonly used kitchen timers. Such a timer is a mechanism that enables observation and control of time elapsed. A duration is specified to the timer, and an alarm is given when the corresponding time has elapsed. This timer mechanism differs from the classical interrupt mechanism in computers, in that different timer alarms are not keyed to different prespecified alarm handlers.

We identified the need for at least three types of timers: "stand alone" timers, "periodical" timers, and "active" timers.

We used stand alone timers to set an "interrupt" to occur at the time when it is necessary to end a "stand alone" sequence (as a stopping condition). The accessing primitives we used to manipulate timers of this type were:

set-std-alone-timer(timer-identifier, time, unit_time);
to set the timer;

bool-value = ring-std-alone-timer(timer-identifier);
to determine when time has elapsed;

cancel-std-alone-timer(timer-identifier);
to cancel the timer before it ends;
(makes the value of ring-std-alone-timer TRUE).

Periodical timers were used to model an "interrupt" that must occur periodically. They are mainly used to control operations that must be performed "from time to time" such as stirring the contents of a casserole, or checking a dish in the oven. After a periodical timer has given an alarm it is automatically reset again until explicitly unset. The accessing primitives we used to manipulate timers of this type were:

set-period-timer(timer-identifier, time, unit_time);
to set the timer;

bool-value = ring-period-timer(timer-identifier);

to know when time is elapsed;

cancel-period-timer(timer-identifier);
to cancel the timer and prevent it from being automatically
set again;
(makes the value of ring-period-timer TRUE).

Active timers were used to model the control of an active sequence. An active timer alarm was used as a reminder that it is time to stop some activity

set-active-timer(timer-identifier, time, unit_time);
to set the timer;

bool-value = ring-active-period-timer(timer-identifier);
to know when time is elapsed;

cancel-active-timer(timer-identifier);
to cancel the timer before it ends;
(makes the value of ring-active-timer TRUE).

eg.: in procedure BAKE_STOCK_INGREDIENT of recipe "SUPREMES DE VOLAILLES AU FOIE GRAS"

```
/* bake stirring from time to time, for 30 minutes
  or until nicely browned */

set_period_timer(BAKE_STOCK_INGREDIENTS_2, 5, MINUTE);
set_std_alone_timer(BAKE_STOCK_INGREDIENTS_1, 30, MINUTE);
not_done = TRUE;
while (not_done) do
-- loop until the contents of the casserole is nicely browned --
begin
        until(ring_std_alone_timer(BAKE_STOCK_INGREDIENTS_1)) do
        begin
                when (ring-period-timer(BAKE_STOCK_INGREDIENTS_2)) do
                begin
                        stir(contents-of(LARG_CASS));
                        if(contents-of(LARG_CASS) is-nicely-browned)
                        begin
                                cancel-std-alone-timer(BAKE_STOCK_INGREDIENTS_1);
                                cancel-period-timer(BAKE_STOCK_INGREDIENTS_2);
                                not_done = FALSE;
                        end
                end
        end;
        if (not(contents_of(LARG_CASS) is_nicely_browned))
                set_std_alone_timer(BAKE_STOCK_INGREDIENTS_1, 5, MINUTE);
        else
        begin
                not_done = FALSE;
                cancel-period-timer(BAKE_STOCK_INGREDIENTS_2);
        end;
end;
```

Timers are declared in a "Declare timers" section. Each timer is of one of the three known types: stand alone timer, active timer and period timer.
eg.: in procedure BAKE_STOCK_INGREDIENT of recipe "SUPREMES DE VOLAILLES AU FOIE GRAS"

Declare timers

```
        std_alone_timer BAKE_STOCK_INGREDIENTS_1;
        period_timer BAKE_STOCK_INGREDIENTS_2;
end;
```

Cooking is clearly a real-time process. Thus it should come as no surprise that recipe programs must draw heavily upon the use of timer mechanisms. We must confess that our timer trichotomy is rather ad hoc, and we are not convinced that we have a properly complete and exhaustive set of timing capabilities described here. Further, as contemporary programming languages rarely address real-time programming issues effectively, we had few programming language models upon which to draw. We were struck by the fact that recipes convey real-time information with remarkable clarity and terseness, especially as compared to the more rigorous programming constructs used in our recipe programs. This would seem to pose a challenge to programming language designers.

### 6.6. The use of assertions.

Recipes also contain many indications of errors and anomalies that can occur and often indicate what to do about them. Much of this is expressed implicitly, much is covered in introductory sections or appendices of cookbooks, but much of it is expressed explicitly within the recipe. Thus it was necessary for us to devise a mechanism for detecting errors or anomalies and a mechanism for dealing with them. We used the assert statement to specify these conditions, and an Ada-like exception handling mechanism to deal with them.

It is not uncommon to see phrases such as "... at this point the sauce should look like....." and "....you should now be able to see whip marks in the cream....." embedded in recipes. Sometimes there are also remedial instructions in case the dish is not developing in accordance with these conditions. We expressed these phrases as assertions, not unlike the assertions found in annotation languages such as Anna [LuH85].
eg.:
In procedure FIX_SAUCE we translate the comment "the sauce should have a light coating consistency" by:

```
assert(contents_of(SAUCEPAN) has light coating consistency)
else raise ADJUST_SAUCE_CONSISTENCY;
```

### 6.7. Error or exception handling.

The error handling mechanism we used was borrowed from Ada. We distinguish between two types of errors: "predefined errors", and "user defined" errors.

The class of "predefined errors" consists of those (such as "tool not available"), that can happen in any recipe-program, and can be handled in the same way in all these recipe-programs (unless explicitly overridden). We consider that these errors are automatically handled by the "executing agent" (in our case, the cook) in the same way that "divide by zero" is automatically handled by runtime hardware/software systems. These "predefined errors" therefore appear to be handled in a rule-based way. Each predefined error corresponds to a condition. If a condition turns out to be true, the corresponding rule (implemented as an error handler) is applied. If none of the conditions is true, the execution of the program goes on. Thus, here we see evidence of the value of a recipe programming language which offers support for a rule-based programming paradigm in addition to an algorithmic paradigm.

The class of "user defined" errors consists of those which require specialized handling, which must be defined by the person who translates the recipe into the recipe program.

The error handling mechanism has the following characteristics:

- in the case of "user defined" errors, the recipe programmer is responsible for specifying the conditions under which the error (exception) occurs ; this is done by writing a statement of the form:
  ```
  if CONDITION raise EXCEPTION;
  ```
- for "user-defined" errors, the recipe programmer must also provide the exception handler to be called when the CONDITION occurs;
- we found that the scope of the exception handler was sometimes better modeled as the scope of the procedure in which it was declared, and sometimes better modeled as the scope of the program in which it is declared (in

which case it is accessible from all the procedures defined in the scope of the program);

- handlers for the "predefined errors" were declared as being global to an entire program but might just as well have been defined as global to an entire cookbook. There was no need to use an "if CONDITION then raise EXCEPTION" construct to trigger these handlers.
- we also found it useful to enable the exception handler itself to specify if execution is to be resumed at the point where it was called or if it must resume at some other point in the program (for example to enable redoing a step);

eg.: In the example given in appendix 1, we have:

```
Program SUPREMES_AU_FOIE_GRAS(SERVING, 4);

Declare exceptions -- exceptions global to the program --
  predefined-exception TOOL_NOT_AVAIL;
end;
                              .
                              .
                              .
-- procedure FIX_SAUCE --

Procedure FIX_SAUCE;
exception       ADJUST_SAUCE_CONSISTENCY;
begin
/* Add creme fraiche and bring to a boil over medium high heat. Simmer
   sauce, skimming as necessary, for about 15 minutes or until reduced
   by half. (the sauce should be a light coating consistency)
*/
                              .
                              .
                              .
        assert(contents_of(SAUCEPAN) has light coating consistency)
        else raise ADJUST_SAUCE_CONSISTENCY;
                              .
                              .
                              .
        exception ADJUST_SAUCE_CONSISTENCY(recipient)
        TOOL recipient;  -- recipient containing the sauce --
        begin
                ING CORN_STARCH;
                ING WATER;

                if (contents_of(recipient) too liquid)
                begin
                        recipient = add_to(CORN_STARCH, recipient);
                        until (contents_of(recipient) has light coating consistency) do
                        begin
                        let_simmer(contents_of(recipient));
                        end;
                end;
                else -- contents_of(recipient) too sirupy --
                        recipient = add_to(WATER, recipient);
                until (contents_of(recipient) has light coating consistency) do
                begin
                let_simmer(contents_of(recipient));
                end;
        end;

end;    -- end of FIX_SAUCE --
                .
```

```
                .
                .
exception TOOL_NOT_AVAIL(tool_needed)
TOOL tool_needed
begin
        if (cannot find replacement for tool_needed)
        wait-for-tool
        else get_replacement-tool(tool_needed);
        resume;
end;

end; -- end_of Program SUPREMES_AU_FOIE_GRAS --
```

## 7. THE VIRTUAL MACHINE.

### 7.1. Introduction.

Our study of recipes as pieces of code would not be complete if we did not make some comments about the virtual machine on which they are assumed to execute. The main components of this virtual machine are tools, whose management can be modeled by an operating system resource management model, and the set of cooking operations necessary to create dishes. We shall study these two components in more detail in the remainder of this section.

### 7.2. Tools.

#### 7.2.1. Introduction.

We were able to model the management of cooking implements by treating them as non-consumable resources that must be acquired before they are used, and released when they are no longer needed. This is one aspect of the use of tools. Another aspect is that some tools are containers for ingredients and mixing media for ingredients.

#### 7.2.2. Tools as resources.

Before being used in a recipe, it seems important that tools be declared in a *Declare TOOLS* section of the recipe program. By setting up this declaration section at the front of the program we make it easy for a human execution agent or an automatic scheduling program to readily identify the types and quantities of tool resources which will be needed. This section contains a sequence of *get_tool* initialization primitives. These primitives are intended to model the process of selecting the needed tools and making them accessible during the actual cooking activity

eg.: The following sequence is extracted from the "Declare TOOLS" section of the recipe in appendix 1.

```
TOOL OVEN = get_tool(oven);
TOOL LARG_CASS = get_tool(large casserole);
TOOL BURNER[2] =  for i=1 to 2 get_tool(burner);
TOOL WOOD_SPOON = get_tool(wooden spoon);
TOOL COLANDER = get_tool(colander);
TOOL BOWL = get_tool(bowl);
```

Finally, we modeled the action of acquiring the tool during cooking by the *acqu_tool* primitive. Conversely, the action of returning a tool which is no longer needed to the pool of available resources is indicated by the *release_tool* primitive.
eg.: The following sequence in which a wooden spoon is acquired to stir the contents of a casserole, and then released, is part of the FIX_STOCK_INGREDIENTS procedure in appendix 1:

```
acqu_tool(WOOD_SPOON);
stir(contents_of(LARG_CASS), WOOD_SPOON)
release_tool(WOOD_SPOON);
```

Usually, when a tool is needed and is not available, the cook must either wait for it to "be released" or must find a replacement for it. We modeled this situation as an exception, handled by the "TOOL_NOT_AVAIL" exception handler (see the section "exception and error handling").

### 7.2.3. Tools as containers.

Tools such as bowls, pans or skillets are containers for ingredients and mixing media for ingredients. We found it useful to be able to consider the contents of any such tool as a new "ingredient" that can itself be modified. For example, we have:

```
add_to(BUTTER_TO_SAUTE_BACKS, LARG_CASS);
let_melt(contents_of(LARG_CASS));

add_to(chop(CHICKEN_BACKS_FOR_STOCK), LARG_CASS);
```

In this sequence, the first statement adds butter to the large casserole. The second one performs an operation (letting melt) on the contents of the casserole, namely the butter. The "contents_of" accessing function is used to show that "let_melt" operates on the contents of the casserole, and not on the casserole itself. The third instruction is similar to the first one; as a result of it, the casserole now contains chicken backs in addition to the melted butter.

Thus, in this case tools are really just being used as "handles" for objects being stored in what we can call "the recipe's persistent object store".

## 7.3. The operations.

### 7.3.1. Introduction.

In order to be able to translate instructions written for a cook as a recipe into a recipe program, written in a programming language, we need a set of operations well adapted to the cooking domain. We were able to distinguish four classes of cooking operations, and they seemed to be sufficient for the expression of the recipe programs which we translated.

The first class of instructions consists of operations which are very basic. They do not require any cooking skills, and are accomplished in a single step. We call them the primitive operations or *primitives*.

The second class consists of operations are those which have to be initiated by the cook and require some time after their initialization, but they they can continue without the cook doing anything. We call these the *let operations*.

A third class of operations which we call *simple operations* includes the ones that require some intervention of the cook but are not very technical.

Finally, the fourth class is the class of *technical operations*. They are usually described in the body of the recipe using dedicated technical cooking terms. They embody rules for cooking.

We found it reasonable to represent the invocation of all of these operations by a procedure call. We found that some instructions had side effects, modifying an ingredient or the contents of a recipient, while others did not. eg.:

```
roll_into(BREAST, CRUMBS);
(after the operation, the BREAST is coated with CRUMBS);

pour_into(CRUMBS, KITCHEN_PLATE);
(after the operation, the plate has crumbs in it);

set_heat(BURNER, high_heat);
(sets the temperature of the burner to high);
```

Before proceeding further, it is important to point out that the first class of operations can be thought of as being completely described and eminently automatable by a robotlike device, while the fourth class of operations requires insight, knowledge, training, and expertise.

It is important to ponder the formal mechanisms necessary to completely describe and assimilate the fourth class operations into a recipe program, as these seem to tax the descriptive power of recipe programs most severely. We believe that these operations and related predicates, such as those used in some assertions to identify errors, constitute a knowledge base upon which cooking rests. Although we did not really explore this possibility, we believe that a rule-based formalism such as the prolog language is likely to prove quite adaptable to effective expression of these operations and predicates.

We observed, however, that rule-based operators and predicates seem to play a supplementary, rather than a dominant, role in our recipe programs. Initially we had been prepared to believe that recipe programs might well be best modelled simply as rule sets. Our experience has indicated, however, that recipes seem better modelled with an algorithmic, concurrent, real-time language, in which a significant, but modest, number of primitive operations must be left unelaborated due to difficulties in modelling them algorithmically. Conceptually we consider that these operations must be bound to an expert practitioner. Insofar as the only available expert is a human, this simply means that the recipe program cannot be executed without an expert human. At such time as it is possible to model the human cook's expert knowledge, the prospect of completely automated execution of a recipe program is raised. It was not our intention, however, to reduce recipes to machine executability. Rather we sought to model recipes and to understand the programming constructs upon which they draw. Whether expert knowledge is embodied in humans or exotic software does not distort the overall model of the recipe programs we have created.

In the next subsection, we give examples of the four classes of operations, and in particular examples of elaboration on technical operations.

### 7.3.2. Examples of operations.

#### 7.3.2.1. Examples of primitive operations.

```
set_heat(BURNER1, high_heat);
turn_off_heat(BURNER1);

pour_into(BREAD_CRUMBS, KITCH_PLATE);
add_to(BUTTER_TO_SAUTE_BACKS, LARG_CASS);
```

#### 7.3.2.2. Examples of *let* operations.

```
let_drain(contents(COLANDER));
let_melt(contents(LARG_CASS));
let_dry(what_is_on(RACK));
let_boil(contents(LARG_CASS));
let_simmer(contents(LARG_CASS));
```

### 7.3.2.3. Examples of simple operations.

```
bring_to_boil(contents(LARG_CASS));

brush_with(BREAST, EGG_WHITE);
roll_into(BREAST, BREAD_CRUMBS);

preheat_oven(OVEN, 450);
stir(contents(LARG_CASS), WOOD_SPOON);
```

### 7.3.2.4. Examples of technical operations.

```
skim(contents(SAUCEPAN));
```

The "skim" operation can be elaborated as follow:

```
skim(LIQUID);
begin
        acqu_tool(SKIMMER);
        until (no foam left on top of LIQUID) do
        begin
                plunge_into(SKIMMER, LIQUID);
                orient_towards_top(SKIMMER, LIQUID);
                remove_going_towards_top(SKIMMER, LIQUID);
                clean_tool(SKIMMER);
        end;
        release_tool(SKIMMER);
end;            -- end_of skim --
```

```
deglaze(LARG_CASS, WOOD_SPOON, VERMOUTH);
```

The "deglaze" operation can be elaborated as follow:

```
deglaze(RECIPIENT, BURNER_NUM, LIQUID);
begin
        pour_into(LIQUID, RECIPIENT);
        put_over(RECIPIENT, BURNER[BURNER_NUM]);
        set_heat(BURNER[BURNER_NUM], high);

        acqu_tool(WOOD_SPOON);
        until (no solidified juice left inside_of(RECIPIENT)) do
        begin
                scrap_with(inside_of(RECIPIENT), WOOD_SPOON);
                stir(contents_of(RECIPIENT));
        end;
        release_tool(WOOD_SPOON);

        turn_off_heat(BURNER[BURNER_NUM]);
end;            -- end_of deglaze --
```

### 7.4. Characterization of the operations and accessing functions.

As in the case of more familiar programs, performance is a key issue in recipe programs. It is important to synchronize certain subactivities, and it is important to know the length of time it will take to execute both the entire recipe and its key subsections. Thus we found it important to create a mechanism for helping in evaluating the duration of given sequences of operations. We hypothesized that, to each of the recipe program's operations and accessing functions, there was associated a value which is an estimate of the maximum time required to perform the operation. This value can then be accessed by using the following construct:

value = duration(operation-name, unit);

the parameter "unit" is specified when the duration of the operation is a function of the magnitude of the ingredient on which the operation is performed. The value returned is the maximum time required to perform the operation on a unit.

eg.:

value = duration(chop, pound),

returns the maximum time required to chop a pound of ingredient of the type for which chop is an accessing primitive.

While this sort of annotation is necessary in order to support automatic parallelization and timing estimation, it is not sufficient. It is also necessary to know if a given operation is of active or stand alone type. An indicator is therefore associated with each operation and accessing function. Its value is retrieved using the following construct:

value = is_stand_alone(operation-name);

## 8. CONCLUSIONS.

From this study we believe we gained some worthwhile insights into how adequate contemporary programming languages are in supporting the expression of the sorts of processes which humans need to carry out. We continue to believe our initial premise that, from studying natural processes such as cooking, we can infer how humans are able to successfully communicate process information to each other. We continue to believe our original assumption that the natural paradigms used in communicating such processes must therefore be of interest and value to programmers who are attempting to specify all other kinds of processes. Thus we also continue to believe that the ways in which we attempt to express computational processes, ranging from text and numerical manipulation applications to the software development processes by which these application processes are created, ought to draw upon the same process description mechanisms and paradigms as those used in natural process descriptions such as recipes.

This study has convinced us that there is a great deal of similarity between the mechanisms used in conveying recipe processes and the mechanisms contained in contemporary computer languages for use by programmers in specifying computer applications, but that there are also some differences which are very much worth studying. The similarities should be encouraging to language designers and of particular interest to those who are concerned with establishing effective programming style. The differences should be of interest to language designers, software developers, and particularly to those engaged in the relatively new field of studying software development and maintenance processes.

We now summarize the similarities and differences in the following subsections. Before beginning, however, it seems important to place in the proper perspective the most striking apparent difference--namely the disparity between the lengths of the recipes we studied and the lengths of the corresponding recipe programs.

The reader has surely noticed that recipes are strikingly more terse than their corresponding recipe programs. On closer examination, it becomes clear that this is because recipes are simply higher level summaries of considerably more detailed and complete specifications, namely the recipe programs. It is these recipe programs that

correspond directly to computer program code, and are comparable to such code in length. Thus recipes should be considered to be the analogs of process design specifications. As such they are a compelling sort of design notation. We believe that they are well worth studying as instances of an effective natural design notation. On the other hand, they should not be confused with code, written in a natural coding language. Thus our conclusions are not based upon a comparison of recipes with programming languages, but rather upon comparison of the salient features of the derived recipe programs with the features of programming languages.

### 8.1. Similarities between recipe program features and programming language features.

When we started this study, we were not sure about which of the contemporary programming languages was the most adequate to serve as a basis for our recipe language. We had the feeling that we would need algorithmic constructs, but were ready to experiment with rule-based languages. However, it became very quickly obvious that recipes have a very strong algorithmic structure. It turned out that a great deal of the procedural aspects of recipes can be expressed using the basic control flow constructs (if-else, until-do, for, ...), and the procedural decomposition mechanism provided by a language such as Pascal.

The data declaration and manipulation features of a Pascal-like language also proved to be well suited to manipulating the ingredients of the recipes. Thus, ingredients are handled as variables declared to be of a known ingredient type.

However, we realized that a Pascal-like language was not powerful enough to translate all the constructs that appear in recipes. We therefore decided to explore the constructs provided by other languages (sometimes less well-known or commonly used than Pascal). First, we adopted the use of a type hierarchy structure (using type inheritance), to model the ingredient domain. Second, we introduced a "parbegin-parend" statement such as the one used in concurrent languages, and a critical section mechanism, to deal with parallelism. Third, we adopted the Ada exception handling mechanism. Finally, we introduced an assertion mechanism to specify error conditions, and checking points.

However, even with these less commonly-used programming languages constructs, we did not have enough features in our recipe language.

### 8.2. Differences between recipe program features and programming language features.

The first thing that obviously did not exist in programming languages, was a set of operations well suited to the cooking domain, or the ability to build such a set. "Cooking primitives" must be assumed, that are part of the virtual machine "on which our recipe programs run". We have already indicated that many of these operations seem best modelled as rules, not ideally expressed algorithmically.

In addition, the similarities between cooking tools (skillets, pans, burners, ...) and operating system resources, almost immediately struck us. However, resource management primitives are not usually a feature of programming languages (they are more commonly included in operating systems command languages). We therefore had to introduce resource management primitives - get-tool, acqu-tool, release-tool - in our recipe language.

Similarly, we felt the need for an interrupt (or similar) mechanism such as the ones usually provided at the level of operating systems. Thus, we introduced the timer construct.

Finally, there are a couple of recipe features that we did not quite manage to implement in our recipe language. First, we noticed some non-determinism in the order in which to execute some of the steps of a recipe. It is often the case that a small set of operations are independent of one another and do not have to be executed in a given order. Contemporary programming languages do not provide constructs that allow for such non-determinism. Second, we are not entirely satisfied with the classical type hierarchy structure we used, but we did not find another construct that would really suffice to model the real structure that cooks put upon ingredients.

### 8.3. Summary.

We found that a language capable of effectively supporting the expression of recipe programs must be quite broad and powerful. We believe that a language which is basically algorithmic can suffice, but only if it also

supports concurrency and real-time constructs. Further, however, it must incorporate data type specification that is more complex than what is available in conventional languages. It should probably also support rule-based function evaluation. While resisting the conclusion that such a powerful language is really necessary, we are also persuaded that humans have no apparent difficulty in understanding designs (ie. recipes) that draw simultaneously upon all of these mechanisms.

This conclusion stands in contrast to the recent trend towards simpler, more streamlined languages. Early results (eg. [BoJ66]) suggested that a very small set of control constructs could suffice to express any algorithmic program. This seemed to embolden language designers to create minimal languages with clean semantics. More recently language designers have investigated similarly clean designs to support non-algorithmic languages. Others have explored concurrency constructs and object-oriented languages stressing the importance of careful typing. Real-time programming languages have been less carefully explored.

We conclude that humans "design" and "program" effectively and naturally using a mixture of all of these types of programming mechanisms. It is not surprising that all are being explored, because there is clear evidence that all are natural. What seems most surprising is that humans are quite effective in mixing the paradigms. This suggests that it is necessary to create languages which cleanly mix paradigms if we are to create languages capable of cleanly and satisfactorally supporting the programming of human processes.

## 9. FUTURE WORK

It must be emphasized that the work reported here is based upon close study of a modest number of recipes. Thus the results are hardly conclusive. They seemed to us to be surprising and suggestive. Thus we felt it was reasonable to report them. We, nevertheless, feel that it is important to continue the work of creating recipe programs from recipes in order to gain assurance that these conclusions are more broadly supported and to begin to gain some quantitative results on the relative use and importance of various programming mechanisms. In particular we intend to experiment with ruled-based languages.

Such further work might indicate the desirability of recoding some recipe programs in languages which are not fundamentally algorithmic to test our hypothesis that such languages are indeed less effective than traditional algorithmic ones.

We also believe it will be useful to study what makes recipes such successful vehicles for expressing design. Their manifest terseness and obvious widespread popular appeal suggest that they are well worth emulating in software design applications. It seems that this is all based upon the sort of widely agreed upon base of expert knowledge which simply does not exist in such new disciplines as software engineering, but which might exist in older, more classical application programming domains. Perhaps design in these domains could profit from some of the conventions used in recipes.

Finally we are puzzled by the near total absence of requirements specifications from the cooking domain. Occasionally a cookbook might be addressed totally to solving a single well-specified problem (eg. "meals in less than one hour," or "cooking for bachelors"), or a recipe might be aimed at a particular problem ("getting rid of all those left-over egg whites"). By and large, however, recipes stand alone as designs of solutions without any specifications of the problems that they are to solve. Such higher level designs of how these recipes are to meet problems seem to reside entirely in the heads of people, along with the specifications of the problems themselves. This seems to fly in the face of "good software engineering," which preaches that designs can only be evaluated in the context of the requirements specifications which they are supposed to satisfy. Yet cooking recipes are routinely certified as "good" or "bad" apparently quite independently of requirements for them. Again, it may well be that this is entirely attributable to the widespread existence of an agreed-upon knowledge base (ie. what tastes good to humans). We believe that this is worth further investigation.

Ultimately we believe that the value of this research is that it attempts to harness what we can learn by observing how humans have innately sought to describe and carry out problem solving. We believe that this has the potential to teach us much about what can be most effective when formalized for software development processes such as design and programming. For this reason we believe that other natural problem solving activities such as instruction writing and beaureaucratic procedure specification should be studied as well. Making inferences from the base of natural phenomenology is an activity that is central to most well-established sciences. It has not been widely carried out in Computer Science. This is probably because it has not been clear what phenomena are worth observing and what inferences might be drawn. We hope that this paper makes some worthwhile suggestions in this

regard.

*ACKNOWLEDGEMENTS*

The authors wish to gratefully acknowledge the financial support of the US Department of Energy, through grant DE-FG02-840ER132283, the US Defense Advanced Research Projects Agency and the National Science Foundation, through grant CCR8705162, and the American Telephone and Telegraph Company.

The authors also wish to thank Stuart I. Feldman, Paul Smolensky, Gary J. Nutt, Janis Osterweil and Cedric J. Demeure for many stimulating conversations, and for their perceptive comments and suggestions, made in the course of this work. The students in Prof. Osterweil's Software Process Programming Seminar, Spring 1987, also provided continuing useful comments and observations.

*REFERENCES:*

[BoJ66] C. Bohm and G. Jacobpini, ''Flow-Diagrams, Turing Machines, and Languages with Only Two Formation Rules'', *Comm. of the ACM 9* (May 1966), 366-371.

[GoR83] A. Goldberg and D. Robson, *Smalltalk-80: The language and its Implementation*, Addison Wesley, 1983.

[LuH85] D. Luckham and F. Henke, ''An Overview of ANNA, a Specification Language for Ada'', *IEEE Software 2* (March 1985), 9-22.

[Mil86] C. Millau, *Dinning in France*, Distributed by Workman Publishing, New York, New York, 1986.

[Ost87] L. Osterweil, ''Software Processes are Software Too'', *Proc. 9th International Conf. on Software Eng. IEEE Cat# 87CH2432-3*, Monterey, California, March 1987.

[RoR73] I. S. Rombauer and M. Rombauer-Becker, *Joy of cooking*, New American Library - A Plume Book, New York, New York, November 1973.

# APPENDICES

## Appendix 1

Recipe "SUPREMES DE VOLAILLE AU FOIE GRAS"

(from the American edition of "Dinning in France" by Christian Millau)
original recipe followed by its recipe program translation.

## Appendix 2

Recipe "CHEESE SOUFFLE"

(adapted from "Joy of cooking" by Irma S. Rombauer and Marion Rombauer Becker)
original recipe followed by its recipe program translation.

## Appendix 3

An example of ingredient type hierarchy

APPENDIX 1
recipe : SUPREMES DE VOLAILLE AU FOIE GRAS

*Claude Deligne*

## *SUPREMES DE VOLAILLE AU FOIE GRAS*

## CHICKEN BREASTS STUFFED WITH FOIE GRAS

| | |
|---|---|
| 2 | tablespoons unsalted butter |
| 5 | pounds chicken backs, chopped |
| 1 | large carrot, finely chopped |
| 7 | shallots, finely chopped |
| 1 | small onion, finely chopped |
| ½ | small celery root (celeriac), finely chopped |
| ¼ | pound mushrooms, finely chopped |
| 1 | clove garlic, finely chopped |
| ½ | cup dry vermouth |
| 1 | quart rich chicken stock |
| 1 | bouquet garni |
| 2 | large chicken breasts, halved, boned, and skinned |
| 5 | ounces duck foie gras, cut into 8 slices |
| | Salt and freshly ground pepper |
| 1 | egg white |
| 1¼ | cups fresh breadcrumbs (from 5 slices white bread, crusts removed) |
| ½ | cup crème fraiche |
| ¾ | cup unsalted butter, clarified |

Preheat oven to 450 degrees.

In a large casserole, melt butter. Add chicken backs and cook over medium heat, stirring for about 10 minutes or until very lightly browned. Add carrot, shallots, onion, celery root, mushrooms, and garlic; stir well, and place in oven. Bake, stirring from time to time, for about 30 minutes or until carcasses and vegetables are nicely browned. Pour contents of casserole into a colander set over a bowl to drain.

Set casserole over medium-high heat and deglaze with vermouth; boil, scraping bottom of casserole with a wooden spoon, for about 3 minutes or until liquid is syrupy. Add stock and bouquet garni and stir well. Return drained carcasses and vegetables to casserole and bring to a boil over high heat. Reduce heat to medium and simmer, skimming as necessary, for 45 minutes.

Meanwhile, remove any large tendons from chicken breasts. Detach fillet from underside of each breast and set aside. With a sharp knife, cut a lengthwise slit in each breast to form a pocket. Place 2 slices of foie gras in each pocket and press to seal shut; replace fillets to completely seal openings.

Bring water to a boil in a steamer. Season breasts with salt and pepper and place them in steamer, smooth side down. Cover and steam for 6 minutes; remove and pat dry. When breasts have cooled slightly, brush them with egg white and roll in breadcrumbs to coat well. Set aside on a rack to dry for about 15 minutes.

Strain rich stock through a fine-mesh strainer set over a saucepan; press hard on bones and vegetables to extract all liquid. Add crème fraiche and bring to a boil over medium-high heat. Simmer sauce, skimming as necessary, for about 15 minutes or until reduced by half. (The sauce should be of a light-coating consistency.)

Just before serving, heat clarified butter in a medium skillet. Add chicken breasts, in batches if necessary, and fry for 2 to 3 minutes per side or until nicely browned. Drain on paper towels. Reheat sauce, if necessary. Spoon some of the sauce onto 4 plates and place a chicken breast half on each. Pass remaining sauce in sauceboat.

Makes 4 servings

*Ed. note: The original version of this recipe calls for Bresse chickens. If you can find large, free-range chickens, by all means use them. Remove the breasts for this dish, reserve the legs for another use, and use the carcasses for the sauce.*

**Translation of the recipe : SUPREMES DE VOLAILLE AU FOIE GRAS**

**Program SUPREMES_AU_FOIE_GRAS(SERVING, 4);**

Declare INGREDIENTS

```
 ING BUTTER_TO_SAUTE_BACKS is T_BUTTER amount_is 2 unit_is TABLESPOON
        attributes_are(salt: TRUE) access_is();
 ING CHICKEN_BACKS_FOR_STOCK is T_CHICKEN_BACK amount_is 2 unit_is POUND
        attributes_are() access_is(chop);
 ING CARROT_FOR_STOCK is T_CARROT amount_is 1 unit_is UNIT
        attributes_are(size: LARGE) access_is(peel, wash, finely_chop);
 ING SHALLOT_FOR_STOCK is T_SHALLOT amount_is 7 unit_is UNIT
        attributes_are() access_is(peel, wash, finely_chop);
 ING CELERY_FOR_STOCK is T_CELERY_ROOT amount_is 0.5 unit_is UNIT
        attributes_are() access_is(peel, wash, finely_chop);
 ING MUSH_FOR_STOCK is T_MUSHROOM amount_is 0.25 unit_is POUND
        attributes_are() access_is(brush_under_water, finely_chop);
 ING GARLIC_FOR_STOCK is T_GARLIC amount_is 1 unit_is CLOVE
        attributes_are() access_is(peel, finely_chop);
 ING VERMOUTH_TO_DEGLAZE is T_VERMOUTH amount_is 0.5 unit_is CUP
        attributes_are(dry : TRUE) access_is();
 ING STOCK_FOR_RICH_STOCK is T_STOCK amount_is 1 unit_is QUART
        attributes_are(flavor : chicken) access_is();
 ING BQT_GARNI_FOR_STOCK is T_BOUQUET_GARNI amount_is 1 unit_is UNIT
        attributes_are(storage : DRIED) access_is();
 ING BREAST_TO_STUFF[2] = for i = 1 to 2
 {
        is T_CHICKEN_BREAST amount_is 1 unit_is UNIT
        attributes_are() access_is(halve, bone, skin);
 }
 ING STUFFING_FOIE_GRAS is T_DUCK_FOIE_GRAS amount_is 5 unit_is OUNCE
        attribute_are() access_is(slice_into(8));
 ING SALT is T_SALT amount_is ; unit_is ;
        attributes_are() access_is();
 ING PEPPER is T_BLACK_PEPPER amount_is ; unit_is ;
        attributes_are() access_is(grind);
 ING EGG_WHITE_FOR_COATING is T_EGG_WHITE amount_is 1 unit_is UNIT
        attributes_are() access_is();
 ING CRUMB_FOR_COATING is T_BREAD amount_is 5 unit_is SLICE
        attributes_are(fresh : FALSE; type : white)
        access_is(reduce_to_crumbs);
 ING CREME_FOR_SAUCE is T_CREME_FRAICHE amount_is 0.5 unit_is CUP
        attributes_are() access_is();
 ING BUTTER_TO_FRY_BREASTS is T_BUTTER amount_is 0.75 unit_is CUP
        attributes_are(salt : FALSE) access_is(clarify);
 ING WATER_TO_STEAMER_BREASTS is T_WATER amount_is ; unit_is ;
        attributes_are() access_is();
end;
```

Declare TOOLS:

```
 TOOL OVEN = get_tool(oven);
 TOOL LARG_CASS = get_tool(large casserole);
 TOOL BURNER[2] =  for i=1 to 2 get_tool(burner);
 TOOL WOOD_SPOON = get_tool(wooden spoon);
 TOOL COLANDER = get_tool(colander);
 TOOL BOWL = get_tool(bowl);
 TOOL SKIMMER = get_tool(skimmer);
 TOOL SHARP_KNIFE = get_tool(sharp knife);
 TOOL STEAMER = get_tool(steamer);
 TOOL BRUSH = get_tool(brush);
 TOOL KITCH_PLATE = get_tool( kitchen plate);
```

```
  TOOL RACK = get_tool(rack);
  TOOL STRAINER = get_tool(fine mesh strainer);
  TOOL SAUCEPAN = get_tool(saucepan);
  TOOL SKILLET = get_tool(medium skillet);
  TOOL TABLE_PLATE[4] = for i=1 to 4 get_tool(table plate);
  TOOL SAUCEBOAT = get_tool(sauceboat);
  TOOL SPOON = get_tool(spoon);
  TOOL PAPER_TOWEL ;= get_tool(paper towels);
end;

Declare variables

  integer i, num, num2, num3, num4;
  ING HALF_BREAST[4], foie_gras_slices;
  boolean not_done;
end;

Declare required_skill

  extern    procedure DEGLAZE();
  extern    procedure SKIM();
  extern    predicate has_light_coating_consistency;
  extern    predicate is_syrupy;
  extern    predicate is_reduced_by_half;
  extern    predicate is_nicely_brown;
end;

Declare exceptions
  predefined-exception TOOL_NOT_AVAIL;
end;

-- main part of the recipe --
begin

        parbegin

                begin                   -- begin task 1 --
                        FIX_RICH_STOCK;
                        FIX_SAUCE;
                end;                    --end task 1 --

                begin                   -- begin task 2 --
                        STUFF_BREASTS;
                        PRE_COOK_BREASTS;
                        COAT_BREASTS;
                end;                    -- end task 2 --

        parend;

        begin_critical_section
        -- Once the breasts are fried they must be served without waiting --

                FRY_BREASTS;
                SERVE;

        end_critical_section

end;

-- Procedure FIX_RICH_STOCK and its subprocedures --

Procedure FIX_RICH_STOCK;
begin
        FIX_STOCK_INGREDIENTS;
        BAKE_STOCK_INGREDIENTS;
```

```
        FIX_STOCK;
        SIMMER_STOCK;
end;                -- end_of FIX_RICH_STOCK --


Procedure FIX_STOCK_INGREDIENTS;
begin

        Declare timers
                active_timer        FIX_STOCK_INGREDIENTS_1;
        end;

/* preheat the oven to 450 degrees
*/
        acqu_tool(OVEN);
        preheat_oven(OVEN, 450);

/* in a large casserole, melt butter
*/
        acqu_tool(LARG_CASS);
        num = acqu_tool(BURNER);
        set_heat(BURNER[num], medium);
        put_over(LARG_CASS, BURNER[num]);
        add_to(BUTTER_TO_SAUTE_BACKS, LARG_CASS);
        let_melt(contents_of(LARG_CASS));

/* Add chicken backs and cook over medium heat stirring for about 10
  minutes or until very lightly browned. Add carrot, shallots, onion,
  celery root, mushrooms, and garlic;
*/
        add_to(chop(CHICKEN_BACKS_FOR_STOCK), LARG_CASS);

        acqu_tool(WOOD_SPOON);

        set_active_timer(FIX_STOCK_INGREDIENTS_1, 10, MINUTE);
        not_done = TRUE;
        while (not_done) do
        -- loop until the contents of the casserole is very lightly browned --
        begin
                until(ring_active_timer(FIX_STOCK_INGREDIENTS_1)) do
                begin
                        stir(contents_of(LARG_CASS), WOOD_SPOON);
                        if (contents_of(LARG_CASS) is_very_lightly_browned)
                        begin
                                cancel_active_timer(FIX_STOCK_INGREDIENTS_1);
                                not_done = FALSE;
                        end
                end;
                if (not(contents_of(LARG_CASS) is_very_lightly_browned))
                        set_active_timer(FIX_STOCK_INGREDIENTS_1, 5, MINUTE);
                else not_done = FALSE;
        end;



        release_tool(WOOD_SPOON);

        add_to(finely_chop(wash(peel((CARROT_FOR_STOCK), LARG_CASS))));
        add_to(finely_chop(wash(peel((SHALLOT_FOR_STOCK), LARG_CASS))));
        add_to(finely_chop(wash(peel((ONION), LARG_CASS))));
        add_to(finely_chop(wash(peel((CELERY_FOR_STOCK), LARG_CASS))));
        add_to(finely_chop(wash(peel((GARLIC_FOR_STOCK), LARG_CASS))));
        add_to(finely_chop(brush_under(water((MUSH_FOR_STOCK), LARG_CASS))));
/* stir well
*/
```

```
        acqu_tool(WOOD_SPOON);
        stir(contents_of(LARG_CASS), WOOD_SPOON)
        release_tool(WOOD_SPOON);

        turn_off_heat(BURNER[num]);
        remove_from(LARG_CASS, BURNER[num]);
        release_tool(BURNER[num]);
end;                -- FIX_STOCK_INGREDIENTS --


Procedure BAKE_STOCK_INGREDIENTS;
begin

        Declare timers
                std_alone_timer     BAKE_STOCK_INGREDIENTS_1;
                period_timer        BAKE_STOCK_INGREDIENTS_2;
        end;

/* and place in oven; Bake, stirring from time to time, for 30 minutes or
  until carcasses and vegetables are nicely browned.
*/
        put_into(LARG_CASS, OVEN);
        acqu_tool(WOOD_SPOON);

        set_std_alone_timer(BAKE_STOCK_INGREDIENTS_1, 30, MINUTE);
        set_period_timer(BAKE_STOCK_INGREDIENTS_2, 5, MINUTE);
        not_done = TRUE;
        while (not_done) do
        -- loop until the contents of the casserole is nicely browned --
        begin
                until(ring_std_alone_timer(BAKE_STOCK_INGREDIENTS_1)) do
                begin
                        when (ring-period-timer(BAKE_STOCK_INGREDIENTS_2)) do
                        begin
                                stir(contents-of(LARG_CASS));
                                if(contents-of(LARG_CASS) is-nicely-browned)
                                begin
                                        cancel-std-alone-timer(BAKE_STOCK_INGREDIENTS_1);
                                        cancel-period-timer(BAKE_STOCK_INGREDIENTS_2);
                                        not_done = FALSE;
                                end
                        end
                end;
                if (not(contents_of(LARG_CASS) is_nicely_browned))
                        set_std_alone_timer(BAKE_STOCK_INGREDIENTS_1, 5, MINUTE);
                else
                begin
                        not_done = FALSE;
                        cancel-period-timer(BAKE_STOCK_INGREDIENTS_2);
                end;
        end;

        release_tool(WOOD_SPOON);

        turn_off_heat(OVEN);
        remove_from(LARG_CASS, OVEN);
        release_tool(OVEN);

/* Pour contents of casserole into a coolander set over a bowl to drain
*/
        acqu_tool(COLANDER);
        acqu_tool(BOWL);
        set_over(COLANDER, BOWL);
        pour_into(contents_of(LARG_CASS), COLANDER)
        let_drain(contents_of(COLANDER));
```

```
end;                    -- end_of BAKE_STOCK_INGREDIENTS --

Procedure FIX_STOCK;
begin

        Declare timers
                std_alone_timer         FIX_STOCK_1;
        end;

/* Set casserole over medium-high heat and deglaze with vermouth;
  boil, scraping bottom of casserole with a wooden spoon, for
  about 3 minutes or until liquid is syrupy.
*/
        acqu_tool(BURNER[num]);
        set_heat(BURNER[num], medium_high);
        put_over(LARG_CASS, BURNER[num]);

        deglaze(LARG_CASS, num, VERMOUTH_TO_DEGLAZE);

        bring_to_boil(contents_of(LARG_CASS));

        set_std_alone_timer(FIX_STOCK_1, 3, MINUTE);
        not_done = TRUE;
        while (not_done) do
        -- loop until the liquid is sirupy --
        begin
                until(ring_std_alone_timer(FIX_STOCK_1)) do
                begin
                        let_boil(contents_of(LARG_CASS))
                        if (contents_of(LARG_CASS) is syrupy)
                        begin
                                cancel_std_alone_timer(FIX_STOCK_1);
                                not_done = FALSE;
                        end
                end;
                if (not(contents_of(LARG_CASS) is syrupy))
                        set_std_alone_timer(FIX_STOCK_1, 1, MINUTE);
                else not_done = FALSE;
        end;

/* Add stock and bouquet garni and stir well. Return drained carcasses
  and vegetables to casserole and
*/
        add_to(STOCK_FOR_RICH_STOCK, LARG_CASS);
        add_to(BQT_GARNI_FOR_STOCK, LARG_CASS);

        acqu_tool(WOOD_SPOON);
        stir(contents_of(LARG_CASS), WOOD_SPOON)
        release_tool(WOOD_SPOON);

        add_to(contents_of(COLANDER), LARG_CASS);
        release_tool(COLANDER);
        release_tool(BOWL);

end;                    -- end_of FIX_STOCK --

Procedure SIMMER_STOCK;
begin

        Declare timers
                period_timer            SIMMER_STOCK_1;
                std_alone_timer         SIMMER_STOCK_2;
        end;
/* bring to a boil over high heat. Reduce heat to medium and
*/
```

```
        set_heat(BURNER[num], high_heat);
        bring_to_boil(contents_of(LARG_CASS));
        set_heat(BURNER[num], medium_heat);

/* simmer, skimming as necessary for 45 minutes.
*/

        set_period_timer(SIMMER_STOCK_1, 10, MINUTE);
        set_std_alone_timer(SIMMER_STOCK_2, 45, MINUTE);
        not_done = TRUE;
        until(ring_std_alone_timer(SIMMER_STOCK_2)) do
        begin
                let_simmer(contents_of(LARG_CASS));
                when (ring-period-timer(SIMMER_STOCK_1)) do
                begin
                        skim(contents_of(LARG_CASS));
                end;
        end;

/* Strain rich stock through a fine-mesh strainer set over a saucepan;
  press hard on bones and vegetables to extract all liquid.
*/
        acqu_tool(STAINER);
        acqu_tool(SAUCEPAN);
        set_over(STRAINER, SAUCEPAN);
        pour_into(contents_of(LARG_CASS), STRAINER);
        press_to_extract_liquid(contents_of(STRAINER));
        release_tool(LARG_CASS);
        release_tool(STRAINER);

end;            -- end_of SIMMER_STOCK --

-- procedure FIX_SAUCE --

Procedure FIX_SAUCE;
begin

        Declare exceptions
                exception  ADJUST_SAUCE_CONSISTENCY;
        end;

        Declare timers
                period_timer        FIX_SAUCE_1;
                std_alone_timer     FIX_SAUCE_2;
        end;

/* Add creme fraiche and bring to a boil over medium high heat. Simmer
  sauce, skimming as necessary, for about 15 minutes or until reduced
  by half (the sauce should be a light coating consistency).
*/
        add_to(CREME_FOR_SAUCE, SAUCEPAN);
        set_heat(BURNER[num], medium_high);
        put_over(SAUCEPAN, BURNER[num]);

        bring_to_boil(contents_of(SAUCEPAN));

        set_period_timer(FIX_SAUCE_1, 5, MINUTE);
        set_std_alone_timer(FIX_SAUCE_2, 15, MINUTE);
        not_done = TRUE;
        while (not_done) do
        -- loop until the sauce is reduced by half --
        begin
                until(ring_std_alone_timer(FIX_SAUCE_2)) do
                begin
                        let_simmer(contents_of(SAUCEPAN));
```

```
                        when (ring-period-timer(FIX_SAUCE_1)) do
                        begin
                                skim(contents_of(SAUCEPAN));
                                if(contents-of(SAUCEPAN) is_reduced_by_half)
                                begin
                                        cancel-std-alone-timer(FIX_SAUCE_2);
                                        cancel-period-timer(FIX_SAUCE_1);
                                        not_done = FALSE;
                                end
                        end
                end;
                if (not(contents-of(SAUCEPAN) is_reduced_by_half))
                        set_std_alone_timer(FIX_SAUCE_2, 3, MINUTE);
                else
                begin
                        not_done = FALSE;
                        cancel-period-timer(FIX_SAUCE_1);
                end;
        end;

        assert (not(contents_of(SAUCEPAN) has light coating consistency))
        else raise ADJUST_SAUCE_CONSISTENCY(SAUCEPAN);

        turn_off_heat(BURNER[num]);
        remove_from(SAUCEPAN, BURNER[num]);
        release_tool(BURNER[num]);

        exception ADJUST_SAUCE_CONSISTENCY(recipient)
        TOOL recipient; -- recipient containing the sauce --
        begin
                ING CORNSTARCH_TO_THICKEN_SAUCE is T_CORNSTARCH amount_is ; unit_is ;
        attributes_are() access_is();
                ING WATER_TO_LIQUEFY_SAUCE is T_WATER amount_is ; unit_is ;
        attributes_are() access_is();

                if (contents_of(recipient) too liquid)
                begin
                        add_to(CORNSTARCH_TO_THICKEN_SAUCE, recipient);
                        until (contents_of(recipient) has light coating consistency) do
                        begin
                                let_simmer(contents_of(recipient));
                        end;
                end;
                else -- contents_of(recipient) too sirupy --
                        add_to(WATER_TO_LIQUEFY_SAUCE, recipient);
                        until (contents_of(recipient) has light coating consistency) do
                        begin
                                let_simmer(contents_of(recipient));
                        end;
        end;

end;    -- end of FIX_SAUCE --
```

**-- Procedure STUFF_BREASTS --**

```
Procedure STUFF_BREASTS;
begin
/* Meanwhile, remove any large tendons from chicken breasts. Detach fillet
  from underside of each breast and set aside. With a sharp knife, cut a
  lengthwise slit in each breast to form a pocket. Place two slices of
  foie gras in each pocket, and press to seal shut; replace fillets to
  completely seal openings.
  [...] Season breasts with salt and pepper
*/
        acqu_tool(SHARP_KNIFE);
```

```
        for i=1 to 2
        begin
                remove_from(large_tendons, halve(bone(skin(BREAST_TO_STUFF[i]))));
        end;
        release_tool(SHARP_KNIFE);

        for i = 1 to 2
        begin
                detach_from(fillet_of(BREAST_TO_STUFF[i]), BREAST_TO_STUFF[i]);
        end;

        acqu_tool(SHARP_KNIFE);
        for i = 1 to 2
        begin
                slit_lengthwise(BREAST_TO_STUFF[i]);
        end
        release_tool(SHARP_KNIFE);

        for i = 1 to 2
        begin
                foie_gras_slices = get_part_ing(slice_into(8)(STUFFING_FOIE_GRAS), slice, 2);
                put_into(foie_gras_slices, BREAST_TO_STUFF[i]);
                press_to_seal(BREAST_TO_STUFF[i]);
                put_on(fillet_of(BREAST_TO_STUFF[i]), seal_of(BREAST_TO_STUFF[i]));
                assert(BREAST_TO_STUFF[i] is sealed);

                season_with(BREAST_TO_STUFF[i], SALT);
                season_with(BREAST_TO_STUFF[i], PEPPER);
        end;
end;            -- end of STUFF_BREASTS --
```

**-- procedure PRE_COOK_BREASTS --**

```
Procedure PRE_COOK_BREASTS;
begin

        Declare timers
                std_alone_timer         PRE_COOK_BREASTS;
        end;

/* Bring water to a boil in a steamer.
*/
        acqu_tool(STEAMER);
        add_to(WATER_TO_STEAMER_BREASTS, STEAMER);

        num2 = acqu_tool(BURNER);
        set_heat(BURNER[num2], high_heat);
        bring_to_boil(contents_of(STEAMER));

/* [...] and place them (the breasts)
  in steamer, smooth side down. Cover and steam for 6 minutes;
*/
        for i = 1 to 2
        begin
                place_into(BREAST_TO_STUFF[i], STEAMER);
                if(not(smooth_side_of(BREAST_TO_STUFF[i]) is down))
                        put_on_other_side(BREAST_TO_STUFF[i]);
        end;

        put_on(cover_of(STEAMER), STEAMER);

        set_std_alone_timer(PRE_COOK_BREASTS, 6, MINUTE);
        until(ring_std_alone_timer(PRE_COOK_BREASTS)) do
        begin
                let_simmer(contents_of(STEAMER))
```

```
        end;

/* remove and pat dry.
  When breasts have cooled slightly
*/
        turn_off_heat(BURNER[num2]);
        remove_from(STEAMER, BURNER[num2]);
        release_tool(BURNER[num2]);

        for i = 1 to 2
        begin
                remove_from(BREAST_TO_STUFF[i], STEAMER);
        end;

        release_tool(STEAMER);

        for i = 1 to 2
        begin
                pat_dry(BREAST_TO_STUFF[i]);
        end;

        let_cool(BREAST_TO_STUFF);
end;            -- end_of PRE_COOK_BREASTS --
```

**-- procedure COAT_BREASTS and its subprocedures --**

```
Procedure COAT_BREASTS;
begin
        DO_COATING;
        LET_DRY_COATING;
end;            -- end_of COAT_BREASTS --

Procedure DO_COATING;
begin
/* brush them (the breasts) with egg white, and roll in bread crumbs to coat
  well. Set aside on a rack,
*/
        assert(BREAST_TO_STUFF are cool);

        acqu_tool(KITCH_PLATE);
        acqu_tool(RACK);
        acqu_tool(BRUSH);
        pour_into(reduce_to_crumb(CRUMB_FOR_COATING), KITCH_PLATE);

        for i = 1 to 2
        begin
                brush_with(BREAST_TO_STUFF[i], EGG_WHITE_FOR_COATING);
                roll_into(BREAST_TO_STUFF[i], CRUMB_FOR_COATING);

                assert(BREAST_TO_STUFF[i] is well coated);

                set_on(BREAST_TO_STUFF[i], RACK);
        end;

        release_tool(KITCH_PLATE);
        release_tool(BRUSH);

end;            -- end_of DO_COATING --

Procedure LET_DRY_COATING
begin

        Declare timers
                std_alone_timer     LET_DRY_COATING;
        end;
```

```
/* to dry for about 15 minutes.
*/
        set_std_alone_timer(LET_DRY_COATING, 15, MINUTE);
        until(ring_std_alone_timer(LET_DRY_COATING)) do
        begin
                let_dry(what_is_on(RACK));
        end;

end;                    -- end of LET_DRY_COATING --
```

**-- Procedure FRY_BREASTS --**

```
Procedure FRY_BREASTS;
begin

        Declare timers
                std_alone_timer         FRY_BREASTS_1, FRY_BREASTS_2;
        end;

/* Just before serving, heat clarified butter in a medium skillet. Add
  chicken breasts, in batches if necessary, and fry for 2 or 3 minutes
  per side, or until nicely browned. Drain on paper towels.
*/
        num3 = acqu_tool(BURNER);
        set_heat(BURNER[num3], medium);
        put_over(SKILLET, BURNER[3]);
        add_to(clarify(BUTTER_TO_FRY_BREASTS), SKILLET);

        until (contents_of(SKILLET) is frying) do
        begin
                let_heat(contents_of(SKILLET));
        end;

        add_to(BREAST_TO_STUFF, SKILLET);
        release_tool(RACK);

        set_std_alone_timer(FRY_BREASTS_1, 3, MINUTE);
        not_done = TRUE;
        while (not_done) do
        -- loop until the contents of the skillet is nicely browned --
        begin
                until(ring_std_alone_timer(FRY_BREASTS_1)) do
                begin
                        let_fry(contents_of(SKILLET));
                        if(contents-of(SKILLET) is-nicely-browned)
                                begin
                                        cancel-std-alone-timer(FRY_BREASTS_1);
                                                                not_done = FALSE;
                                end
                end;
                if (not(contents_of(SKILLET) is_nicely_browned))
                        set_std_alone_timer(FRY_BREASTS_1, 1, MINUTE);
                else not_done = FALSE;
        end;


        for i = 1 to 2
        begin
                turn_over(BREAST_TO_STUFF[i])
        end;

        set_std_alone_timer(FRY_BREASTS_2, 3, MINUTE);
        not_done = TRUE;
        while (not_done) do
        -- loop until the contents of the skillet is nicely browned --
```

```
        begin
                until(ring_std_alone_timer(FRY_BREASTS_2)) do
                begin
                        let_fry(contents_of(SKILLET));
                        if(contents-of(SKILLET) is-nicely-browned)
                                begin
                                        cancel-std-alone-timer(FRY_BREASTS_2);
                                                                not_done = FALSE;
                                end
                end;
                if (not(contents_of(SKILLET) is_nicely_browned))
                        set_std_alone_timer(FRY_BREASTS_2, 1, MINUTE);
                else not_done = FALSE;
        end;

        turn_off_heat(BURNER[num3]);
        remove_from(SKILLET, BURNER[num3]);
        release_tool(BURNER[num3]);

        for i = 1 to 2
        begin
                remove_from(BREAST_TO_STUFF[i], SKILLET);
                put_on(BREAST_TO_STUFF[i], PAPER_TOWEL);
                turn_over(BREAST_TO_STUFF[i], PAPER_TOWEL);
                assert(BREAST_TO_STUFF[i] is drained);
        end;
end;            -- end of FRY_BREASTS --
```

**-- Procedure SERVE and its subprocedures --**

```
Procedure SERVE;
begin
/* Reheat sauce, if necessary.
*/
        if (contents_of(SAUCEPAN) not hot)
        begin
                REHEAT_SAUCE;
        end;

        SERVE_PLATES;
end;            -- end_of SERVE --

Procedure REHEAT_SAUCE;
begin
        num4 = acqu_tool(BURNER);
        set_heat(BURNER[num4], medium);
        put_over(SAUCEPAN, BURNER[4]);

        until (contents_of(SAUCEPAN) hot) do
        begin
                let_heat(contents_of(SAUCEPAN));
        end;

        turn_off_heat(BURNER[num4]);
        remove_from(SAUCEPAN, BURNER[num4]);
        release_tool(BURNER[num4]);
end;            -- end_of REHEAT_SAUCE --

Procedure SERVE_PLATES
begin
/* Spoon some of the sauce onto 4 plates and place a chicken breast half on
  each. Pass remaining          sauce in sauceboat.
*/
        for i= 1 to 2
        begin
```

```
                cut_into(BREAST_TO_STUFF[i], 2, HALF_BREAST[(i-1) * 2 +1], HALF_BREAST[(i-1) * 2 + 2]);
        end;

        for i=1 to 4
        begin
                acqu_tool(TABLE_PLATE[i]);
                pour_into(get_part_ing(contents_of(SAUCEPAN), tablespoon, 3), TABLE_PLATE[i]);
                put_into(HALF_BREAST[i], TABLE_PLATE[i]);
        end;

        acqu_tool(SAUCEBOAT);
        pour_into(contents_of(SAUCEPAN), SAUCEBOAT);
        release_tool(SAUCEPAN);

end;                    -- end of SERVE_PLATES --

-- exception handlers global to the program --
exception TOOL_NOT_AVAIL(needed_tool)
TOOL needed_tool;
begin
        if (cannot find replacement for needed_tool)
                wait-for-tool
        else get_replacement-tool(needed_tool);
        resume;
end;

end; -- end_of Program SUPREMES_AU_FOIE_GRAS --
```

## APPENDIX 2

### recipe : CHEESE SOUFFLE

CHEESE SOUFFLE

**4 Servings**

Preheat oven to 350 degrees.
Prepare :
**1 cup of bechamel sauce**
Bring to a boil. Remove from heat half a minute. Add, stirring well:
**5 tablespoons grated Parmesan cheese**
**2 tablespooons grated Gruyere cheese**
**3 beaten egg yolks**
Beat until stiff, but not dry:
4 egg whites
Fold into the cheese mixture. Pour into one 7-inch souffle baker.
Bake for about 25 to 30 minutes or until set.

BECHAMEL SAUCE

**1 cup**

Melt over low heat:
**3 tablespoons butter**
Add and blend over low heat for three minutes:
**3 tablespoons flour**
Stir in slowly:
**1 cup of milk**
Cook and stir the sauce with a wire whisk until thickened and smooth.
Season with salt and pepper, to taste.
Cook over low heat until for 20 minutes, stirring from time to time.

**Translation of the recipe : CHEESE SOUFFLE**

```
Program CHEESE_SOUFFLE(SERVING, 4);

Extern program BECHAMEL_SAUCE(CUP, 1);

Declare INGREDIENTS

 ING PARMESAN is T_PARMESAN amount_is 5 unit_is TABLESPOON
          attributes_are() access_is(grate);
 ING GRUYERE is T_GRUYERE amount_is 2 unit_is TABLESPOON
          attributes_are() access_is(grate);
 ING EGG_YOLK is T_EGG_YOLK amount_is 3 unit_is UNIT
          attributes_are() access_is(beat);
 ING EGG_WHITE is T_EGG_WHITE amount_is 4 unit_is UNIT
          attributes_are() access_is();
end;

Declare TOOLS

 TOOL OVEN = get_tool(oven);
 TOOL CASS_FOR_MIXTURE = get_tool(casserole);
 TOOL WOOD_SPOON = get_tool(wooden spoon);
 TOOL ELEC_WHIPPER = get_tool(electric whipper);
 TOOL BOWL_FOR_WHITES = get_tool(bowl);
 TOOL BURNER = get_tool(burner);
 TOOL BAKER = get_tool(7_inches souffle baker);
end;

Declare variables

 boolean  not_done;
end;

Declare required_skill

 extern   procedure whip_egg_whites;
 extern   procedure fold_egg_whites;
 extern   predicate  souffle_is_set;
 extern   predicate  is-lumpy;
end;

Declare exceptions

 predefined-exception TOOL_NOT_AVAIL;
end;

-- main part of the recipe --
begin
          FIX_BECHAMEL_SAUCE;

          FIX_CHEESE_MIXTURE;

          BAKE_SOUFFLE;
end;

Procedure FIX_BECHAMEL_SAUCE;
begin
/* Prepare 1 cup of bechamel sauce */
          acqu_tool(CASS_FOR_MIXTURE);
          add_to(FIX(BECHAMEL_SAUCE(CUP, 1)), CASS_FOR_MIXTURE);
end;                    -- end_of FIX_BECHAMEL_SAUCE --
```

```
Procedure FIX_CHEESE_MIXTURE;
begin

 Declare timers
  std_alone_timer    FIX_CHEESE_MIXTURE_1;
 end;

/* preheat oven to 350 degrees */
        acqu_tool(OVEN);
        preheat_oven(OVEN, 350);
/* Bring to a boil. Remove from heat half a minute */
        acqu_tool(BURNER);
        set_heat(BURNER, high_heat);
        put_over(CASS_FOR_MIXTURE, BURNER);
        bring_to_boil(contents_of(CASS_FOR_MIXTURE));
        turn_off_heat(BURNER);
        remove_from(CASS_FOR_MIXTURE, BURNER);
        release_tool(BURNER);

        set_std_alone_timer(FIX_CHEESE_MIXTURE_1, 0.5, MINUTE);
        until(ring_std_alone_timer(FIX_CHEESE_MIXTURE_1)) do
        begin
                let_stand(CASS_FOR_MIXTURE);
        end;

/* Add, stirring well: 5 tablespoons grated Parmesan cheese, 2 tablespooons
  grated Gruyere cheese, 3 beaten egg yolks.
*/
        acqu_tool(WOOD_SPOON);
        mix_in(PARMESAN, CASS_FOR_MIXTURE, WOOD_SPOON);
        mix_in(GRUYERE, CASS_FOR_MIXTURE, WOOD_SPOON);
        mix_in(beat(EGG_YOLK), CASS_FOR_MIXTURE, WOOD_SPOON);
        release_tool(WOOD_SPOON);

/* Beat until stiff, but not dry: 4 egg whites;
*/
        acqu_tool(BOWL_FOR_WHITES);
        put_into(EGG_WHITE, BOWL_FOR_WHITES);
        acqu_tool(ELEC_WHIPPER);
        whip_white(contents_of(BOWL_FOR_WHITES), stiff_but_not_dry, ELEC_WHIPPER);
        release_tool(ELEC_WHIPPER);

/* Fold into the cheese mixture. Pour into one 7-inch souffle baker.
*/
        acqu_tool(WOOD_SPOON);
        fold_white_egg(contents_of(BOWL_FOR_WHITES), contents_of(CASS_FOR_MIXTURE), WOOD_SPOON);
        release_tool(BOWL_FOR_WHITES);
        release_tool(WOOD_SPOON);

        pour_into(contents(CASS_FOR_MIXTURE), BAKER);
        release_tool(CASS_FOR_MIXTURE);


end;  -- end_of FIX_CHEESE_MIXTURE --

Procedure BAKE_SOUFFLE;
begin

 Declare timers
  std_alone_timer    BAKE_SOUFFLE_1;
  period_timer       BAKE_SOUFFLE_2;
 end;

/* Bake for about 25 to 30 minutes or until set.
*/
```

```
        put_into(BAKER, OVEN);

        set_std_alone_timer(BAKE_SOUFFLE_1, 25, MINUTE);
        set_period_timer(BAKE_SOUFFLE_2, 5, MINUTE);
        not_done = TRUE;
        while (not_done) do
        -- loop until the souffle is set --
        begin
                until(ring_std_alone_timer(BAKE_SOUFFLE_1)) do
                begin
                        let_bake(contents_of(BAKER));
                        when (ring-period-timer(BAKE_SOUFFLE_2)) do
                        begin
                                if(souffle-is-set)
                                begin
                                        cancel-std-alone-timer(BAKE_SOUFFLE_1);
                                        cancel-period-timer(BAKE_SOUFFLE_2);
                                        not_done = FALSE;
                                end;
                        end;
                end;
                if (not(souffle-is-set))
                  set_std_alone_timer(BAKE_SOUFFLE_1, 5, MINUTE);
                else
                begin
                        not_done = FALSE;
                        cancel-period-timer(BAKE_STOCK_INGREDIENTS_2);
                end;
        end;

        remove_from(BAKER, OVEN);
        release_tool(OVEN);

end; -- end_of BAKE_SOUFFLE --

-- exception handlers global to the program --
exception TOOL_NOT_AVAIL(needed_tool)
TOOL needed_tool;
begin
  if (cannot find replacement for needed_tool)
    wait-for-tool
  else get_replacement-tool(needed_tool);
  resume;
end;

end; -- end_of program CHEESE_SOUFFLE --


Program BECHAMEL_SAUCE(CUP, 4);

Declare INGREDIENTS

  ING BUTTER is T_BUTTER amount_is 3 unit_is TABLESPOON
        attributes_are() access_is();
  ING FLOUR is T_FLOUR amount_is 3 unit_is TABLESPOON
        attributes_are() access_is();
  ING MILK is T_MILK amount_is 1 unit_is CUP
        attributes_are() access_is();
  ING PEPPER is T_BLACK_PEPPER amount_is ; unit_is ;
        attributes_are() access_is(grind);
  ING SALT is T_SALT amount_is ; unit_is ;
        attributes_are() access_is();
end;

Declare TOOLS
```

```
  TOOL WOOD_SPOON = get_tool(wooden spoon);
  TOOL CASS = get_tool(casserole);
  TOOL W_WHISK = get_tool(wire whisk);
  TOOL BURNER = get_tool(BURNER);
end;

Declare variables

  boolean   not_done;
end;

Declare required_skill

  extern    predicate   is-thick-and-smooth;
  extern    predicate   is-lumpy;
end;

Declare exceptions

  predefined-exception TOOL_NOT_AVAIL;
  exception REMOVE_LUMPS_FROM_SAUCE;
end;

-- main part of the program --
begin
            FIX_BECHAMEL_SAUCE;
            COOK_BECHAMEL_SAUCE;
end;

Procedure FIX_BECHAMEL_SAUCE;
begin

  Declare timers
    active_timer        FIX_BECHAMEL_SAUCE_1;
  end;

/* Melt over low heat: 3 tablespoons butter
*/
            acqu_tool(BURNER);
            acqu_tool(CASS);
            set_heat(BURNER, low_heat);
            put_over(CASS, BURNER);
            add_to(BUTTER, CASS);
            let_melt(contents_of(CASS));
/* Add and blend over low heat for three minutes: 3 tablespoons flour
*/
            add_to(FLOUR, CASS);
            acqu_tool(WOOD_SPOON);

            set_active_timer(FIX_BECHAMEL_SAUCE_1, 3, MINUTE);
            until(ring_active_timer(FIX_BECHAMEL_SAUCE_1)) do
            begin
                        blend(contents_of(CASS), WOOD_SPOON);
            end;

/* Stir in slowly: 1 cup of milk
   Cook and stir the sauce with a wire whisk until thickened and smooth.
*/
            stir_in_slowly(MILK, WOOD_SPOON);
            release_tool(WOOD_SPOON);

            acqu_tool(W_WISK);
            until(contents_of(CASS) is-thick-and-smooth) do
            begin
                        stir(contents_of(CASS), W_WISK);
```

```
                    if (contents_of(CASS) is-lumpy)
                      raise REMOVE_LUMPS_FROM_SAUCE;
            end;
            release_tool(W_WISK);

/* Season with salt and pepper, to taste.
*/
            season_with(contents_of(CASS), PEPPER);
            season_with(contents_of(CASS), SALT);

end; -- end_of FIX_BECHAMEL_SAUCE --

Procedure COOK_BECHAMEL_SAUCE;
begin

 Declare timers
   std_alone_timer     COOK_BECHAMEL_SAUCE_1;
   period_timer        COOK_BECHAMEL_SAUCE_2;
 end;

/* Cook over low heat until for 20 minutes, stirring from time to time.
*/
            acqu_tool(WOOD_SPOON);

            set_std_alone_timer(COOK_BECHAMEL_SAUCE_1, 20, MINUTE);
            set_period_timer(COOK_BECHAMEL_SAUCE_2, 5, MINUTE);
            until(ring_std_alone_timer(COOK_BECHAMEL_SAUCE_1)) do
            begin
                    let_cook(contents_of(CASS));
                    when (ring-period-timer(COOK_BECHAMEL_SAUCE_2)) do
                    begin
                            stir(contents_of(CASS), WOOD_SPOON);
                            if (contents_of(CASS) is-lumpy)
                              raise REMOVE_LUMPS_FROM_SAUCE;
                    end;
            end;

            release_tool(WOOD_SPOON);
end; -- end_of COOK_BECHAMEL_SAUCE --

-- exception handlers global to the program --

exception REMOVE_LUMPS_FROM_SAUCE(recipient)
TOOL recipient -- recipient containing sauce --
begin
TOOL ELEC_BLENDER = get_tool(electric blender);

            acqu_tool(ELEC_BLENDER);
            pour_into(contents_of(recipient), ELEC_BLENDER);
            run_electric_blender(ELEC_BLENDER);
            pour_into(contents_of(ELEC_BLENDER), recipient);

            release_tool(ELEC_BLENDER);
end; -- end_of exception REMOVE_LUMPS_FROM_SAUCE --

exception TOOL_NOT_AVAIL(needed_tool)
TOOL needed_tool;
begin
 if (cannot find replacement for needed_tool)
   wait-for-tool
 else get_replacement-tool(needed_tool);
 resume;
end;

end; -- end_of program BECHAMEL_SAUCE --
```

## APPENDIX 3

### An examples of ingredient type hierarchy

```
TYPE T_VEGETABLE
attributes;
accessing functions
{
   wash;
   chop;
   finely chop;
   dice;
   slice;
}
subtypes -- of T_VEGETABLE --
{
        TYPE T_CRUNCHY
        attributes;
        accessing functions
        {
                peel;
                grate;
        }
        subtypes -- of T_CRUNCHY --
        {
                TYPE T_ONION-FAMILY
                attributes;
                accessing functions;
                subtypes -- of T_ONION FAMILY --
                {
                        TYPE T_ONION;
                        TYPE T_SHALLOT;
                        TYPE T_GARLIC;
                }

                TYPE T_CARROT;
                attributes
                {
                        in_enum{SMALL, MEDIUM, LARGE} size;
                }
                accessing functions;
                subtypes;
        };

        TYPE T_SQUISHY
        attributes;
        accessing functions
        {
                seed;
                peel;
        }
        subtypes-- of T_SQUISHY --
        {
```

```
                TYPE T_TOMATO
                attributes:
                {
                        boolean ripe;
                }
                accessing functions;
                subtypes;
        };

        TYPE T_MUSHROOM
        attributes;
        accessing functions
        {
                brush-under-water;
        }
        subtypes;

        TYPE T_ LEAFY
        attributes
        {
                boolean faded;
        }
        accessing functions
        {
                detach-leaves;
        }
        subtypes -- of T_LEAFY --
        {
                TYPE T_LETTUCE;
                TYPE T_CABBAGE;
        }
} -- end of T_VEGETABLE subtypes --

TYPE T_SOLID-FAT
attributes;
accessing functions;
subtypes -- of T_SOLID-FAT --
{
        TYPE T_BUTTER-TYPE
        attributes
        {
                boolean salted;
        }
        accessing functions
        {
                melt;
                clarify;
                refrigerate;
                cream;
        }
        subtypes -- of T_BUTTER-TYPE --
        {
                TYPE T_BUTTER;
                TYPE T_MARGARINE;
```

```
                TYPE T_SOLID-SHORTENING;
        };

        TYPE T_CREAM-TYPE
        attributes;
        accessing functions
        {
                whip;
        }
        subtypes -- of T_CREAM-TYPE --
        {
                TYPE T_CREME-FRAICHE;
                TYPE T_WHIPPING-CREAM;
                TYPE T_SOUR-CREAM;
        };
};

TYPE T_LIQUIDS
attributes;
accessing primitives
{
        warm-up;
        boil;
}
subtypes -- of T_LIQUIDS --
{
        TYPE T_ALCOHOL
        attributes
        {
                integer degree; -- degree of alcohol --
        }
        accessing functions;
        subtypes -- of T_ALCOHOL --
        {
                TYPE T_BRANDY;
                TYPE T_VERMOUTH
                attributes
                {
                        boolean dry;
                }
                accessing functions;
                subtypes;
        };

        TYPE T_STOCK
        attributes
        {
                in-enum{veal, beef, chicken} flavor;
        }
        accessing functions;
        subtypes;

        TYPE T_WATER
        attributes;
```

```
        accessing functions;
        subtypes;
};

TYPE T_FOWL
attributes;
accessing primitives
{
        bone;
        skin;
        halve;
        pluck;
}
subtypes -- of T_FOWL --
{
        TYPE T_POULTRY
        attributes;
        accessing functions;
        subtypes -- of T_POULTRY --
        {
                TYPE T_CHICKEN;
                attributes;
                accessing functions;
                subtypes -- of T_CHICKEN --
                {
                        TYPE T_CHICKEN-BACK;
                        TYPE T_CHICKEN-BREAST;
                }
                TYPE T_DUCK;
                TYPE T_TURKEY;
        };

        TYPE T_GAME-BIRD
        attributes;
        accessing functions
        {
                hang;
        }
        subtypes -- of T_GAME-BIRD --
        {
                TYPE T_PHEASANT;
                TYPE T_QUAIL;
        };

}

TYPE T_SPICE
attributes;
accessing functions
{
        grind;
}
subtypes -- of T_SPICE --
{
```

```
        TYPE T_PEPPER
        attributes
        {
                in-enum{mild, medium, hot} taste;
        }
        accessing functions;
        subtypes -- of T_PEPPER --
        {
                TYPE T_BLACK-PEPPER;
                TYPE T_WHITE-PEPPER;
                TYPE T_CAYENNE;
        };

        TYPE T_NUTMEG;

        TYPE T_SALT;
}

TYPE T_HERBS
{
attributes
{
        in-enum{FRESH, DRIED, FROZEN} storage;
}
accessing functions
{
        chop;
}
subtypes -- of T_HERBS --
{
        TYPE T_PARSLEY;
        TYPE T_CHIVE;
        TYPE T_TARAGON;
        TYPE T_BASIL;
        TYPE T_MINTH;
        TYPE T_BOUQUET-GARNI;
};

TYPE T_EGG
attributes
{
        boolean fresh;
        in-enum{small, medium, large, jumbo} size;
}
accessing functions
{
        hard-cook;
        beat;
}
subtypes -- of T_EGG --
{
        TYPE T_WHOLE_EGG
        attributes;
        accessing functions
```

```
        {
                poch;
                scramble;
        }
        subtypes;

        TYPE T_EGG-YOLK
        attributes;
        accessing functions;
        subtypes;

        TYPE T_EGG-WHITE
        attributes;
        accessing functions
        {
                whipped;
        }
        subtypes;
}

TYPE T_BREAD
attributes
{
        boolean fresh;      -- fresh or dry --
        in_enum{white, wheat, rye} type;
}
accessing functions
{
        slice;
        toast;
        reduce-to-crumbs;
}
subtypes
{
        TYPE T_BAGUETTE;
        TYPE T_SANDWICH;
};
```