FoodCalc v. 1.3 19981218

Introduction

FoodCalc is a simple program to calculate intake of nutrients when given a list with amounts of different foods consumed and another list with contents of nutrients for the different foods (a food table).

Programs which does variants of such food calculations abound, but most have a fairly limited scope: Some are tied up to a specific food table; some requires that you type in all information, even if you already has the data in a computer file; and some do not handle reductions of nutrients due to cooking, or do so in an idiosyncratic way.
FoodCalc also has a limited scope, but inside this scope it tries to be both flexible and efficient. For starters FoodCalc expects the user to be a programmer or at least someone with enough skills to prepare text files and submit commands on a command line. If you, the reader, do not feel you fit those expectations, you should ask a programmer for initial help in using the FoodCalc program.

FoodCalc expects you to already have both the list of consumed amounts and the food table available in computer text files. If this is not the case, you must first enter the data in an editor or in some other system, e.g. a database system or an interview support system. Most such systems will then be able to export the data in a form understandable by FoodCalc. Also the commands telling FoodCalc how to do the calculations must be entered in a text file before the execution of the FoodCalc program. This may not seem very user-friendly, but it makes it easy repeatedly, systematically and well documented to do the same calculations but with different data, or to do different calculations with the same data. Also it makes it easy for programmers to use FoodCalc as a part of a larger system.
The calculations done by FoodCalc are rather simple, however it does have some flexibility, especially in how to do reductions of nutrients when foods are cooked by boiling, frying, etc.

FoodCalc is very efficient, making it possibly quite quickly to compute intakes for tens or even hundreds of thousands of people with each hundreds or thousands of foods consumed. The time taken for FoodCalc to do the calculations will however largely depend on the computer used. At the time of writing FoodCalc is available for Windows 95, Windows NT and HP-UX. Also the source code is available, and can be easily compiled on most systems that have an ANSI C compiler.

The FoodCalc program was written by Jesper Lauritsen. The writing of the initial version and some of the later changes was largely funded by the Diet, Cancer and Health project at the Danish Cancer Society.

The program is in the public domain. You may use it and change it in any way you like. However, you should always give due credit to Jesper Lauritsen. Also, neither Jesper Lauritsen nor the Danish Cancer Society can in no way be responsible for any damages done by this program. Also, neither Jesper Lauritsen nor the Danish Cancer Society gives any guarantees what so ever regarding the functionality or correctness of this program.

This documentation is in two parts. The first part is a user's guide with the sections:

The second part is a reference guide with the sections:

The reference guide also have detailed information about all the commands: Log:, Verbosity:, Commands:, Save:, Decimal point:, Separator:, Comment:, Blip:, Foods:, Groups:, Recipes:, Food weight:, Ingredients:, Cook:, Weight cook:, Set:, Group set:, Recipe set:, No-calc fields:, Text fields:, Input:, Input format:, Input fields:, Input *fields:, Input scale:, Cook field:, Reduce field:, Weight reduce field:, Recipe reduce field:, Recipe weight reduce field:, Non-edible field:, Output:, Output format:, Output fields:, Transpose:, Group by:, Where:.

Food table (foods file)

The basis for the calculations is a database with the contents of nutrients for different foods – this is what we call a food table. As nutrients FoodCalc understands both macro nutrients (total energy, total fat contents, total protein contents, etc.) and micro nutrients (vitamins, minerals, etc.) (it would perhaps be more correct to talk about food components instead of nutrients). Indeed FoodCalc does not distinct between different types of nutrients (except possibly by some rule you specify for reductions when cooked). Also FoodCalc treat all foods in the food table equal, and the foods may be basic foods, composite foods, or even whole meals.
Each food must be identified by a number, which we call the food id. The numbers must be positive integers, but they do not need to be consecutive and you can choose any numbering scheme you like.
For each food should be listed the values specifying the contents of each nutrient. All such values should be given with respect to the same amount of the food. Typically you will specify the values for 100 g of the food (we call this the food table amount), but you may choose any amount, as long as you use the same amount for all nutrients and for all foods. You do not normally have to specify the food table amount explicitly, but it may be a good idea to do so anyway, and we will return to how to do that.
The units of each nutrient may be different (but each nutrient should off course be specified in the same units for all the foods).
The food table must be in the form of a text file. This means you must prepare it with a text editor, a word processor (where you export to a text file) or perhaps export it from some database system. We call such a file a foods file. The following is an example of a very small foods file:

```
foodid,energy,fat,water,vit_d,calcium
381,766,12.1,68.9,0.60,7.0
1146,949,0.2,44,0,0
```

The first line lists the names of the columns in the table. We call the columns for fields, so we say the first line is a list of the field names. Each field is either a food id (food#) or a nutrient (energy, fat, etc.) (Later we will introduce other types of fields). Although it is not stated explicitly, this example specifies all values per 100 g of the foods (which are defatted pork collar and apricot marmalade from the Danish national food table). The nutrients are total energy in kJ, total fat in g, total water in g, vitamin D in µg, and calcium in mg. But note that all we specify is the names of the fields.
It is very important to note that FoodCalc does not handle missing values. If you leave out values from the table, FoodCalc will simply use zero for such values.

Consumed amounts (input file)

How you express the consumed amounts of different foods will of course depend somewhat on how you collected this information. As a minimum you must specify a list with food id and the amount consumed. The food id must of course be defined in the food table. The amount can be in grams, in pounds, or in any other unit, but you must of course use the same unit consistently. You do not have to use the same unit as you use in the food table. You can have any number of additional values like person identifications, food occasion identifications, etc.
Again the information must be in a text file, and we call such a file an input file. The following is a small example:

```
Person_id,food,amount
101,381,9
101,1146,20
102,381,11.5
```

Again the first line names the fields (we have three fields here). In this example the file contains information about a number of persons, which each has consumed a number of foods (this could for example be data from a food frequency questionnaire or perhaps from a food diary). The first field, "person_id", identifies the person. The second field, "food", identifies the food consumed. The third field, "amount", is the amount consumed in grams.

Food calculations (commands file)

When you want to do food calculations, you will always need at least three text files: the foods file; the input file; and a commands file that specifies how to do the calculations.
Let us say we have the foods file in the example above in the file "foods.txt" and the input file in the other example above in the file "input.txt". If we want to get consumed nutrients for each consumed food we need the following simple commands file:

```
; this is an example
Foods: "foods.txt" foodid
Input: "input.txt" food amount
Input scale: 0.01 ; = 1/100
Output: "output.txt"
```

The commands file can contain comments in lines starting with a semi-colon. You can also have comments at the same line as a command, starting with a semi-colon (as after the "Input scale:" command in the example above). Commands in the file are each on one line, and such command lines always start with the name of the command followed by a colon and then the arguments to the command. Most commands can be entered in any order and FoodCalc does not differentiate between upper and lower case letters.
The "Foods:" command specifies the name of the file containing the food table and the name of food id field. The "Input:" command specifies the name of the file containing the consumed amounts and the name of the food id field and the amount field in that file. The "Input scale:" command specifies the proportion between the units of the input file amount and the food table amount. I this example the units of the input file is grams and the food table amount is 100 grams, giving an input scale value of 0.01 (= 1/100). Finally the "Output:" command specifies the file name of the file where the resulting values are written.

If the commands are in the file "example.fc", we can execute the calculation be entering the following at a command prompt (a DOS-prompt or a UNIX shell-prompt, depending on your operating system):

```
foodcalc example.fc
```

This will make a log file with the name "example.log", telling a little about what was done and if there were any errors. After running FoodCalc you should always check for errors in the log file. The log file looks like this:

```
FoodCalc v. 1.2
Tue Jun 09 16:25:24 1998
```

```
Read 4 commands from file example.fc.

Read foods file foods.txt. Foods: 2. Fields:
foodid energy fat water vit_d calcium

The food table contains 2 foods and the fields:
foodid energy fat water vit_d calcium

Read input file input.txt. Lines: 3. Fields: 3
person_id food amount

Wrote output file output.txt. Lines: 3. Fields: 9
person_id food amount foodid energy fat water vit_d calcium
```

The resulting output file will contain the following:

```
person_id,food,amount,food#,energy,fat,water,vit_d,calcium
101,381,9,381,68.94,1.089,6.201,0.054,0.63
101,1146,20,1146,189.8,0.04,8.8,0,0
102,381,11.5,381,88.09,1.3915,7.9235,0.069,0.805
```

As you can see, there is one line in the output file for each line in the input file. You can also see that it contains all the fields from the input file as well as all the fields from the foods file.
The calculations done here are extremely simple. The grams consumed from the input file is simply multiplied with the input scale factor and with the values from the foods file (for example do you get the value for energy in the first line as 9*0.01*766 = 68.94).

Group by

In the above example we got one line in the output file for each line in the input file. However, most often you want those results aggregated to some level. If for example we want the total intake for each person, we can add the following command to the commands file:

```
Group by: person_id
```

This will result in the following output file:

```
person_id,energy,fat,water,vit_d,calcium
101,258.74,1.129,15.001,0.054,0.63
102,88.09,1.3915,7.9235,0.069,0.805
```

FoodCalc has summarized the nutrients values for all the consumed foods for each person, resulting in one line for each person in the input file. You should note that the output file now only contains the group by field and the nutrients fields from the foods file. This is because it would be meaningless to summarize any of the other fields.

Let us take a more elaborated example. Let us say we have information about foods consumed for a number of persons at a number of occasions (for example meals in a single day). The food occasion is coded in an extra field, "fco", in the input file. Also we have grouped all the foods in a number of groups (vegetables, fruits, meat, etc.). The food group is coded in an extra field, "food_group", in the foods file.
If we now want total nutrients intake for each person at each food occasion for each food group, we can do it with:

```
Group by: person_id,fco,food_group
```

If the first one or more fields are input fields then the input file must be sorted on these input fields (previous version of FoodCalc had more limitations on the use of the "Group by:" command, but all these limitations has now been lifted).

Transposing

Let us say that we use the commands:

```
Group by: person_id, food_group
Output fields: person_id, food_group, energy
```

and that the result is the following output file:

```
person_id,food_group,energy
101,1,68.94
101,3,189.8
102,1,88.09
```

For each person there is a number of lines, and each line has the energy intake contributed from one food group. However, if we want to import those data into a statistical or graphical system for analysis, that is not a very nice form. Let us say that all foods in the food table has food_group values between 1 and 4. We can then use the commands:

```
Group by:person_id
Output fields: person_id
transpose: food_group 4 energy
```

and the result will then be the following output file:

```
person_id,energy1,energy2,energy3,energy4
101,68.94,0,189.8,0
102,88.09,0,0,0
```

There now is only one line for each person but one energy field for each food_group. We say that we transpose energy on the food_group. You can have any number of "Transpose:" commands.

Cooking

Many food tables contains only (or perhaps mostly) uncooked foods. But most people like cooked foods, and cooking can substantially change the amount of nutrients in the foods. FoodCalc’s understanding of cooking is very simple but also quite flexibly.

FoodCalc cooks by using what we call reduce fields. If we have a foods file with the fields:

```
Food_id,vit_a,sodium,calcium,vit_a_boil,vit_a_fry,min_boil,min_fry
```

We can then use the following commands:

```
Cook: boil vit_a_boil vit_a
Cook: boil min_boil sodium,calcium
Cook: fry vit_a_fry vit_a
Cook: fry min_fry sodium,calcium
```

The fields vit_a, sodium, and calcium are nutrient fields and the fields vit_a_boil, min_boil, vit_a_fry and min_fry are reduce fields. The commands specifies that if we are boiling a food, we should reduce the vitamin A content with the factor in the vit_a_boil field, and we should reduce the Sodium and Calcium content with the factor in the min_boil field. If we are frying a food, we should reduce the vitamin A content with the factor in the vit_a_fry field, and we should reduce the Sodium and Calcium content with the factor in the min_fry field.

In this way you fully control how much which nutrients should be reduced by each type of cooking. You can specify any number of cooking methods with any names you like and you can use as many reduce fields as necessary.
This method allows different reduction factors for each food and for each cooking method (you can use groups files as explained later to use the same reduction factors for groups of foods).

The reduce fields should have values, which are the factor the nutrient should be reduced by. Let us say that for a given food the vitamin A contents should be reduced by 10% when boiling, then the vit_a_boil field should have the value 0.1. You can also specify negative values meaning that the content of the nutrient should be increased. This may be useful for water in some foods when boiling and for fat in other foods when frying.

You use an extra field in the input file to specify if and how to cook the food consumed. This field should be specified with the "Cook field:" command, as in the line below:

```
Cook field: cook boil,fry
```

The first argument to the "Cook field:" command tells the name of the field in the input file, which specifies the cooking. In this example the field "cook" is used. The second argument should be a list with the names of the cooking methods used. The "cook" field should have the value zero if the food should not be cooked. It should have the value one if it should be cooked by the first cooking method listed in the command (in this case it is "boil"), it should have the value two if it should be cooked by the second method listed, and so on.

We will later show how to do cooking and other reductions in other ways.

You should note that when an input line specifies that the consumed food is to be cooked, then the amount field in the input file should specify the *uncooked* weight of the food consumed. If you do not like this, you can instead use recipes as explained later.

Calculating new fields

In our very first example we had an "energy" field in the foods file. However, this is not a good field to have in the food table. This is because total energy usually is calculated from other fields in the food table, and if cooking reduces the values of these other fields you must also reduce the energy in a perhaps non-obvious way.
A common method to calculate the total energy contents is to calculate: 17*protein_total + 38*fat_total + 17*carbohyd_total + 30*alcohol. If the foods file contains the fields protein_tot,

fat_tot, carb_total and alcohol with values in grams, we can make a new field, which will contain the total energy in kJ with the command:

```
Set: energy = 7*protein_total + 38*fat_total + 17*carbohyd_total + 30*alcohol
```

You can make any number of new fields with the "Set:" command. These new set fields can be used almost anywhere you can use a food table nutrient field.

The set fields are calculated for each food that is calculated, and if the "Group by:" command is used the values of the set fields are aggregated just like any food table nutrient field. However that means that we can not use the "Set:" command to for example calculate the fat energy percent (think about it, and you will se why). Therefor there is a variant:

```
Group set: fat_energy_pct = 38*fat_total / energy * 100
```

The group set fields are calculated *after* the group by aggregations and will therefor calculate the fat energy percent correctly. You may wonder if you can not just always use the "Group set:" command instead of the "Set:" command. The answer is, that you very often can do that (for example it would probably be a good idea to make energy a group set field instead of a group field). But for technical reasons you can only use group set fields in the "Output fields:", while you can use set fields in many other commands.

Another useful field to calculate could be the weight of the food consumed (this will be equal to the amount field in the input file if the food is not cooked, but it may be different if it is cooked). This field we will later use in the "Food weight:" command, and therefor we make it a set field and not a group set field. If you have the fields above and also the fields "ash" and "water" in the foods file, you can calculate it with:

```
Set: weight = protein_tot + fat_tot + carb_tot + alcohol + ash + water
```

Groups files and more than one foods file

If you have groups of foods which has the same value for one or more fields you may use groups files. This is most often useful for reduce fields, where you often only will have generic values for groups of foods.
To use a groups file you must have one or more group reference fields in the foods file. These fields specifies which group the food is a member of. Such group ids must be positive integers. If all the group reference fields has a value of zero for some food, this food is not a member of any group.
The groups file is a text file very much like a foods file. It should contain the group id fields and there should be a line for each group. The groups file can contain any number of additional fields.

If we have a foods file like:

```
Food_id,vit_a,group_id
400,3,1
401,2,3
402,2,1
403,3,0
```

And a groups file with name "groups.txt" like:

```
Group_id,vit_a_boil,vit_a_fry
1,0.1,0.2
2,0.2,0.1
3,0.4,0.4
```

Then if we have the command line:

```
Groups: "groups.txt" group_id
```

It will be just as if we had a foods file like:

```
Food_id,vit_a,group_id,vit_a_boil,vit_a_fry
400,3,1,0.1,0.2
401,2,3,0.4,0.4
402,2,1,0.1,0.2
403,3,0,0,0
```

You can have any number of group files, and you can also have more than one foods file if you want. There can usually be two reasons to have more than one foods file. You may want to have some foods in one file and other foods in another file (for example if you have different sources for the data). In this case you should use different food id values in the different files. Also you may want to have some fields in one file and other fields in another file (for example you may want to have nutrient fields in one file and reduce fields in another file). In this case you should make sure that all foods are in both files with the same food id values. If a food is not in one of the foods files, all fields defined in that file will get the value zero.
You specify several foods files simply by using several "Foods:" commands.

We call the combination of all groups and foods files for the food table (however, we will se that recipes files will add foods to the food table and that the "Recipe set:" command can add fields to the food table).

Recipes files

You can add new foods to the food table by combining other foods in the foods table according to recipes, which specifies how much of each ingredient (an existing food) to use and if and how the ingredient should be cooked.

You specify recipes in recipes files, which looks very much like input files, as in the example below:

```
Recipe_id,food,amount,cook
3001,400,30,1
3001,401,110,1
3001,403,5,0
3002,400,50,1
```

The file contains four fields: The first field is the recipe id, which must be a positive integer. Recipe ids must be unique and must also be different from all food ids (actually, after the recipe is read it will be a food in the food table with food id equal to the recipe id.) The second field is the food id of the ingredient, which should have values which can be found in a foods file. The third field is the amount, which can be in grams or any other unit exactly as in input files. And finally there may be a cook field exactly as in input files. The file should have one line for each ingredient in each recipe.

Alternatively you can use an other format of the recipes file, which we call the star format. With the star format the example above looks like this:

```
*Recipe_id
food,amount,cook
*3001
400,30,1
401,110,1
403,5,0
*3002
400,50,1
```

The two examples define exactly the same recipes. The examples defines two new foods(/recipes): Food(/recipe) 3001 which is composed of 30 grams of food 400, boiled, 110 grams of food 401, also boiled, and 5 grams of food 403, uncooked. And food 3002 which only is 50 grams of food 400, boiled.

You use recipes files with the "Recipes:" command, like in the example below:

```
Recipes: "recipes.txt" recipe_id food amount
```

The "Recipes:" command takes as arguments the name of the recipes file, the name of the field with recipes ids, the name of the field with food ids for the ingredients, and the name of the field with the amount of the ingredient.

As you remember, all nutrient values in the foods and groups file were per some implicit amount of food (for example per 100 grams), and we called this amount for the food table amount. The amounts of the ingredients may sum to any value for a recipe, but for us to use the recipe as a new food in food table we must normalize the total amount to the food table amount. Also, cooking may change the amounts for some of the ingredients. The FoodCalc program does this normalizing automatically, but you must tell it how to know the weight of a cooked food and what the value of the food table amount is.
The weight of a cooked food was discussed in the "Calculating new fields" section, and you can only use recipes if you have fields, which sum to the cooked weight of a food (however, if you do not use cooking, you can simply have a "weight" field in the foods file that has the value 100 for all foods). You specify this with the "Food weight:" command, like below:

```
Food weight: 100 protein_tot,fat_tot,carb_tot,alcohol,ash,water
```

The first number is the value of the food table amount and the sum of the fields listed should be the weight of the food. If you had defined a new "weight" field like in the calculate example previously, you could simply use:

```
Food weight: 100 weight
```

In the example above, the recipe 3002 contains only the food 400, boiled. Why make such recipes, when you can specify food 400, boiled, directly in the input file? The difference is that if you use food 400, boiled, in the input file, you should specify the *un-cooked* weight consumed. But if you use recipe 3002 in the input file, you should specify the *cooked* weight consumed.

FoodCalc allows you to specify a negative amount for an ingredient. This is probably only meaningful if that ingredient is water (with zero content of all nutrients except the content of water). In that case the interpretation can be that you remove water from the recipe because the water evaporate while cooking the recipe. The water content in recipes is very important because the proportion of water

influences the level of all nutrients when you take a given amount of the total recipe.
You must of course be very careful when you combine this method with the use of reduce fields.

When defined, the recipes behave just like foods defined in a foods file, and you reference recipes just like simple foods in your input file. However, when you group by a field in the food table (for example by the food id or by some food group), you may want to group by the ingredients instead of by the recipes. For example, if you are interested in the amount of vitamin A coming from vegetables, you will group by some food group separating the vegetables from other foods. But if recipes are like foods, then you will not get the contribution from carrots used in recipes. To group by the ingredients instead of by the recipes, you should use the command:

```
      Ingredients: keep
```

You should only use "Ingredients: keep" when you have to, as it causes FoodCalc to run somewhat slower.

Often you will have some grouping of the foods in you foods file, expressed by one or more group fields. When you add recipes to the food table, these recipes/foods will get the value zero for all fields that are not nutrient fields, and this means recipes always will have the value zero for such group fields. You can change this by having fields in the recipe file with the same name as group fields. The values from the recipes file will then be used in the food table if you use the "Ingredients:" command with either the sum argument or the keepx argument (keepx is just like keep, except that keep does not copy values from the recipes file to the food table).

Reducing from input and recipes

We talked previously about how you can use reduce fields when cooking. A specific cooking method chooses specific reduce fields from the food table.
You can also use reduce fields from the input file or from recipes files. Such reduce fields are not related to the cook field – an input/recipe reduce field will always be used regardless of any cooking method specified. These reduce fields are probably most interesting in recipes.

Let us say you have an extra field in your recipes file with the name water_loss, and you have a field in the food table with the name water. The water_loss field contains the proportion of water lost in the given ingredient when the recipe is cooked. You tell this to FoodCalc with the command:

```
      Reduce field: water_loss water
```

As you can see, the "Reduce field:" command is exactly like the "Cook:" command except you do not specify a cooking method and that the reduce field is in the recipe and/or input file instead of in the food table.
You can combine the use of reduce fields from the food table and from the recipe file. For example you can use reduce fields from a groups file to specify loss of micro nutrients depending of the cooking method, combined with a reduce field in a recipe file to specify loss of water for specific ingredients in specific recipes. You must of course be very careful if you reduce a nutrient field with both a recipe reduce field and a food table reduce field!

Weight reductions

The "Reduce field:" command uses reduce fields which specify how much to reduce as a fraction of the value of the nutrient. Alternatively you can use what we call weight reduce fields, which specify how much to reduce as a fraction of the weight of the food.
To use weight reduce fields, FoodCalc must know the food table amount, and it must know how to calculate the weight of a food. You specify this with the "Food weight:", just as when you use recipes, as explained above.
Some countries has a tradition of specifying reductions of water and fat in this way. You tell FoodCalc to use weight reduce fields with the "Weight reduce field:" command as in the example below:

```
Food weight: 100 protein_tot,fat_tot,carb_tot,alcohol,ash,water
Weight reduce field: water_reduc water
```

Let us say that a recipe contains 20 grams of an ingredient which has a water_reduc value of 0.2. Then the water field in the example above will be reduced with 20*0.2 = 4.

When reducing the the contents of fat you may like to specify it as a weight reduction. We usually call "fat" a macro nutrient. You also may have several micro nutrients which are different components of fat, and you can choose to reduce them with the same factor as the fat macro nutrient, as in the example below:

```
Weight reduce field: fat_reduc fat,fat_mono,fat_poly,fat_other
```

Let us say that a recipe contains 20 grams of an ingredient which has a fat_reduc value of 0.1. Then the fat field in the example above will be reduced with 20*0.05 = 2, just like in the water example above. If the ingredient has a fat value of 30 grams per 100 grams of the food, then the resulting fat contents for the ingredient will be 30/100*20 - 2 = 4. This means the fat content has been reduced with the fraction 2/(30/100*20) = 0.333. In the example above the fields fat_mono, fat_poly and fat_other will then be reduced with the same fraction 0.333.
If you have a list of fields in the "Weight reduce field:" command, the first field may be a calculate field. For example, if the sum of the fields fat_mono, fat_poly and fat_other equals the total fat contents in the food, the you do not need the fat field in the food table, but can instead calculate it with the "calculate:" command.

You can also use weight reduce fields when cooking, just by using the "Weight cook:" command instead of the "Cook:" command, as in the example below:

```
Weight cook: boil water_boil water
Weight cook: fry water_fry water
Weight cook: boil fat_boil fat,fat_mono,fat_poly,fat_other
Weight cook: fry fat_fry fat,fat_mono,fat_poly,fat_other
```

Recipe reductions

As mentioned earlier it is usually most useful to do reductions in recipes files instead of in input files. But note, that the commands "Cook:", "Weight cook:", "Reduce field:" and "Weight reduce field:" all specifies reductions on the ingredient level. Some times you do not want to be that specific, but do just want to use the same reductions for all ingredients in a recipe. You can do that with the "Recipe reduce field:" and "Recipe weight reduce field:" commands, as in the example below:

```
Recipe reduce field: water_reduc water
Recipe weight reduce field: fat_reduc fat,fat_mono,fat_poly,fat_other
```

The recipe reduce fields should be star fields if you use the star format, otherwise the value of the recipe reduce field should be the same for all ingredients.
The "Recipe reduce field:" command is actually just an alias for the "Reduce field:" command (think about it, and you will see it is the same), but you can use the "Recipe reduce field:" command instead of the "Reduce field:" command, to document that you reduce with the same fraction for all ingredients.
The "Recipe weight reduce field:" works like this: The first field in the list of nutrient fields in the command are summarized for all the ingredients (let us call this n_sum), and also the weight of all the ingredients are summarized (let us call this w_sum). If we call the value of the reduce field for r, FoodCalc then calculates a fraction to reduce by: (r * w_sum) / n_sum. All the nutrient fields in the command are then reduced by this fraction for all the ingredients in the recipe.

Calculating recipe set fields

The "Recipe set:" somewhat like the the "Set:" command in that it calculates an expression and puts the resulting value in a field. However, the "Recipe set:" commands are only calculated when the recipes files are read, not when the input file is read. The expression can contain fields from the recipes files and the resulting field will be put in the food table. This means that the resulting field can then be used just like any other food table field. There are several very specialized uses for this.

If a recipes file contains the field recipe_group then the command:

```
Recipe set: recipgrp = recipe_group
```

will move the value of recipe_group to the new food table value recipgrp for all recipes.

The previous "Recipe reduce field:" example could have been written:

```
Recipe set: water = water * (1 - water_reduc)
```

This is of course not very interesting, but you can see that you can use the "Recipe set:" command to do other very specialized reduction etc.

Finally, if you for example have a glycemic_index field in your recipes files, you can use:

```
Recipe set: glycemic_load = carbonhydrate * glycemic_index
```

to calculate a new glycemic_load field in the food table containing the glycemic load for all recipes.

Non-edible part

Many foods consist naturally of a non-edible part (for example the bone in a lamb chop) and an edible part (for example everything but the bone in a lamb chop). The food table (the foods and groups files) should contain nutrient values for only the edible part of foods, and the input and recipes fields should normally have amount fields specifying the amount eaten.

However, if you have a field in the food table, which as value has the non-edible fraction of the foods, as they are normally purchased, you can choose to specify the total weight (both edible and non-edible parts) in the amount fields in input and recipes fields, as in the example below:

```
non-edible field: non_edible
```

In the example the food table field "non_edible" should contain the non_edible fraction, and all amount fields in input and recipes files should then contain the total weights.
More interesting, you may specify that only some amounts are total weights and that others are only the weight of the edible part, as in the example below:

```
non-edible field: non_edible non_ed_flag
```

The second argument to the "Non-edible field:" command should be the name of a field. If this field is not in an input or recipes file, then all the amounts in this file should be the weight of the edible part. If the field is in a file, then it should have the value zero or one. If it is zero, the amount should be the weight of the edible part. If the value is one, the amount should be the total weight of that food.

Selecting foods

It is possibly to tell FoodCalc to ignore some foods or to ignore all but some foods. This is mostly interesting when used with recipes files and the "Ingredients: keep" command.
You use the "Where:" command to select foods. The example commands below select only foods where the food_id field has the value 236 or the food_group field has the value 4 or 7:

```
where: food_id = 236 or group_id in (4,7)
```

As an other example the following command will select all foods with main_group larger than 7 or foods with main_group equal 2 and sub_groups 3, 4 or 7:

```
where: main_group > 6 or (main_group = 2 and sub_group in (3,4,7))
```

Reference section

The rest of this document is reference information, which probably is of most interest for programmers who want to use FoodCalc together with other systems.

Data files

FoodCalc uses a number of data files: foods files, input files, output files, groups files and recipes files. All of these files are normally text files, and they share a common format (only the input and output files may have a different format as explained under the "Input format:", "Input fields:" and "Output format:" commands).

Any line in data files which are empty (contains nothing but blanks) or where the first non-blank character is a semi-colon (;) are comment lines that are ignored by FoodCalc. By blanks we mean spaces or tabs. Other lines can also contain a semi-colon. Then the part of the line from the semi-colon to the end of the line is ignored.
You can instruct FoodCalc to use any other character than semi-colon as the comment character for all or for just a single data file (se the "Comment:" command). This can be useful if you want to use the semi-colon character as separator character.

A logical line may be continued on the next line by putting an equal sign (=) in the first position of the next line.

Normal format

The first non-comment line in a data file should list the names of the fields in the file. The names must be separated by the separator character which is usually a comma (,) and possibly with some blanks before and/or after the separator character. The field names may be any length and may contain any characters. However, if a name contains blanks, the separator character or the comment character it should be enclosed in double-quotes ("). For the letters a-z the case of the letter is not significant. Below are three examples of legal lines with field names:

```
Food_id,vit_a,vit_b ; this is a comment
; this is a comment line
Foodid, vitaminA, vitaminB
# , "Vitamin A" , "Vitamin B" , ; this line continues below
= "A long and very stränge field name"
```

You can instruct FoodCalc to use any other character than comma as the separator character for all or for just a single data file (see the "Separator:" command). This is often useful in countries, where they use comma as decimal point in numbers. If you use space as separator character the field names only needs to be separated by blanks (spaces or tabs).

Any non-comment lines after the list of field names should be a list of values. On each such line there should be the same number of values as the number of field names. The values must be separated with the separator character and blanks in the same way as the field names are separated. The values should normally be numeric, but you can use the "Text fields:" command to specify that some fields can contain any value. Such text fields are always just ignored when the data file is read. If a text field contains any separator, continuation, or comment characters, it should be enclosed in

double-quotes (").
You can leave out a value (but not any of the separator characters), but this will be interpreted exactly as the number zero (FoodCalc does not handle missing values). Numeric values can in general be integers or contain fractions (they can be floating point numbers), and they can be positive or negative (however some types of fields only use the integer part of numbers and some types of fields requires the numbers to be positive). There must not be any spaces in the number and you cannot use thousand separators. The following is three examples of lines with values for four field, where the last field is a text field:

```
1001,23,4.5,xx ; this is a comment
; this is a comment line
0203, 0023, -4.0, "x,;"
23 , , ; this line continues below
= -0.000004, xx yy
```

You can change the character used as decimal point for all or for just a single data file (se the "Decimal point:" command). This is often useful in countries where the comma is used as decimal point (note, that if you change the decimal point to comma, you must also change the separator character to something else.)

Star format

Above we described what we called the normal format in data files. Alternatively you can use a star format in data files. Star format is most useful for recipes files, but you can use it in any data file.
The first non-comment line in a star format file should start with the star character (*), which should be followed by a list of field names. Those fields we call star fields. The second non-comment line should be list of field names. The file will contain the fields from both these two lines.
If any line after the two field name lines starts with a star, the star should be followed by the same number of values, as the number of star fields. For any following lines not starting with a star, the star fields will have the values from the previous star line.
Any lines after the two field name lines not starting with a star, should contain the same number of values as the number of non-star fields.

The normal format file below:

```
field1, field2, field3
1, 2, 3
1, 4, 5
2, 6, 7
2, 8, 9
```

will be interpreted by FoodCalc exactly like the star format file below:

```
*field1
field2, field3
*1
2, 3
4, 5
*2
6, 7
8, 9
```

Standard files

When specifying the name of a file, you can use the special file name "-". The meaning of this depends on the type of file:

```
        Log file: standard error
        Output file: standard output
        All other types of files: standard input
```

The meaning of standard error, output and input will depend on your operating system.

Commands files

When executing the FoodCalc command you can specify one or more commands files as argument and FoodCalc will read all the files. From inside a commands file you can also read other commands files with the "Commands:" command. If you do not give any arguments to FoodCalc, it will read commands from standard input.
In commands files you can use comments and continuation lines in the same way as in data files. The order of the commands in a commands file does not matter and the order of several commands files does not matter (however, for some commands the order of several uses of this command does matter).

Commands

All commands start with a keyword followed by a colon (:). Upper and lower case does not matter in a keyword, and you may use extra blanks. The following is a list of all the commands that FoodCalc understands. Commands marked with + may be used more than once, other commands can only be used at most one time. Commands marked with ! must be used at least one time, other commands are optional.

	Log: file-name
	Verbosity: number
+	Commands: file-name
	Save: file-name
	Decimal point: character
	Separator: character
	Comment: character
	Blip: number
!+	Foods: file-name [id-field sep-char dec-point-char com-char]
+	Groups: file-name [id-field-list sep-char dec-point-char com-char]
+	Recipes: file-name [recipe-field food-field amount-field sep-char dec-point-char com-char]
	Food weight: weight field-list
	Ingredients: keep|keepx|sum
+	Cook: type-name reduce-field field-list
+	Weight cook: type-name weight-reduce-field field-list
+	Set: field = expression
+	Group set: field = expression

+	Recipe set: field = expression
+	No-calc fields: field-list
+	Text fields: field-list
!	Input: file-name [id-field amount-field sep-char dec-point-char com-char]
	Input format: text|bin-native
	Input fields: field-list
	Input *fields: field-list
	Input scale: scale
	Cook field: cook-field type-list
+	Reduce field: reduce-field field-list
+	Weight reduce field: weight-reduce-field field-list
+	Recipe reduce field: reduce-field field-list
+	Recipe weight reduce field: weight-reduce-field field-list
	Non-edible field: non-edible-field [flag-field]
!	Output: file-name [sep-char dec-point-char]
	Output format: text|text-no-head|bin-native
	Output fields: field-list
	Group by: field-list
+	Transpose: field number field-list
	Where: logical-expr

Arguments to commands must be separated by one or more blanks. If an argument contains blanks, commas or semi-colons it must be enclosed in double-quotes ("). If a number is used as argument, you must use the point character (.) as decimal point. If an argument is a list of values or names the elements in the list must be separated by commas. If some arguments in the list above is enclosed in square brackets ([ ]) they are optional. You can leave out all optional arguments or just the last one or more arguments.
If an argument is a *field-list* you can list field names separated by commas. You can also specify a range of fields by typing the name of the first field, two dashes (--), and the name of the last field. Single fields and ranges of fields may be in the same *field-list* separated by commas. The order of food table fields will be the order they are added to the food table by "Foods:" and "Groups:" commands.

Calling FoodCalc

You execute FoodCalc by entering the following at a command prompt or in a script file:

```
    foodcalc [options] [file-names]
```

The file names should be names of commands files. If more than one file is specified, the ones following the first will be read as if the commands were entered at the bottom of the first file. If no file names are given as argument, FoodCalc will read commands from standard input.

You can optionally give one or more of the following options as arguments before any file name arguments:

-v *number*	The same as using the "Verbosity:" command. This will also override any "Verbosity:" commands in the commands files.
-l *file-name*	The same as using the "Log:" command. This will also override any "Log:" commands in the commands files.
-i *file-name*	This file will be used instead of the file name given as argument to the "Input:" command. This is most useful when also using the -s command.
-o *file-name*	This file will be used instead of the file name given as argument to the "Output:" command. This is most useful when also using the -s command.
-s *file-name*	See the description of the "Save:" command for information about the -s option.

Log: command

```
            log: file-name
```

The "Log:" command specifies the name of a file where the FoodCalc log will be written. If you do not use the "Log:" command, FoodCalc will take the name of the commands file given as argument to FoodCalc, strip out an ending ".fc" or ".txt" and append ".log". The resulting file name will be used for the log file. If you do use the "Log:" command, it should be the first command in the commands file.

Verbosity: command

```
            verbosity: number
```

The "Verbosity:" command specifies how verbose FoodCalc will be when writing the log file. The higher the number, the more verbose FoodCalc will be. As default will be used the number 50, which is pretty verbose. Currently the following levels are defined:

50	Default.
40	Do not print warning messages.
30	Do not print the name of the log file to standard error.
20	Do not print any informational messages about files read and written.
10	Do not print any error messages.
1	Do not make a log file.

Commands: command

```
       +        commands: file-name
```

The "Commands:" command is optional and can be used more than once. It specifies an additional commands file, and the commands in this file will be processed as if they were entered at the bottom of the current commands file.

Save: command

```
        save: file-name
```

The save command is very special. When it is used, FoodCalc will read all the commands, foods, groups and recipes files, but it will not read the input file nor write the output file. Instead it will save all the information from the commands, foods, groups and recipes files in the file given as argument to the "Save:" command, using an efficient binary format.
At a later time you can then execute the FoodCalc command with:

```
       foodcalc -s save-file
```

When executed like this, FoodCalc will read the information in the binary save-file and then run as if it was given the original commands, foods, groups and recipes files. However, you may override the names of the log file, input file and output file from the original commands file by using the -l, -i and -o options.

This is mostly useful if you run FoodCalc repeatedly with different (small) input files but with the same foods, groups and recipes files. FoodCalc can read the binary save file much faster than the original foods, groups and recipes files.

Decimal point: command

```
            decimal point: character
```

Specifies the character to use as default decimal point for all data files (you may override it for specific data files). If you do not use the "Decimal point:" command, FoodCalc will use the point character (.) as default decimal point.

Separator: command

```
            separator: character
```

Specifies the character to use as default separator for all data files (you may override it for specific data files). If you do not use the "Separator:" command, FoodCalc will use comma (,) as default separator.

Comment: command

```
            comment: character
```

Specifies the character to use as default comment character for all data files (you may override it for specific data files). If you do not use the "Comment:" command, FoodCalc will use semi-colon (;) as default comment character.

Blip: command

```
        blip: number
```

If the "Blip:" command is used, FoodCalc will write the number of lines read from the input file to standard error (usually to the terminal/screen), whenever it has read as many lines as is specified as argument to the "Blip:" command.

Foods: command

```
        !+        foods: file-name [id-field sep-char dec-point-char com-char]
```

Tells FoodCalc to read the foods file with the name given as the first argument. A foods file is a data file, where each line defines a food. One of the fields in the file must be the food id field, which identifies the foods. The food id field will be a no-calc field. Food id values must be unique positive integers. If the second argument to the "Foods:" command is used it must be the name of the field, which is the food id field. Otherwise the first field in the file is used as food id field. The third, fourth and fifth arguments can be used to specify the separator character, the decimal point character and the comment character for the foods file.
You must always have at least one "Foods:" command, and you can have as many as you need. If you use more than one foods file, the files do not need to have the same fields and they do not need to contain the same foods. The foods files are read in the same order as the "Foods:" commands are given. If more than one foods files defines the same field for the same food, the food table will get the value from the last foods file. If a foods file define a food but none of the foods files defines all fields for this food, the missing fields will get zero values.
If you use more than one foods file, you will usually have files with same fields but different foods, or files with different fields (except the food id field) but the same foods. However there may be situations where it is useful to combine this.

Groups: command

```
        +        groups: file-name [id-field-list sep-char dec-point-char com-char]
```

Tells FoodCalc to read the groups file with the name given as the first argument. A groups file is a data file, where each line defines a group. One or more of the fields in the file must be the group id fields, which identifies the group. Group id values must be unique positive integers. If the second argument to the "Groups:" command is used it must be the names of the group id fields. Otherwise the first field in the file is used as a single group id field. The third, fourth and fifth arguments can be used to specify the separator character, the decimal point character and the comment character for the groups file.
The values in the groups file will be expanded into the food table. Before the expansion there must already be fields in the food table with the same names as the group id fields in the groups file. We call these for group reference fields. All the fields in the groups file will be added to the food table. For all foods in the food table, the group reference fields must be either zero or have a value, which can be found as group id in the groups file. If it is found in the groups file, all the values for that group

will be used for the given food. If the food reference values is zero all the group fields will get zero values for the given food. The group reference fields will be a no-calc fields.
You can use as many groups files as you need. Groups files are expanded in the order the "Groups:" commands are given.

Recipes: command

```
+       recipes: file-name [recipe-field food-field amount-field sep-char
        dec-point-char com-char]
```

Tells FoodCalc to read the recipes file with the name given as the first argument. A recipes file is a data file, where each line defines an ingredient in a recipe. One of the fields in the file must be the recipe id field, which identifies the recipe. Recipe id values must be unique positive integers (they must be unique across all recipe files and foods files). An other field must be the ingredient id field, which identifies the ingredient in the recipe. Ingredient id values must be found in the food table, and this identifies the food in the ingredient. A third field must be the amount field, which specifies the amount used of the ingredient. If the second, third and fourth argument are given to the "Recipes:" command, they must be the names of the recipe id, ingredient id and amount fields. Otherwise the first, second and third fields in the file will be used. The fifth, sixth and seventh arguments can be used to specify the separator, decimal point and comment characters for the recipes file.
All ingredients for a recipe must come right after each other in the recipes file. If you manually edit your recipes files you will usually use the star format with the recipe field as a star field.
FoodCalc will calculate the total nutrients content for each recipe, which are then normalized as described under the "Food weight:" command. The resulting values will be entered in the food table with a food id equal to the recipe id. The effect is that the recipe can then be used as any other food.
If you use "Ingredients: sum" and a recipe file has any fields with the same name as non-calc fields in the food table, then these recipe fields should have the same value for all ingredients in a recipe, and these values will be copied to the food table fields when the recipe(/food) is entered into the food table. If you use the star format, you should make such recipe fields star fields.
You can use as many recipes files as you like. When a "Recipes:" command is given you must also give a "Food weight:" command.

Food weight: command

```
food weight: weight field-list
```

Specifies the food table amount and which fields sum to the weight of the food. The first argument specifies for which amount of food the nutrient values in the food table are given. This is what we call the food table amount, and is typically 100 grams.
The second argument to the "Food weight:" command should be a list of fields, which are in the food table or which are set fields. The sum of the values of these fields (after any reductions) should express the total weight of the food (typically you should specify fields containing the values of all the macro nutrients: total protein, total fat, total carbohyd, alcohol, ash and water). In recipes all nutrients in all ingredients will then be normalized with the same factor, so that this sum for all ingredients equals the food table amount.

Ingredients: command

```
        ingredients: keep|keepx|sum
```

The "Ingredients:" command can be used with one of the keywords "keep", "keepx" or "sum" as argument. If sum is specified, recipes will behave exactly like they were a single food. This is the default. If keep or keepx is used, then recipes will behave as if each of the ingredients was specified separately in the input file.
If sum or keepx is used any fields from the recipes files with the same names as fields in the food table will be copied to the food table when the recipes are entered into the table. If keep is used such fields keep the value that the ingredients had in the table.

Cook: command

```
        +       cook: type-name reduce-field field-list
```

The "Cook:" command specifies how to do reductions of nutrients when cooking. The first argument should be the name of the cooking method. You can choose these names freely. The second argument must be the name of a field in the food table (this field will be a no-calc field). This field is called a reduce field. The third argument must be a list of names of fields in the food table. The meaning is, that when cooking by the given method, the values of all fields in the third argument should be reduced by the fraction, which is the value of the reduce field. If the reduce field has the value zero, there will be no reductions. If it is positive it should be less than or equal to one. If it is negative, the effect is to increase the values of the fields in the third argument.
Usually you will have more than one cooking method and for each cooking method you will usually have several "Cook:" commands, because you have a number of reduce fields in the food table, which each reduces different nutrients.

Weight cook: command

```
        +       weight cook: type-name weight-reduce-field field-list
```

The "Weight cook:" command is a variation of the "Cook:" command. The only difference is, that the second argument is not a reduce field but a weight reduce field. The weight reduce field should be in the food table and it should specify a fraction of the weight of the food. The first field in the field list will be reduced with an amount equal to the fraction of the weight of the food. Any other fields in the field list will be reduced with an fraction equal to the fraction the first field was reduced with. The first field in the list must be a food table field or a set field, any additional fields must be food table fields.
The "Weight cook:" command is normally only useful, when the first field in the field list is a macro nutrient (or ash or water). You can freely combine the use of "Cook:" and "Weight cook:" commands. If you use the "Weight cook:" command you must also use the "Food weight:" command.

Set: command

```
+       set: field = expression
```

The "Set:" command will make a new field with the name of field to the left of the "=" and value equal the expression. You can use expression with the following EBNF syntax:

```
Exp         ::=         Exp1  { +|-  Exp1 }
Exp1        ::=         Exp2  { *|/  Exp2 }
Exp2        ::=         [ - ] Exp3
Exp3        ::=         Number
            |           Field-name
            |           (  Exp  )
```

Any fields in the expression should be food table fields or previous set fields. The set fields will be calculated for each food that is calculated, and if the "Group by:" command is used it will be aggregated just like a nutrient fields. For each command it will be noted if a set field can be used as argument, but in general set fields can be used almost all places where a food table nutrient field can be used.

Group set: command

```
+       group set: field = expression
```

The "Group set:" command will make a new field very much like the "Set:" command, but if the "Group by:" is used the group set fields will be calculated after the group aggregations are done. This means you can use the "Group set:" command to calculate things like indexes and percents (e.g. percent of total energy coming from fat). Group set fields can only be used in the "Output fields:" command and in later "Group set:" commands.

Recipe set: command

```
+       recipe set: field = expression
```

The "Recipe set:" command will make a new field somewhat like the "Set:" command, but the field is calculated for each recipe ingredient and then the field is added to the food table (if the field already is in the food table its value will be changed to the value of the expression). Therefor recipe set fields can be used just like any other food table field.
Any fields in the expression can be food table fields, set fields or fields from the recipes fields.

Calculate: command

```
+       calculate: new-field additive-list
```

The "calculate:" command only exists for backward compatibility. You should now use the "set:" command!
You can use the "calculate:" command to make new fields, which will as values have the weighted sum of (possible reduced) values of fields in the food table. The first argument should be the name of the new field. The second argument should be a list of names of fields in the food table (or of fields previously defined with a "calculate:" command). The values of those fields will be summarized. You may enter a number before each field name in the list (but also separate the number and the name with a comma). The value of such a field will then be weighted with that number before it is added to the sum.

No-calc fields: command

```
      +       no-calc fields: field-list
```

The "No-calc fields:" command takes a list of names of fields in the food table as argument. The fields will be no-calc fields. Note, that many other commands has the side effect of making fields no-calc. Fields that are not no-calc are supposed to be nutrient fields. No-calc fields will not be weighted with amounts, will not be aggregated by the "Group by:" command and will not be reduced by cooking.

Text fields: command

```
      +       text fields: field-list
```

The "Text fields:" command takes a list of names of fields as argument. All fields with one of these names in the foods, groups, recipes and input files will be text fields. Text fields can have any value, but are totally ignored by FoodCalc. All fields which are not text fields can have only numeric values.

Input: command

```
      !       input: file-name [id-field amount-field sep-char dec-point-char com-
              char]
```

Tells FoodCalc to read the input file with the names given as the first argument. An input file is a data file, where each line gives the amount consumed of a given food. One of the fields must be the consumption id field. Consumption id values must be found in the food table, and this identifies the food consumed. Another field must be the amount field, which specifies the amount consumed. If the second and third arguments are given to the "Input:" command, they must be the names of the consumption id field and the amount field. Otherwise the first and the second fields in the file will be used.
FoodCalc will multiply the amount with the input scale and this value will be multiplied with all nutrient fields in the food table (nutrient fields are all fields that are not no-calc). After that any reductions will be made and any new fields calculated.

Input format: command

```
input format: text|bin-native
```

The "Input format:" command can be used to specify the format of the input file, and it can be used with either the keyword "text" or the keyword "bin-native" as argument. If the keyword text is used the input file should be a data file (however, see also the "Input fields:" command"). This is the default.
If the keyword bin-native is used, the "Input fields:" command must be used to specify the names of the fields. The input files must only contain the values, and these should be in the form of binary double precision floating point numbers without any end of line markers or any end of file markers. This format will be different on different platforms. If you generate the file with a C program, you should use variables of type double, and you should open the file with the "wb" mode. If you generate the file with SAS you should use a FILE command with "LRECL=(8*number-of-fields) RECFM=F", and you should put with the format RB8.
You should only use the bin-native input format when you have large amounts of data and want to improve the running time. FoodCalc can read a bin-native file somewhat faster than a data file.

Input fields: command

```
input fields: field-list
```

Normally the input file is a data file, where the names of the fields are in the first non-comment line in the file. Instead you can use the "Input fields:" command to name the fields. If you do that, you must not also have the names of the fields in the input file.
You must always use the "Input fields:" command if you use the "Save:" command or if you use the "Input format: bin-native" command.

Input *fields: command

```
input *fields: field-list
```

You can only use the "Input *fields:" command, when you also use the "Input fields:" command. When you use the "Input *fields:" command the input file should be in star format, except without the two field name lines. The fields in the "Input *fields:" command will be the star fields and the fields in the "Input fields:" command will be the non-star fields.

Input scale: command

```
input scale: scale
```

The argument to the "Input scale:" command should be a number, and that number is multiplied with all input amounts and all recipe amounts. If your food table has nutrient values pr. *n* units of food, and your amount fields uses the same unit, then you should use the value of 1/*n* as argument to the "Input scale:" command. If you do not use the "Input scale:" command the value one will be used.

Cook field: command

```
          cook field: cook-field type-list
```

The first argument to the "Cook field:" command should be the name of a field. If that field name is found in the input file, that input field will be a cook field. If the field name is found in a recipes file, that recipe field will be a cook field. You may have a cook field in just the input file or just in a recipe file. Or you can have a cook field in both the input file and the recipes file – in this case, the cook fields must have the same name!
The second argument must be a list of names of cooking methods. All these cooking methods must be defined with "Cook:" commands.
If a cook field has the value zero, it specifies that this food should not be cooked. Otherwise it must have a positive value less than or equal to the number of cooking methods in the second argument to the "Cook field:" command. The value then selects the cooking method which has the same ordinal number in the list as the value of the cook field. How the cooking is done must be defined with "Cook:" commands.

Reduce field: command

```
    +       reduce field: reduce-field field-list
```

The "Reduce field:" command is used to specify reductions of nutrients depending on values in the input file or in recipes files. The first argument should be the name of a field. If that field is found in the input file, that input field will be a reduce field. If the field name is found in a recipes file, that recipe field will be a reduce field. You may have a reduce field in just the input file or just in a recipe file. Or you can have a reduce field in both the input file and the recipes file – in this case, the reduce fields must have the same names! The second argument must be a list of names of food table fields. All the values of the food table fields will be reduced with the fraction that is the value of the reduce field. If the reduce field has the value zero there will be no reductions. If it is positive, it should be less than or equal to one. If it is negative, the effect is to increase the values of the food table fields. You can define as many reduce fields as you need.

Weight reduce field: command

```
    +       weight reduce field: weight-reduce-field field-list
```

The "Weight reduce field:" command is a variation of the "Reduce field:" command. The only difference is, that the second argument is not a reduce field but a weight reduce field. The weight reduce field should be in the input file and/or in the recipes files, and it should specify a fraction of the weight of the food. The first field in the field list will be reduced with an amount equal to the fraction of the weight of the food. Any other fields in the field list will be reduced with an fraction equal to the fraction the first field was reduced with. The first field in the list must be a food table

field or a set field, any additional fields must be food table fields.
The "Weight reduce field:" command is normally only useful, when the first field in the field list is a macro nutrient (or ash or water). If you use the "Weight reduce field:" command you must also use the "Food weight:" command.

Recipe reduce field: command

```
        +       recipe reduce field: reduce-field field-list
```

The "Recipe reduce field:" command is used to specify reductions of nutrients depending on values in recipes files. The first argument should be the name of a field. If the field name is found in a recipes file, that recipe field will be a reduce field. The recipe reduce field should have the same values for all the ingredients in a recipe. If you use the star format, you should make the reduce field a star field.
The second argument must be a list of names of food table fields. For all ingredients in a recipe all the values of the food table fields will be reduced with the fraction that is the value of the reduce field. If the reduce field has the value zero there will be no reductions. If it is positive, it should be less than or equal to one. If it is negative, the effect is to increase the values of the food table fields.
The "Recipe reduce field:" command is actually just an alias for the "Reduce field:" command.

Recipe weight reduce field: command

```
        +       recipe weight reduce field: weight-reduce-field field-list
```

The "Recipe weight reduce field:" command is a variation of the "Recipe reduce field:" command. The only difference is, that the second argument is not a reduce field but a weight reduce field. The weight reduce field should be in the recipes files, and it should specify a fraction of the weight of the food. The recipe weight reduce field should have the same values for all the ingredients in a recipe. If you use the star format, you should make the weight reduce field a star field.
The first field in the list of nutrient fields in the command are summarized for all the ingredients (let us call this n-sum), and also the weight of all the ingredients are summarized (let us call this w-sum). If we call the value of the reduce field for r, FoodCalc then calculates a fraction to reduce by: (r * w-sum) / n-sum. All the nutrient fields in the command are then reduced by this fraction for all the ingredients in the recipe.
The first field in the list must be a food table field or a set field, any additional fields must be food table fields. The "Recipe weight reduce field:" command is normally only useful, when the first field in the field list is a macro nutrient (or ash or water). If you use the "recipe weight reduce field:" command you must also use the "Food weight:" command.

Non-edible field: command

```
            non-edible field: non-edible-field [flag-field]
```

The food table (the foods and groups files) should contain nutrient values for only the edible part of foods, and the input and recipes fields should have amount fields specifying the amount eaten, except if you use the "Non-edible field:" command.

The first argument to the "Non-edible field:" command should be the name of a field in the food table, which as value has the non-edible fraction of the foods, as they are normally purchased.
If the second argument is not specified, then all amount fields in input and recipes files should contain the total weights (the weight of both the edible and non-edible parts).
If the second argument is specified, it should be a field name. If this field is *not* in an input or recipes file, then all the amounts in this file should be the weight of the edible part. If the field is in a file, then it should have the value zero or one. If it is zero, the amount should be the weight of the edible part. If the value is one, the amount should be the total weight of that food (the weight of both the edible and non-edible parts).

Output: command

```
!       output: file-name [sep-char dec-point-char]
```

Tells FoodCalc to write the output file with the name given as the first argument. The output file is a data file (however, se also the "Output format:" command), where each line will list the nutrients consumed. The second and third argument can be used to specify the separator and decimal point characters.
If you do not use the "Group by:" command, there will be one output line for each input line. If you do use group by, there will be one output line for each (sub)group.

Output format: command

```
output format: text|text-no-head|bin-native
```

The "Output format:" command can be used to specify the format of the output file, and it can be used with one of the keywords "text", "text-no-head" or "bin-native". If the keyword text is used the output file will be a data file. This is the default. If the keyword text-no-head is used, the output file will be like a data file, but without a line with the names of the fields.
If the keyword bin-native is used, the output file will only contain the values in the form of binary double precision floating point numbers without any end of line markers or any end of file marker. This format will be different on different platforms. If you read the resulting file with a C program, you should use variables of type double, and you should open the file with the "rb" mode. If you read the file with SAS, you should use an INFILE command with "LRECL=(8*number-of-fields) RECFM=F", and you should get with the informat RB8.
You should only use the bin-native output format when you have large amounts of data and want to improve the running time. FoodCalc can write a bin-native file somewhat faster than a data file.

Output fields: command

```
output fields: field-list
```

This command specifies which fields are written to the output file. If you do not use the "Group by:" command, you can specify any input field, any food table field, any set field and any group set field. If you do use the "Group by:" command, you can specify only nutrient table fields (fields that are not

no-calc), set fields, group set fields, and group by fields. Any input field or no-calc field will be written just as it was read. Nutrient fields will be weighted with the amount and the input scale, possibly reduced by reduce fields, and possibly summarized by a group by.
If you do not use the "Output fields:" command and do not use the "Group by:" command, the default will be to output all fields from the input file, all fields from the food table, all set fields, and all group set fields. If you do not use the "Output fields:" command but do use the "Group by:" command, the default will be to output all the group by fields, all the nutrient fields, all set fields, and all the group set fields.
You should only output the fields you need, because FoodCalc runs faster and uses less memory when fever fields are output.

Group by: command

```
group by: field-list
```

Tells FoodCalc to group the foods according to the values of one or more group by fields, and to summarize the nutrients in each group. The argument to the "Group by:" command should be a list of field names of fields in either the input file or the food table. These fields are called group by fields and they will be no-calc fields. If the first one or more fields in the field-list are input fields, the input file must be sorted ascending on these fields. Only the integer part of the values of these fields will be used.
Each nutrient field will be summarized for all foods where all the group by fields has the same values.

Transpose: command

```
transpose: field number field-list
```

Tells FoodCalc to output the fields in the field-list transposed on the field in the first argument. The "Transpose:" command can only be used when the "Group by:" is also used. The field in the first argument it the field transposed on and should be a field from the food table and it will be a no-calc field.
Each field in the field-list will be transposed and should be a field in the food table or a set field. For each of these fields will be output *number* fields as specified in the second argument. The output fields will be named the same as the transposed field but with the numbers one to *number* as suffix. To the output field with number *n* will be added the value of the transposed field when the integer part of the value of the field transposed on equals *n*. If the field transposed on does not have a value >= one and <= *n*, then the transposed field will not be added to any of the output fields.

Where: command

```
where: logical-expr
```

The argument to the "Where:" command should be a logical expression with the following EBNF syntax:

```
Lexp         ::=        Lexp1 { or Lexp1 }
Lexp1        ::=        Lexp2 { and Lexp2 }
Lexp2        ::=        [ not ] Lexp3
Lexp3        ::=        Val  =|<>  Val
             |          Val  >|>=  Val  [ >|>=  Val]
             |          Val  <|<=  Val  [ <|<=  Val]
             |          Val  [ not ]  in ( Val { , Val } )
             |          ( Lexp )
Val          ::=        Field-name  |  Number
```

Any fields in the logical expression must be food table fields or set field. The command selects only foods where the logical expression evaluate to true.

If: command

```
+        if: field value-list
```

The "if:" command only exists for backward compatibility. You should now use the "where:" command!
The first argument to the "if:" command should be the name of a field in the food table. This field will be a no-calc field. The second argument should be a list of numbers. The command selects only foods where the value of the field equals one of the values in the second argument. If more than one "if:" command are used, a food is selected if just one of the commands select the food.

If not: command

```
+        if not: field value-list
```

The "if not:" command only exists for backward compatibility. You should now use the "where:" command!
The first argument to the "if not:" command should be the name of a field in the food table. This field will be a no-calc field. The second argument should be a list of numbers. The command deselects foods where the value of the field equals one of the values in the second argument. If more than one "if not:" command are used, a food is deselected if just one of the commands deselects the food.

Version history

Version	Comment
v1.0	Initial version
v1.1	Treats carriage return as blank space. This allows us to read DOS/Windows files on UNIX systems. Fixes a few bugs.

v1.2 Allow comments at end of lines in data and commands files.
Changeable comment character in data files. New "Comment:" command.
Continuation lines (=) in data and commands files.
Text fields in data files. New "Text fields:" command.
Star format in data files. New "Input *fields:" command.
Recipe fields with same name as non-calc fields in the food table will now be copied to the food table when Ingredients: sum is used.
New "Recipe weight reduce field:" command.
New "Recipe reduce field:" command is an alias for the "Reduce field:" command.
The "recipe sum:" command is now called "Food weight:".
New "Weight cook:" and "Weight reduce field:" commands.
New "Non-edible field:" command.
New -v option and new "Verbosity:" command.
New -l, -i and -o options. The -s option now takes only one argument.

v1.3 Added new "Set:", "Group set:" and "Recipe set:" command to do more general calculations at different points of the FoodCalc processing. The "calculate:" command is now obsolete.
Added new "Where:" command to make more general selections of foods. The "if:" and "if not:" commands are now obsolete.
Added new "Transpose:" command to transpose nutrient fields on a field expressing some grouping of the foods. This is often more handy than doing a "Group by:" on a food table field.
Most limitations of the "Group by:" command has been lifted. You can now group on several food table fields and the food table fields do not have to be last in the list of group by fields.
Groups files can now be identified by more than one field. For example by main group and sub group.
New keepx argument to the "Ingredients:" command keeps ingredients like the keep argument but moves values from the recipes file to the food table like the sum argument does.