Notater
Documents
19/2012

Statistics Norway
Statistisk sentralbyrå

Anne Abelsæth

Tutorial: Development of data entry- and CAPI applications in CSPro

Documents 19/2012

Anne Abelsæth

Tutorial: Development of data entry- and CAPI applications in CSPro

Statistisk sentralbyrå • Statistics Norway
Oslo–Kongsvinger

Documents In this series, documentation, method descriptions, model descriptions and standards are published.

ISBN 978-82-537-8339-0 Printed version
ISBN 978-82-537-8340-6 Electronic version
ISSN 1891-5906
Subject:
Published March 2012

Print: Statistics Norway

Symbols in tables	Symbol
Category not applicable	.
Data not available	..
Data not yet available	…
Not for publication	:
Nil	-
Less than 0.5 of unit employed	0
Less than 0.05 of unit employed	0.0
Provisional or preliminary figure	*
Break in the homogeneity of a vertical series	—
Break in the homogeneity of a horizontal series	|
Decimal punctuation mark	.

Preface

Statistics Norway has for more than 15 years been involved in institutional development cooperation with sister-organisations. This work is based upon the professional competence and experience gained both in Norway and through the cooperation with several partners over these years. But in order to stay in front of statistical development we are trying to move ahead by developing and documenting new methods.

Over the last years we and our partners have moved ahead from the old key punching to optical reading and scanning as well as various types of CAPI- and CATI-systems. With the current development of small devices such as e-pads we have met an increasing demand for a new step in data-entry. But rather than embarking upon tailor-made systems we have been waiting for the common data-entry systems for national statistical institutes to be adapted to the new devices and operating systems.

However we still find a lack of text books and tutorials for CSPro. The almost 500 pages Users Guide and the 40 pages Getting Started document do serve well for expert users, but in order to build capacity within the statistical community we do think a tutorial is highly needed. Hence we have so far prepared a tutorial for the Data Entry and CAPI parts of CSPro and we hope to expand the document with new chapters. Hence we recommend checking www.ssb.no whether versions with a further coverage may be available.

We would like to thank Mr. Greg Martin at US Bureau of Census for extensive comments and help. However, the US Bureau of Census is not responsible for any possible mistakes and shortcomings, they are the full responsibility of Statistics Norway.

Since this is a document in development we would be very happy to receive comments and suggestions about this tutorial. Please address these to Anne.Abelsaeth@ssb.no

Abstract

This is a tutorial for the software CSPro, covering how to building Data entry and CAPI applications (Computer Assisted Personal Interview). Using a typical census questionnaire as a case study, we will learn how to create the dictionary, how to design the forms, and how to code the logic like skips and checks for the questionnaires.

Contents

1. Introduction

CSPro is a free software package used by hundreds of organizations and tens of thousands of individuals for entering, editing, tabulating, and disseminating census and survey data. CSPro is designed to be as user-friendly and easy to use as possible, yet powerful enough to handle the most complex applications. It can be used by a wide range of people, from non-technical staff assistants to senior demographers and programmers. The funding for CSPro comes from USAID.

CSPro is used by different institutions and organisations who do surveys. This includes National Statistical Offices, NGOs, Universities, Hospitals and businesses. It can be downloaded from the website of the U.S. Census Bureau (See below for URL).

The most common kinds of surveys in which CSPro is used are censuses (population and housing; agriculture; and economic), Demographic and labour force surveys, Household income and expenditure surveys, etc.

This document is meant as an introductionary tutorial for developing CAPI questionnaires (Computer Assisted Personal Interview) and data entry applications in CSPro, and does not cover all sides of the software package. For a complete reference, please see the CSPro manual which can be downloaded from the U.S. Census Bureau's website. This tutorial will cover the whole process of developing a CAPI questionnaire. This includes

- Defining the metadata for the survey as a CSPro dictionary (containing information about the questions, what kind of data will the answer be, name of the variable holding the data etc.)
- Defining the code lists (possible replies) to the given questions (for instance, a question about the respondent's sex can have replies "1 = male" and "2 = female")
- Designing the forms for the questionnaire.
- Programming the skips and controls and other logic of the questionnaire.
- Deploying the application on a laptop, tablet PC or similar.

The main areas of CSPro not covered in this tutorial are tabulation and batch editing data.

1.1. About this tutorial

This tutorial will cover the whole process of developing a CAPI questionnaire. This includes
The layout of the tutorial is in several parts: First the most common tasks when using CSPro are exemplified in the development of a simple census questionnaire, and some additional issues are discussed in the end of each chapter. The second part covers more complex, but still frequently used subjects.

The structure is the following:

- Development of the dictionary of the example application
- Discussion of topics about dictionaries not covered by the example
- Creation of forms for the example application
- Further discussion about forms
- Programming the logic of the example
- Further discussion about programming
- Miscellaneous topics
- Appendices

1.2. Useful links and addresses

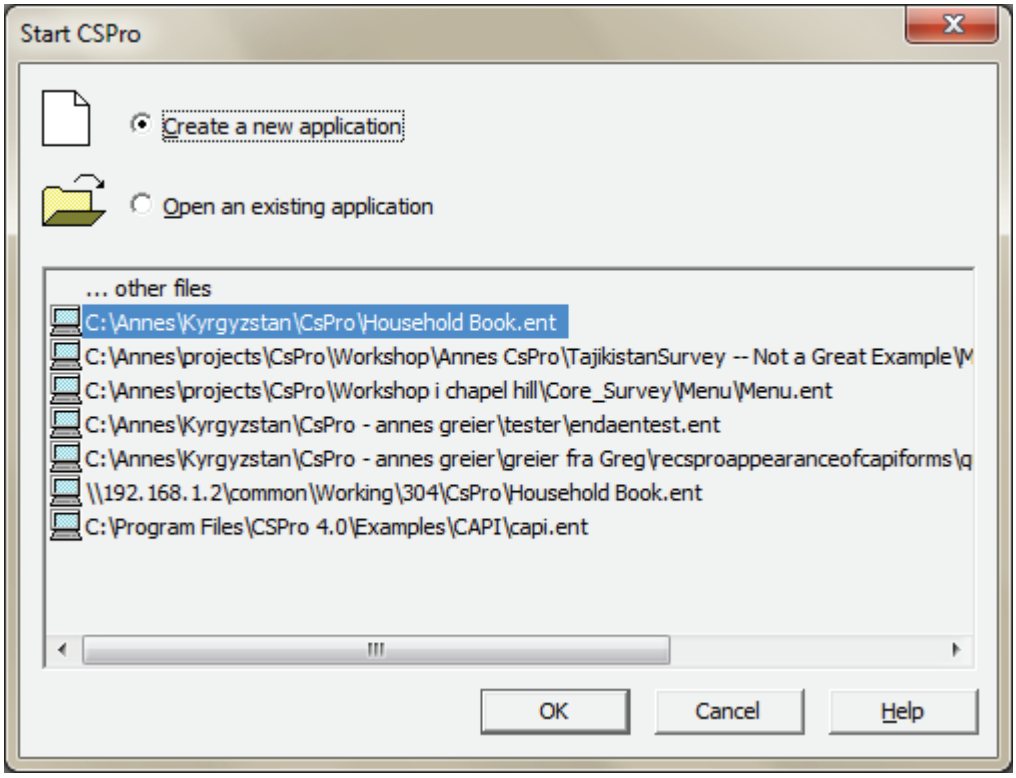
- U.S. Census Bureau website: http://www.census.gov/ipc/www/cspro
- CSPro Users website: http://www.csprousers.org
- CSPro on Twitter: http://twitter.com/cspro
- Mailing list for CSPro questions: cspro@lists.census.gov

2. Creation of the example application

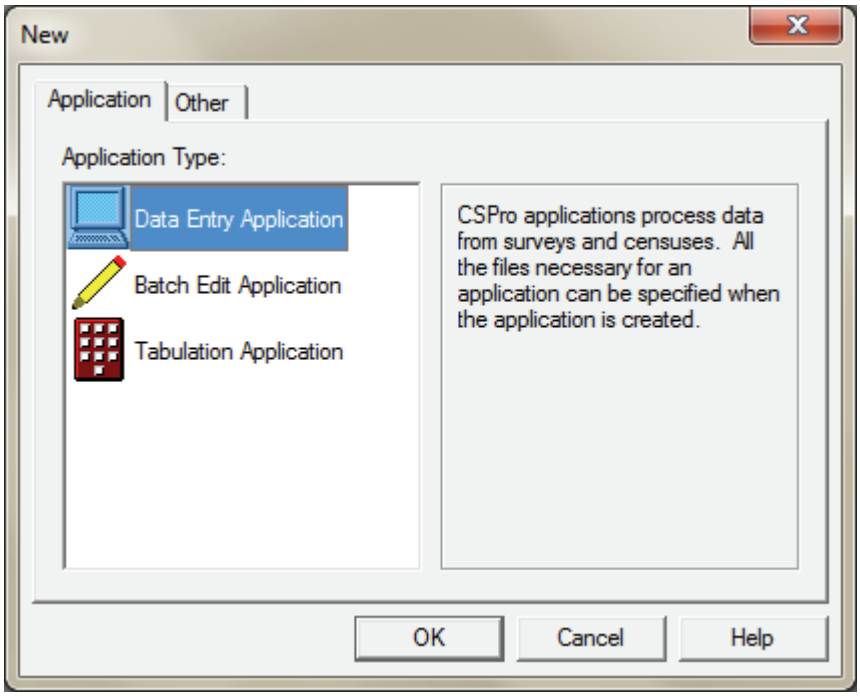
We will use the questionnaire in the appendix B to create our CAPI application. The layout of the questionnaire is fairly common for a household survey: One part for identifying the household, one part with questions for the household as a whole, and then the third part for each of the members of the household.

We assume that you already have installed CSPro. If not, download it from the US Census Bureau website, and follow their instructions to download and install.

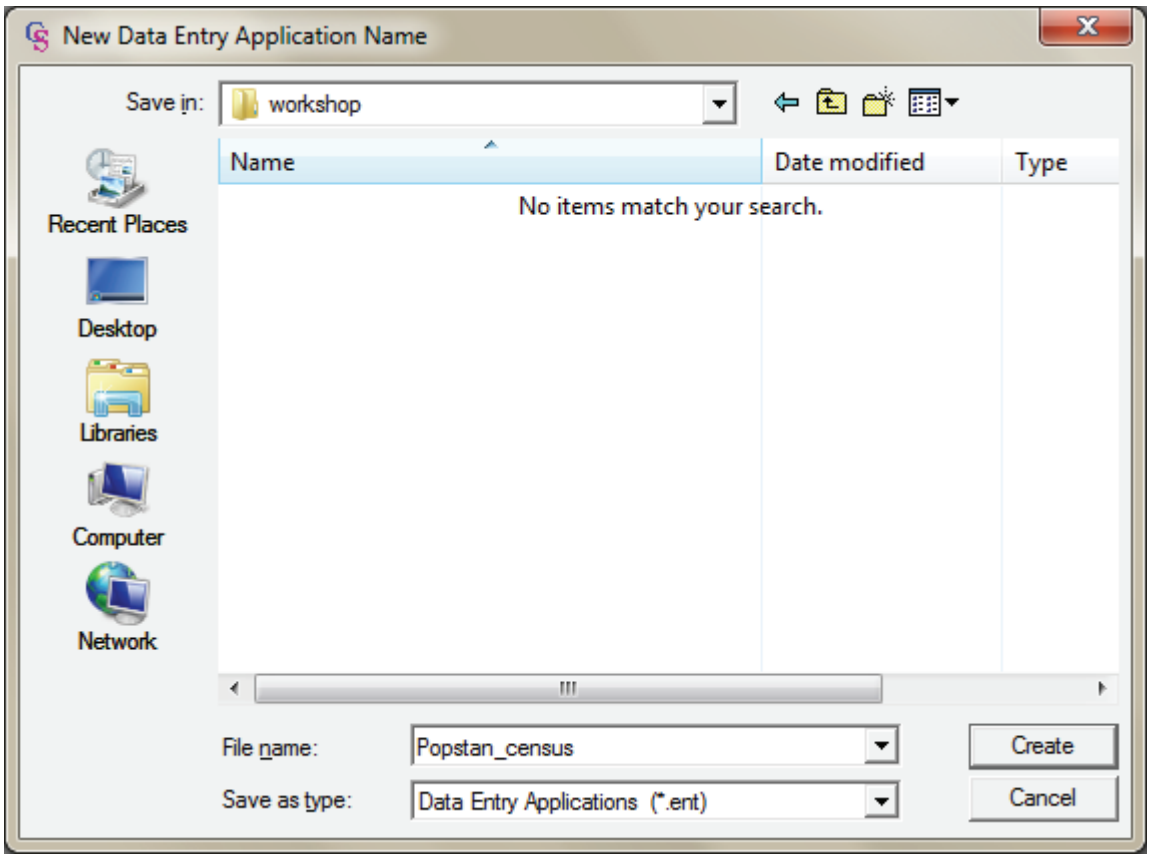
To create a new application, please start up CSPro, and tick “Create a new application”, then “OK”:



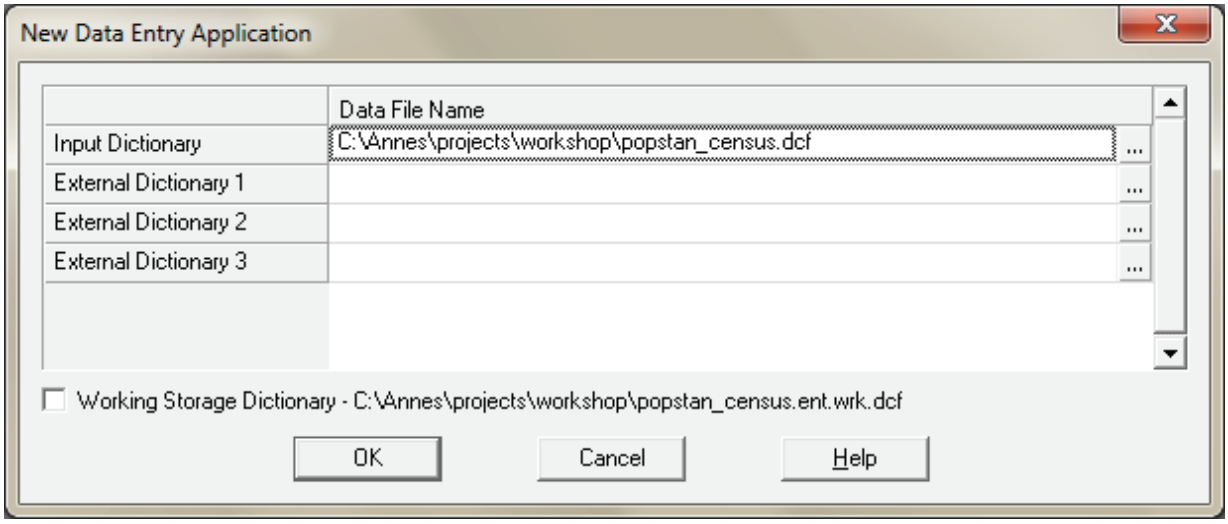
You then get a question about what kind of application you want to create. Select the default “Data Entry Application” and click “OK” again.



Then you have to give the application a location and a name (we chose the name popstan_census, and put it in a workshop folder somewhere convenient on the hard drive)

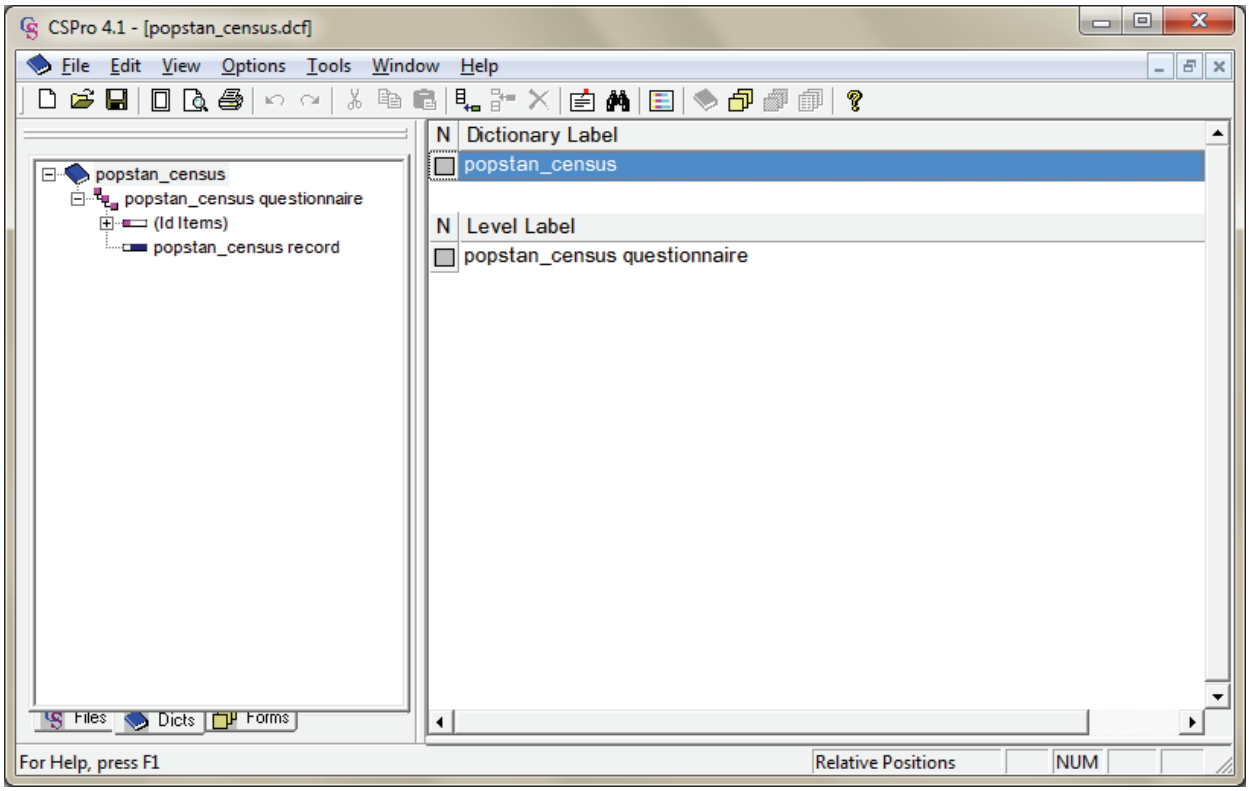


Finally CSPro asks for the input dictionary. The default suggestion is a dictionary in the same folder and with the same name as the the application itself, only with the extension .dcf instead of .ent. (See chapter 9.2 for more about file extensions)



One application can have several dictionaries. We will talk more about this in chapter 9.3. For now, accept the default dictionary and click “OK”. CSPro will then tell you that the dictionary does not exist, and ask you if you want to create it. Click “yes” to this.

CSPro has now created a skeleton for your application, which you can see in the left part of the CSPro window:



In the above picture, the first line on the left (having a blue book as the icon) is the dictionary itself. The next line is the questionnaire (or rather the level. An application can have more than one level). Then the ID of the record comes (this is the (Id items) line), and finally a record that CSPro has created for us named popstan_census record.

3. Creating the dictionary of Popstan Census Questionnaire

In CSPro jargon, a dictionary is the place to store the questions and the metadata about the questions of a questionnaire: For each question we want the interviewer to ask (or the operator to enter in data entry applications), we need to decide

- What kind of data is to be entered?
- How many positions are needed for storing the data?
- How to name the question?
- What label should be put on the question?
- If the field is numeric: Do we need decimals?
- What are the possible replies to a given question?
- …?

The dictionary is also important for deciding the flow of the data entry application:

- What parts of the questionnaire should be grouped together?
- Are there parts of the questionnaire that are to be repeated (e.g. one set of questions for the household, and one set of questions for each of the members of the household)?
- What variables do we want to use for identification of the different cases?
- …?

Every CSPro application needs at least one dictionary, but multiple applications can share the same dictionary, and a dictionary can exist without an application.

It is important to think through the flow and layout of the application at this stage. In our example questionnaire, it makes sense to have one record type containing the information about the household, and one record type containing information about the members of the household. And as a household can have many members, the latter record type has to be repeated several times, one for each member. But first we should decide what should be the identification elements.

3.1. Identification elements

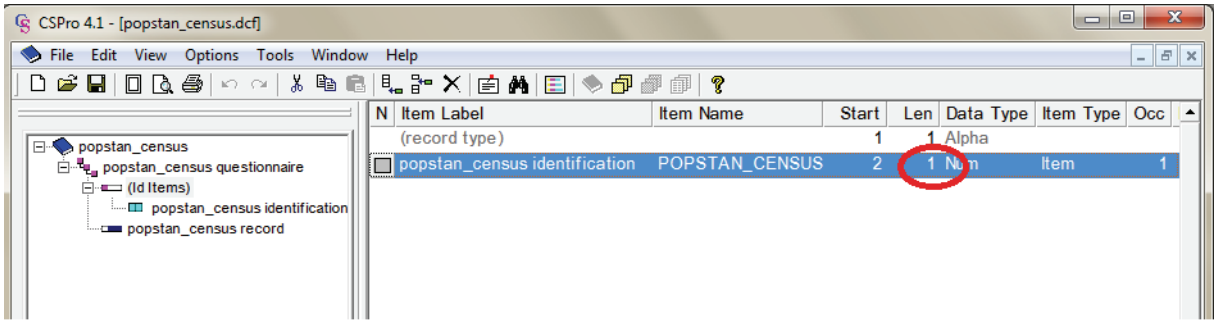
The identification elements are needed to uniquely identify the records or households in our survey. It is a good principle to keep the number of elements of the identification to a minimum.

In household surveys or censuses, the ID almost always consists of geographical data, for instance a combination of province, district, details about location within the district, and maybe a household number in the end. In other kinds of surveys there might be other indicators that uniquely identify the records depending on what kind of data to have entered.

Consider our example questionnaire (appendix B). The candidates for the identification elements here are to be found top right on the page: The items "Province", "District", "Village", "EA" (Enumeration Area) and "Housing unit number" together uniquely identify the household.

3.1.1. Creating the ID elements

Expanding the (Id Items) in CSPro (click on the plus sign in the beginning of the line on the left side of the display), will reveal that CSPro has already created an id item for you, as seen in the following picture:



This Id Item is generally not very useable, however, as it only has length 1 (the red circle), meaning that the survey only can have 10 records, identified by the numbers 0 to 9. We start creating ID items by deleting the one that already exists. This is done by right-clicking on it in the left pane and choose "delete item".

To create our own Id items do one of the following:

- Right click in the right panel on the "(Id Items)", and choose "Add item",
- or select "Edit" – "Add item" from the menu.

CSPro then wants information about the item. In our case the first id item is province which has two digits in the paper version of the questionnaire. Enter the following data:

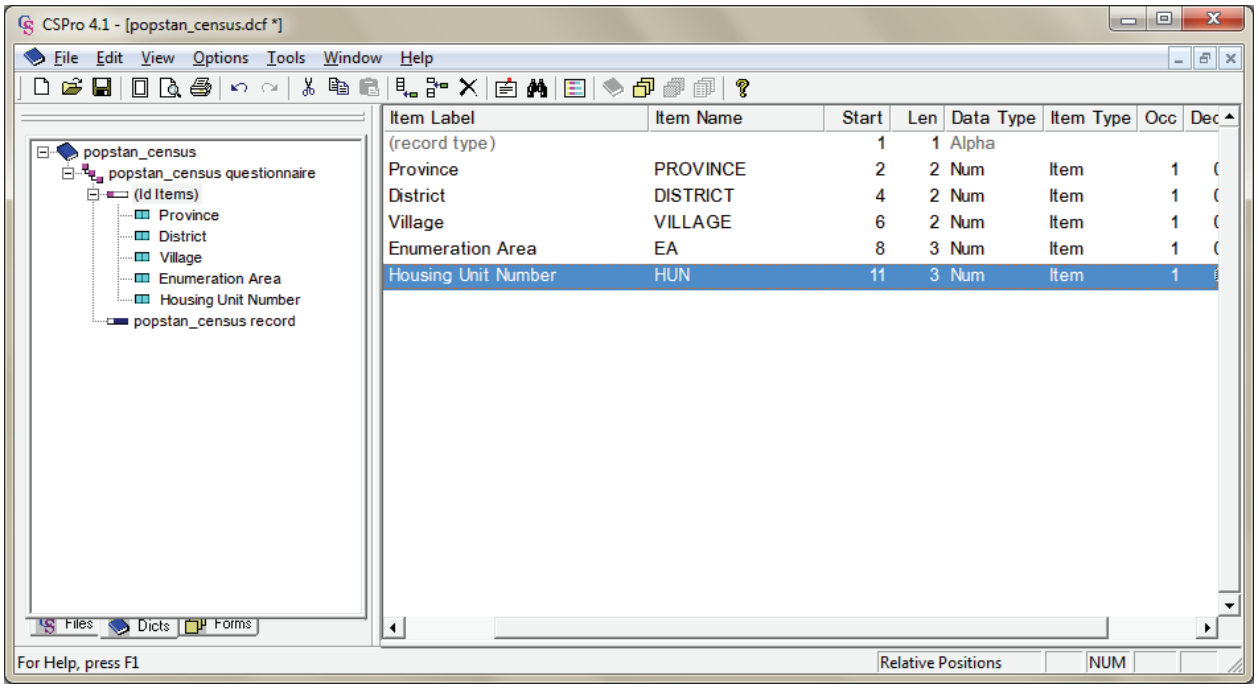
- Label field: province
- Name- and start fields: Accept CSPro defaults (PROVINCE and 2 respectively)
- Len: 2 (This is the number of digits – or length of the item)

The rest of the default values are ok, so to finish working on this item by hitting the tab button repeatedly, or by pressing control-enter.

We need more id items than just the province. Next ones are district and village. Create these just as you created the province item.

Then we need the EA (enumeration area) and housing unit number. These can be created as the former items, with the difference that length should be 3 instead of 2. We could also consider giving them different labels and names, say “Enumeration area” and “EA” for the former one, and “Housing unit number” and “HUN” for the latter.

To stop entering more items, press esc on a new line. When all the id items have been entered, your application should look like this:



3.1.2. Questions about the total number of persons in the Household

On the paper questionnaire, right under the identification items are questions about the total number of persons in the household, and also about number of males and females. We choose to let CSPro count the persons rather than having the interviewer asking these questions. Hence we do not include the questions in the dictionary.

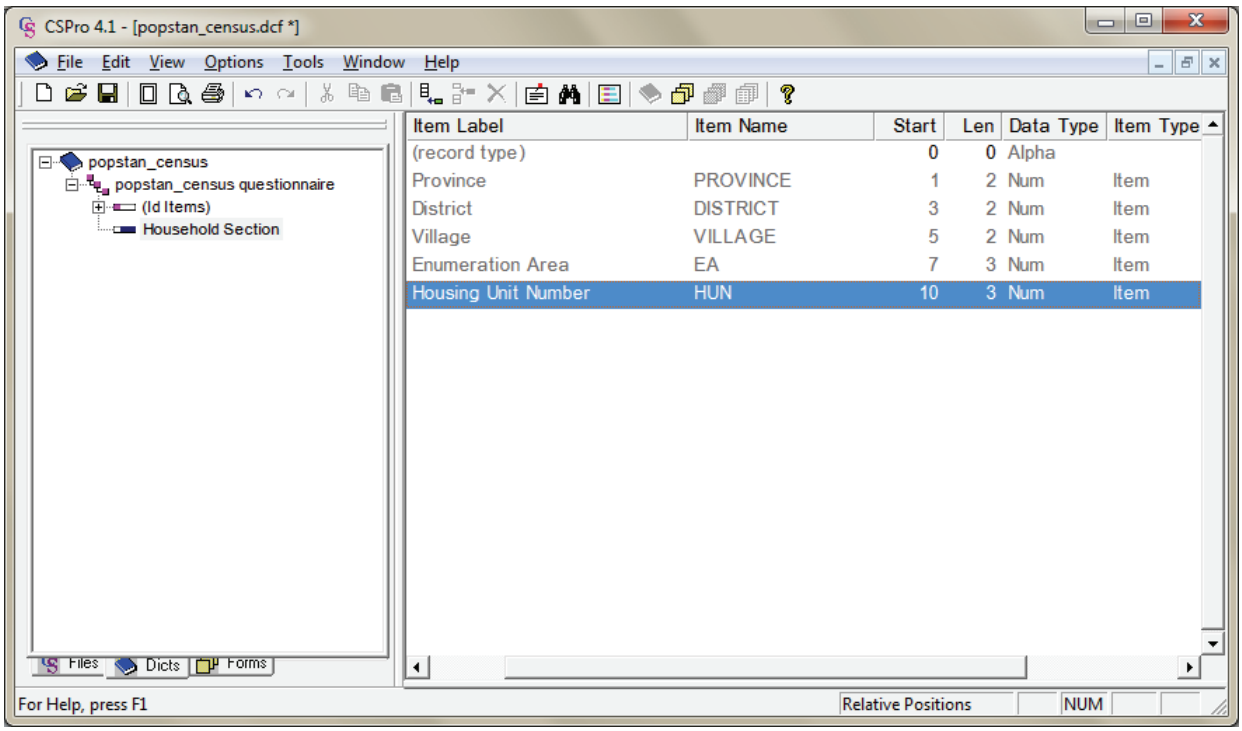
An alternative approach could have been to ask about the total number of persons in the household, and then loop through one person at the time.

And of course we do not need sheet number and total number of sheets found in the paper version, as there are no sheets in the electronic version.

3.2. The Household record

Now that we have our id items, we can work on the household record. The questions here are to be asked once for each household in the survey. Before we start, we should delete the record CSPro created for us by right-clicking on the “popstan_census record” and select “delete record”.

Create a new record by right-clicking on the “popstan_census questionnaire”, and select “Add record”. CSPro is again going to ask you for details about the record. Enter for instance “Household Section” and “HOUSEHOLD” as label and name respectively, and leave the other fields as they are, as the defaults are OK in this case. Your application in CSPro should now look something like this:



(The record we just added is still empty of content, but you can see it in the left pane. The ID items in the right pane will always be visible).

Now that we have the household record, we can start entering questions to it. After this is done, we shall enter the value sets, defining what values the enumerator is allowed to enter for each question.

3.2.1. A note on item labels and item names

There are different standards on how to name variables, and this topic is important for two reasons: Firstly the name of the variables is used for referring to the element when we program the logic of the questionnaire. Choosing good names makes this task easier. And secondly the variable names serve as documentation for the data file after the survey is completed, which of course is very important.

A name of a dictionary item has to be a single word (i.e. no spaces is allowed), it can be up to 32 characters long, and it has to start with a letter. All the names inside of one single dictionary have to be unique and cannot conflict with a CSPro reserved word (see the list in the CSPro manual).

The label is a longer description of the element. It can be up to 255 characters long, and it can contain spaces.

(When doing CAPI surveys, there is one additional field to add text relying to the question: The “CAPI question”. This is only one field, but it uses two different fonts; one containing the text that the enumerator is to read out loud while doing the interview, and the other one for giving the enumerator additional instructions if needed. This means that the label field in the dictionary is not necessary for the interviewer – only for documenting the data. We shall talk more about CAPI questions in section 5.4)

There are three common approaches for naming variables in CSPro if the questionnaire has numbered questions:

- The variable name contains both number and content: P02_NAME, P03_RELATION, P04_SEX
- The variable name is only about the content of the question: NAME, RELATION, SEX
- The variable name is the same as the question number: P02, P03, P04

We recommend using the third approach – to name the variables with the question numbers. This ensures very good documentation as long as the questionnaire is available with the data (which it should be). In addition it is short and easy to use when programming.

In addition, we recommend using labels containing both question number and a (very) brief description of the content.

3.2.2. Back to the household record

There are two ways to enter questions to the dictionary. Either right click somewhere in the right pane and select "add item", or select "edit – add item" from the menu.

The first question we want to add is "What is the type of this housing unit?" and it is numbered H01, so enter the following details for this question:

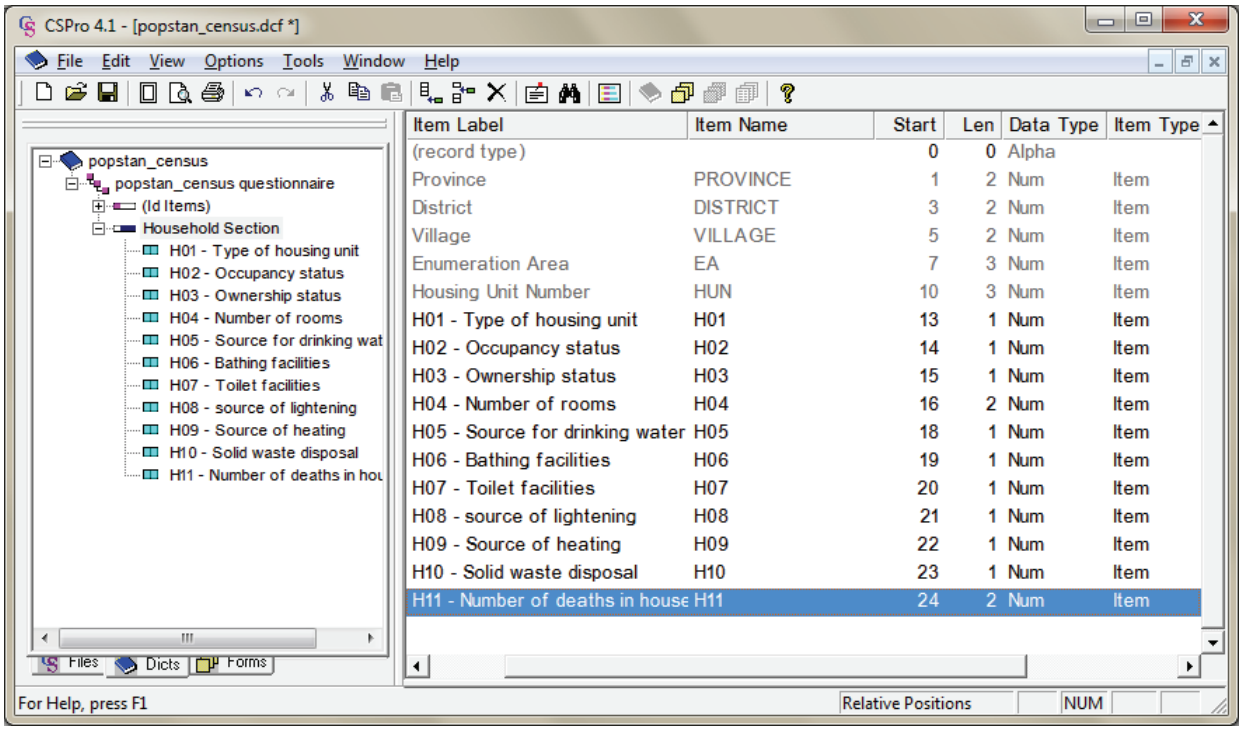
- **Item label:** H01 – Type of housing unit
- **Item name:** H01

In this case, the rest of the details should be left as CSPro suggests as defaults, so hit tab repeatedly, or press control-enter to finish the item.

However, the rest of the fields have the following meanings:

- **Start:** This is the start position of the variable in the data file. Deciding the start position should almost always be left to CSPro, as it is very easy to mess things up. The exception is if you are reading from an external file with spaces between the fields.
- **Len:** This is the length of the variable. In the question above, there are only 4 possible replies, so length = 1 (default) is correct.
- **Data type:** Can either be num (for numeric values), or alpha (for text values). Alpha should only be used when the input is a text, for instance the name of a person. If the input is one of a set number of possible answers, numeric should be used, and the code list should be declared as a value set (see paragraph 3.3)
- **Item type:** This field can either be Item or Subitem. We will talk more about subitems in chapter 3.4.2.1
- **Occ:** Defines the number of consecutive repetitions – or occurrences - of the item in the record. We will talk more about this in chapters 5.7 and 6.1.
- **Dec:** Defines the number of decimal places (if any) in the item.
- **Dec Char:** if set to "yes", the data will be stored in the data file with an explicit decimal character. This only applies to items where Dec is greater than 0.
- **Zero Fill:** states whether the numeric data item should contain leading zeroes or blanks.

Continue to add items from the household section in the questionnaire the same way: One for each question until your application looks like this:



3.3. Adding value sets to the household questions

In the paper version of the questionnaire, the enumerator hardly ever writes any text in the question fields. He or she is instead supposed to code the possible answer to a question to numbers according to the code list in the question fields. We want the same in the electronic questionnaire, so we have to define what numbers to use for each of the possible replies. These code lists are called Value sets, and each question can have zero, one or several value sets associated with it. Value sets can also be defined programmatically (more about that in 8.6).

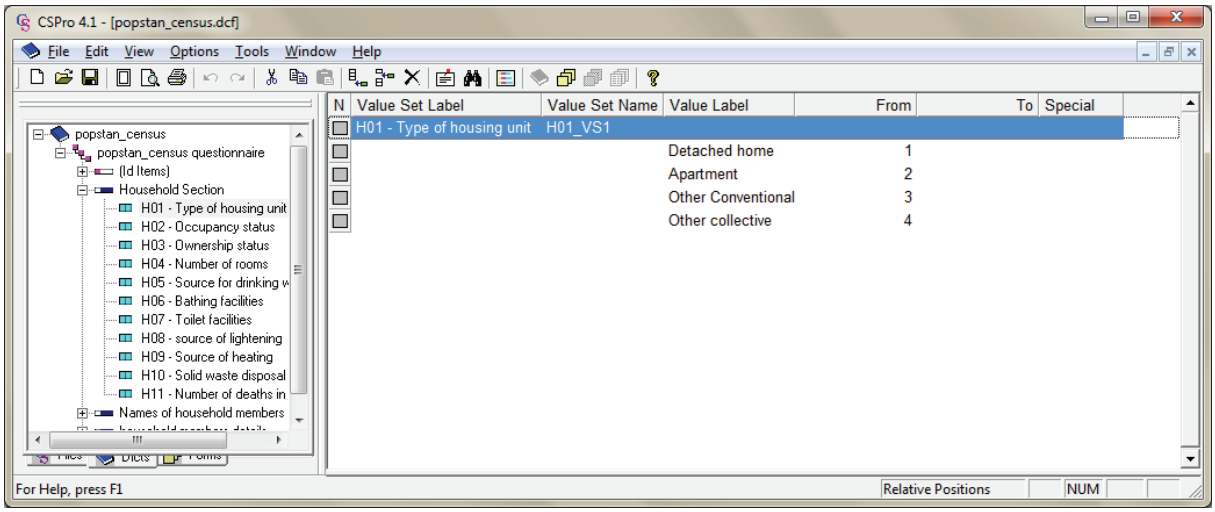
To add the value set of the first question: Highlight the item in the left pane, right click and select "Add value set".

We want to create a value set having the following values:

1 Detached home
2 Apartment
3 Other Conventional
4 Other Collective

Keep the default values for "Value set label" and "value set name", and press tab or enter until you reach the field "Value label". Here you enter "Detached home", and in the "From" field, enter 1. Leave the "To" field empty, as there are no intervals in this value set.

Continue entering the value label and from fields for each of the possible types of housing units. When you are finished, it should look like this:



Value sets are done the same way for all of the questions of the household section, except for the questions about number of rooms and number of deaths in the household. These questions should not have any value sets.

3.4. The population record

The population record is going to be a repeating record, as there can be several people living in one household, so we first have to decide what is the maximum number of members in a household. At first sight, it looks like the paper version of the questionnaire only allows 10 members, but have a closer look at the column containing the person number of the household (the very first column): There is a blank field before the numbers 1, 2, .., 0. This is there to make it possible to use several questionnaires for one household: The enumerator just adds the number of the extra questionnaires in front of the existing numbers.

In the upper right corner of the questionnaire the enumerator is supposed to enter how many sheets he used for the household, and this field allows a two digit number, so each household can have a very large number of members in the paper version of the questionnaire.

Let us limit the number of members of a household to 30 people. This should be plenty for all "normal" households.

Then we have to decide how to structure the record(s). There are two possibilities: Either we make a big matrix looking more or less like the matrix in the paper version of the questionnaire, or we split the record in two, asking first for all the names of the people in the household in a smaller matrix, and then loop through each person asking the rest of the questions repeatedly.

As this manual focuses on CAPI interviewing rather than traditional data entry, the first option is not good because of the small data screens of the tools commonly used for interviewing: The screen would be overpopulated by input fields, and scrollbars both vertically and horizontally would be needed, which is not good in a CAPI interviewing setting.

Splitting the population record in two records requires some programming to get things to work. We will get back to how to do it later.

3.4.1. Creating the record holding the names of the household members

Add a new record by right clicking on the "popstan_census questionnaire" level in the left pane and select "Add record" as we did earlier. Enter Record label and record name, and let CSPro handle the Type value as earlier. When considering the "required" field: Remember the second question of the household section about occupancy status of the unit. If this status is "vacant", no people live in the house, so we have to select "no" for required.

The Max field defines the maximum number of times the record will appear in the questionnaire. Enter 30 here.

Then we need to add the items of the record: We do not need a field for the person number, as CSPro manages to keep track of this for us. The only two items we need are first name (data type "alpha" with length 20) and last name (also "alpha" of length 20).
(In the questionnaire, the first and last names are in the same field, but we prefer to split them in two).

3.4.2. Creating the record containing details of the members of the household

We then need to make a record containing the detailed information about the members of the household too. Create the record just like the former one, and add the items for question numbers P03, P04 and P05 as explained earlier.

3.4.2.1. The Date of birth item – using subitems

In the paper version of the questionnaire, the birth date is split into day, month and year. In an electronic version, it often comes handy to have dates simultaneously as only one item – the full date, and at the same time split up in parts. We can achieve this by using *subitems***.**

First create the day, month and year items as explained earlier for the other items. Then highlight the three items just created, right click and select "Convert to subitems". CSPro will then create an item above the parts of the birthdate where you can define label and name of the item.

The "main" item automatically gets the data type "Alpha". This is to avoid problems i

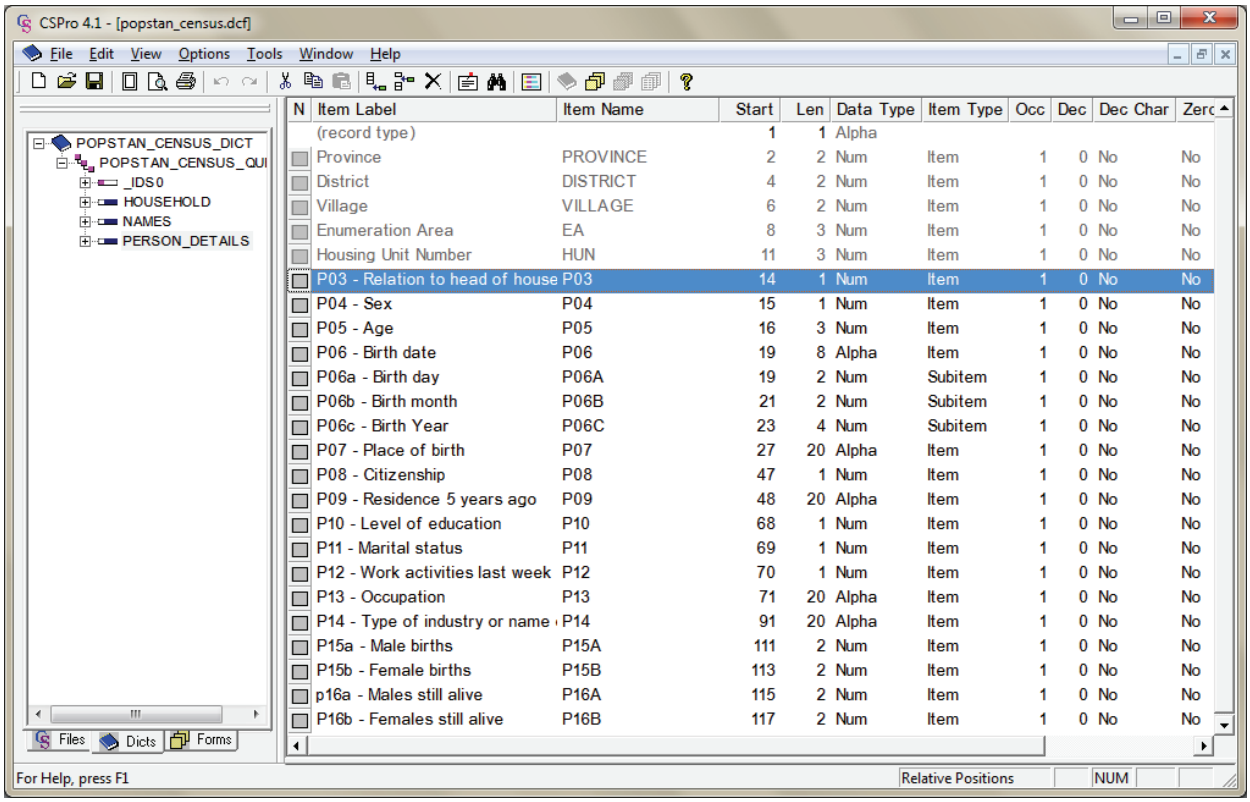
3.4.2.2. Alpha items vs. code lists – the rest of the member record

There still are some issues to consider when designing the dictionary for this questionnaire:

- What is the best way to deal with question P07 and P09? Alpha fields like in the paper questionnaire require more time to enter, and the probability for errors is high.
  Maybe a better way to do this is to have the provinces and names of countries as a value set to choose from. Or if the list of countries is long, make it two fields: one for provinces and one for countries.
- How about question P13: If we manage to find the right detail level for the value set, this field too is better to have as numeric field.
- Question P14: This question should be two questions, not one: One asking about type of industry (again with a code list for the legal values), and one asking about the name of the employer.
- Question P15 and P16 – consider why it is not a good idea to make subitems into P15a, p15b, p16a and p16b.

All of the bullet points above are about questionnaire design rather than CSPro, so we leave these questions for now, and just finish the record keeping the layout of the paper version as it is.

When the person details record is finished, it should look something like this:



We have now finished the dictionary for our example questionnaire, but before discussing how to make the forms, we have a few more issues to talk about dictionaries.

4. More about creating dictionaries

4.1. More about the identification element and repeating records

4.1.1. Meaningless ID

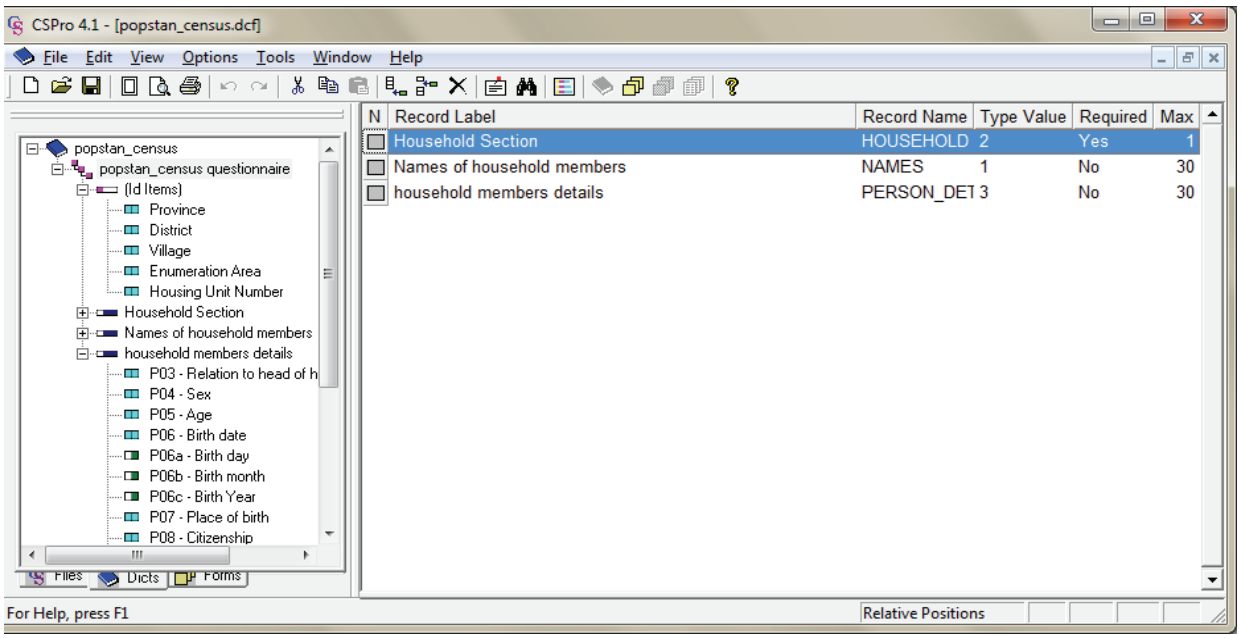
Sometimes – though extremely rarely – we have questionnaires in which there are no candidates for ID elements. What then? CSPro still requires that each record has a unique ID.

The best way to deal with this is to handle the ID element(s) programmatically, so that the interviewer or operator never has to think about this at all. A common way to do this is to use the computer clock to get the time stamp of the moment the record is entered. The time stamp is of the form yyyymmddhhmmss (year – month – day – hour – minute – second, 14 digits)

4.1.2. IDs for repeating records – Record type

In our example questionnaire, we have a housing record and two household member records, and the last two records can appear up to 30 times for each housing record. We still only have one ID for a housing unit that can contain 61 lines in the data file. How does CSPro deal with this?

The answer is the record type field. As seem below, the popstan_census application has three records, and hence three record types; the household section has type value 2, the names record has 1 and the member details 3.



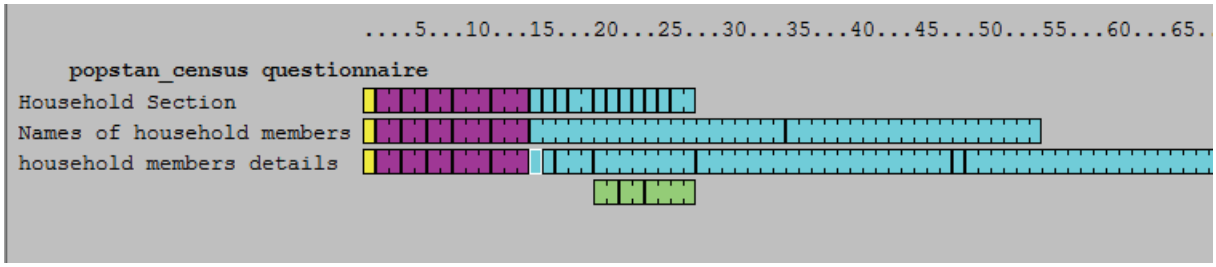
CSPro uses the record type to keep track of the data lines within the same ID. Here is an example of a data file from our application:

```
2 1 2 1  1  1111 2111111 0
1 1 2 1  1  1John               Smith
1 1 2 1  1  1Barbara            Smith
1 1 2 1  1  1Junior             Smith
3 1 2 1  1  111 6613 619980slo              1Oslo              711Professor          University
3 1 2 1  1  122  55  7 719890slo              1Oslo              611teacher            school
3 1 2 1  1  131   312 220100slo              1
2 1 2 1  1  2211 5223123 0
1 1 2 1  1  2Oliver             Jones
3 1 2 1  1  211 7819 21956Wherever          2Wherever          142
```

The column in red is the type column, while the blue columns are the ID elements. The very first line is of type 2, hence a housing record. Next three lines are type 1, name records, and the next three are type 3, details records. It is obvious from the example data that the order of the records within an ID is important, as the data for John Smith is the first line of type 3.

4.2. Viewing the Dictionary layout

To see how the layout of the data file will be, select View – Layout from the menu. The lower right part of CSPro becomes like this:



This is an overview on where each item in a record is located and how much space has been allocated.

The different colors have the following meaning:

- Yellow rectangle denotes record type
- Magenta rectangles denotes Id items
- Cyan/turquoise rectangles denotes Items
- Green rectangles are subitems.

Connections to the items in the dictionaries:

- Click on the item on the layout window to move the cursor to the specific item in the dictionary window
- Double click on an item to show its value set(s).

4.3. Multiple value sets in the dictionary

As we saw in the popstan census application, value sets tell CSPro what values are acceptable as input data for an item. Value sets are optional, and if no value set is present, CSPro accepts any values for the item, given that it has the right data type.

An item can also have multiple value sets. This is often used if the application is a multi-language application (see chapters 6.3 and 8.5). Another common use for multiple value sets is classification of age groups into for instance the following sets:

- Discrete values (0, 1, 2, … 98; Not reported 99)
- By five years (0-4 years, 5-9 years, … 60 and over)
- By category: Infant: 0 years, Child: 1-12, Teenager: 13-19, Adult: 20-59, Senior: 60-98

This classification is not important in a data entry/CAPI application, as only the first value set will be used to check the validity of entered data, but can be used for tabulation applications when processing the data at a later stage.

In chapter 8.5.2 we will see how the programmer can change what value set is active for an item, and can even generate a value set dynamically by using logic.

4.3.1. Special values

CSPro has three "special values" that describe certain kinds of data:

- Not Applicable: the item is blank (e.g. education level of a 5-years old)
- Missing: the codebook had a value for missing (or not stated) and you assign this value to be missing.
- Default: the item has an invalid value (e.g., your program logic assigned a three-digit value to a two-digit field)

By default CSPro ensures that keyed data fits in the value set and is not blank, but if desired CSPro can accept blank data or out of range data

4.4. Numeric Items with decimals

Dealing with numeric items, we also have to consider decimals and how to represent them in the data file:

- Is a decimal fraction needed for this item? If so, how many digits are necessary to the right of the decimal point?
- Should the item be saved to the data file with a decimal point? (This is a purely cosmetic indicator, though it does have bearing on the length of the item.)
- Zero Fill: Do you want the unused spaces to the left of a number padded with zeroes?

This is the number 3.14 stored using various item attributes:

Numeric, Length: 4, Decimal: 2, Decimal Character: Yes, Zero Fill: Yes	3.14
Numeric, Length: 6, Decimal: 2, Decimal Character: Yes, Zero Fill: Yes	003.14
Numeric, Length: 6, Decimal: 2, Decimal Character: No, Zero Fill: Yes	000314
Numeric, Length: 6, Decimal: 2, Decimal Character: Yes, Zero Fill: No	3.14
Numeric, Length: 6, Decimal: 3, Decimal Character: Yes, Zero Fill: No	3.140
Alphanumeric, Length: 6	3.14

4.5. Relative versus absolute positioning: making dictionaries from existing files

By default, CSPro will automatically assign the starting position (column number) of each item in your dictionary. This is known as relative positioning, as opposed to absolute positioning, where the person designing the dictionary assign starting positions.

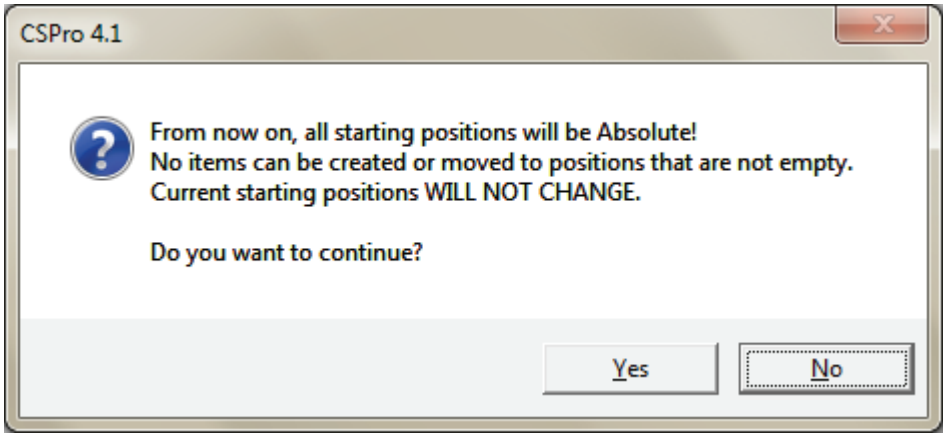
When creating a dictionary, it is highly recommended to use relative positioning, as it is very easy to lose track when doing it yourself. Inserting an item in between other items, or modifying the length of an item, will cause all the other items' starting positions to automatically change, and doing this manually would be a tedious job.

The default order in the data file will be: record type, ID items, record items in the order they appear on the screen.

4.5.1. Absolute positioning

If you are creating a dictionary to match an existing data file, it may be necessary to select absolute positioning. With absolute positioning, you must specify the starting position (column number) of each item in your dictionary, and it becomes your responsibility to make sure that items do not overlap.

To create the dictionary in absolute positioning, select "options" and then un-tick the relative positions, you then get this warning:



...and from here, you are on your own..

The following is how a file with data in relative position might look:

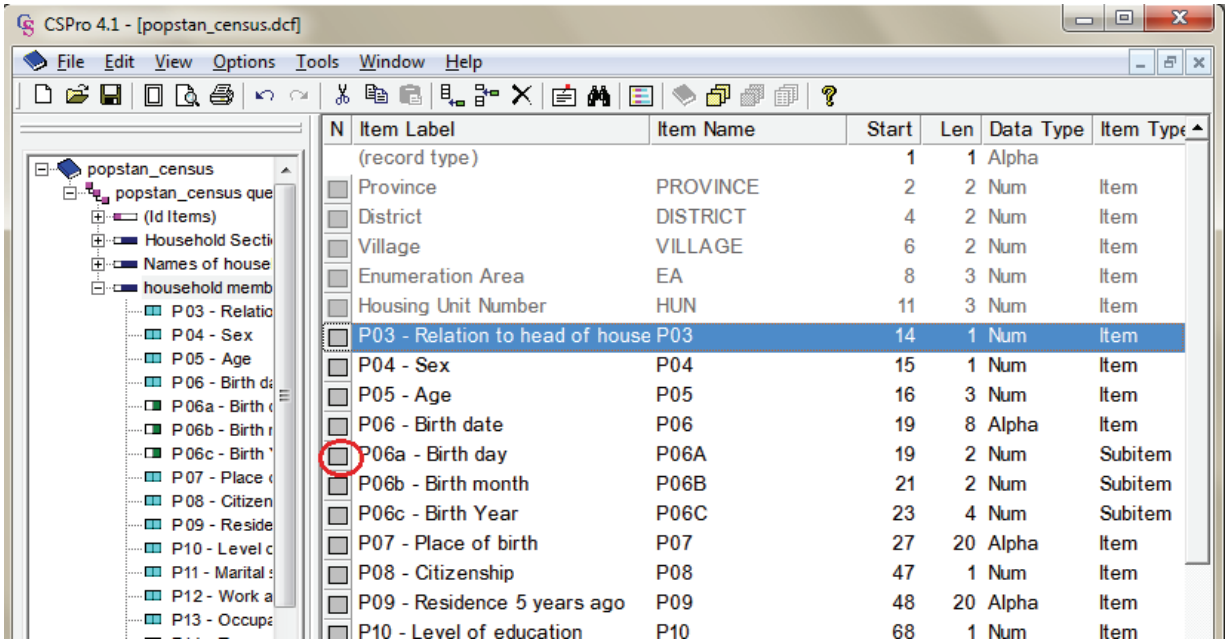
```
11996William Jefferson Clinton
21996Robert Joseph Dole
```

And a file with data in absolute position (one of many possibilities):

```
William Jefferson Clinton          1996      1
Robert Joseph Dole                 1996      2
```

4.6. Documenting the Dictionary elements

It is a good idea to document the dictionary elements – especially if you have done something unexpected or unusual. Left of every element in the dictionary editor is a small gray box under the column heading N:



Clicking on this box brings up a field in which you can write notes about the dictionary element. These notes are stored in the dictionary file but are not visible during data entry

Consider making use of these notes, especially when working with partners on an application

4.7. Modifying the dictionary

The dictionary can be modified again and again up to the moment data entry or CAPI interviewing starts. CSPro detects changes between the dictionary and forms, so if you rename or delete a dictionary item, the field on the form will also be renamed, or will be removed from the form.

But if the data entry or interviewing has started, it is not recommended to change the dictionary unless it is absolutely necessary. If changes are needed, it must be done with great care; make backups of the dictionary before the modification so that you always have a dictionary to read data that was entered at any time of the data entry operation.

4.7.1. Adding Fields to the Dictionary or Modifying item lengths after data entry has started

If, after the data entry process has begun, some fields need be added to the dictionary, one option is to simply add them to the end of any given record. This way the data that already exists will have blanks for the new values, but the data can still be read by the new dictionary.

However, if adding the fields to the end of a record is not practical, you can insert them in the record, but then all existing data must be reformatted to the new dictionary format.

Also if the lengths of some items need to be increased, the existing data file has to be reformatted. However, if the length of some items will be decreased, it may be possible to use absolute positioning to make your old data files readable. Likewise, deleting an item from the dictionary can be done in a way that does not require reformatting, but again absolute positioning must be used

4.8. Dictionary Macros

There are some undocumented dictionary macros that might come in handy – especially if some of the people in the development team do not know CSPro. To use them, right click on the dictionary in the tree in the left pane.

4.8.1. Copy and paste dictionary items between CSPro and Excel

Names and labels of dictionary items, or value sets, can be copied to Excel format, modified in Excel, and then pasted back to CSPro.

This can be particularly useful if you want coworkers who do not know how to use CSPro to help with the creation of the dictionary, perhaps by adding values to the codebook (value sets).

- To copy from CSPro to Excel: select the kind of data you want to copy "Copy all Names/Labels" or "Copy all Value Sets". Then in excel, the data is pasted by edit- paste.
- To Copy from Excel to CSPro: First make sure the data in the spreadsheet is on the correct form, then highlight the area, copy it to the clipboard, and select "Paste all Names/Labels" or "Paste all value Sets" according to what you have been working on in Excel.

4.8.2. Generate sample or random data files

The dictionary macros also include a possibility to generate random data – or sample data files. This is not much needed for the data entry/interviewing process, but can be handy if applications for processing the data after data entry are developed before the data entry process has begun.

5. Creating the forms of the Popstan census questionnaire

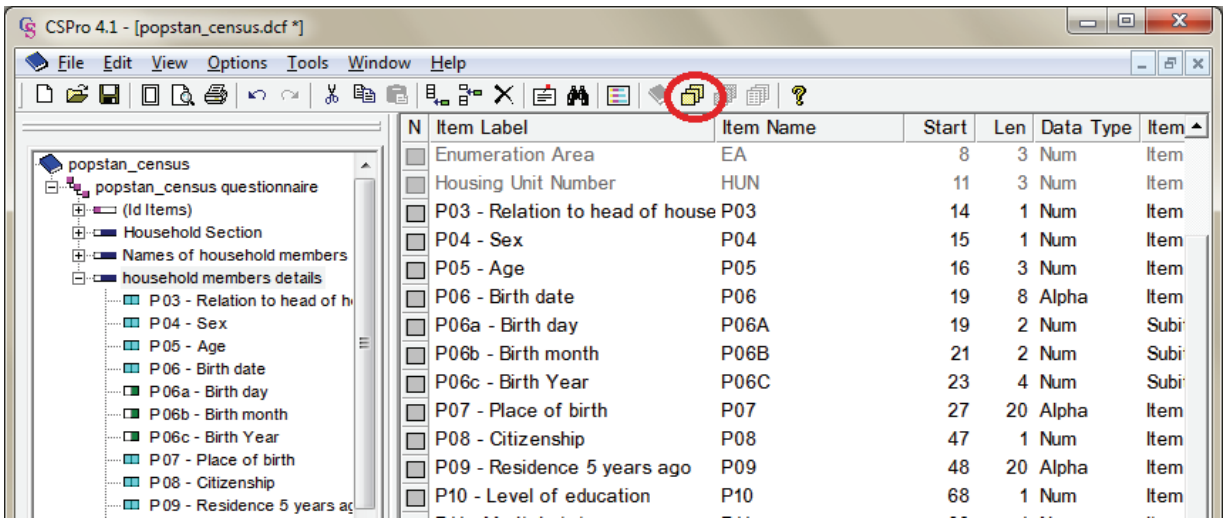
5.1. General about forms

A form of an application is the equivalent of the paper in paper based surveys. A form is a slate where you can put a collection of fields, texts and rosters or repeating items, and it is what the interviewer sees of the application when he does his interviewing.

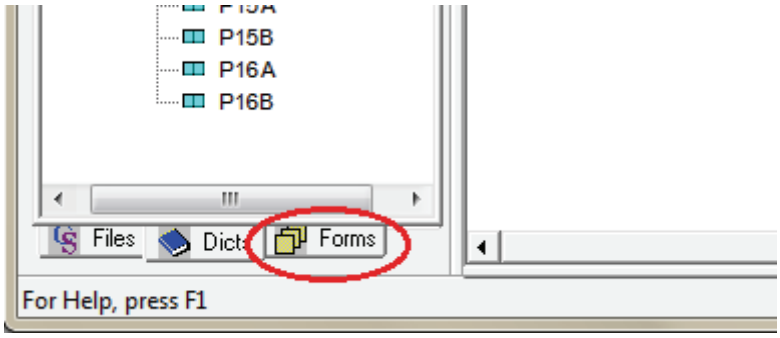
Before we begin designing forms, it is good to have a plan about how many forms we need and about the contents of each form. In general it is best to have one or more forms for each record type.

It is easiest to create the forms after most of the work on the dictionary is done. In our example survey, we now have three records plus the Id items. This splits naturally into three or four forms dependent on whether you want a separate form for the Id Elements. In our example we will let the ID items be a part of the household form.

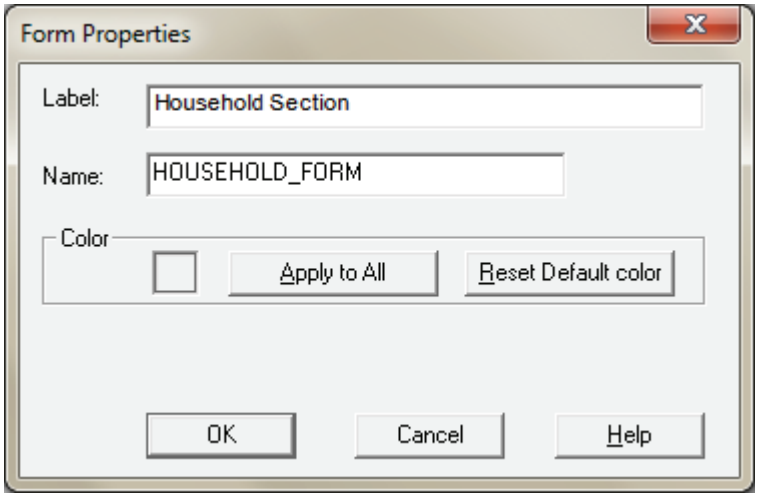
To change the view to the forms "canvas" – a place to design the forms, click on the yellow form icon on the toolbar:



As you can see, the right part of CSPro changes, while the left part remains the same – showing the tree structure of your dictionary. To change this side too, click on the Forms tab on the bottom of the pane:

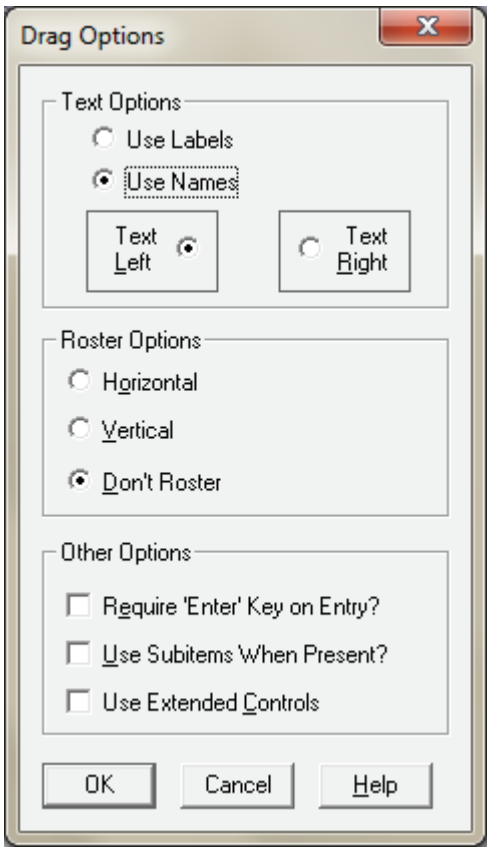


CSPro has already created a form for us – called "popstan_census questionnaire". We do not want this name of the form, however. To change it: right click on it and select properties. Change the name and label, for instance like this:



5.2. Adding questions to the form

First we add the ID items to the top of the form. To do this, we first need to get the dictionary tree back in the left pane: Click on the Dicts tab in the bottom of it, and use the mouse to drag the (Id items) onto the form canvas. You can drag one item at the time or all of them together. CSPro then pops up the following:



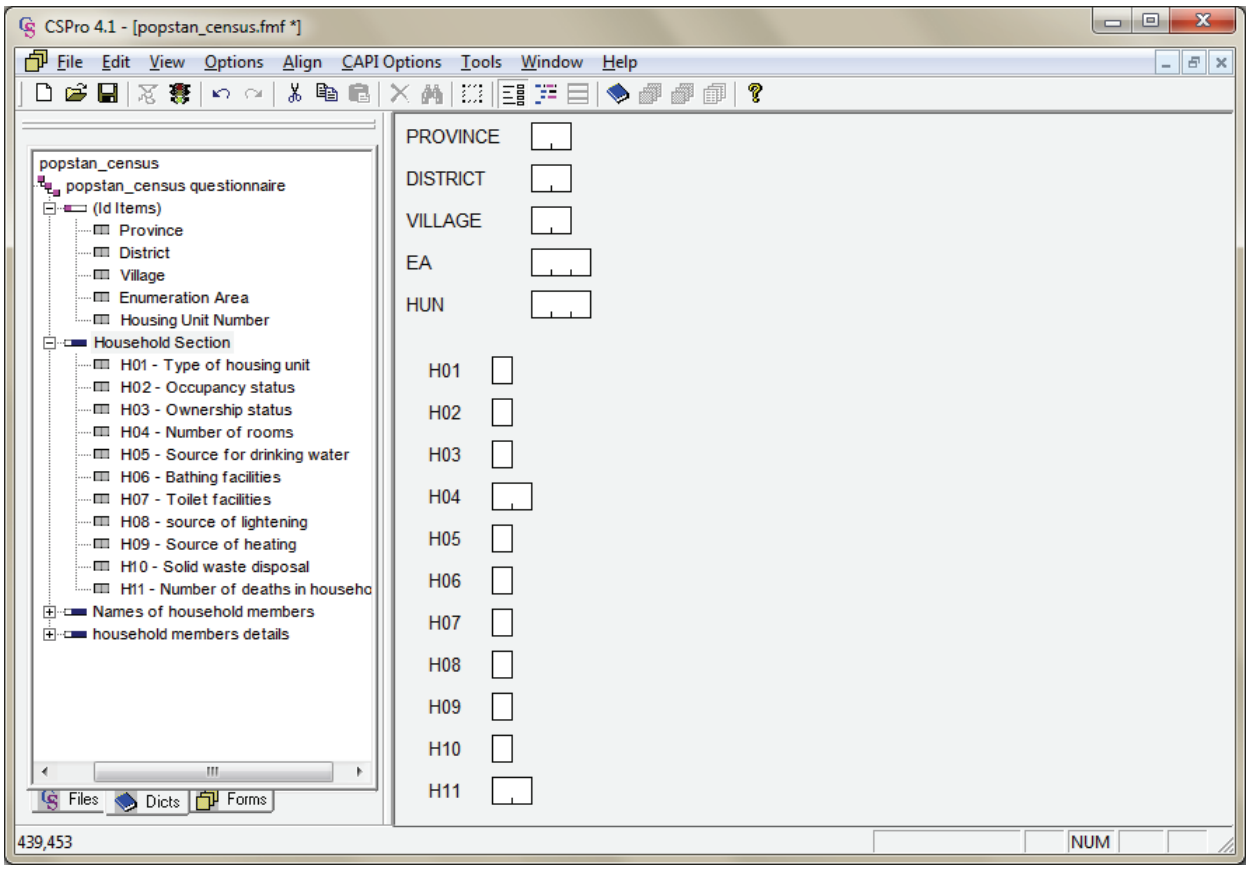
Here you can decide how the form shall look and behave:

“Text Options” lets you choose whether the labels or the names of the input fields should be displayed in the form. When doing CAPI surveys, the text that the interviewer reads to the interviewees is displayed on top of the screen, so it is common to use only names instead of labels in this case.

If the application you are making is a traditional data entry application, labels can be better to use as they are more descriptive, helping the keyer to know what he is doing.

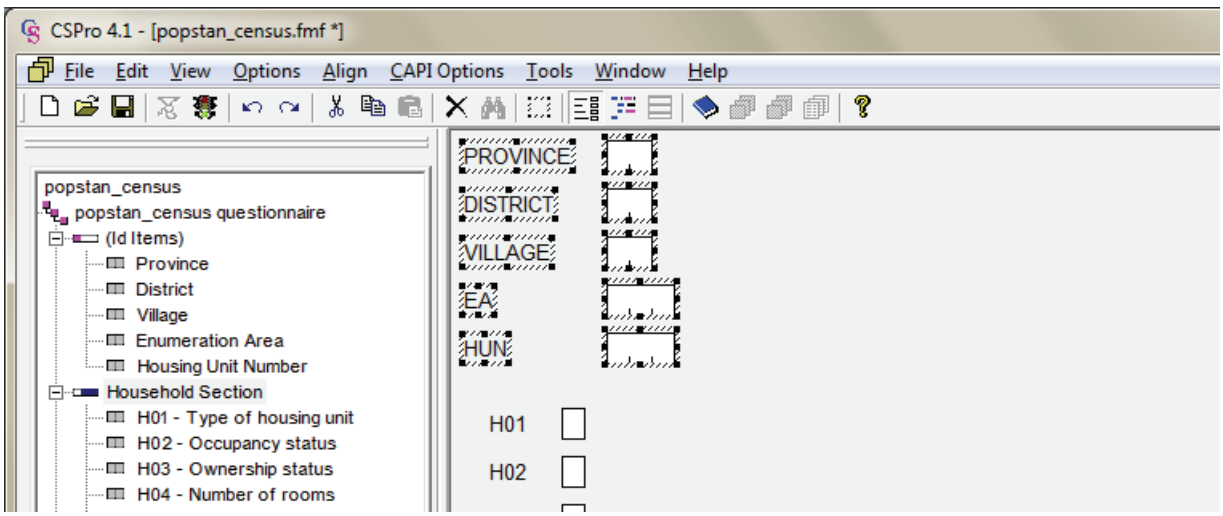
The Roster options only affect dictionary records and items with more than one occurrence. Chosing “Horizontal”, CSPro makes a roster or a matrix of the input fields having the occurrences as the rows and the fields as columns. Choosing “Vertical” makes the opposite roster (occurrences as columns and the fields as rows). “Don’t roster” will make the form repeat. We will use these options in the next two forms.

Drag the household section record over to the canvas too, and you have something looking like this:



5.3. Making the form look better

The above form does not look very nice, so we have to change the layout of it. We can move one item at a time using the mouse to drag it, or we can mark several items by placing the mouse next to it and dragging it to "cover" the fields that you want to move. The fields then get highlighted like in the following picture, and you can move all at once by placing the mouse on one of the items and dragging it. The fields can also be moved with the arrow keys.



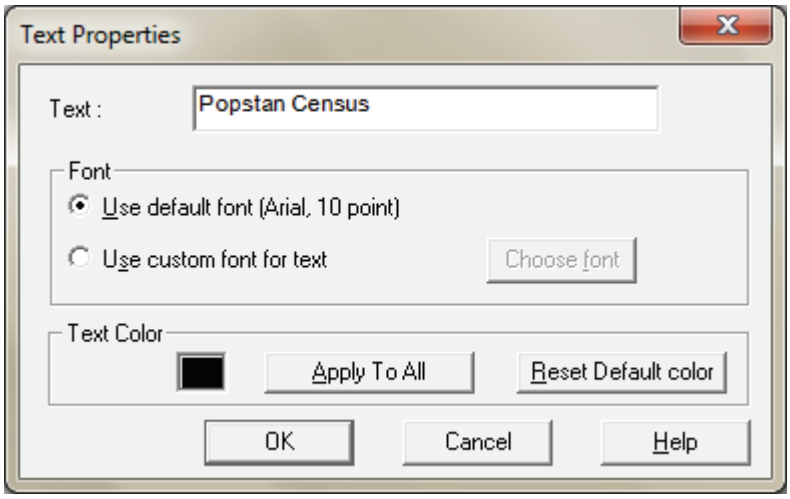
5.3.1. Order of execution of the elements in the form

When using the forms, the focus of the input fields will be in the same order as they are added to the form – which is the same order as they appear in the forms tree in the left pane.

If you need to change the order of the fields, just move them to the desired position in the forms tree. Similarly, moving a form in the tree changes the order that forms will be entered.

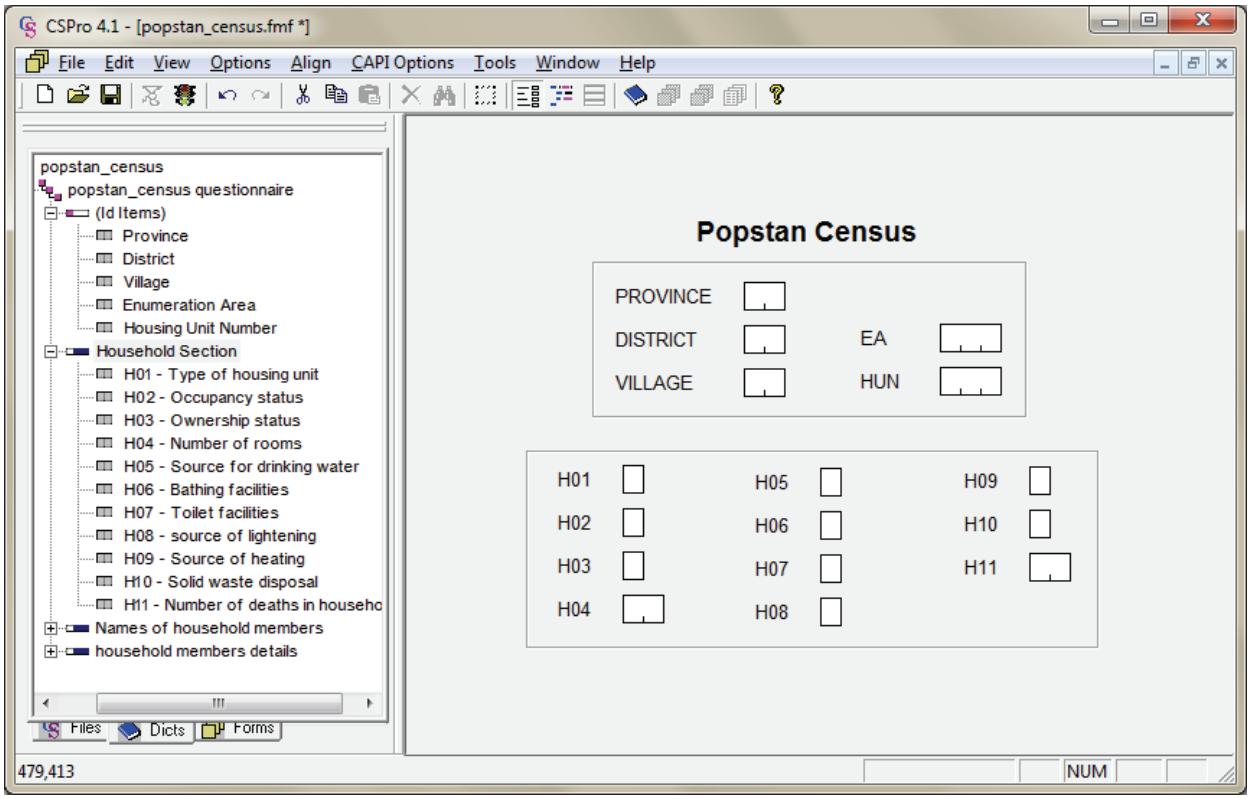
5.3.2. Adding texts and boxes to the form

We can add text to the form by right clicking on the canvas and selected “add text”. We then get a window to enter the text and select colours and fonts.



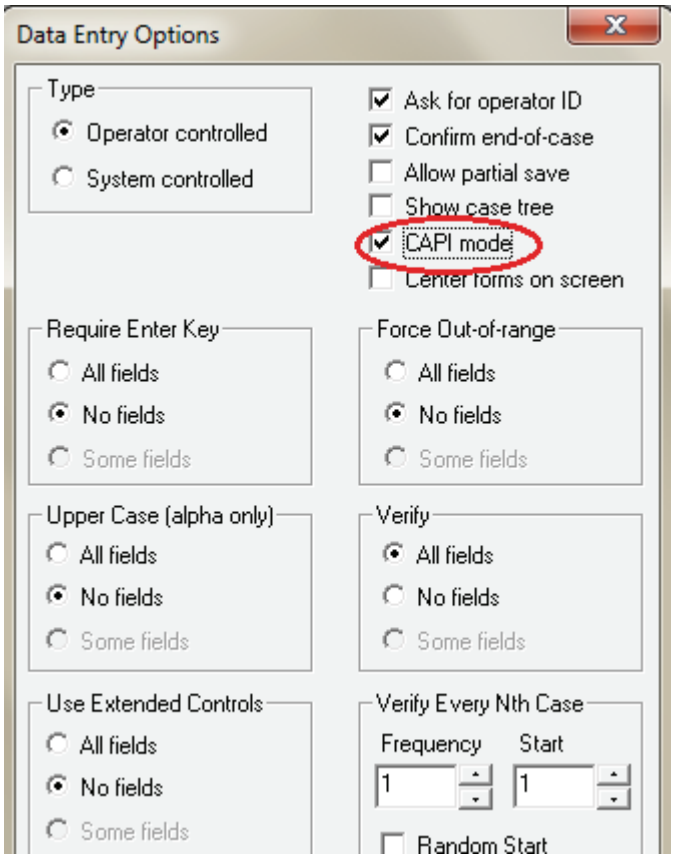
To add a box to neatly group elements together, right click again and choose “add boxes”. A “box tool” is then show. Choose the kind of box you want, and put them on the canvas by clicking and dragging the mouse over the area you want it. Close the box tool to stop adding boxes.

After adding texts and boxes, and moving items around, the form might look like this:



5.4. Adding CAPI questions and texts

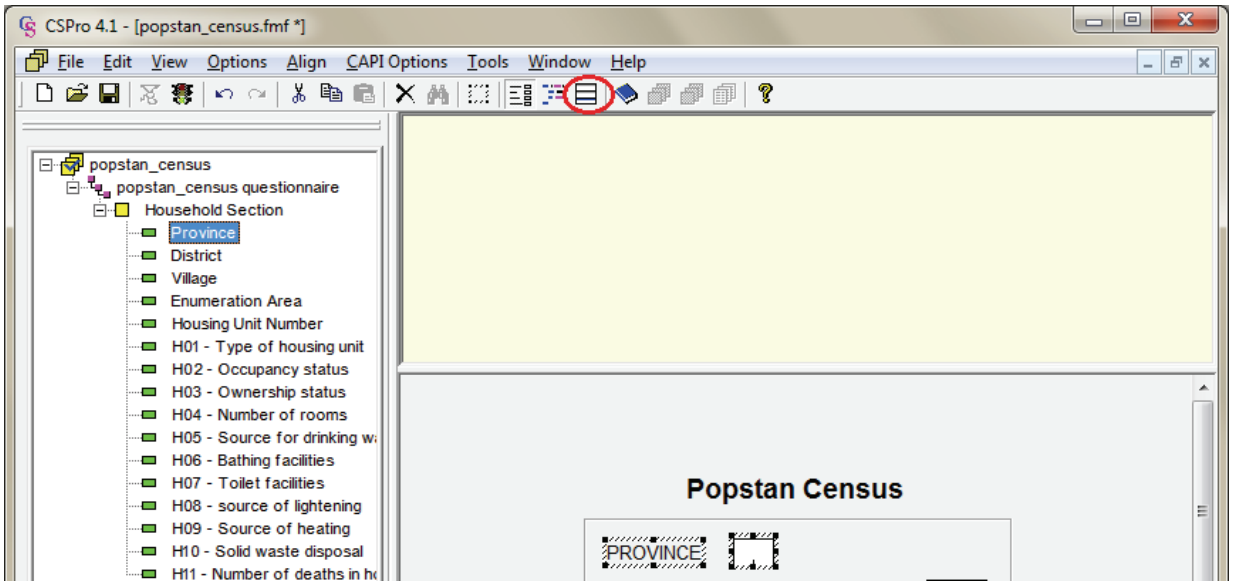
To add CAPI questions, we first need to tell CSPro that this is a CAPI application. Choose Options – Data Entry… from the menu, and tick "CAPI mode" in the resulting data entry options window:



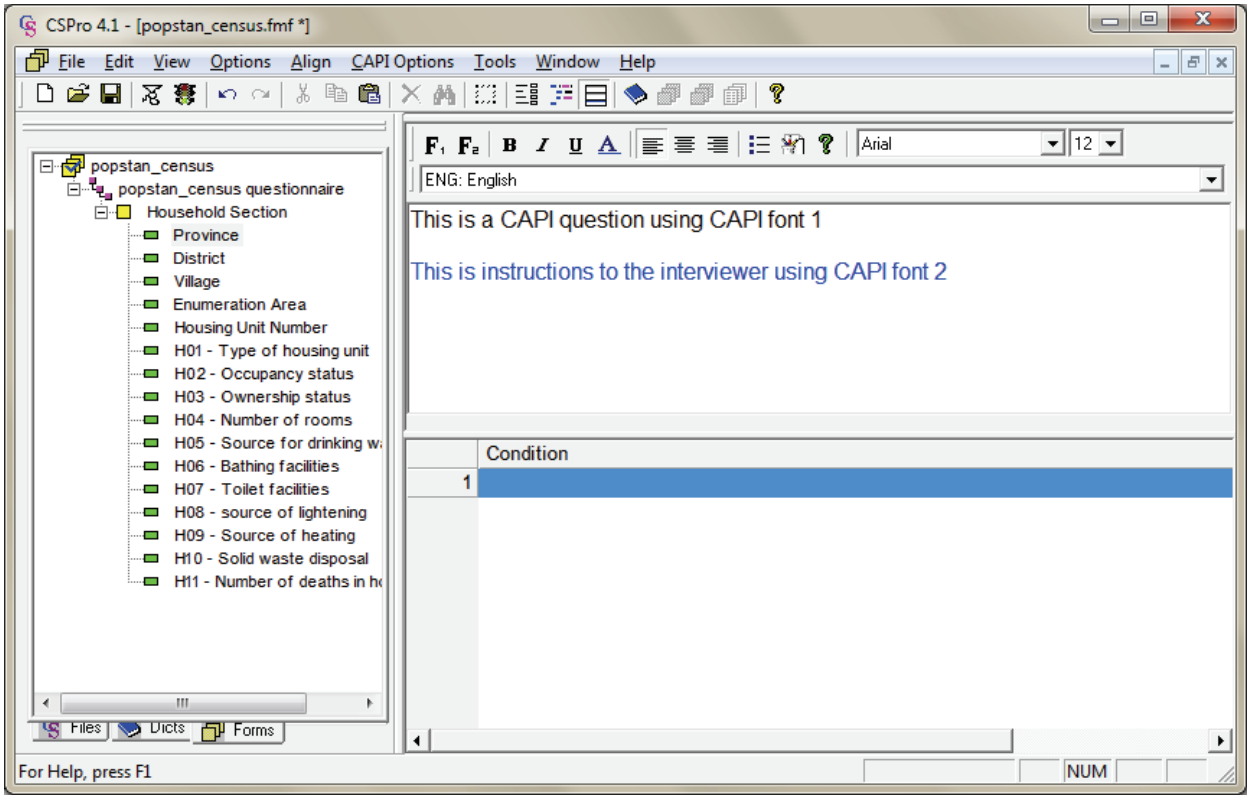
(We will in paragraph 5.6 return to the data entry options window and explain more details.)

When pressing OK, the Form has changed slightly: On top there is a yellow, empty field. This is where the questions for the interviewer to read, as well as additional instructions to him – if needed.

To add CAPI question to an item: Make sure the Form tree is shown in the left pane (click the Forms tab on the bottom of the window), and highlight the item you want to make the question for. Then click the CAPI questions button on the toolbar (shown below):



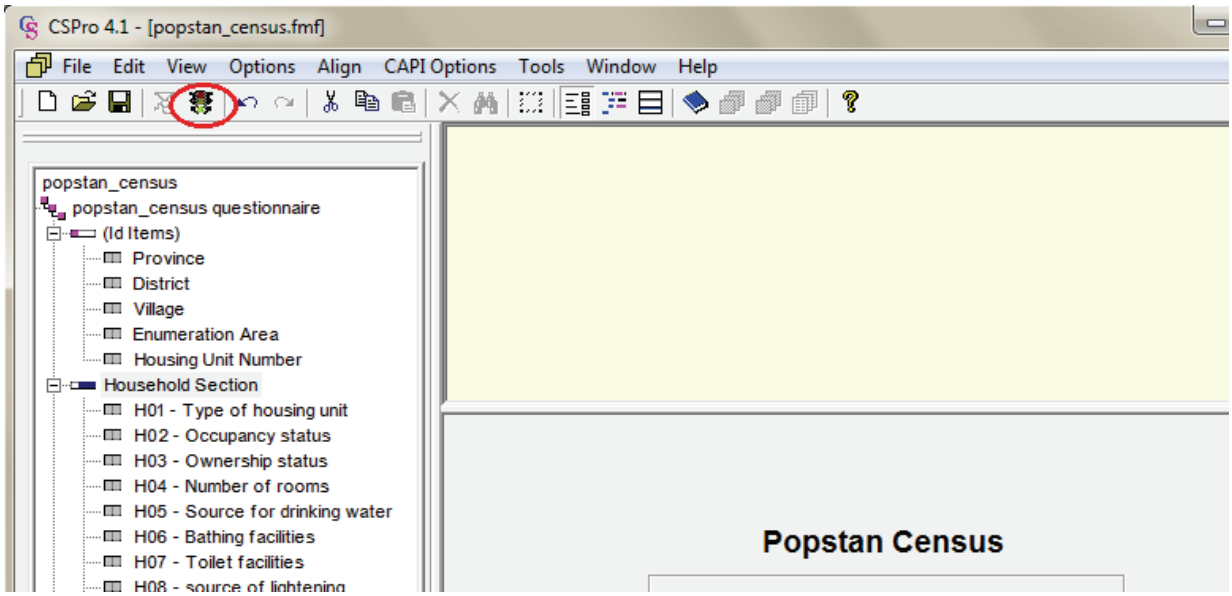
The CAPI question editor has two default fonts. The first one is to use for the questions, and the other one to use for additional comments or information to the interviewer like this:



After entering all the CAPI questions (and instructions if any), we are ready to test the first form of the application.

5.5. Testing the application

To test or run the application, click on the traffic light icon on the toolbar as shown:



CSPro then first asks for the data file: Where to save the data that is entered. Navigate to where you want the files to be saved, and enter a file name. CSPro does not give the data file any file extensions automatically, so it is recommended to name your file with the preferred extension, for instance .dat or similar. Please see chapter 9.2 for a comment on how to structure the files.

After naming the data file, CSPro asks for an Operator ID. This is nice for creating statistics of how the data entry operators or interviewers perform. Any data can be entered here, for example name or initials of interviewer.

5.6. More about the Data entry options window

As seen earlier, selecting "Options" – "Data entry..." pops up a window giving options on how the application should behave. We saw that we can define the application to be a CAPI application. Here we shall explore some of the other options available.

5.6.1. Turning off the question about Operator ID

The question about Operator ID can be turned off. This is handy during the development of the application, as much testing is needed, and it is annoying to have to answer this question every time. To turn it off, select Options – Data Entry Options… from the toolbar, and un-tick "Ask for operator ID" (but do remember to turn in back on when the application is finished).

5.6.2. Partial save

There is an option "Allow partial save". Whether we want to allow this when the survey is done, or not is not a topic of this tutorial, but this option comes very handy when debugging the application:

If we are working on a form towards the end of a multiple-form questionnaire and want to test it, we have to enter data for all the previous forms first. By allowing partial save, data can be entered up to the place we want to test, and then partial saved. Next time we want to test, we just continue to enter data for this saved record.

5.6.3. System controlled vs. operator controlled applications

There are two modes the applications can run in: System controlled and operator controlled. There are pros and cons for both modes.

5.6.3.1. Operator controlled mode

The default is operator controlled, and this allows more flexibility for the interviewer: He can use the mouse to move around in the questionnaire, bypassing fields or whole sections of the application. The mouse can also be used to skip to fields after having keyed an invalid response for a value.

Data entry in operator controlled mode is sometimes quicker than system controlled mode, but the data is less accurate.

5.6.3.2. System controlled mode

In system controlled mode, CSPro decides how the interviewer is allowed to move around in the application. It also ensures that the data comes in the format the programmer specified, with skip patterns obeyed and consistency checks passed. The interviewer has to resolve all errors before moving in the questionnaire, which can slow down the process.

The rest of the Data entry options are not that frequently used, so we leave it to the reader to look it up in the CSPro manual.

5.7. Creating the population forms

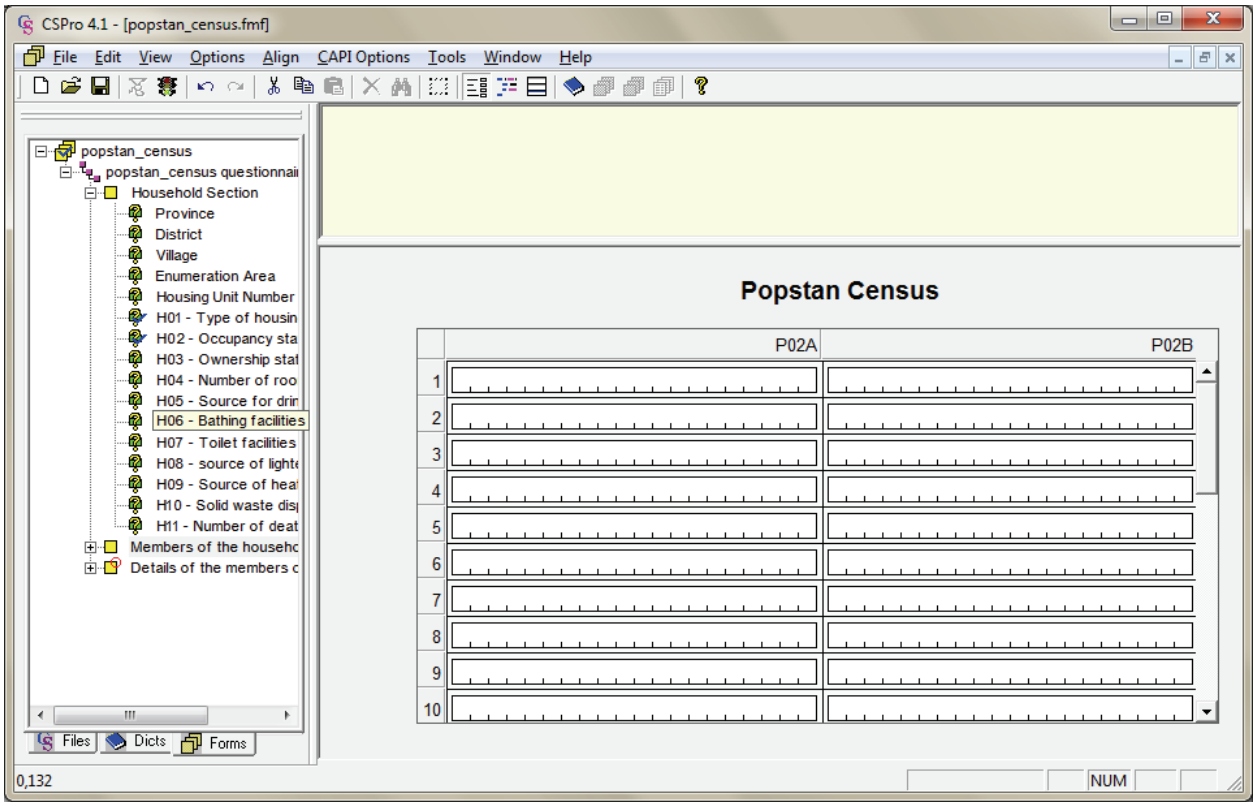
There are two more forms to create in our example application: The form containing names of all the household members, and the form containing details about each of them.

5.7.1. The form containing the names of the household members

To add a new form: make sure that the forms tab is selected in the left pane and that the forms mode is on the right side. Right click on the "popstan_census questionnaire" in the left pane, and select "Add form". Name the form "MEMBERS_FORM" and give it a label.

As earlier, the form is populated with questions by choosing the "dicts" tab on the bottom of the left pane, and then dragging the record containing the names to the form canvas. Choose "Horizontal" in the roster options window.

The result might look something like this:



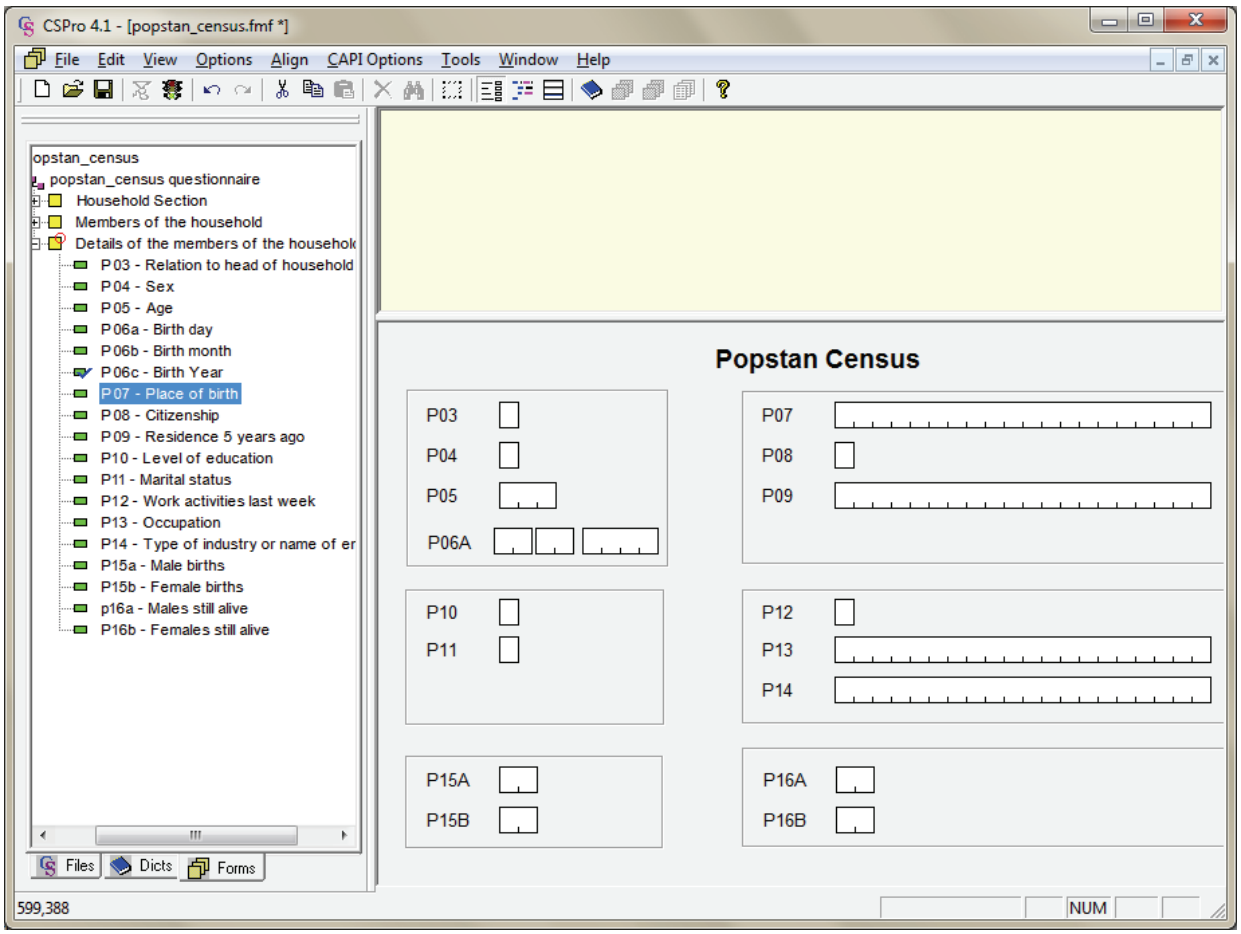
5.7.2. The form containing the details of the household members

Make the details form as you made the previous form, but when dragging the record to the canvas, tick for “do not roster”, to make the form repeat once for each member of the household.

I prefer to use the sub items for the birth date rather than the date itself, as it looks clearer, so delete the date of birth from the form, and add day, month and year subitems.

Note that when doing this, the date subitems end up on the bottom of the forms tree. And as stated earlier, the items on the forms receive focus in the same order as they are in the tree, so they have to be moved up to avoid focus to be jumping back and forth in the form.
Do this by dragging them to where you want it. The form should now look something like this:

:



To finish the creation of forms, add CAPI questions for each of the fields.

6. More about forms

We have now created the forms required for our example application. This chapter contains other important characteristics and features of forms not used in the example.

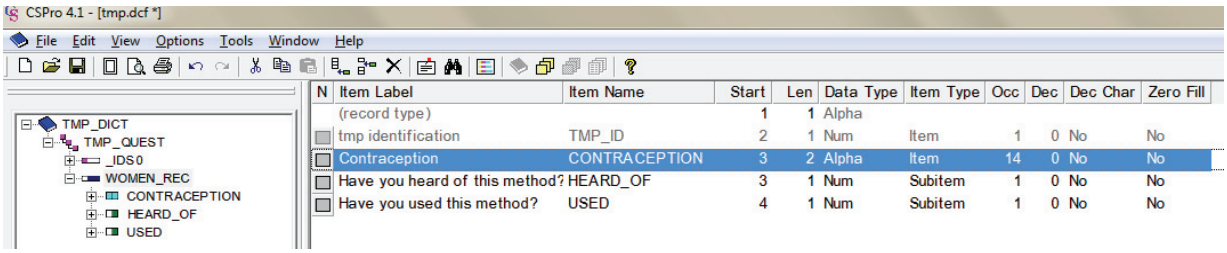
6.1. Rosters in the forms

As we saw above, there are two ways to handle multiply-occurring records in a data entry application – either to have them as a repeating form or as a roster in a single form. In a form with roster, you can also have fields from singly-occurring records, which is not the case for repeating forms.

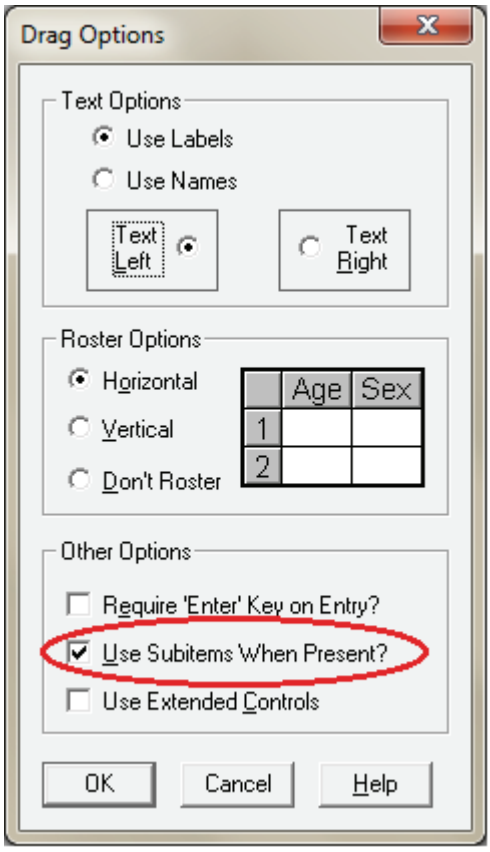
An example of mixing multiply occurring records and single occurring records could be the following:

Say that you have a survey where you ask questions to individuals and you want to ask the women in the population about contraceptive use; for each possible kind of contraceptive, you want to know both whether she has heard about it, and whether she has ever used it.

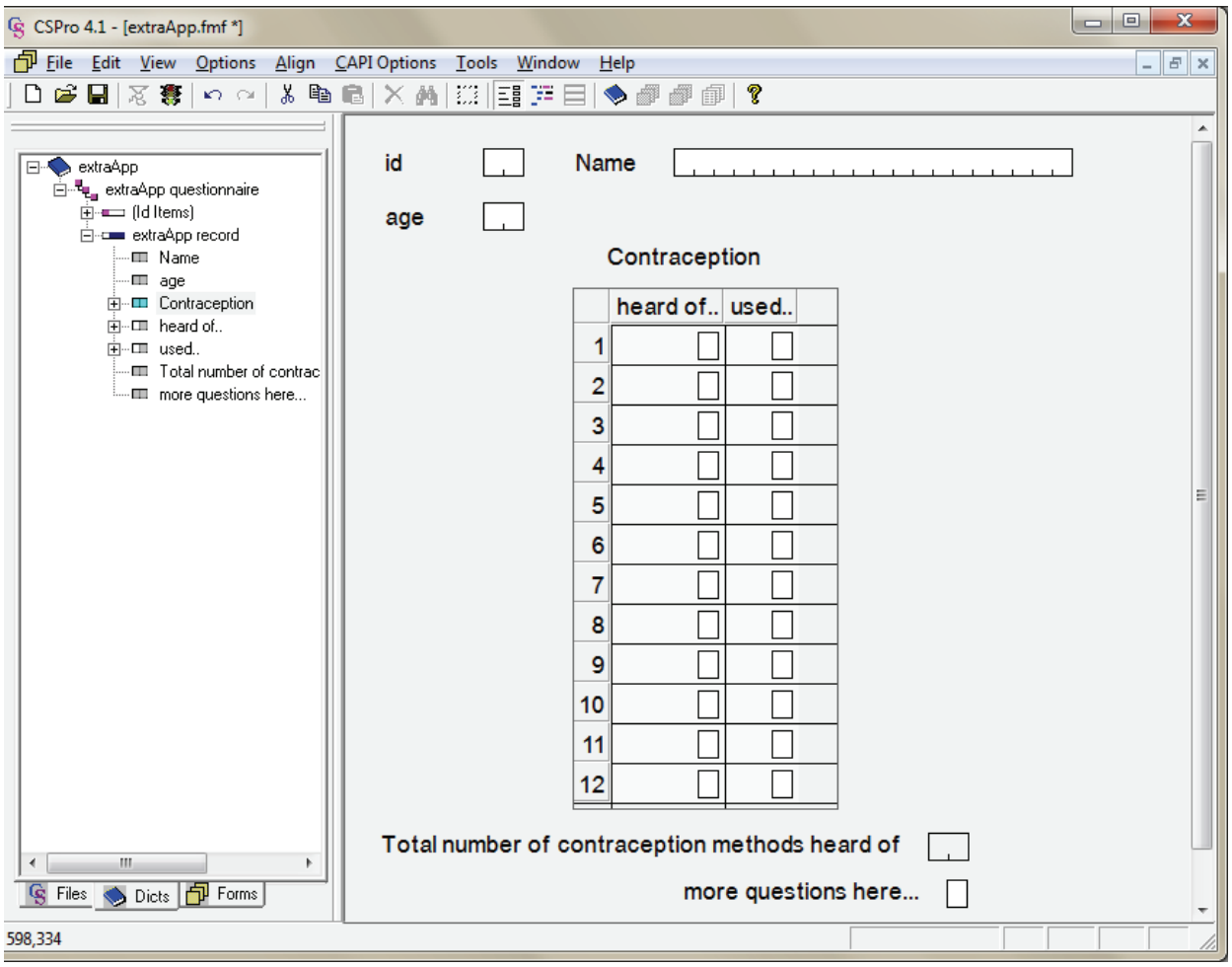
This can be done the following way: When defining the dictionary, create two items; “Have you heard about this contraceptive?” and “Have you used this contraceptive?” Let them both have length 1, and occurrence 1. Then highlight both, right-click and choose convert to subitems. Give the new item a name and a label; keep the other defaults except “occ”: The number of occurrences should be the same as the number of contraceptive methods you want to ask about, like this:



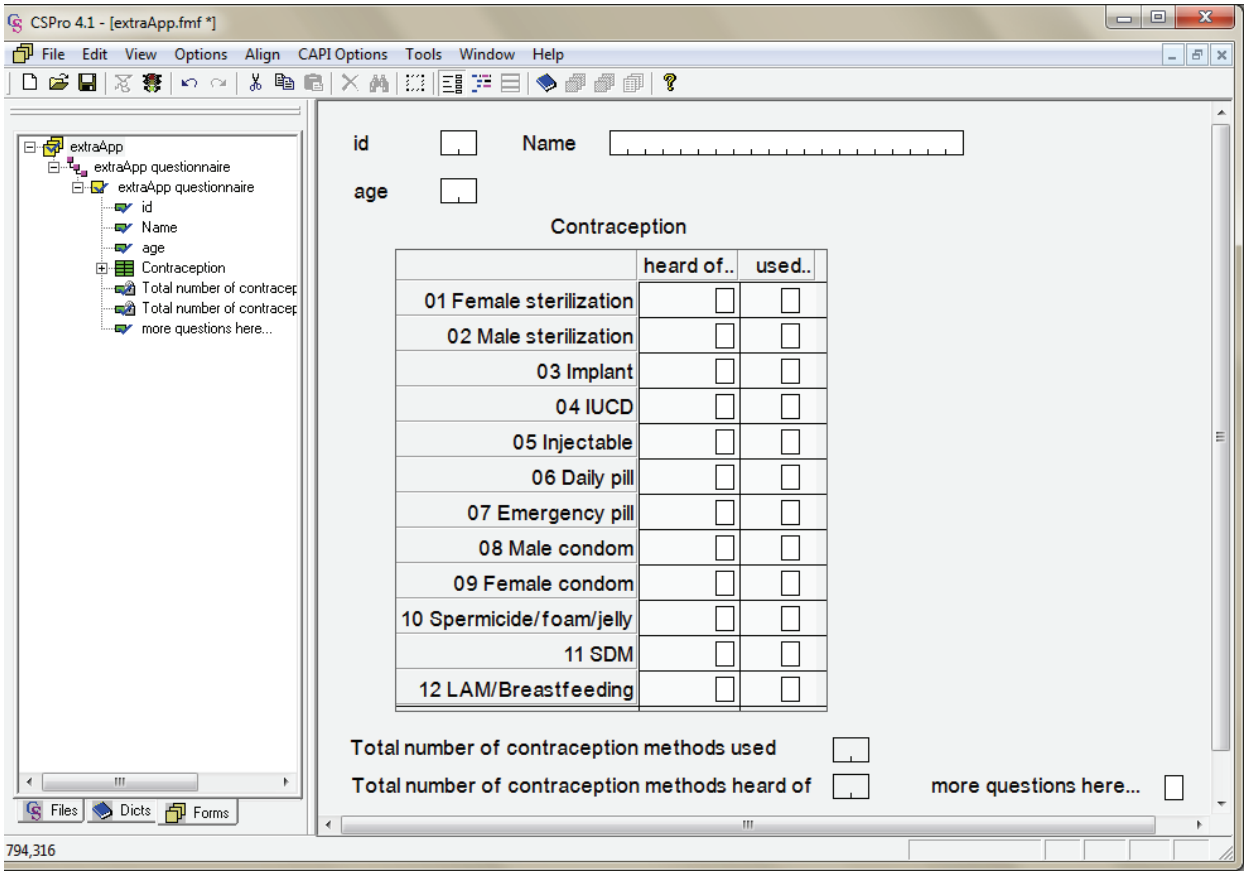
Then we need to make the form for our survey. To get the roster right: Select “Options” – “drag.” from the menu, and tick “Use subitems when present”



Then, when dragging the contraception roster over to the form, the result might look like this:



And after cleaning the whole thing up:

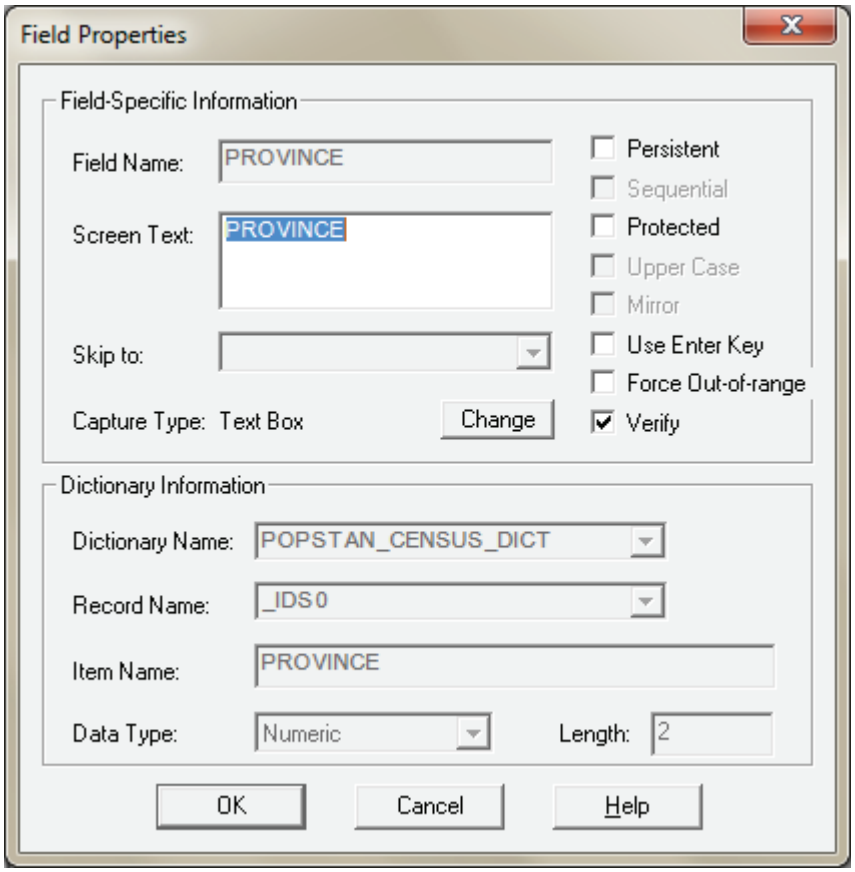


(you can enter the different contraceptive methods by right-clicking on the cells in the leftmost column and editing the content).

We will see later, in chapter 8.2 how to programmatically calculate the total numbers of methods used and heard of.

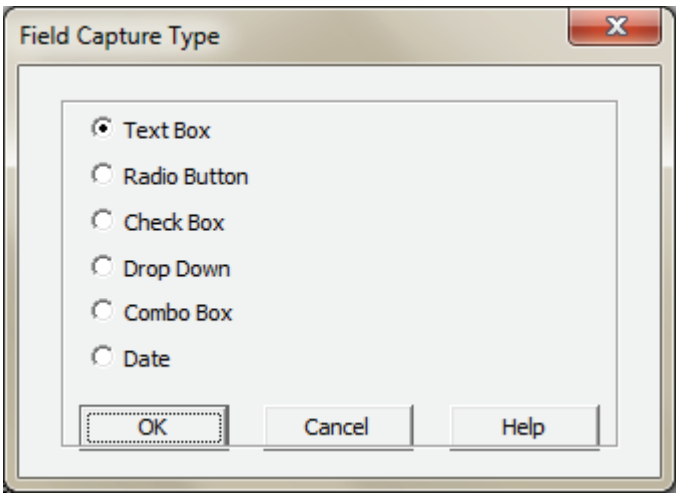
6.2. Field properties

Right clicking on a field in a form and selecting “properties” gives the following window:



Here a lot of options for each field can be configured, for instance the following:

- **Skip to:** (This only works in operator-controlled mode): Gives the possibility to specify what field to skip to. Pressing + will skip from this field to the specified.
  (Generally skips are programmed rather than specified here. See 7.2.1 and 7.2.11.)
- **Persistent:** An ID item will automatically take the value found in the previously entered case (at least one ID item must not be persistent).
  In our Popstan Census example: Let us assume that we know that each enumerator only operates inside one province and district, then we can make these fields persistent, so that the enumerator does not have to enter them more than once.
- **Sequential:** The current item will take the value found in the previously entered case, incremented by 1.
- **Protected:** The field cannot be keyed and must be assigned a value with logic. The value has to be assigned programmatically, though. If this is not done, the application will crash when you try to run it.
- **Upper case:** Alphanumeric fields will be in all uppercase.
- **Mirror:** Shows the value of an already-keyed field for reference.
- **Use Enter key:** Forces the operator/interviewer to press Enter or tab to advance to the next field.
- **Force Out-of-range:** Allows the operator/interviewer to input values not found in the value set.
- **Verify:** Should this field be verified in dependent verification?
- **Capture type:** Gives the following popup window:

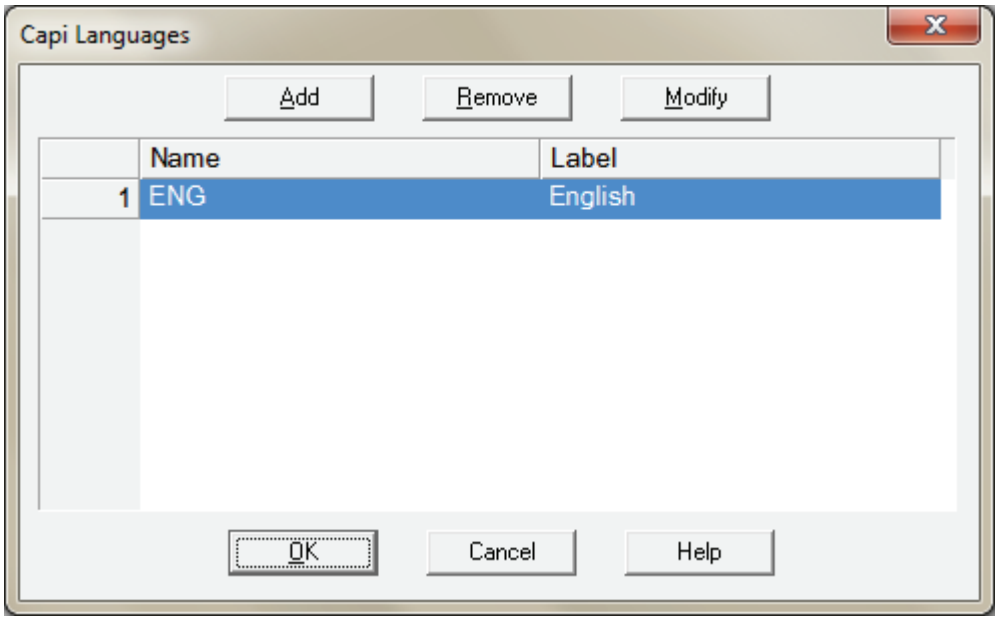


Here you can select an "extended control" for the field.

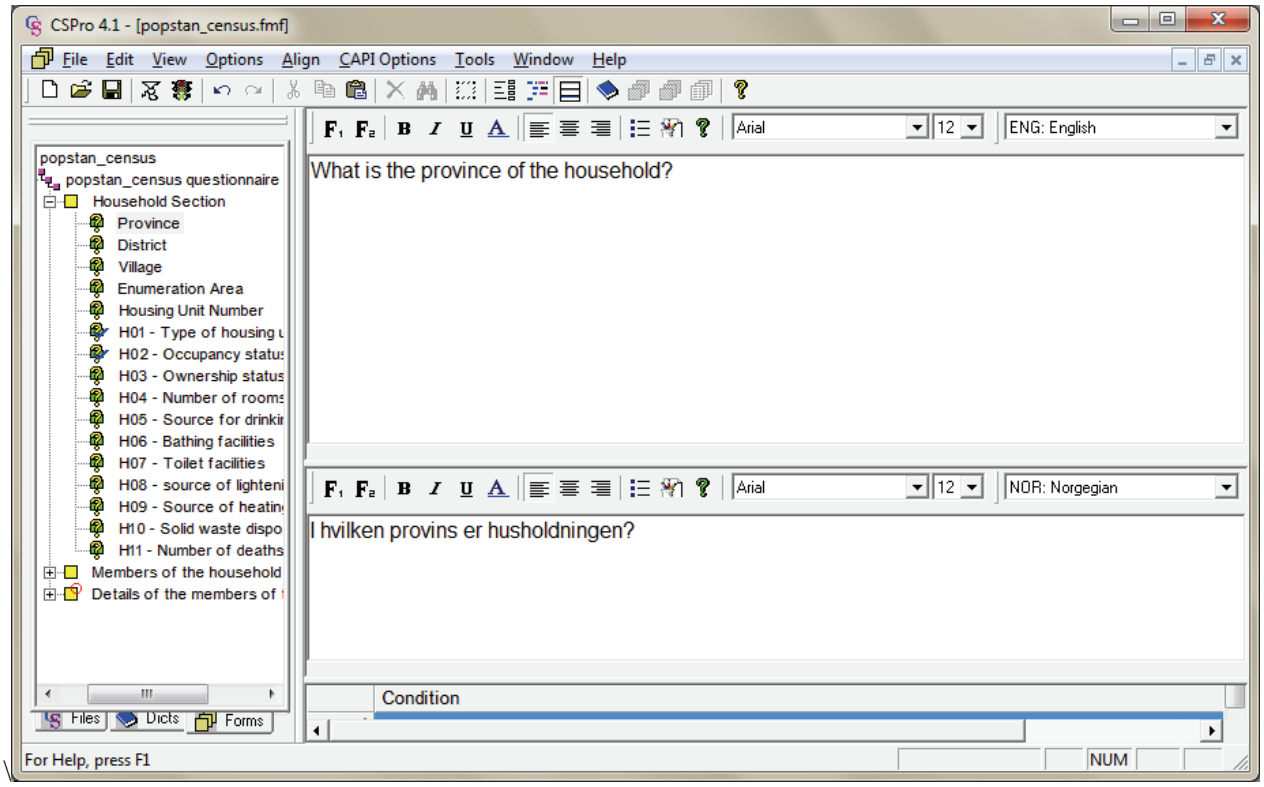
6.3. Multiple languages in CAPI questions

Sometimes multiple languages in an application are needed, and we will see in Chapter 8.5.2 how to deal with multiple languages value sets.

The CAPI questions as described in chapter 5.4 can also be entered as multiple languages:
From the menu, select CAPI options – Define CAPI languages. The following window pops up:



And you can add as many languages as you need. Next time you click the CAPI Questions button on the tool bar, CSPro splits the screen in two – one for each language:



(The above window has the languages English and Norwegian.)

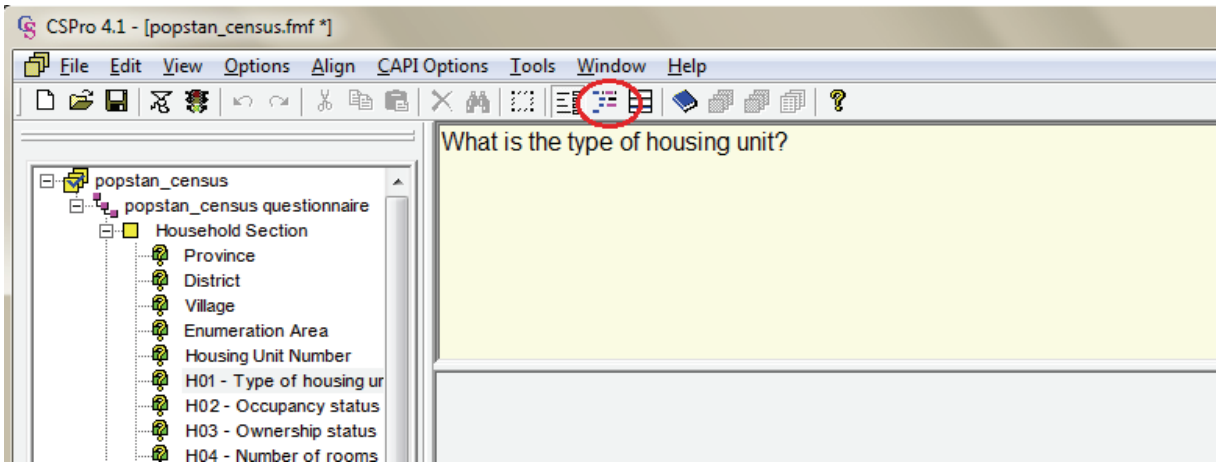
When the enumerator uses the application, he can change the language on the spot by selecting options – change language from the menu, and then selecting the desired language. This is only possible when he is actually adding or modifying data.

7. Programming the Logic for the Popstan Census questionnaire

Only very simple data entry/CAPI applications can be made without writing any logic at all. The moment there are skips or checks in a questionnaire, we have to write logic to describe the desired actions. Programming CSPro is challenging and fun, but at the same time it is very easy to get lost and introduce bugs, so thorough testing is very important.

7.1. Getting started with programming

To open the programming code editor, click on the “logic” icon on the toolbar as shown:

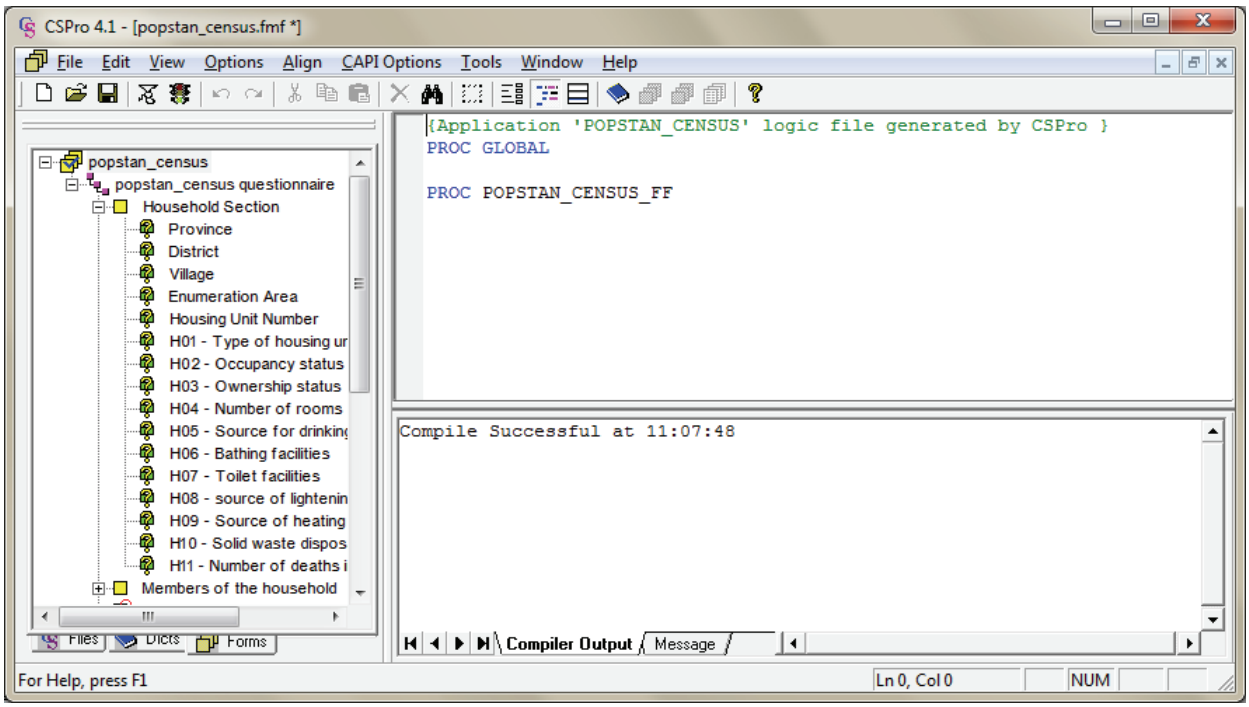


The right pane now changes into two parts: one for writing the code on top, and compiler output under it. The latter is where CSPro gives you error and warning messages.

CSPro logic consists of a collection of statements inside of procedures or functions. The procedures correspond to the items in the questionnaire (or to the form or level itself). In addition there is a declaration section.

7.1.1. Declaration section – PROC GLOBAL

Click on the very top of the forms tree in the left pane to enter the declaration section:



Here variables, arrays, user defined functions and other options are declared. We will talk more about this soon.

7.1.2. Procedural section

This section contains the statements you want to be executed during data entry. Logic can be written for all levels in the questionnaire, but it is most common to write logic on the item, roster, and form levels.

When programming a form, roster, or an item, we first have to decide whether the logic should be executed before or after the "event" (in the case of a form: before the form is displayed, or after the form is finished; in the case of a roster: before the multiple-occurring group gets focus, or when focus is lost; in the case of an item: when the item gets the focus, or when focus is lost.)

If you want the statements to be executed before the event, use the keyword "preproc", if you want it after, use "postproc". Postproc is the default, so if you do not explicitly write the word preproc, the statements will be executed after the event. (There is also the possibility of "onfocus" and "killfocus", but we'll get back to that later.)

7.1.3. Commenting the code

It is a good practice to comment your code to make things more understandable. Comments can be made in two ways: By starting the line with //, or with enclosing the comments in curly brackets {}, like this:

```
//This is a single line comment

{And this is a multiple line comment.
One can write several lines here,
And end the comment with a bracket}
```

Comments in your program are text that is ignored by the compiler, meant to be read by humans. They can also be used when debugging the program, to comment away (disable) code around the place where you are testing.

7.1.4. Variables – and the option set explicit

All data – temporary and permanent – is stored in variables. CSPro has three types of variables in addition to the items in the dictionary:

- Numeric variables – for storing numbers
- Alpha variables – for strings
- Arrays – for storing an array of strings or numbers.

It is highly recommended to start the GLOBAL section by setting the option "set explicit". This makes CSPro require that you declare your variables before using them. If this option is not set, and you misspell a variable somewhere in your code, CSPro will not issue an error message when you compile, but instead create a new variable for you. This again makes debugging your code very hard.

The "set explicit" option is the default setting (see the Options menu), but it is still recommended to set it explicitly in the code, in case you move your application to another computer where this option is turned off.

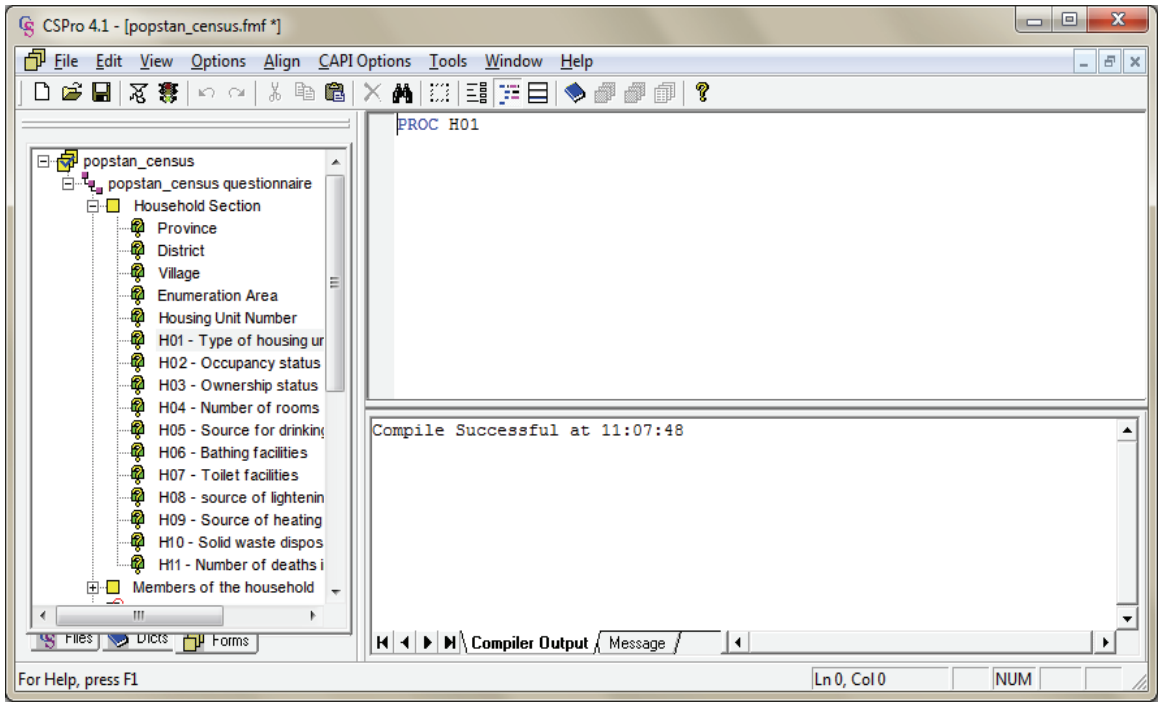
7.2. Logic for the example questionnaire

In this section we will go through the example questionnaire and program all the logic needed.

7.2.1. Skips – logic for question H01

Consider the paper questionnaire: Already in the first question there is a skip: If the type of housing is "Other collective", then skip to question P01 – i.e. we are supposed to skip all the other questions about housing. To write the logic for this, we need to do the following:

- Click on the H01 item in the forms tree on the left pane. CSPro now looks like this:



(Please note that on top of the upper left window it says "PROC H01". This means that we are writing code for question H01.)

- The action we want to happen, should take place *after* the interviewer has entered the answer, so we need to write a postproc.
- Enter the following lines just under the "PROC H01":

```
postproc
    if H01 = 4 then
      skip to MEMBERS_FORM;
    endif;
```

This means
"if the answer to question H01 equals 4, then skip to the form MEMBERS_FORM".

The semi colons at the end of the lines are to tell CSPro that this is the end of the statement. The keyword "postproc" is not necessary because, as mentioned above, postproc is the default behaviour, but it can be good to be explicit about the intentions of the code.

7.2.2. The If - then statement

The if statement executes different statements based on the value of a condition given. The syntax is the following:

```
If condition1 then
    Statements1;
[elseif condition2 then
    Statements2;]
[else
    Statements3;]
Endif;
```

(The [] indicates that this part is optional).

CSPro checks whether condition1 is true, and if so, it executes statements1. If not, it checks whether condition2 is true, and executes statements2. If this also is not true, it executes statements3. One can have as many elseif – then statements as needed.

The conditions in the if-then statement are logical expressions using the following operators:

- Equal: =
- Not equal: <>
- Less than: <
- Less than or equal to: <=
- Greater than: >
- Greater than or equal to: >=
- In
- Not, and, or

One more example of if-then-else statements (This does not really make any sense to do; it is just to show the use of the different operators):

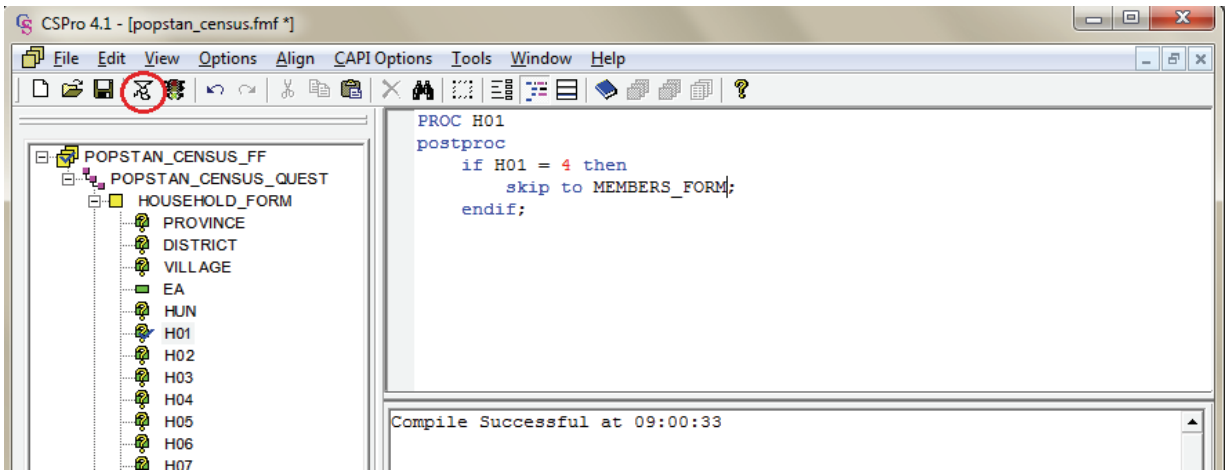
```
if x = 3 then
     z = 6;
elseif x in 4:5 or y in 7:9,12 then
     z = 7;
else if x >= 13 then
     z = 8;
else
     z = 9;
endif;
```

7.2.3. This item ($) – referring to the current item programmatically

Instead of writing H01 in the code above, we can use a dollar sign ($) instead. This is a short way of referring to the data item belonging to the procedure.

7.2.4. Compile the code

To check whether the code is correct, compile it by clicking the compile button as shown:



Run the application by clicking the traffic light icon next to the compile icon as explained earlier. If the code you have written is correct, then you see in the lower right part of CSPro "Compile successful at…" If not, error messages explaining what is wrong appear in this field.

7.2.5. Showing the value set in a pop-up response box

When testing the questionnaire, we discover a problem: How would the interviewer know the codes from the value sets that we defined?

What we need is a way to show the codes with its descriptions to the interviewer. This is done by popping up a response box where he or she can choose the correct answer. It can be done either at application level – once and for all, or at item level – specifying for each question whether we want it or not.

To do it at application level, click on the "popstan_census" element at the very top of the forms hierarchy on the left pane. Navigate downwards in the code editor on the right side until "PROC POPSTAN_CENSUS_FF", and enter the following code:

```
PROC POPSTAN_CENSUS_FF
preproc
      set attributes(POPSTAN_CENSUS_DICT) assisted on (question, responses);
```

The set attributes statement can change the appearance and behaviour of an item in different ways, but used this way, it shows both the CAPI question text and the content of the value sets to the interviewer. If an item has more than one value set, only the first one is shown. This behaviour can be changed with the setValueSet() function. (See paragraph 8.6.)

If we only want the CAPI question and not the value sets shown, use set attributes this way:

```
      set attributes(POPSTAN_CENSUS_DICT) assisted on (question);
```

To only show the value set and not the question, write this:

```
      set attributes(POPSTAN_CENSUS_DICT) assisted on (responses);
```

7.2.6. Ending the interview in the middle of the questionnaire – logic for H02

Next we see that question H02 needs some logic: If nobody lives in the housing unit, there is nobody to whom we can ask the rest of the questions, so we want to save the current record and go to the next housing unit.

This is done by entering the following for the H02 item:

```
PROC H02
  if $ = 2 then
     endlevel;
  endif;
```

The endlevel statement ends the current level, and as our application only has one level, this means that the ends the current record like we wanted.

7.2.7. Stop adding data to a roster or a multiply-occurring form before reaching the end.

If we run the application as it is now, we have to enter 30 members for each household. We want to change this, so that there can be an arbitrary number of people in a household. This can be done in several ways, for instance by adding an item asking how many people there are in the household before entering the names, or by having the interviewer enter a blank name when there are no more names to enter. We choose to use the latter method in our application.

We also need to store the number of people in the household to use in the next form about the details of the members.

Declare the variable for holding the total number of members of the household in the GLOBAL section of the application:

```
PROC GLOBAL

     numeric totalPopulation;
```

Then we can write the code for the first name items like this:

```
PROC P02A
     if $ = "" then  //current input field is blank
          totalPopulation = curocc() - 1;
          endgroup;
     endif;
```

Here, the function curocc() returns the number of the current occurrence in the roster, while the endgroup statement finishes the data entry for the current group (a group is a roster or a multiply-occurring form), hence the above statements translate to the following in normal English:

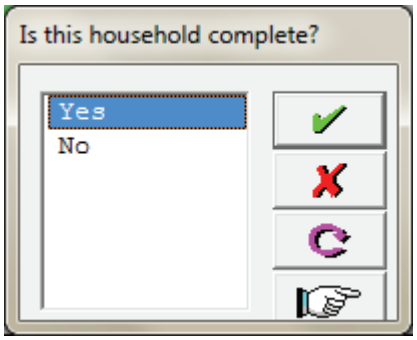
If the current item ($) is blank, the totalPopulation is to be given the value "number of the current occurrence minus one", and the asking for names of more people is to end. (We have to subtract one because we do not want to count the last, empty one.)

What if the interviewer enters a blank name by a mistake? Maybe we should give him a chance to verify that he really has entered all the people before ending the form. This can be done by adding the "accept" and "reenter" statements to the text above:

```
PROC P02A
     if $ = "" then
          if accept("Is this household complete?",
                    "Yes", "No") = 1 then
              totalPopulation = curocc() - 1;
              endgroup;
          else
              reenter;
          endif;
     endif;
```

(Meaning:
If the current item ($) is blank, then:
Pop up a question "Is this household complete?" If the interviewer "accepts" – i.e. says "yes" (1) to this question - totalPopulation is to be given the value "number of the current occurrence minus one", and the asking for names of more people is to end. Else (if the answer is NOT "yes" (1), then "reenter" this field.)

The accept function pops up a window like this:



The interviewer can confirm that he really is finished entering names of the household by hitting enter or clicking the green check button. If he is not finished, he first needs to select "no" before clicking on the green button.

The reason we ask "if accept(..) = 1", is that the function returns a number corresponding to the option selected. In this case, we want to quit adding persons to the household if the interviewer selects "yes", which is the first choice. If he selects "No", the function returns 2, and the else statement is executed instead (see 7.2.8 for more about boolean values)

The general format of the accept funtion is the following:

```
i = accept(heading, opt-1, opt-2[,...opt-n]);
```

Where *heading* is the header of the popup window, and *opt-1, opt-2*[,...*opt-n*] are the list of choices of the operator.
The variable i holds the return value from the function. In our case, we do not need this number later, so we do not store it in a variable.

If the interviewer says "no" to this question, the "else" part of the if-then-else statement is executed, and the interviewer has to reenter the name in the item.

Now we can add as many persons as we want to – up to 30 – to the persons form, but what about the mutiple occurring forms containing the details about the members of the household? We only want the repeating form to be repeated the same number of times as there are members of the household, hence we need to stop repeating the form after a while.

The most natural place to enter the code for this is generally in the last question of the form. But in our case, not all people of the household will be asked this question – only the women in fertile age. So instead, we put it in the preproc of the first question of the form. This question is asked to everybody:

```
PROC P03
Preproc
    //NB: More code to come here later

    if curocc() = totalPopulation + 1 then
        {Dealt with all members of the household. Stop
        showing the  repeating forms}
        endgroup;
    endif;
```

7.2.8. Checking validity of a date – using functions, arrays and boolean values

The questions about sex and age (P04 and P05) do not require any programming, but in question P06, the interviewer asks about the birth date of the interviewee. It is a good idea to program a test for the validity of the date entered. In the example questionnaire, there is only one date to be entered, so we could do it in the postproc of the year-item, but we prefer to write a function to do it, and call this function from the postproc. This way – if

the questionnaire contained more data fields, we only have to write the function once, and use it every time we need it.

First we need an array to store the number of days for each month. An array is a kind of variable that holds more than one value, and is declared like this:

```
Array MyArray(10);
```

Or

```
Array alpha(8) myArray(10);
```

The first one is an array with 10 elements of type numeric (the default type), and the second one is an array of 10 elements containing strings that are up to 8 characters long.
An array can also be declared and initiated in the same statement. The following creates the array "monthLengths" that has 12 elements, and the elements are initiated to the number of days for each month.

```
array monthLengths(12) = 31 29 31 30 31 30 31 31 30 31 30 31;
```

(February generally has 28 days, but we need 29 to allow for leap years).

Then we write the function to check whether a date is valid. A function is declared like this:

```
function [alpha(X)] functionName ([var1 [, [alpha(X)] var2 [,…]]]);
    //statements here
end;
```

Explanation:

- The [alpha(X)] part is only needed if the function is to return a string value. If it returns a numeric, this does not have to be specified, as this is the default return type.
- The var1, var2, .. are parameters which we need to use in the function. We can specify as many parameters as we want to. If any of them are strings, then it has to be specified with the word alpha(x) before it.

In our date checking function, we want it to return true if the date is valid, and false if not. Variables that are either true or false are called *Boolean variables*. They are common in most computer languages. In CSPro we use 0 (false) and 1 (true) instead of the words "true" and "false", hence we want the function to return 0 if the data is not valid, 1 otherwise. To make the function return a value, assign it to the name of the function.

Now we are ready to write the function. Functions must be declared (i.e.written) in PROC GLOBAL:

```
function isValidDate(day, month, year)
    if day > monthLength(month) then
          isValidDate = 0;
    elseif month = 2 and day = 29 then
          //checking for leap year
          if (year % 400 = 0) then
              isValidDate = 1;
          elseif (year % 100 = 0) then
              isValidDate = 0;
          elseif (year % 4 = 0) then
              isValidDate = 1;
          else
              isValidDate = 0;
          endif;
    else
          isValidDate = 1;
    endif
end;
```

And here is the explanation:

- First we check whether the day of month is bigger than the legal number of days for that month. (We declared the monthLength array to be like this:
  - monthLength(1) = 31 because January has 31 days,
  - monthLength(2) = 29 because February can have up to 29 days etc),
    so if for example day = 31 and month = 4, then
    31 > monthLength(4) = 30,
    hence illegal date, and we return 0.

- Then we have to check for leap years: A year is a leap year if it is divisible by 4, but not if it is also divisible by 100, though it is a leap year if it is divisible by 400 (hence the year 2000 was a leap year, while 1900 was not)
  The operator % gives the remainder in an integer division. For instance 10 % 4 = 2, because 2 is the remainder when 10 is divided by 4.

To use the function we just created, go to the item containing the year part of birth date, and put the following code there (note: We will improve this proc later):

```
PROC P06C
if not isValidDate(P06A,P06B, P06C) then
   errmsg("Not a valid date. Please reenter");
   reenter P06A;
endif;
```

The function is used by typing its name, and then to enter the items of the birth date as parameters. We want to write an error message if the date is *not* valid, hence the "if not isValidDate…"

The errmsg is a built-in function which displays an error message. We will come back to this in a moment.

The reenter statement we have already seen before, but this time we specify which field to reenter, and as we do not know which part of the date is invalid, we set the focus to the first part.

7.2.8.1. Exiting a custom made function or procedure

Sometimes you want to exit a function or procedure before the natural program exits it automatically. This can be done with the exit statement. It should, however, be used carefully, as it makes the program harder to read and debug.

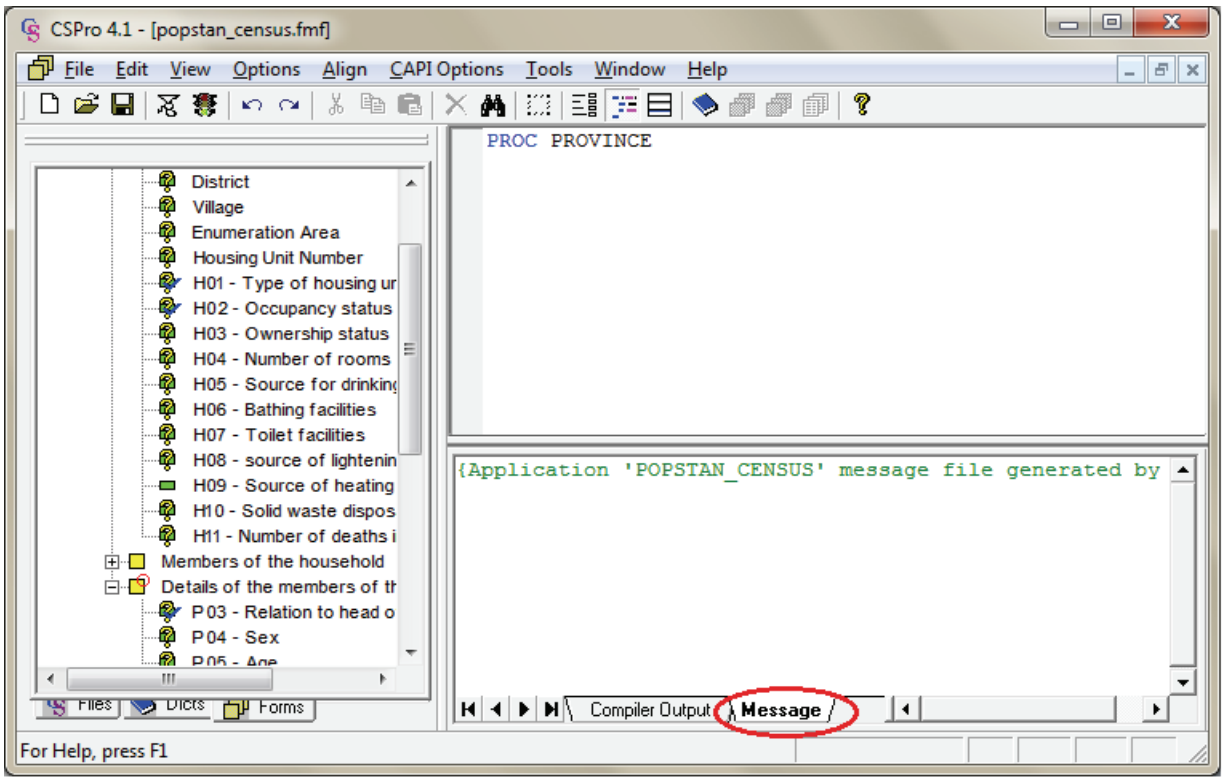
One reason to use exit, is to make it clear that the procedure or function only deals with some special cases, and these special cases requires a lot of code, so that the function itself is long. Or it could be used when searching for special values, and when it is found, exit the search loop (see also the seek function in section 8.3.4)

7.2.9. More about errMsg – error messages

In the paragraph above we used the errmsg() function to tell the interviewer that the data he entered is not valid.

A better way of doing it is to have all the error messages of an application in one place, rather than spread out in the code, and then refer to them by an assigned error number. This is especially important if the application is to be multilingual (see chapter 8.5). Another advantage is that you can reuse the error messages, and if you want to change them later, you only change it one place.

The place to put the error messages is in the lower part of the left pane – under the code editor: click on the tab "Messages":



Enter the error messages on the following form:

```
1 Not a valid date. Please reenter
```

(First an error number, then the error message)

Then you can refer to the error message by the number, like this:

```
PROC P06C
   if not isValidDate(P06A,P06B, P06C) then
      errmsg(1);
      reenter P06A;
endif;
```

We can also put variables in the errMsg statement. In the above example, we might want to put the date entered into the error message.
We put the variables into the text by using the following symbols

%d	inserts a number and display as an integer
%f	inserts a number and display as a decimal value
%s	inserts a text string

In our case, we want three integers with common date formatting characters between them, like this:

```
%d/%d-%d is not a valid date. Please reenter
```

Using the error message then looks like this:

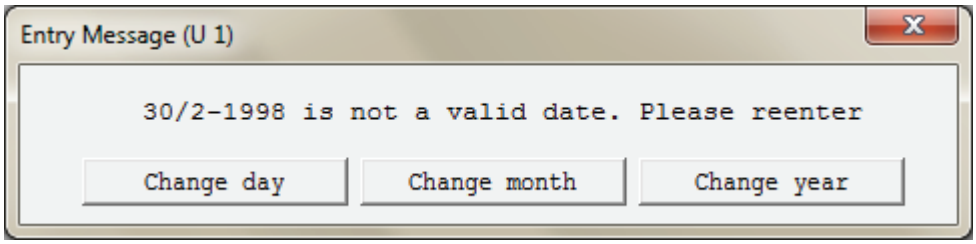
```
errmsg(1, P06A, P06B, P06C);
```

and the displayed message:

We can also let the interviewer decide which field to go back to, to reenter:

```
errmsg(1, P06A, P06B, P06C)
     select("change day", P06A,
            "change month", P06B,
            "change year", P06C);
```

Resulting in the following question from CSPro:



(Each pair of arguments to the select statement gives a button for the interviewer to choose a field to go to).

The final version of the proc P06C becomes:

```
PROC P06C
     //Checking whether the date is valid
     if not isValidDate(P06A,P06B, P06C) then
               errmsg(1, p06A, p06b, p06c)
                         select("Change day", P06A,
                                "Change month", P06B,
                                "Change year", P06C);
     endif;
```

7.2.9.1. Debugging using error messages

Debugging programs is often hard - sometimes it is even hard to know whether a value has been assigned to a variable or not, or what actually happens in an if-then statement.
Use error messages to check whether variables have the values you expect them to, to check that loops works the way you intended them, to verify that if-then-else statements behave as expected, etc.

7.2.10. The other skips in the questionnaire

In the paper questionnaire, there are a few more skips:

- The questions from P09 to end is only to be asked people 5 years or older,
- The questions from P11 to end is only to be asked people 10 years or older,
- If the answer to P12 is "no" and the interviewee is a woman, skip to P15A; if he is a man, go to the next person of the household
- Finally, the questions from P15A are only to be asked women between 15 and 49.

7.2.10.1. The skips at P09, P11 and P15A – skip to next

Let us start with P09. If the person is less than 5 years old, he or she is not to be asked any more questions, so we can continue with the next person. This can either be programmed in the postproc of P08, or in the preproc of P09. The latter is the better, because it intuitively makes more sense, as the logic is in the same place as the skip is happening.

To skip to the next line in the roster, do the following:

```
PROC P09
```

```
preproc
     if P05 < 5 then
          skip to next P03;
     endif;
```

The statement "Skip to next" takes you to the next person, and P03 is the first question of this form. The skip for question P11 and P15A is done the same way.

7.2.10.2. The skip at P12

Here we skip to the next person if the current subject is a male and to P15A if the subject is a woman in fertile age (between 15 and 49)

```
PROC P12
     if $ = 2 and P04 = 1 then
          skip to next P03;
     elseif $ = 2 and P04 = 2 and (P05 >= 15 and P05 <=49) then
          skip to P15A;
     endif;
```

7.2.11. More about skips

In addition to the skips "skip (to next)" and "reenter", there are several other kinds of skips. We shall have a look at them too.

When using all of the different skips, you can either specify where to skip to by the name of the field or by a variable containing the name of the field. The latter version is handy if you need the program to decide where to skip to, for instance in a function.

```
Alpha(10) skipToField;

If age < 5 then
    skipToField = "P10";
else
    skipToField = "P11";
endif;

//do some more processing

skip SkiptoField;
```

7.2.11.1. Reenter

As we have seen, this statement is used to make the interviewer re-enter a field. The field specified has to be *earlier* in the data path than were we currently are (i.e. reenter can only move backwards in the application).

The preproc of the field to reenter will not be executed, but the onfocus will (see more about preproc and onfocus in chapter 8.1)

7.2.11.2. Skip

As we saw in the example application, skip can be used to skip to another field, or to the next occurrence in a roster or a multiply-occurrence form. A skip moves forward in the questionnaire.

When skip to fieldname is used, none of the statements between skip and the preproc of the field we are skipping to, are executed (as opposed to advance (see below)). The items skipped, will be given the value NOTAPPL.

7.2.11.3. Advance

This statement moves forward field by field to the specified field, executing preprocs and postprocs as it goes. It is used to return to a position later in the questionnaire after having returned to a previous part of a questionnaire.

7.2.11.4. Move

This statement allows movement to any field without regard to whether it is before or after the current field. If you do not know whether the field has been entered or not, use the move statement.

If the move statement is used to move to a field before the current field, this statement acts like the reenter statement.
When moving forward in the path, the behavior is dependent on whether advance or skip is specified. The following two examples will act like the skip statement:

```
Move to P10;

Move P10 skip;
```

While the following statement

```
Move p10 advance;
```

acts like the advance statement

7.2.11.5. Noinput

The noinput statement prevents data to be entered in a field, and it can of course only be coded in the preproc or onfocus (before data is entered to the field).

This statement comes in handy if you know the answer to some of the question – for instance if the current person you are asking about is under 16, then you know he or she has never been married:

```
PROC SIVIL_STATUS
PreProc
     If AGE < 16 then
          $ = 6;
          Noinput;
     Endif;
```

7.2.12. Keeping track of the persons of the household of the example questionnaire

When entering data for each of the members of the household, we need to keep track of whom we are talking about. This can be done in the CAPI question area of the questionnaire.

We first need to declare two variables in the GLOBAL section; one to store the full name of the current person, and one to keep track of the person number we are dealing with.

```
Alpha(50) fullName;
Numeric   currIdx;
```

We assign a value to the currIdx (current index) in the preproc of the first question of the form about the details of the household members like this:

```
currIdx = curocc();
```

(As mentioned earlier, curroc() is a function returning the number of the current occurrence in a multiple-record)
Then we deal with the fullName variable. To access the current row of the roster, we can index the item we are interested in. For instance P02A(5) gives us the first name of the fifth person in the grid, and P02A(currIdx) gives us the first name of the person we are currently dealing with.

We want the fullName variable to contain names like this: “FirstName secondName”, i.e. we have to put together – or concatenate - the string P02A and P02B to one string. This is done by the function concat like this:

```
fullName = concat(P02A(currIdx), " ", P02B(currIdx));
```

But remember that we declared the names to be 20 characters long. If somebody has a name shorter than this, the rest of the characters become spaces. We want to get rid of the extra spaces, and that can be done with the strip function like this:

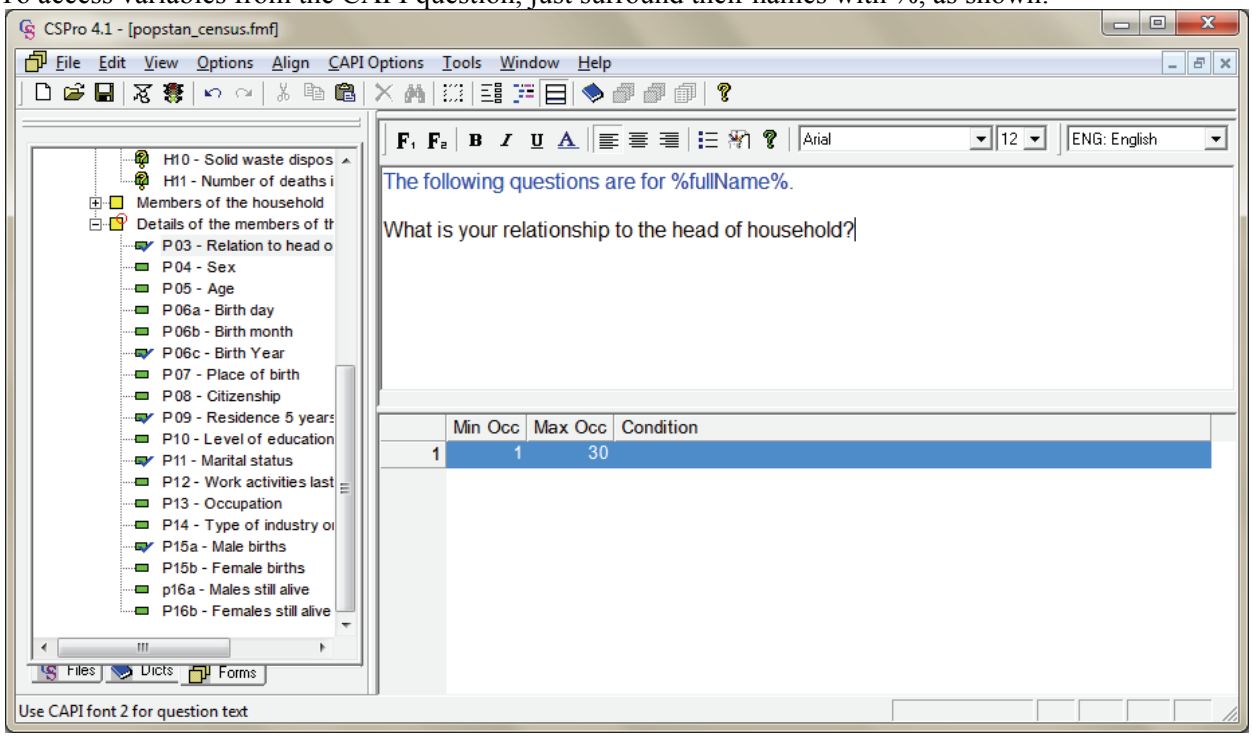
```
fullName = concat(strip(P02A(currIdx)), " ", strip(P02B(currIdx)));
```

The preproc for the item P03 finally looks like this (including the code we wrote in the beginning of the chapter to stop looping through the multiple occurring form when there are no more people to ask questions about):

```
PROC P03
preproc
      currIdx = curocc();
      fullName = concat(strip(P02A(currIdx)), " ", strip(P02B(currIdx))) ;

      if curocc() = totalPopulation + 1 then
           { all done, finish this roster }
           endgroup;
      endif;
```

To access variables from the CAPI question, just surround their names with %, as shown:



7.3. Other logic checks for the example application

There are still a lot of logic checks that can be programmed into the questionnaire, for example:

- Checking whether the age and the birth date correspond, and having the interviewer choose between reentering the age or the date.
- Deciding on minimum and maximum number of rooms in a household.
- Checking whether the number of children still alive is smaller or equal to the number of children given birth to. (As this have to be checked twice, one for each sex, it can be done in a function rather than directly in postproc to avoid copying and pasting code.)
- The provinces and districts should be value sets, and the latter value set should be dynamically built from a file according to what province is selected.

We are, however, not doing this here instead we will discuss more general topics in programming in the following chapter.

8. More about programming

8.1. Order of execution

When writing logic for the CSPro hierarchy, the order of execution is always the same: The proc GLOBAL or the declaration section will be executed when the data entry program starts up: either when the user clicks on the traffic light from CSPro itself or by clicking on a .pff file (more about files in chapter 9.2). Nothing in the GLOBAL section is actually executed; there are only definitions of variables and functions in this part of the logic file.

After the declaration section/Proc GLOBAL the following order of execution happens:

- Application or Forms file is executed. This is trigged the moment you start adding or modifying the data. It can be used to set behavior for all of the forms in one go, or to initialize arrays and similar.
- Level
- Form
- Roster
- Field

Within one type of the objects mentioned above, the order of execution is the following:

- **Preproc**: Executed just before the object is trigged, but only if you move *forward* onto the object. Preproc is NOT executed if you move backwards in the application.
- **Onfocus**: This is executed when the object gets focus or become active or alive, also via reenter statement or by mouse. If both a preproc and an onfocus exist for the same element, preproc is executed before the onfocus.
- **Killfocus**: Is executed when the object stops being active. The killfocus is also executed if a field is protected or if the *noinput* statement is used.
- **Postproc**: This is the default if none of the above are specified. It is executed when you *complete* an object, i.e. when you "flow" of it. It is NOT executed if you leave the field by using the mouse or similar.

8.2. Program loops

If the same task has to be done multiple times, or we have to loop though a lot of data to find what we are looking for, we use one of the loops below to achieve this.

- while
- do
- for
- next
- break

Let us go back to the example from chapter 6.1 where we had a survey asking questions to individuals. To the women in the survey, we could ask questions about number of children, fertility, family planning, and use of contraceptive (the questionnaire is very much simplified compared to this), and for some reason, we want to count how many different contraceptive methods the women has heard about, and also how many of them she has used.

We had a roster of 12 different contraceptive methods with the questions "Have you heard about this method?" and "Have you used this method?". The next fields in the form could be protected fields where we count how many times the woman's answer is yes for each of the two questions. And to do the counting, we loop through the entire roster.

As there are several ways to do this, we shall look at some of them.

8.2.1. While loop

The while statement executes the statements in a loop, as long as a logical condition is true. The syntax is the following:

```
while condition do
  statements;
enddo;
```

The condition can be anything that evaluates to true or false, but it is often on the following form:

```
while counter <= value do
```

The counter variable is just a variable that keeps track of where in the loop we are and when to stop.

In our example, the roster where the women can answer whether they have heard of a contraceptive is called HEARD_OF. In addition we need the counter mentioned above. Programmers often call counters like this "i". We also need a variable to keep track of how many "yes" answers are found so far, let us call it heardOfCounter[1]. The field to display the number of yes-es is called NUM_HEARD_OF, hence the following code:

```
PROC NUM_HEARD_OF
preproc
     numeric i = 1;
     numeric heardOfCounter = 0;

     while i <= totocc(HEARD_OF) do
          if HEARD_OF(i) = 1 then
               heardOfCounter = heardOfCounter + 1;
          endif;
          i = i + 1;
     enddo;

      NUM_HEARD_OF = heardOfCounter;
```

Comments:

- As the NUM_HEARD_OF field is protected, the calculation has to be done in the preproc, otherwise there will be an error message, and the application will crash.
- The structures i = i + 1 and heardOfCounter = heardOfCounter + 1 might look a bit strange in the beginning, but this is the way to increase the value of a variable by one. (One can, of course use any other operator to change the value of the variable, as long as one does not divide by zero)
- If the field NUM_HEARD_OF had not been a protected field, we would not need the extra heardOfCounter variable: We could have assigned the value directly in each loop by doing NUM_HEARD_OF = NUM_HEARD_OF + 1; (In our case, when the field **is** protected, CSPro would have tried to assign notAppl to the variable NUM_HEARD_OF if done like this, and the program would again crash)
- Do not forget to increase the i-variable for each loop! If this is forgotten, the condition in the while statement will always be true, and we have an infinite loop.

8.2.2. The Do loop

The do statement executes the statements inside the loop repeatedly either as long as a logical condition is true, or until a logical condition is not true anymore.

This loop structure is more complex than the while loop, and has the following format:

```
Do [var = expression] while/until condition [by expression]
```

[1] The typing style writing several words without spaces but with uppercase for each new word is called camelCase, and is also frequently used by programmers as variable names generally can not contain spaces.

```
  statements;
enddo;
```

In its simplest form, the do loop is basically the same as the while loop. Our example from the previous paragraph becomes like this:

```
PROC NUM_HEARD_OF
preproc
    numeric i;
    numeric heardOfCounter = 0;

    do i=1 while i <= totocc(HEARD_OF)
       if HEARD_OF(i) = 1 then
            heardOfCounter = heardOfCounter + 1;
       endif;
    enddo;

     NUM_HEARD_OF = heardOfCounter;
```

The differences compared to the while loop are

- Instead of initiating the variable i before the loop starts it is done inside the loop.
- The incrementation of i is implicit. The programmer should not do this himself (hence in this case, we avoid the infinite loop as mentioned above)

The do loop can, however, be used with the "until" keyword instead of "while". In this case the loop goes on until the condition becomes true.

The by clause of the do loop indicates what size the steps of the i variable should be. If, for instance the do statement looks like this:

```
do i=2 while i <= maxValue by 2
```

then only even values of i are considered in the loop.

8.2.3. For loop

The for loop is maybe the type of loop most commonly used in other programming language, but is CSPro, we do not need it that often. The for loop executes statements in the loop for each occurrence of a multiply occurring form, a roster or similar. Here is the format:

```
for numeric-var in [record | item | relation] group do
   statements;
enddo;
```

In our example counting contraceptive methods known to a woman, this translates to:

```
PROC NUM_HEARD_OF
preproc
      numeric i;
      numeric heardOfCounter = 0;

      for i in HEARD_OF do
          if HEARD_OF = 1 then
               heardOfCounter = heardOfCounter + 1;
          endif;
      enddo;
```

which looks a bit weird, because the i-variable we use as the looping variable is not used at all inside the loop. It does, however, keep track of where we are in the loop, and is hence necessary. In addition it cannot be changed programmatically inside the loop – it can only be referenced.

8.2.4. The Next- and break statements – Jumping to the next iteration of the loop – or out if it

Sometimes a loop is used to find a special occurrence of something. If we already know during the execution of an iteration in the loop know that this is not the occurrence we are looking for, we can stop executing the rest of the statements inside of the loop, and jump to the next occurrence. This is done with the next statement.

An example could be to find all the children of the head of a household for further processing. To do this, we use a loop to look through all the members of the household. If the relationship with head of household is not 3 or 4 (son or daughter), then we do not want to process any further, and jump to the next household member.

```
For i in HHMEMBERS do
     if RELATION <> 3 and RELATION <> 4 then
          next;
     endif;
     <a lot of processing the children here>
enddo;
```

If you want to break out of the loop entirely – not only to the next occurrence, the keyword “break” is used. Say that we just want to find the spouse of the head of the household, and nothing more:

```
numeric spouse = 0;
for i in HHMEMBERS do
      if RELATION = 2 then
          spouse = i;
          break;
      endif;
enddo;
```

8.3. Functions and operators

Frequently we need to calculate values in our applications, for instance many skips are based on the age of the interviewee, and this can be calculated from the birth date. Or maybe we want a (protected) field where we sum up values from other fields. To achieve this, CSPro has a selection of predefined functions and operators.

The functions mentioned in this chapter are only the most frequently used ones. To get a list of them all, refer to the CSPro manual.

8.3.1. Operators

CSPro has the following operators:

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Modulo or reminder: %
- Raise to power: ^

The first four probably do not need further explanation, but the modulo/remainder is maybe new to somebody. As we saw when creating the function to check whether a date is correct, this is the remainder when doing integer division. An example:

```
9 % 4 = 1
```

Because 9 divided by 4 equals 2, with reminder 1.

Example of the last operator – raise to the power:

```
3 ^ 2 = 9
```

Or 3 to the power of 2 equals 9.

8.3.2. Numeric functions

The **tonumber()** function takes a string as input and converts it to a number, given that the input string actually is a number. If non-numeric characters are found, the conversion stops.

The format is like this:

```
d = tonumber(string-exp);
```

This is frequently used together with the *sysparm()* function (see 9.1.1) to convert a parameter found in a configuration file (which are always strings) to a numeric.

To do the opposite, convert a number to a string, use the edit() function (see chapter 8.3.3)

Other numeric functions are:

- **Inc(x):** Increases the x variable by 1 (and hence is the same as the expression "x = x + 1". If two arguments are given; inc(x, y) , x is increased by y (ie. x = x + y)
- **Exp(x):** Raises the variable x to the power of e (natural logarithms)
- **Log(x):** Calculates the base 10 logarithm of x
- **Sqrt(x):** Calculates the square root of x
- **Low(x,y,z,..):** Returns the lowest value of the variables x, y and z. If one of the inputs is a special value, it is ignored.
- **High(x,y,z,..):** opposite of low; returns the highest value of the variables given.

8.3.3. String functions

While most of the numeric functions are not that often needed, the string functions comes in handy in many situations. Here are the most important of them:

The **Concat()** function concatenates – or chains – two or more strings together, as we saw in the example application in paragraph 7.2.12.

Edit(): This function converts a number to a string. The reason why it is not called tostring() is because it takes an "edit pattern" as input. This edit pattern is a string containing 9, Z, period or comma, where these symbols have the following meanings:

- **9:** Display a digit. If it is a leading zero, display it as it is.
- **Z:** Display a digit, but if it is a leading zero, display blank.
- **. (period):** display decimal character
- **, (comma):** Display thousands separator character

Any other character in the edit pattern is added to the new string as it is.

The syntax of the edit()-function is like this:

```
newString = edit(pattern, numExp);
```

where numExp is any numeric expression.
Examples:

```
A = edit("Z999",56);
```

The result of this is " 056". The edit() function is often used in combination with the sysdate() function which gives the current date. The following two examples formats the date in different ways:

```
A = edit("99:99:99",sysdate());
```

```
A = edit("99/99/99",sysdate("DDMMYY"));
```

And if you do not want a lot of leading zeroes in a variable containing income, but still want a zero value if there is no income, it can be done like this:

```
A = edit("ZZZ,ZZZ,ZZ9",INCOME);
```

Length(): This function returns the length of a string or a dictionary item as an integer value. If the argument is a dictionary item, it returns the length of the item, rather than the length of the content. (See Strip() function on how to strip away the leading zeroes)

Maketext(): As we saw in the paragraph about errMsg (paragraph 7.2.9), Strings can be formatted so that they contain values of variables or items. This is done in exactly the same way as it is done with the errMsg() (Paragraph 7.2.9). See also the CSPro manual for details.

Strip(): As we saw when we developed the example application (paragraph 7.2.12), the strip() function removes the trailing blanks from a string.

Tolower() and toupper(): Changes all the characters of a string to either lowercase- or uppercase. These functions are often used to compare contents of strings when the case does not matter.

8.3.4. Functions on records (multiple occurrence functions)

Multiple occurrence functions are functions that take multiple values as input and use them to produce a single value as output. Statistical functions (average, sum etc.) are always multiple occurrence functions.

We have already used one of the multiple occurrence functions in the example chapter, namely **curocc(),** while processing multiple occurring items (see 7.2.7). As we remember, the **curocc()** function returns the current occurrence number in a roster or form (If the form does not repeat, curocc() returns 1). There are two similar functions:

Maxocc(): This function returns the maximal number of occurrences in a form or a roster as defined in the dictionary, and will hence always return the same number for a given roster or form. This function, and the next, totocc() are often used in loops (see 8.2)

Totocc(): Returns the total number of non-missing occurrences in a roster or form (i.e. the number of occurrences that have been given input).

seek(): This function seeks and returns the occurrence number of the first item that meets the seeking criteria of the function call. If the item does not exist, it returns 0.

Example:

```
numeric femaleIndex = seek(SEX = 2);
```

Here CSPro seeks for the first occurrence in the roster where sex = 2

Swap(): Swaps two rows in a roster. This is for instance useful if we want the head of the family to be the first row in a roster of family members.

Count(), min(), max() sum() average(): Calculates basic statistical values, and can be restricted to parts of the roster by using a where clause.

- **Count()** returns the number of occurences,
- **min() and max()** return the minimum- and maximum values respectively,
- **sum()** returns the sum of the items,
- **average()** returns the average value.

Example:

```
Numeric averageIncome = average(Income where age > 12);
```

Insert() and delete(): inserts a row or deletes a row of a roster respectively.

8.3.5. The global function OnKey() and execsystem()

Sometimes we want to change the behaviour of certain keys. It could for instance be nice to display help information if the user presses a function key, or maybe to start up other programs external of CSPro (calculator, maybe, or a program to find your gps coordinates?).
This can be done by writing your own code in a function called **OnKey()**. If an OnKey() function exists, every single keystroke from the operator is sent to this function for processing. All the keys are identified by a number code which can be used in the function. See the CSPro manual for a list of which codes belong to which keys.

We want to change our Popstan census application (The example used earlier) so that if the user presses F2, CSPro will open a pdf file containing instructions. The F2 key has the code 113. If any other key is pressed, nothing special should happen.

To start up an external program, in this case Acrobat reader, we use the execsystem statement, which has the following format:

```
execsystem(alpha-exp, [maximized | normal | minimized],
                      [focus | nofocus], [wait | nowait]);
```

The options after the alpha-exp (alpha expression) parameter are optional, and define how the application should look. The alpha-exp contains both the name (and path) of the application to start up, and its parameter if needed. If there are any spaces in the path or file names, the whole expression has to be surrounded with single quotes (‘), while the path and filenames has to be surrounded by double quotes (“).

We need the function to check whether the key pressed has code number 113, if so: start up Adobe Reader, and let the function return 0 to indicate that the key should be ignored after we have dealt with it.

If the keystroke is any other key, we do not want to do anything with it, so we let the function return the key itself.

```
function OnKey(x)
   if x = 113 then
         execsystem('"C:\Program Files\Adobe\Reader\AcroRd32.exe"
    "C:\workshop\Popstan Census Application 2\Doc\popstan_faq.pdf"');
         onkey = 0;
   else
         OnKey = x;
   Endif;
 end;
```

In the above example the two lines about the execsystem should be on the same line, but the page is too narrow. Anyway: Paths should never be coded in the functions or procedure like this, because files can be on different places in different computers, and they might be moved around. Instead string variables containing the path should be declared at the global section in one place, so that changes only have to be done in one place.

8.4. End the whole data entry application – stop()

We have already talked about skips to change the order of the program flow ((see 8.2 and 8.2.11)), but what if you want to force the data entry application to terminate?

This can be done with the **stop()** function, and the parameter to the function decides how it behaves:

- No parameter or 0: Only the current case is stopped. Hence the effect is the same as pressing the stop button on the toolbar.

```
if $ = 99 then
   stop;
endif;
```

- If the parameter is non-zero, the whole application is terminated. This can, for instance be used if we have a menu to let the operator decide what questionnaire to fill out (see 9.4.): One of the menu options could be exit the data entry application.

8.5. Multilingual CAPI applications

Sometimes we need the CAPI application to be multilingual, and to let the interviewer switch between the different languages runtime. When an application is running, there are basically 3 kinds of texts that need to have in several languages: The error messages, the CAPI questions and the texts in the value sets. As stated earlier: In CAPI applications, there is no need to have long explanatory texts on the form itself, as we have the CAPI questions to instruct the interviewer about what to say and how to do it. If we chose to use symbolic names – say question numbers or similar – as texts related to the items in the form, no multiple language versions of the forms are needed.

We have already seen how to make the CAPI questions multilingual (see 6.3). Here we will learn how to deal with the value sets and the error messages.

8.5.1. Multilingual error messages

In 7.2.9 we saw how it was better to have the error messages collected in the messages "window", and referring to them by a message number than to spread them around in the application. Another reason why this is a good idea, is if you want multiple languages in your application. If the error messages are stored in the message window, they will automatically be stored in a separate text field with the same name as your application, but with the file extension .mgf, and it is easy to give this file to a translator who does not have to know anything about CSPro to translate the messages to other languages.

It is, however, important to structure the error messages in a certain way to make this work. Here is one way of doing it:

Let us say that we want our application to have three languages: English, Russian and Kyrgyz. We decide that all English error messages should have a message code between 100 and 199, all Russian error messages should have code between 200 and 299, and Kyrgyz messages codes between 300 and 399.

In addition we always let the same error messages have the same next to digits in all three languages like this:

```
121 This is an English error message
221 Это сообщение об ошибке на английском
321 Бул кыргызча ката жонундо маалымат
```

Doing it this way, it is easy to switch between the languages by first defining a numeric variable in the global section:

```
Numeric MSG_LANG = 0;
```

And also a function to reset this variable:

```
Function resetLanguage()
   If getLanguage() = "ENG" then
            MSG_LANG = 100;
   Elseif getLanguage() = "RUS" then
            MSG_LANG = 200;
   Elseif getLanguage() = "KYR" then
            MSG_LANG = 300;
   Else
```

```
                ErrMsg("Something is seriously wrong with this
                application. Please contact the developer");
      Endif;
    End;
```

The getLanguage() function is a built-in function that returns the string used to describe the name of the current language in the CAPI questions, so if you have named your languages ENG, RUS and KYR, this will work.

Then, to make the error messages multilingual, you just call this function before using the errmsg() function, and when calling the function with the above specified error messages, do the following:

```
errMsg(MSG_LANG + 21);
```

8.5.2. Multilingual value sets – the setValueSets() function

There are two functions to manipulate value sets: The setValueSet() function, and the setValueSets() function. It is the latter we use in multilingual applications, while the former is used if we want to set the actual content of the first value set – the one that is used in the application as default. This might happen if the value set is dependent on replies from the interviewee earlier in the questionnaire. See paragraph 8.6 on how to use the setValueSet() function.

As above, we want to our CAPI application to have three languages to switch between, English, Russian and Kyrgyz.

First, we need to define three value sets for each item in the dictionary that needs value sets – one in each language. As in the case with error messages, it is important to name the value sets structured, so that it is easy to manipulate them programmatically.

Let all of the value sets in English have names ending with _ENG, all the Russian value sets names ending with _RUS, and the Kyrgyz names ending with _KYR. (For instance, the value sets for the item about sex, could have names "SEX_ENG"," SEX_RUS" and "SEX_KYR" with values "Male" and "Female" in English etc.)

To change between the different value sets, we use the setValueSets() function. setValueSets() takes a string expression as input, traverses through all the items in the dictionary in search for value sets. For the items with value sets, it goes through the value sets for this item, searching for one that has a name containing the text string that was a parameter to the function. If this is found, it sets this value set to be the one used in the application.

So in our case, we need to add some lines of code to the resetLanguage() function of the last paragraph (changes in bold):

```
Function resetLanguage()
   If getLanguage() = "ENG" then
      MSG_LANG = 100;
      setValueSets("_ENG");
   Elseif getLanguage() = "RUS" then
      MSG_LANG = 200;
      setValueSets("_RUS");
   Elseif getLanguage() = "KYR" then
      MSG_LANG = 300;
      setValueSets("_KYR");
   Else
      ErrMsg("Something is seriously wrong with this application. Please
      contact the developer");
   Endif;
End;
```

Now we have a 3 languages application. It might, however, be a good idea to add a few more things to the resetLanguage() function, though. For instance if the accept() function is in use (see paragraph 7.2.7), one could

declare variables – named for instance yes and no, and give them corresponding values for each of the languages. The same goes for the string parameters to the select statement (see 7.2.9)

8.6. setValueset() – programmatically change the value set

There are cases when the value set is not known until runtime. Say, for instance that the application first asks for province, and then what district within the province. Then the value set of the district is dependent of the selected province.

A common way to do this is to use lookup files (see paragraph 9.3) to get the names of district and provinces, and then – in the postproc of the province field, populate two arrays with the codes and the names of the district respectively. There is, however, a catch: Due to historical reasons, the indexing of arrays actually starts at 0, not at 1 as all other indexes in CSPro. This fact is generally hidden for the CSPro programmers: The arrays look like their indexing starts at 1, and generally we do not need to know this. One of the few exceptions is the setValueset() function: This function requires that you pass to it zero-based arrays.

In the following example, we assume that an external dictionary called AREA_DICT, to use with the lookup file has been created. In the PROC GLOBAL section, the following variables have to be declared:

```
PROC GLOBAL
 numeric i;
 array aDistrictCodes(99);                  //Codes of the value set
 array alpha(32) aDistrictNames(99);        //Names of the elements
                                          //of the value set
```

And in the proc for the province item we first have to load the districts for the province entered. This is done with the loadcase function:

```
PROC PROVINCE

      PROVINCE_CODE = PROVINCE;
      if loadcase(AREA_DICT, PROVINCE_CODE) = 0 then
                  errmsg("Invalid province, please reenter");
                  reenter PROVINCE;
      endif;
```

Then we copy the district codes and names into the arrays declared in the global section. The noccurs() function returns the numbers of occurences of the province code in the lookup file.

The reason why the indexing in the loop is i-1 rather than the expected i, is because of the aboce mention fact: That the setValueset() function requires zero-based arrays.

```
      do varying i = 1 until i > noccurs(AREA_DICT.AREA_REC)
           aDistrictCodes(i-1) = DISTRICT_CODE(i);
           aDistrictNames(i-1) = DISTRICT_NAME(i);
      enddo;
```

We also need to tell the setValueset() function how many elements there will be. This is done by putting an extra element in the codes array with the value “notappl”:

```
      aDistrictCodes(i-1) = notappl;
```

And then we can call the setValueset() function

```
setvalueset(DISTRICT, aDistrictCodes, aDistrictNames);
```

8.7. Trace – Makes debugging easier

Sometimes it is hard to understand what is actually going on in your application. To find out step by step what is executed when, the *trace* function can be used. This function outputs the statements in the order they are

executed to a separate window, and lets you see what is going on inside of a loop, or when an if-then-else statement is executed.

Trace can be run in two different ways: One possibility is to only output debug messages and nothing else. These debug messages can be more complex if you use the maketext() function to create the message (see 8.3.3)

To use trace this way, turn it on with the following command

```
Trace(on);
```

And then you can send messages to the output window by the following:

```
Trace("This is a message to the message window");

Trace(maketext("A string with argument %d", myVar));
```

To turn it off:

```
Trace(off);
```

The other way is to have trace output all the statements that are executed, do the following:

```
Trace(on);
Set trace(on);
```

(The first statement only turns on the trace window, while the second tells trace to output all the statements executed).

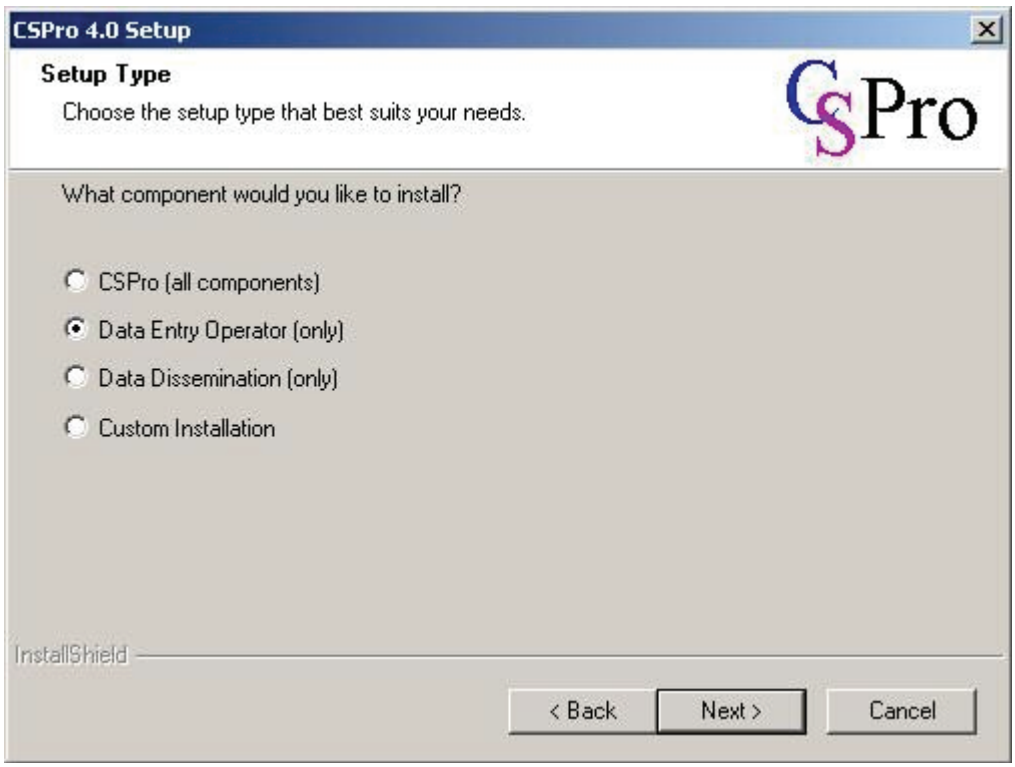
Statements will be output to the trace window until you add this line in the application:

```
Set trace(off);
```

9. Miscellaneous topics

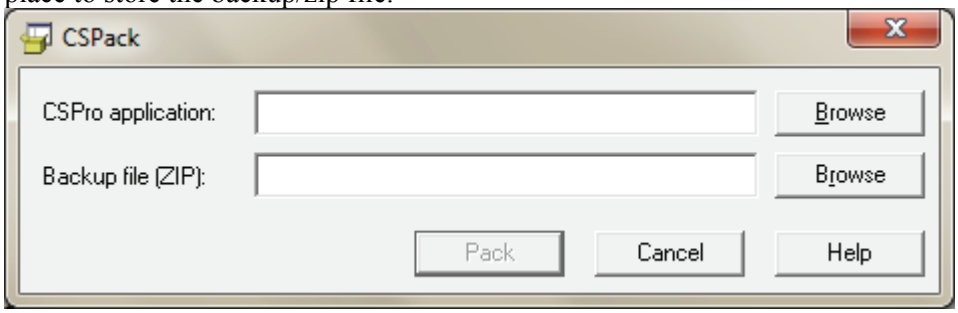
9.1. Making the application ready to run on laptops/tablet PCs

Before you can run a data entry application, you have to install the CSPro data entry operator software. To do this, you need the CSPro installation file (nowadays called CSPro41.exe). After double-clicking on this file, you get a question about what setup type you want. Select "Data Entry Operator (only)":



And click next. Click finish to finish the installation.

Next you need to pack the application into a zip file: Open CSPro and choose tools – Pack Application. CSPro then asks what application to pack. Use the Browse button to navigate to the .ent file, and accept the default place to store the backup/zip file.



Copy the zip file to the tablet pc/laptop, unzip it and double click on the file with the traffic light icon – the .pff file to start up.

9.1.1. Startup parameters: The .pff file and the sysparm() function

You can give CSPro startup parameters by specifying them in the .pff file (Or you can actually build a .pff file programmatically. This is done with applications using menus to select different questionnaires.)

Say for instance that we have a country wide survey, and we know what laptops/tablets to use where for each district. Then, instead of asking about district in the CAPI application, we can put the district code in the in the .pff file on beforehand.

The following is the .pff file that CSPro generated for our example application before we start working on it manually:

```
[Run Information]
Version=CSPro 4.1
AppType=Entry

[DataEntryInit]
```

```
Interactive=Ask

[Files]
Application=.\popstan_census.ent
InputData=.\Data\popstan.dat


[Parameters]
```

Not much exiting here. The lines in `[brackets]` are just comments to make it easier to see what is what. The two first lines of instructions, tells us that we are running CSPro version 4.1, and that this is a data entry application, while the "Interactive=Ask" lets the interviewer decide what types of messages to display.

Next we see that the application is popstan_census.ent, and that this file is situated in the same folder as the .pff file itself (the ".\" means this folder). The last line with instruction tells us that the data file will be named "popstan.dat", and this is situated in the Data folder which is to be found in the current folder.

You can define several different parameters in the [DataEntryInit] section of the file, the most commonly used being:

- OperatorID=John: If each interviewer has his own laptop/tablet, you might as well set the operator ID in the pff file instead of asking for it each time he starts up.
- StartMode=add: This automatically sets the Data entry program to be in add mode. Other options are "Modify or "Verify".
- Lock=Verify,Stats: defines the operations which the user does not have access to. In this case, he cannot run the data entry program in verify mode, and he has not access to read the statistics of the data entry. The Options are "add"; "modify", "verify" and "stats", separated by commas.
- FullScreen=Yes: If the application should be run in fullscreen or not.

In addition to the [Files] section there can be an [ExternalFile] section. If this is present, it means that a second dictionary is linked to the data entry application. (Using more than one dictionary is for the time being not covered in this tutorial, please refer to the manual for information).

When giving input parameters to the application, one has to divide between those being a part of the ID and those not. In the example mentioned in the beginning of this paragraph, having district and maybe province code as input parameters in the pff file, we assume that these variables are part of the ID, and hence should be put in the [DataEntryIDs] section rather than in the [Parameters] section. It can be done in the following way:

```
[DataEntryIDs]
Province=12
District=05
```

As these fields are the same for all housing unit throughout the province and district, these variables have to be given the option "Persistent" in the Field properties (right-click on the field in the form and select "persistent").

In the [Parameters] section, you can define your own startup parameter the following name:

```
[Parameters]
Parameter=your choice
```

This is mostly used as command line parameter, but you can access it from the application, by using the sysparm() function as a string. Once it is received in the program, it can be parsed for further usage.

9.2. File types and folder structure

After working with CSPro for a while, one notices that it creates lots and lots of files. Most of them are text files in a format readable for CSPro, and generally one does not have to relate to them.

To understand what files are which, it is recommended to turn on "show file extensions" in Windows (In Explorer, select Tools -> Folder Options and untick "Hide extensions for known file types")

Here is an overview of the most important files of the data entry part of CSPro, though (For other file extension, please refer to the CSPro Manual):

- **ent**: This is the Date Entry Application File – the main file that pulls together all the other files needed for the data entry application. To open the application, open this file.
- **dcf:** This is the dictionary file. The dictionary is independent of the rest of the CSPro application, and can be changed without changing the rest of the application. It is important to be careful if doing this, because it might destroy applications dependent on it.
- **fmf**: It is the form file
- **app**: the logic for the data entry application
- **mgf**: The message file. Here are the error messages that you specified in the "Message" tab under the editor where you write the logic is saved (see 7.2.9). If you, for instance, need the error messages to be translated to other languages, this can be done directly in this file by translators who do not necessarily know CSPro.
- **qsf**: the CAPI question text file. This is, however, not structured as simple as the message file, so to translate CAPI questions, the translators have to use (a small part of) CSPro.
- **pff**: This is the file used to start the application. It is also just a text file, and if parameters for the application are needed, they can be specified here (see 9.1)

In addition there are a lot of data files containing the actual data collected by the application. If, as recommended earlier, the main data file is given a name with the file extension .dat (CSPro does not automatically give the data files a file extension, you have to specify it yourself), here is a list of the various extensions:

- dat: The main data file
- idx: This is the data file index used by CSPro to ensure that each case has a unique id. The file is binary, and neither can nor should be changed manually.
- log: Contains statistics about the operator: Operator ID, start and end time, number of keystrokes, etc.
- not: If editnote() or putnote() is used, the comments are saved in this file. Each record contains case id, name of data item of which the note belongs, and the note itself.
- sts: Data file status. Contain information about partial save and verification status.

9.2.1. Structuring your files

As seen above, CSPro generates a lot of files, and it is a good idea to create a file hierarchy to keep track of them. Especially if the application is big with several dictionaries, lookup files, forms, etc. But even if it is a small, simple application, it is recommended to at least make a separate folder for the data files.
The following is a suggestion of a folder structure:

- Application folder
  - Data
    - Backup
    - <all the data files>
  - Dictionaries
  - Entry
    - <here the actual application files is stored: fmf, app, ent>
  - Lookup
    - <If external lookup files are used, put them here>
  - …

9.3. Using lookup files

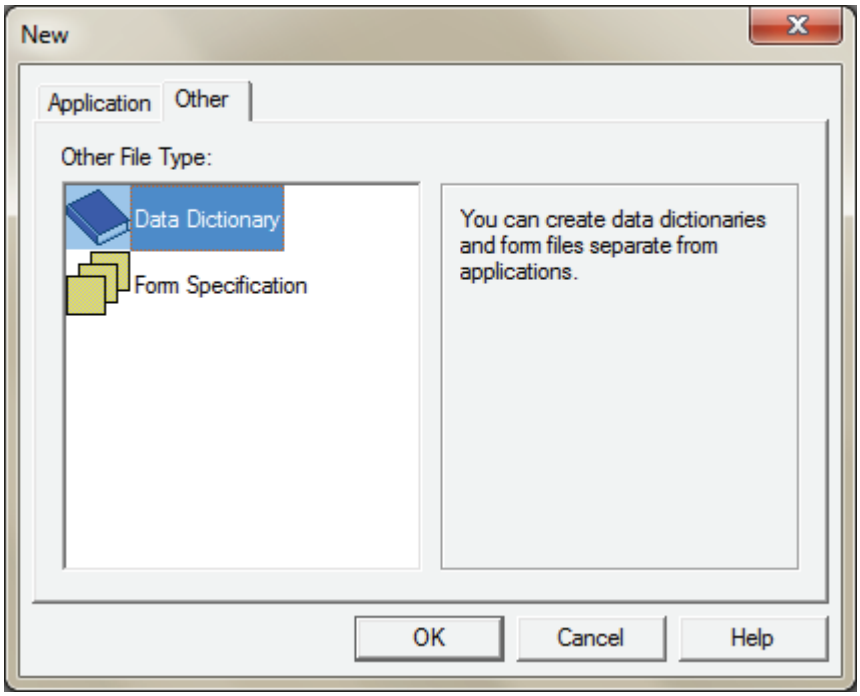
We want the interviewers to type as little as possible. Text fields are often a source of errors in the data, so it is much better to have the interviewers select from a value set rather than type a value. To achieve this, an ASCII text file can be used as a lookup file. This can be used for:

- Geographic codes and names. Your application could show the name corresponding to the code the user keyed.
- Industry and occupation codes. Your application could make sure the user keys a valid code. It could also display the names associated with the code.
- Last year's data. Your application could look up a corresponding field from last year's data and calculate a percentage change.
- Dynamically create value sets according to the input of the interviewer
- Generalized menu choices. Your application could read a lookup file and show the contents on the screen as a menu, then convert the user's choice to a code.

To use lookup files, we first need to create an extra data dictionary and associate this with the file to use as a lookup file. Then we create the main dictionary, and connect the relevant items in it with the items of the lookup dictionary.

9.3.1. The dictionary of the lookup file

When starting up CSPro after selecting "Create a new application" and clicking OK, make sure you change the tab in the next window to other, and choose "Data Dictionary", then press OK:

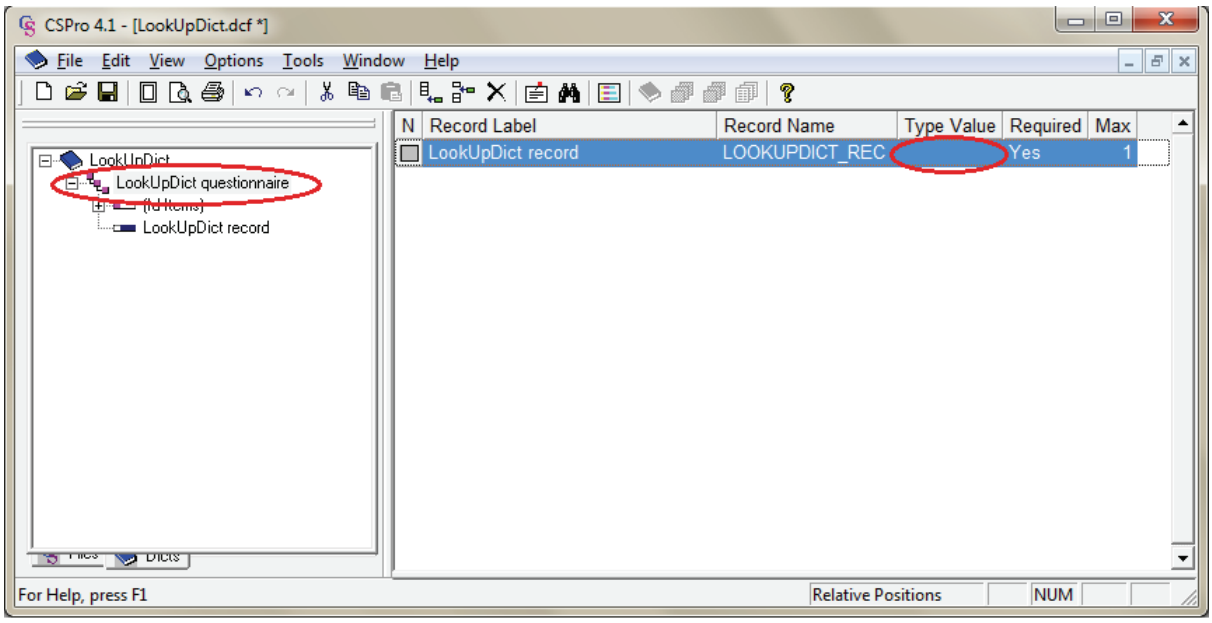


Give the dictionary a name (e.g.LookUpDict), and press OK, as before.

Say that our lookup file is a text file containing two digits codes for province and district respectively, and then, if the code for district is 00, the name of the province, else the name of the district. The following is an excerpt from the file:

```
0000Popstan
0100Artesia
0101Dongo
0102Idfu
0103Jummu
0104Kars
…
```

As the lookup file does not have record type as the first digit, we have to remove the Type value from the record: Highlight the LookUpDict questionnaire in the left pane, then right-click on the LookUpDict record, choose modify record, and remove the "1" in the type value:

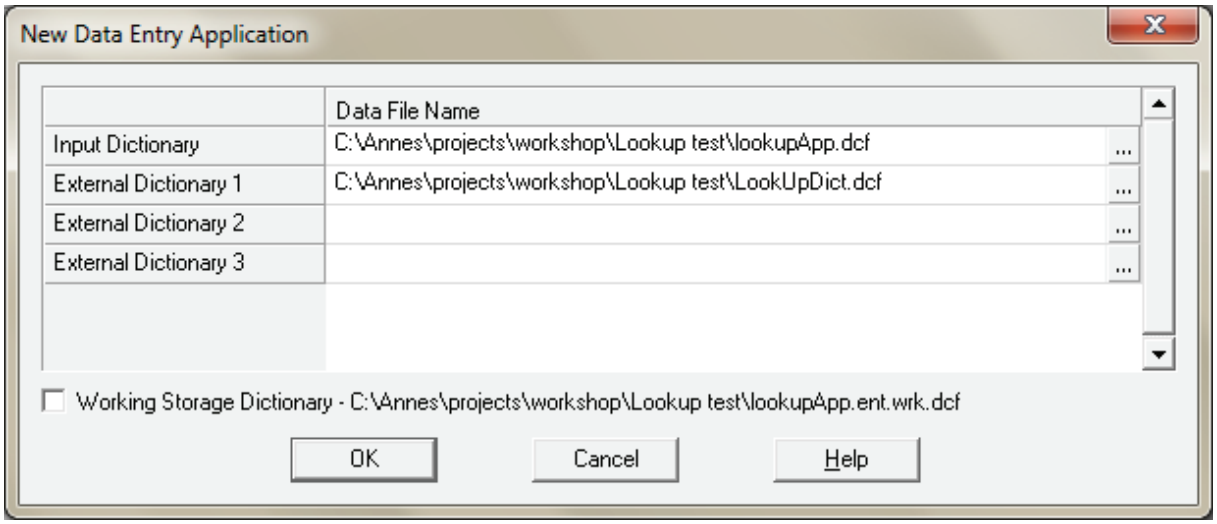


Then make the ID fields Province and districts, but give them names that show they are lookup variables, for instance LF_PROVINCE and LF_DISTRICT (this is important because we also need ID fields for the application dictionary, and we need a way to distinguish between them).
In the record, make an alpha field containing the name of the province or district.

Now we are finished making the lookup dictionary, and ready to make our application. Save and close the LookUpDict.

9.3.2. The main application

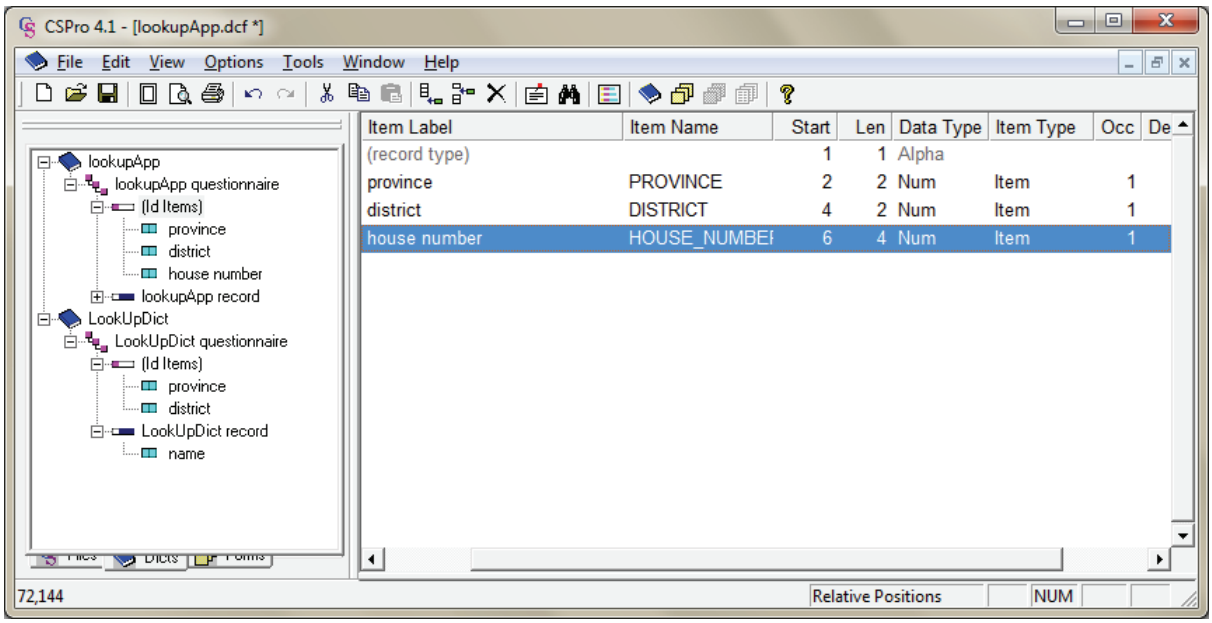
Make a new application as done before, but when asked about input dictionary: accept the default input dictionary, and also add the dictionary just created as an external dictionary:



(At this stage, CSPro does not need to know which file to use as lookup file, it will ask for it first time (and every time) you run the application)

Most of the development of the application is straight forward: Make the ID elements, then the other elements and the forms etc. The only thing new is how to connect the two dictionaries and also to read the file. This is done by programming just a few lines.

Say that you end up with the following dictionary:



And then make two forms: One where the interviewer types the province and district he is working in, and one for the rest of the fields.

The programming is the following: In the first form, for the "DISTRICT" field:

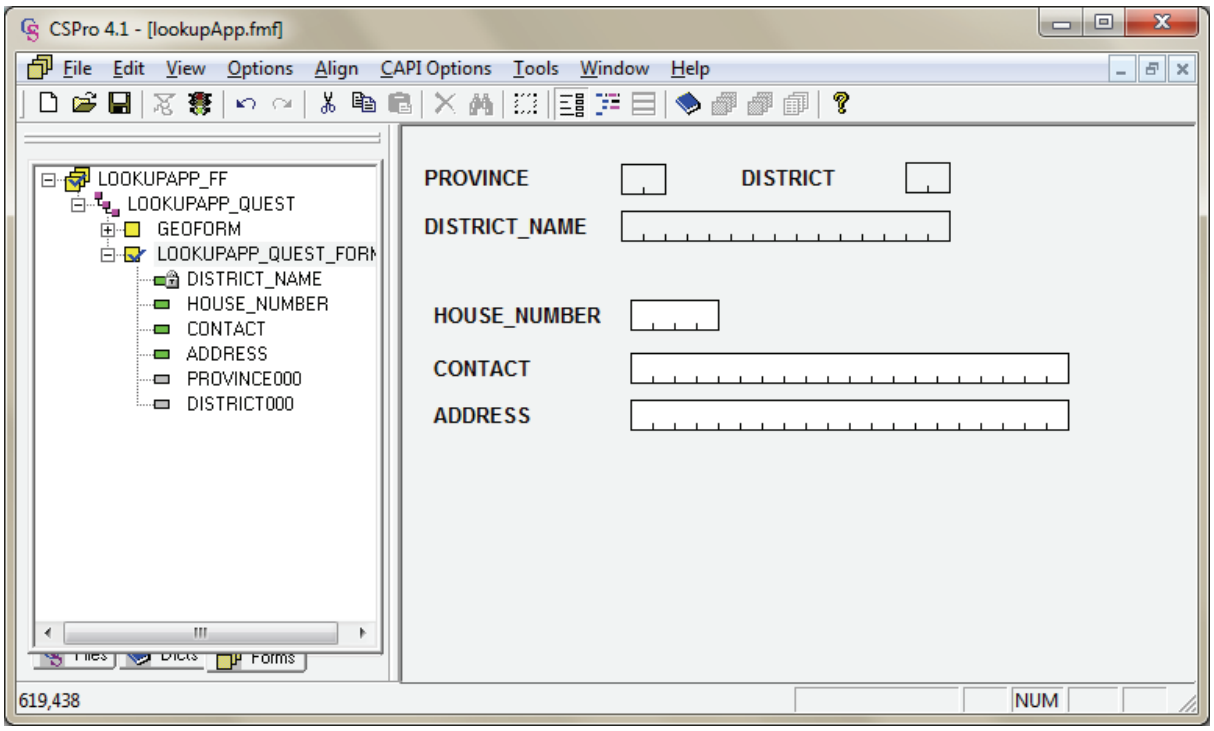
```
PROC DISTRICT
  LF_PROVINCE = PROVINCE;
  LF_DISTRICT = DISTRICT;

  numeric found = loadcase(LF_LOOKUPDICT, LF_PROVINCE, LF_DISTRICT);

  if found <> 1 then                    //record is not found
        errmsg("Could not find province %d and district %d",
               PROVINCE, DISTRICT);
      reenter;
  endif
```

Here the two first lines of the procedure assign the values that the interviewer typed in the PROVINCE and DISTRICT fields to the corresponding fields in the lookup dictionary. The next line uses the loadcase() function to read the specified case into the memory of CSPro. When this is done, all the variables defined in the lookup dictionary are available to the application.

We want to display the province and district variables in the next form together with the name of the district. Say the form looks like this:



(the fields from the external dictionary are all protected, as we do not want the interviewer to change these.)

By using the preproc of the **form** (not the DISTRICT_NAME field), we can display the name of the district on the form like this:

```
PROC LOOKUPAPP_QUEST_FORM
preproc
       DISTRICT_NAME = LF_NAME;
```

(Notice that the opposite of what we did earlier is done here: We assign the value of the item in the lookup dictionary to the field in the form)

9.3.3. Using lookup files not in fixed format

The above example uses a lookup file where all the variables are of fixed length. CSPro only accepts files of this format, so what if we have a lot of variables which are not of fixed lengths? There are several ways to deal with this – for instance import the whole record into one variable, and then use string functions to split it up into substrings or numeric. The problem with doing it this way, is that we lose the easy indexing and searching implicitly found in the loadcase() function, so that we have to search though the whole file every time we look up a variable.

A better way to do it is to make a mix of the two approaches: Make sure the ID variables are in fixed positions (or maybe rather as many as possible of the variables, regardless of whether they are ID items or not), and then import the rest of the variables to a long alpha variable which you parse into substrings. This requires, of course, that all the variables of fixed lengths are in the beginning of the record.

9.4. Creating a menu application for multiple questionnaires

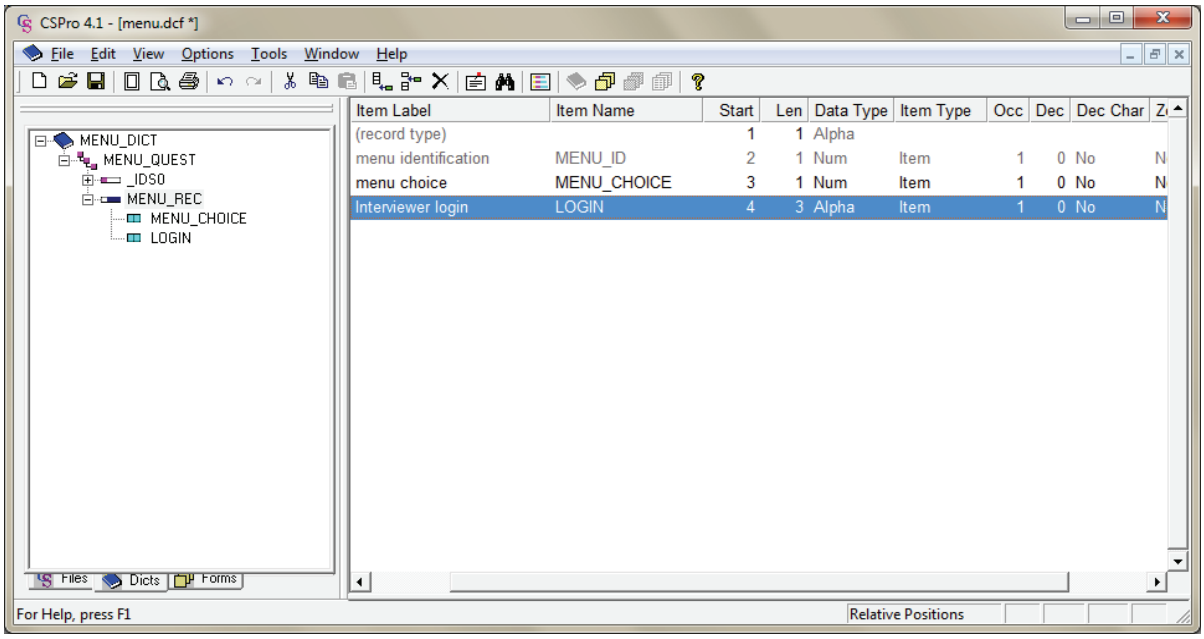
Say that we have a survey system consisting of several different questionnaires, and we want the interviewers to be taken to a menu system when he starts up CSPro, so that he can choose what questionnaire to use on the fly.

In this paragraph we will make such a system step by step. We assume the different questionnaires are already finished, so that what is left, is only the menu system.

9.4.1. The dictionary

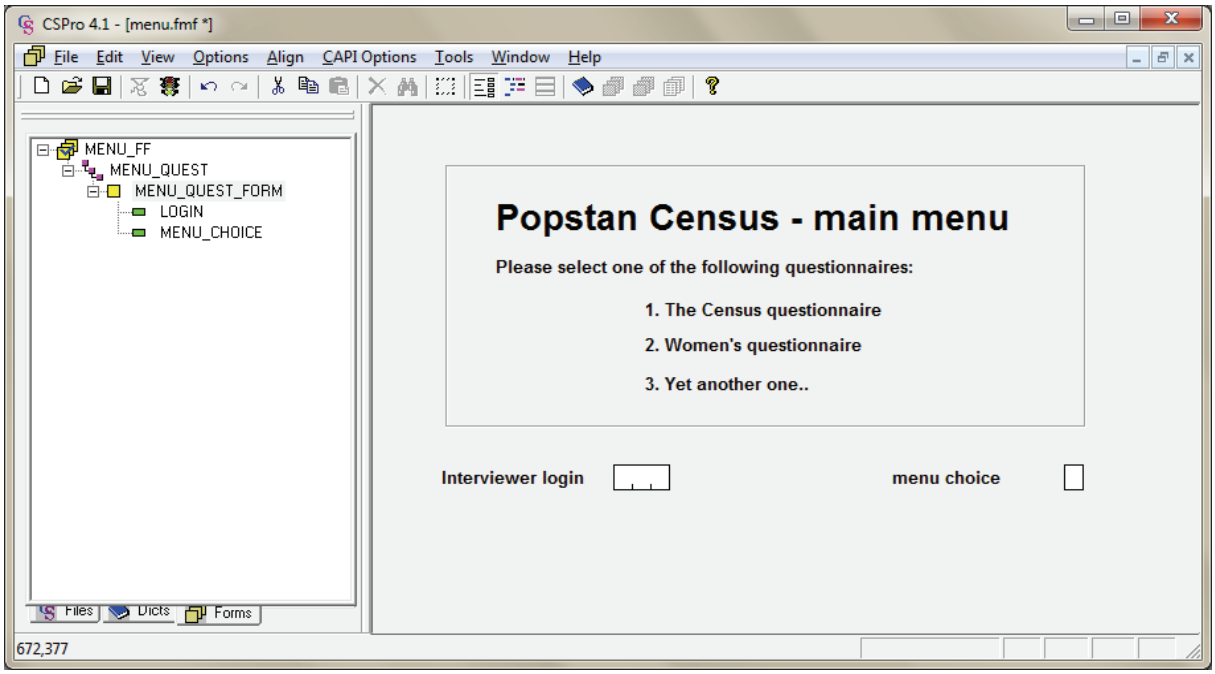
The dictionary of the menu system is a very small one: As CSPro requires that there is an ID item; we have to make one – even though we do not need it for the menu system. Just keep the default item that CSPro made for us.

In addition we need an item to hold the choice of the interviewer, and that is all that is really needed. Of course other fields can be added – to use in the actual questionnaires or to use as start-up parameters for the questionnaires. We will add a field where the interviewer can type his name or login code, and then we will save the data in a file with a name containing this information. Hence the dictionary might look something like this:



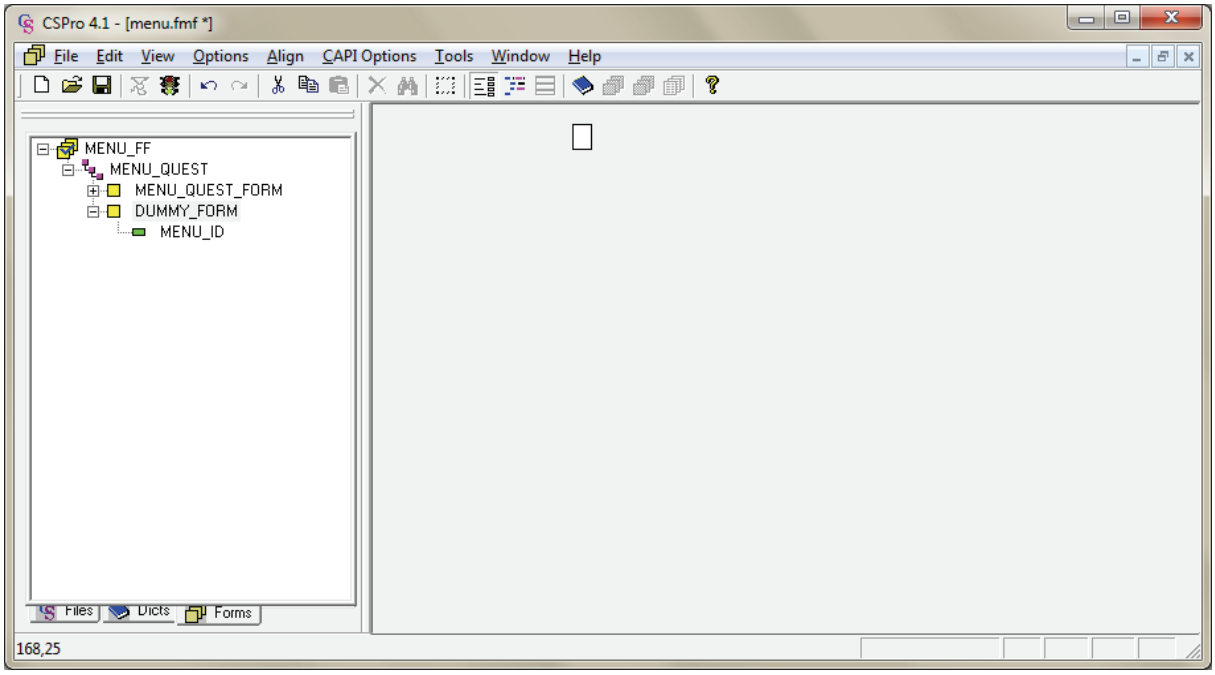
9.4.2. The Menu form(s)

Then we have to make the menu form – something like this:



As you can see, there is no ID item on this form, and as stated in the beginning of this tutorial, all applications have to use the ID item. As we do not need it, and we do not want to mess up the nice looking menu with items not in use, we make an additional form where we put the ID. This form will never be display to the user – or

actually even be used, as we choose what form will be the next programmatically according to the user's choice. I prefer to make this dummy form as minimal as possible, to indicate that it is not in use. Something like this:



9.4.3. The logic

In paragraph 9.1.1 we talked about the pff-file. This file is used as the start-up file for the interviewers, and it contains instructions to CSPro on how to launch the application. In our menu application, we shall programmatically create one pff-file for each of the different menu choices on the fly, and then start it up. This way the interviewer will be taken to the right questionnaire.

We first need to create variables to hold names for the pff file, data file and data folder. In addition we need a variable of type *file* to hold a "pointer" to the actual file (not only a name). As before, these variables have to be declared in the GLOBAL section:

```
PROC GLOBAL

      alpha (50) pffFileName = "launcher.pff";
      alpha (60) dataFileName, applicationName;

      file pffFile;
```

We also need some new functions to be able to write normal text files (which the pff file actually is):

- Setfile() function: This function assigns a physical file to a dictionary or a declared file – either the name of an external dictionary or an external file must be specified. And the external file must have been declared with the file statement (as we did above). The syntax is this:

  ```
  OK = setfile(external-dict-name | file-name, alpha-exp,
                        [update | append | create]);
  ```

  the last argument; update, append or create, indicates what we want to do with our file.

- fileWrite() function: This function write one line to a text file:

  ```
  OK = fileWrite(file-name, alpha-expr[,p1[, p2[,…pn]]);
  ```

  As above, the file-name has to be declared with the file statement in proc GLOBAL

- close(): After writing to a file, we need to use this function to close it, and hence release it so that other applications can use it
- filecopy(): To make a backup of the data file in case something goes wrong
- execpff(): to execute the pff file that we create programmatically.

Here is the code for the MENU_CHOICE field:

```
PROC MENU_CHOICE

      //first assign names to application and datafile
      //according to what questionnaire is chosen

      if $ = 1 then                                  //census questionnaire
           applicationName = "..\\Popstan Census\\census.ent";
           dataFileName = maketext(
                              "..\\Popstan Census\\Data\\census_%s.dat",
           LOGIN);
      elseif $ = 2 then                              //women's questionnaire
           applicationName = "..\\Popstan Census\\womenQuest.ent";
           dataFileName = maketext(
                     "..\\Popstan Census\\Data\\women_%s.dat",
           LOGIN);

      else
           errMsg("Illegal choice!");
           reenter;
      endif;


      //CREATE A PFF FILE TO LAUNCH
      setfile(pffFile,strip(pffFileName),create);
          //then write in it
          filewrite(pffFile,"[Run Information]");
          filewrite(pffFile,"Version=CSPro 4.1");
          filewrite(pffFile,"AppType=Entry");

          filewrite(pffFile,"[DataEntryInit]");
          filewrite(pffFile,"Interactive=Ask");
          filewrite(pffFile,"OperatorID=%s",LOGIN);
          filewrite(pffFile,"StartMode=add");
          filewrite(pffFile,"FullScreen=Yes");

          filewrite(pffFile,"[Files]");
          filewrite(pffFile,"Application=%s", strip(applicationName))
          filewrite(pffFile,"InputData=%s", strip(dataFileName));

          close(pffFile);

          //backup the data file in case of problems
          filecopy(maketext("%s",strip(dataFileName)),
                                 maketext("%s_Backup_%d",
                     strip(dataFileName),sysdate("YYYYMMDD")));

          //lanuch pff file
          execpff(strip(pffFileName));
```

```
//close the menu program(?)
stop(1);
```

Just a little explanation is needed. Consider the line:

```
applicationName = "..\\Popstan Census\\census.ent";
```

It might look a bit strange with the two dots and then the two backslashes. Let us talk about the dots first: It is important to avoid writing the whole path of a file or folder in the code, because if we later move the file or folder, then we have to change the path in the code too. In order to avoid this, we use *relative* paths; i.e. relative to where "we" are (or, more correctly, where the application is). The symbols for relative paths are the following:

- . (dot): symbols the current folder
- .. (two dots): symbols the folder *above* the current folder.

Hence "..\\..\\MyFile.dat" is located two steps up from where the application is.

The reason why we need two \-es instead of only one which is the common sign for folders, is that only one is used for other symbolisations, for instance: \n means newline, \t means tab-button etc, hence \\ simply means \.

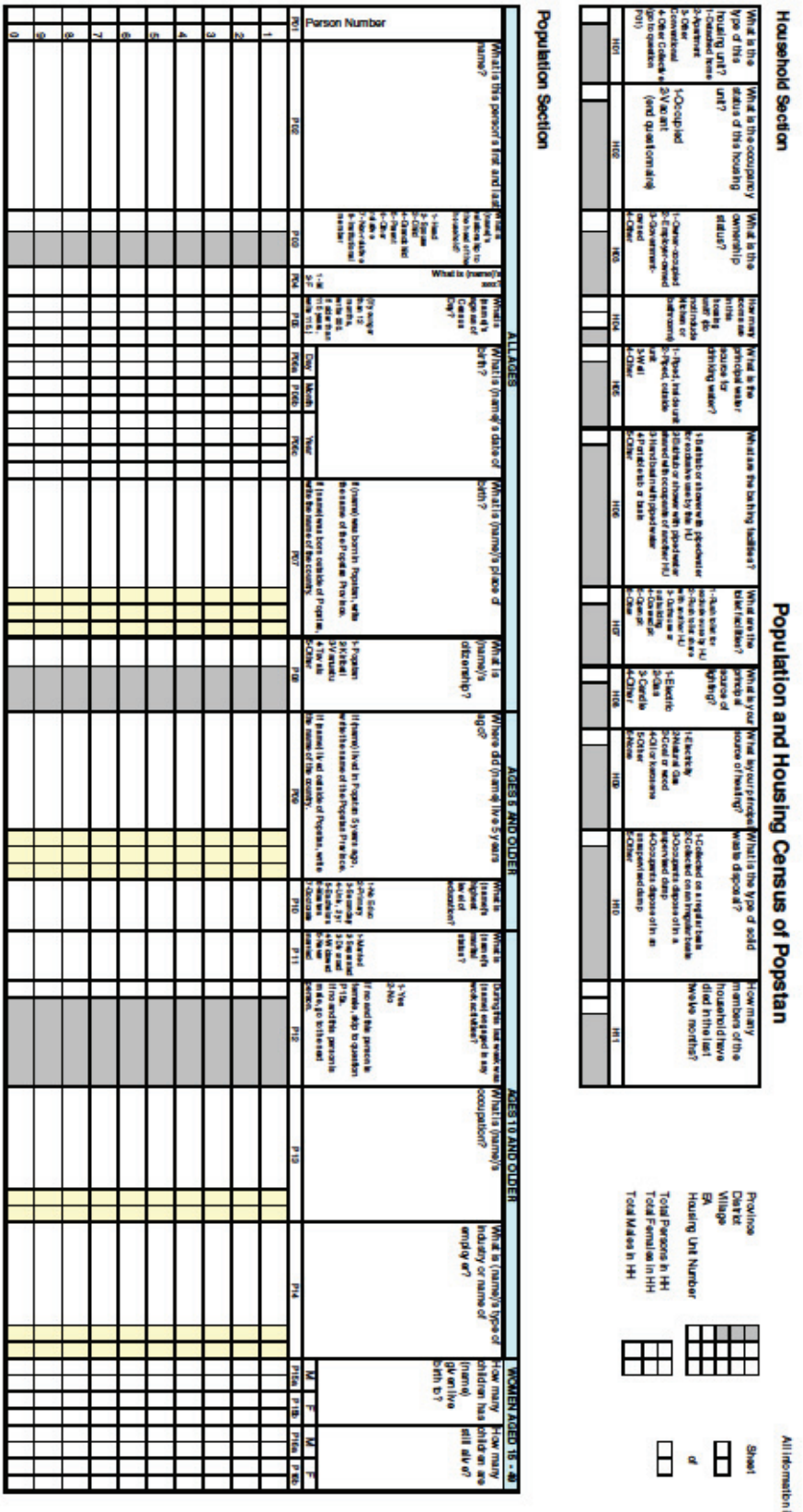
So the above expression means that the census.ent application is to be found one level up, then "down" in the Popstan Census folder.

Appendix A: Programming standards

(Found at the CSProUsers.org website)

1. Backup your code, and all components of the application, frequently. Create a system of version control.
2. Compile code frequently while working on it, and compile code with set explicit checked. This will help identify syntax errors early.
3. Use the help system (by pressing F1) and the reference window while writing logic.
4. Use // for line comments and { } for multiline comments. Comment the code while writing it, not afterwards, as it will probably never get done in that case. Remember that it is likely that you will not be the only person who looks at your code.
5. Maintain indentations for code within conditional statements (IFs), loops (DOs/FORs), and functions.
6. Write CSPro statements in lowercase.
7. Write variable names from the dictionary in uppercase.
8. Write globally declared variable names in mixed case, starting with a lowercase word. Make all global variables at least three characters long.
9. Declare all constants in PROC GLOBAL, not in the application-level preproc.
10. Use locally declared variables for loop counters and when only one PROC or function uses the value of the variable. Remember that the value of the local variable will be reset (to 0) each time the PROC is executed.
11. Use functions when an operation is generalizable or if the same operation will be called repeatedly from different PROCs.
12. Use standard prefixes for variable names. For example:
    a. cnt – counts of an item (such as cntPop and cntHouseholds)
    b. hld – variable used to "hold" the value of the previous household (e.g., hldHeadSex)
    c. ptr – pointer to a record (for instance ptrEldest is the record number of the eldest resident)
13. Maintain an error message numbering scheme so that both subject matter specialists and programmers can understand where error messages occur. For example
    a. (1 digit) record type number
    b. (2 digits) question number of the primary variable
    c. (2 digits) error number for this variable
14. Take advantage of free support from the U.S. Census Bureau at: cspro@lists.census.gov

Appendix B: Example questionnaire



B

Returadresse:
Statistisk sentralbyrå
NO-2225 Kongsvinger

Statistisk sentralbyrå

Oslo:
Postboks 8131 Dep
NO-0033 Oslo
Telefon: 21 09 00 00
Telefaks: 21 09 49 73

Kongsvinger:
NO-2225 Kongsvinger
Telefon: 62 88 50 00
Telefaks: 62 88 50 30

E-post: ssb@ssb.no
Internett: www.ssb.no

ISBN 978-82-537-8339-0 (trykt)
ISBN 978-82-537-8340-6 (elektronisk)
ISSN 1891-5906

ISBN 978-82-537-8339-0

9 788253 783390

Statistisk sentralbyrå
Statistics Norway

Statistisk sentralbyrå

19/2012

Tutorial: Development of data entry- and CAPI applications in CSPro

Design: Siri Boquist