Session 1: Review & New Features CSPro in 7.1 and 7.2

At the end of this lesson participants will be able to:

- Use blocks to show multiple fields on a screen on Android
- Use functions in question text to create complex fills
- Use regular expressions in validation rules
- Deploy applications to mobile devices over the internet

Review Exercise

Before we jump into new topics, we will start with a group exercise to review what we learned in last years workshop. We will implement a CAPI application for a simple household sample survey. This survey consists of two questionnaires: a household questionnaire and a listing questionnaire. The paper questionnaires and the specification documents for the survey are included in the workshop materials on the course website. An initial implementation of the household questionnaire in CSPro is also in the course materials, however only section A and a few questions from section B have been implemented. During this exercise we will implement the remaining sections of the household questionnaire as well as the listing questionnaire. Split up into five groups. Each group will be assigned one section to implement in CSPro. This includes creating items in the dictionary, forms, question text, implementing skip patterns and consistency checks.

- Group 1: section B of the household questionnaire (don't do the prefills for first and last name or the validation of the phone number)
- Group 2: section C of the household questionnaire
- Group 3: section D of the household questionnaire
- Group 4: section E of the household questionnaire (don't do the complex question text in E01)
- Group 4: listing questionnaire

Groups working on the household questionnaire should all start with the CSPro application on the course website and your sections to it. For the listing questionnaire, create a new CSPro application.

Once all teams have completed their sections, each team will present their work and we will combine them together to have on application for the household and one for the listing.

In a future lesson we will implement a menu program for our survey.

New features CSPro 7.1 and 7.2

Since the workshop last year, we released the final version of CSPro 7.1 and a beta release of version 7.2. The remainder of this lesson will focus on the new features in these releases that were not covered in the last workshop.

Blocks

One limitation of earlier versions of CSPro is that on mobile it was not possible to show multiple questions on the same screen. Starting with version 7.2 this is possible using blocks. Blocks are a new feature in version 7.2 that provides a new way of grouping related fields together. You create a block in

the forms tree by selecting the fields you want to group together, right clicking and choosing "Add Block".

Let's put the year, month and day of birth into a block so that they will be displayed together on the tablet. You can see that the three fields are displayed on the same screen, stacked one on top of the other.

Blocks can also have question text. Let's remove the question text from the month, year and day fields and replace it with just the block question text "When was %FIRST_NAME% born?". This makes the screen less busy.

Note that on mobile when the blocks are displayed on the same screen, the onfocus and preprocs for fields in a block are only called when the screen containing the block is first displayed. The postprocs are only called when the user hits next to leave the screen containing the block. On mobile, the procs are NOT called when you move from field to field within the block. This means that you can't do dynamic value sets on fields in a block that rely on values of other fields in the block. For example, we cannot use a dynamic value set to create the correct set of days for the month chosen. It also means that all the skips and consistency checks will only be run at the end of the block so it would not make sense to put questions **B04** (has cell phone) and **B05** (phone number) in the same block since **B05** could be skipped based on the value of **B04**. This limitation is only on mobile. On desktop the block fields are shown as regular fields and the procs are called when you enter/leave the fields as if there was no block.

Blocks themselves have all the same PROCs that fields do. The block preproc and onfocus are called before the preproc/onfocus of the first field in the block and the block postproc and killfocus are called after the postproc/killfocus of the last field in the block. In other words, if you have a block with two fields, FIELD1 and FIELD2, the procs will be called in the following order as you move through the block:

Mobile	Desktop
Hit next to move into block BLOCK preproc BLOCK onfocus FIELD1 (preproc, onfocus) Hit next to move out of block FIELD1 (killfocus, postproc) FIELD2 (preproc, onfocus) FIELD2 (killfocus, postproc) BLOCK (killfocus, postproc)	Hit next to move into FIELD1 BLOCK (preproc, onfocus) FIELD1 (preproc, onfocus) Hit next to move from FIELD1 -> FIELD2 FIELD1 (killfocus, postproc) FIELD2 (preproc, onfocus) Hit next to move from FIELD2 -> FIELD3 FIELD2 (killfocus, postproc) BLOCK (killfocus, postproc)

We can put our logic for the fields in the block in the block procs to avoid any differences in behavior between desktop and mobile. In other words, we put the code we would normally put in the onfocus of the fields into the onfocus of the blocks, code from the field preprocs to the block preproc etc…

Just like fields, block names can used as the target for a skip or reenter. On Desktop this will skip or reenter the first field on the block and mobile it will go to the whole block. To maintain the same

behavior on both platforms it is best to use the block name rather than a field name when skipping to or reentering a block. This has the added benefit that it will still work correctly if you reorder the fields in the block.

Blocks can also be used without displaying the fields together on a mobile device. Block procs allow you to put a **skip** or **ask if** in the preproc of the block to skip the entire block without worrying about what field to skip to. This is useful when you want to skip an entire section of the questionnaire but want to be able to reorder the sections later on without having to change what field to skip to.

For example, in section D, if **D02** is "no" we need to skip to the next section which we have implemented by skipping the first field in section E, **CONSUMED_ANY**. However, if we were to reorder the sections of the survey or insert a new section between sections D and E, we would have to remember to change this `skip` to go to the first field in the new next section. If instead, we put fields **D03** and **D04** into a block we can then put an `ask if` in the block preproc. This way CSEntry will automatically figure out which section to skip to.

```
PROC FOOD_INSECURITY_AND_CAUSES_BLOCK
preproc

// Only ask months and causes if had an incident in last 12 months
ask if HAD_INCIDENT_LAST_12_MONTHS = 1;
```

Another limitation of blocks is that they cannot be nested. If we put questions **D03** and **D04** into a block as described above, we cannot then also create a second block inside to show the reasons in **D04** on the same screen.

Group exercise

Put the first and last name fields on the same screen on mobile using a block. Remove the question text on the individual fields and add question text for the block itself. Move the validation checks that first and last name cannot be blank to the block postproc.

Functions in question text

It is currently cumbersome to use both the first and last name in the question text for the questions in sections B and C. It would be better to combine the first and last names in just one place. We could do this by creating a logic variable or another dictionary variable and assigning it a value in logic, however this can be tricky because we need to make sure to keep the variable up to date. Since CSPro 7.1 it is possible to use a function as a fill in question text. We can write a function to combine the FIRST_NAME and LAST_NAME variables into a full name:

```
function string GetFullName()
    GetFullName = strip(FIRST_NAME) + " " + strip(LAST_NAME);
end;
```

We can then use this function in our question text. For example "Is %GetFullName()% male or female?".

Regular expressions

Validating the formatting of alpha variables can be difficult in CSPro logic. For example, validating that a phone number contains only digits, spaces and dashes in the correct place can take 10 of lines of code to do correctly. Many other programming languages use regular expressions to do these kinds of complex validations.

Starting with CSPro 7.2 there is a new function `regexmatch`() that allows you to check if a string matches a regular expression. Regular expressions are a way to provide a pattern and check if an input string matches the pattern. The simplest regular expressions just match the characters in the pattern exactly with the characters in the input string. To check if a string is exactly "hello" then you could do:

```
if regexmatch(INPUT_FIELD, "hello") then
```

Of course, this is not particularly interesting since we could do the same thing with the equals operator:

```
if INPUT_FIELD = "hello" then
```

Where regular expressions become interesting is when you introduce character classes which, when included in the pattern, will match different sets of characters in the input string. The most common classes are:

.	Match any single character
\w	Match a word character (letter or number)
\W	Match a non-word character
\d	Match a digit
\D	Match a non-digit character
\s	Match a whitespace character (space or tab)
\S	Match a non-whitespace character

Using character classes, you could match 3 numbers, followed by a space, followed by 4 numbers with:

```
if regexmatch(INPUT_FIELD, "\d\d\d \d\d\d\d") then
```

This can be made even more simply using qualifiers, which let you control how many repetitions of a character to match:

n+	Matches a sequence of one or more repetitions of n (at least one n)
n*	Matches zero or more consecutive repetitions of n (any number of n)
n{X}	Matches a sequence of exactly X n's
n{X,Y}	Matches a sequence of X to Y n's
n{X,}	Matches a sequence of at least X n's

Using n{X} syntax we could simplify our test to:

```
if regexmatch(INPUT_FIELD, "\d{3} \d{4}") then
```

This will match most input that doesn't match the pattern, however you can trick by adding other characters to the start or end of the input string. For example, "123-4567ajsdiajsdio" will also match. We can fix this by adding "^" at the start of the pattern and "$" at the end of the pattern to force it to match the entire input string. "^" checks for a match at the start of the input string and "$" ensures that the match ends at the end of the input string.

```
if regexmatch(INPUT_FIELD, "^\d{3} \d{4}$") then
```

You can create your own character classes similar to \d and \w using brackets:

[abc]	Match any one of the characters between the brackets
[^abc]	Match any one character that is NOT one of the characters between the brackets
[0-9]	Match any character between the brackets (any digit)
[^0-9]	Match any character NOT between the brackets (any non-digit)

For example, to match an input string containing only spaces, capital letters and apostrophes you could do:

```
if regexmatch(INPUT_FIELD, "^[A-Z\s']*$") then
```

To match against characters like ^,$,+,* and – that have special meaning in regular expressions you need to escape them by placing a slash in front of them. For example, to check if a string contains only the characters . and [and] you would use the regular expression "\.\[\]".

Testing and debugging regular expressions can be time consuming to do in CSPro. We recommend testing your regular expression patterns first in an online regular expression tester like https://regex101.com/. Just make sure to set the mode to "Javascript" since that is the type of regular expression that is supported by CSPro.

Let's use a regular expression to verify that the telephone number in **B05** is correctly formatted. In the US, a correctly formatted telephone number has 3 digits followed by a dash, followed by three digits, another dash and then four digits. For example, 123-456-7890. How would we match this in CSPro?

```
PROC PHONE_NUMBER

// Ensure that phone number is correctly formatted
if not regexmatch(PHONE_NUMBER, "^\d{3}-\d{3}-\d{4}$") then
    errmsg("Invalid telephone number. Please use format XXX-XXX-XXXX.");
    reenter;
endif;
```

Since regular expressions are used in a lot of programming languages, there are many good tutorials online to learn more. You can often find ready-made regular expressions online for common validations that might you want to do.

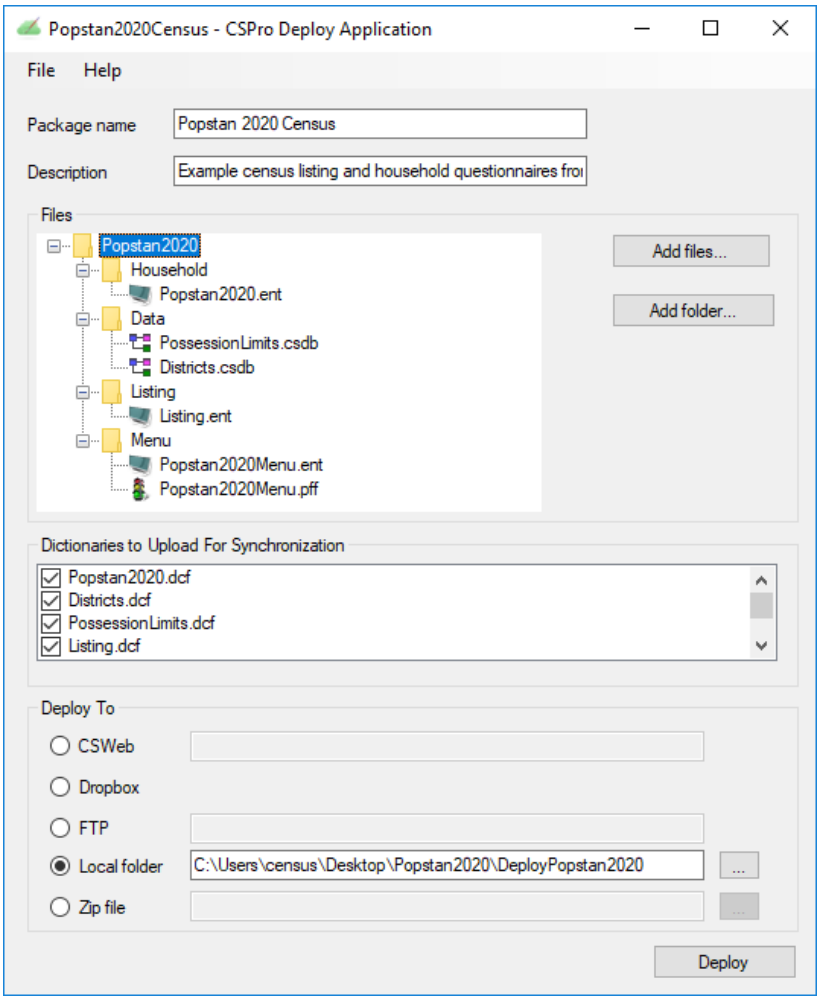
Group exercise

Modify the phone number validation for B05 to support Rwandan cell phone numbers.

Application deployment

Prior to version 7.1, it was difficult to deploy applications to a mobile device, particularly when you had multiple files to publish and deploy like a menu program, a data entry program and lookup files.

CSPro 7.1 provides a new *Deploy Applications* tool that is available from the tools menu. This tool automates the process of publishing and deployment. Drag the whole PopstanSampleSurvey folder to the Files window. The tool will add the entry applications (.ent files) and .pff files found in the folder and its subfolders to the list of files to deploy. By default, it does not add data files since data files are often just used for testing. However, for our application we need to add the lookup file for the consumption items. We can do this by dragging the individual files onto the files window.



Once the files have been added, enter the name and an optional description for the application. Under *Deploy To* choose Local Folder and click on "..." to choose the folder to deploy to. Click the Deploy button. This will automatically create .pen files for each data entry application and copy them, along with any other files from the Files window, into the deployment folder chosen. The deployment folder can then be copied directly to the mobile device. The folder structure of the deployment folder will match the folder structure we are using for development and testing on the desktop so there will be no problems with paths.

Once you have set up the deployment the first time you can save the deployment specification file to simplify the process for future deployments.

In addition to using the tool to deploy our application via USB, we can also use it to deploy it to a server. To deploy to a server simply change the "Deploy To" to the type of server to upload to. You can deploy to a CSWeb server, Dropbox or FTP. The deploy tool will upload all of the files that you have dropped onto the files list to the server. It will also upload some or all of the dictionaries in your application to the server so you don't have to upload them using the CSWeb web page.

Once you have uploaded the application to the server you can download it to a mobile device by choosing "Add Application" from the menu in the applications listing screen. This will ask you for the server details and then display a list of applications available on the server. Once you download an application it is copied to the CSEntry directory of your device.

You can also deploy updated versions of your application to the server and use the same procedure to download the updated application to your device.

Exercises

1. Use the deploy applications tool to deploy your application to a server. If you have a Dropbox account you can use that (or sign up for a free one). You can also use the CSWeb server at http://csweb.teleyah.com but you will want to make sure to give your application a unique description by adding your name to it so that you can tell it apart from applications that other participants upload.
2. Use a regular expression to add a consistency check to the first and last name fields so that they contain only lowercase and capital letters, spaces, dashes (-), and apostrophes (').
3. In the listing questionnaire, group **L09** (male residents) and **L10** (female residents) onto the same screen in mobile. Move the consistency checks to the postproc of the block.
4. Put the quantity and units' questions in **E03** and **E04** into blocks so that they will be displayed on the same screen on mobile.
5. Implement the question text fill for **E01**. Create a function to compute the date 7 days ago and use it as a fill in the question text. You may find the built in functions sysdate(), dateadd() useful.

Session 2: Advanced Data Entry

At the end of this lesson participants will be able to:

- Use the trace function to aid in debugging
- Implement flow control using a navigation field
- Implement flow control using multiple applications
- Collect and analyze paradata

Debugging Techniques

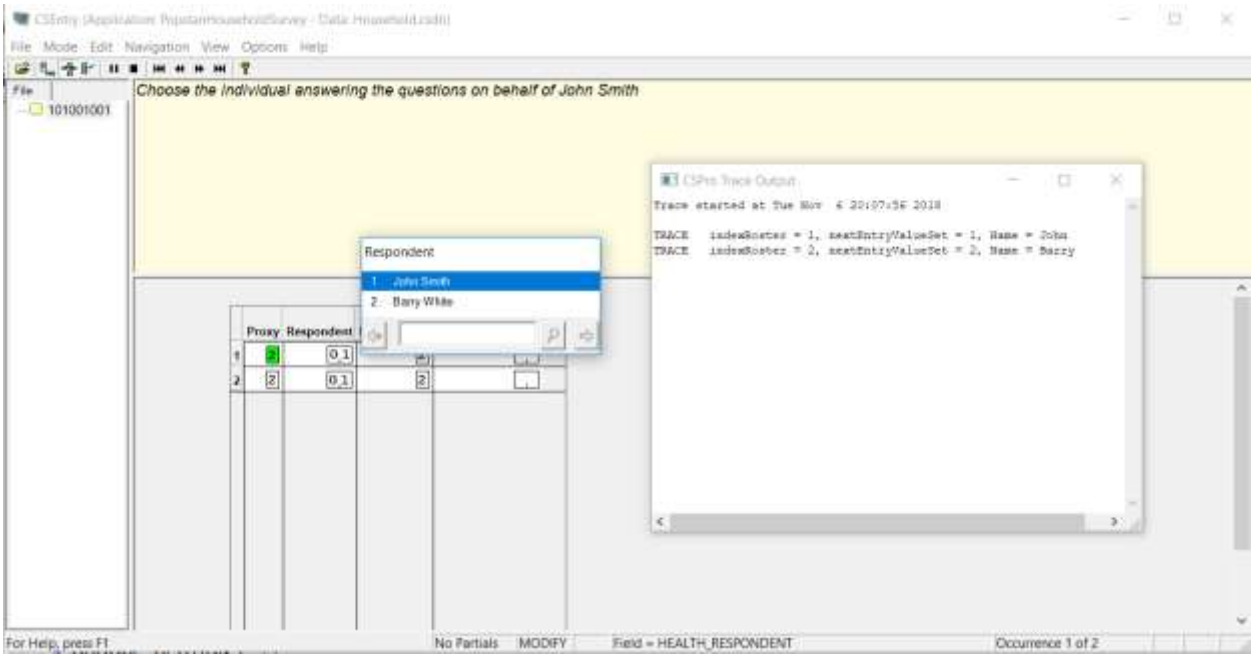
Diagnosing and fixing bugs in a CSPro application can be tricky. It is often useful to be able visualize the values of variables and the flow of the program. You can do this using the **errmsg**() function. For example, if you have code that is building a dynamic value set such as in the health respondent line number (**C02**) we can show a message each time through the loop to display values of the variables we are using:

```
onfocus
// Create the value set for respondent from all household members 10 and over
numeric indexRoster;
numeric nextEntryValueSet = 1;
do indexRoster = 1 while indexRoster <= totocc(HOUSEHOLD_MEMBERS_ROSTER)
    errmsg("indexRoster = %d, nextEntryValueSet = %d, Name = %s",
           indexRoster, nextEntryValueSet, FIRST_NAME(indexRoster));
    if AGE(indexRoster) >= 10 then
        labels(nextEntryValueSet) = strip(FIRST_NAME(indexRoster)) + " " +
                                    strip(LAST_NAME(indexRoster));
        codes(nextEntryValueSet) = indexRoster;
        nextEntryValueSet = nextEntryValueSet + 1;
    endif;
enddo;
codes(nextEntryValueSet) = notappl;
setvalueset($, codes, labels);
```

This gives us useful information, however, we have to click through a lot of **errmsg** dialogs to see the results. Instead we can use the **trace**() function. Instead of showing a dialog box for each message, trace writes all the messages to a single window that stays visible. To show the **trace**() window you call **trace(on)**. To write a message to the trace window you call trace the same way you would call errmsg:

```
onfocus
// Create the value set for respondent from all household members 10 and over
numeric indexRoster;
numeric nextEntryValueSet = 1;
trace(on);
do indexRoster = 1 while indexRoster <= totocc(HOUSEHOLD_MEMBERS_ROSTER)
    trace("indexRoster = %d, nextEntryValueSet = %d, Name = %s",
          indexRoster, nextEntryValueSet, FIRST_NAME(indexRoster));
    if AGE(indexRoster) >= 10 then
        labels(nextEntryValueSet) = strip(FIRST_NAME(indexRoster)) + " " +
                                    strip(LAST_NAME(indexRoster));
        codes(nextEntryValueSet) = indexRoster;
        nextEntryValueSet = nextEntryValueSet + 1;
    endif;
enddo;
codes(nextEntryValueSet) = notappl;
setvalueset($, codes, labels);
```

The messages written to the trace window will look like:



In addition to your own messages, CSPro automatically adds messages to the trace window for every preproc and postproc that is run. This allows you to see the order of procs that are run and where your own trace messages fit in.

Flow Control

In long and complex surveys, you often would like to be able to complete sections in the survey out of order. For example, if there is no one at home who can answer the health section (C) you could instead complete the food security section (D) and later come back to the health section. However, in system-controlled mode, CSPro forces you to complete the questionnaire in a linear fashion. You must complete all prior forms before starting on the next form so you have to complete section C before starting section D. There are, however, a couple of ways to work around this behavior to permit answering the

sections out of the usual order. The first technique uses skips to move around the questionnaire and the second technique implements a single questionnaire as multiple data entry applications.

Flow Control with a Navigation Field

The first way to implement flow control in CSPro is to add a navigation field into the application that gives the user a list of choices of sections to jump to. To do this, we add a new item in the dictionary named NAVIGATION. We can add it to the interview record. The value set will be:

Section B – 1
Section C – 2
Section D – 3
Section E – 4

We will create a new form named "navigation form" in between section A and section B and drop this variable onto it. In the postproc of the field we will use skip to move the section chosen.

```
PROC NAVIGATION

onfocus

// Clear previous choice
$ = notappl;

postproc

// Go to selected section
if $ = 1 then
    skip to SECTION_B_HOUSEHOLD_ROSTER_FORM;
elseif $ = 2 then
    skip to SECTION_C_HEALTH_FORM;
elseif $ = 3 then
    skip to SECTION_D_FOOD_SECURITY_FORM;
elseif $ = 4 then
    skip to SECTION_E_CONSUMPTION_FORM;
endif;
```

To go back to the navigation field after completing a section we can add a userbar button that uses the reenter command to go back to the navigation field.

First, we need a function in the proc global.

```
function GoToNavigation()
    reenter NAVIGATION;
end;
```

Then we need to add this function to the userbar:

```
userbar(clear);
userbar(add button, "Navigation", GoToNavigation);
userbar(show);
```

Normally we would add the function to the userbar in the form file preproc but in this case we only want to show the button when we are already at or past the navigation field, otherwise we might skip over some of the id-items. We can do that by placing this code in the preproc of the navigation form. To make sure that the userbar button doesn't get shown after adding a case and then starting a new case we can add code to clear the userbar in the questionnaire preproc.

There are consistency checks and roster control fields in section C and D that use values from section B. If we skip over section B to section C or D, these consistency checks will fail because the values in section B have been skipped and are considered blank. We could try using **visualvalue**() to access the values but not only does this complicate our code, but if the interviewer tries to complete section C before completing section B, even with visualvalue, the values will still be blank. To avoid this, we can move the navigation form to be after section B. This forces the interviewer to complete section B before using the navigation menu. Now in the NAVIGATION postproc we need to use reenter instead of skip for section B.

```
postproc

// Go to selected section
if $ = 1 then
    reenter SECTION_B_HOUSEHOLD_ROSTER_FORM;
elseif $ = 2 then
    skip to SECTION_C_HEALTH_FORM;
elseif $ = 3 then
    skip to SECTION_D_FOOD_SECURITY_FORM;
elseif $ = 4 then
    skip to SECTION_E_CONSUMPTION_FORM;
endif;
```

This allows us to navigate easily but it has a pretty terrible side effect. If we complete sections B and C, go to the navigation field and then complete sections D and E we see that the data we entered for sections B and C is gone. This is because skipped fields are set to blank when the case is saved.

To solve this, we can use the advance command to go to the end of the questionnaire without skipping. Let's add another option to the value set for NAVIGATION called "Validate and Complete". When the interviewer chooses this option, we use advance to go from the NAVIGATION field to the end of the questionnaire. Advance will run all of the consistency for all of the fields and will stop if it encounters any errors. This will ensure that no sections are left blank and no inconsistencies were added by entering data out of order.

```
PROC NAVIGATION

postproc

// Go to selected section
if $ = 1 then
    reenter SECTION_B_HOUSEHOLD_ROSTER_FORM;
elseif $ = 2 then
    skip to SECTION_C_HEALTH_FORM;
elseif $ = 3 then
    skip to SECTION_D_FOOD_SECURITY_FORM;
elseif $ = 4 then
    skip to SECTION_E_CONSUMPTION_FORM;
elseif $ = 9 then
    // Validate and complete
    advance;
endif;
```

This fixes the problem of skipped fields getting erased, but only if the interviewer makes sure to use the NAVIGATION field to select validate and complete. If instead they just complete the final section after having skipped earlier sections, the skipped sections will still be erased. To prevent this, we will force the interviewer to return to NAVIGATION before completing the questionnaire. We will add a new field named QUESTIONNAIRE_COMPLETE and put it on a new form at the end of the questionnaire. In the preproc for QUESTIONNAIRE_COMPLETE we will reenter NAVIGATION to force validation. In order to allow completion of the questionnaire from the NAVIGATION field, we will add a logic variable named inFinalValidation. We will set it to zero in the preproc of NAVIGATION and set it to 1 before calling advance. In the preproc of QUESTIONNAIRE_COMPLETE we will check the value of inFinalValidation and only return to NAVIGATION if inFinalValidation is false. This way, the only way to get to the end of the questionnaire will be to do it by choosing "validate and complete" from NAVIGATION.

```
PROC QUESTIONNAIRE_COMPLETE
preproc

// Only allow completion of questionnaire if coming
// from NAVIGATION, otherwise return to NAVIGATION
if not inFinalValidation then
    reenter NAVIGATION;
endif;
$ = 1;


PROC NAVIGATION

preproc
// By default not in final validation
inFinalValidation = 0;

onfocus

// Clear previous choice
$ = notappl;

postproc

// Go to selected section
if $ = 1 then
    skip to SECTION_B_HOUSEHOLD_ROSTER_FORM;
elseif $ = 2 then
    skip to SECTION_C_HEALTH_FORM;
elseif $ = 3 then
    skip to SECTION_D_FOOD_SECURITY_FORM;
elseif $ = 4 then
    skip to SECTION_E_CONSUMPTION_FORM;
elseif $ = 9 then
    // Validate and complete
    inFinalValidation = 1;
    advance;
endif;
```

We can make the questionnaire complete field protected and set it to true when the questionnaire has been validated.

We now have a working flow control scheme but we can improve it by showing that status of each section in the navigation field. To do that we add a variable in the dictionary for the completion status of each questionnaire with complete – 1 and incomplete – notappl. At the end of each section we set the status variable for the section to complete. In the NAVIGATION field we can show the status in the question text. To ensure that sections do not stay marked as complete when they are modified, we need to also set the status to incomplete at the beginning of each section. Since each section is on its own form, we can do this in the preproc and postproc of the form for the section.

```
PROC SECTION_D_FOOD_SECURITY_FORM
preproc
// Start section, mark incomplete
SECTION_D_COMPLETE = notappl;
postproc
// End section, mark complete
SECTION_D_COMPLETE = 1;
```

To show the status in the question text we can use a function for the fill:

```
// For displaying section status in question text for
// navigation field
function string getSectionStatus()

    getSectionStatus =
        "Section B: " + getlabel(SECTION_B_COMPLETE_VS1, SECTION_B_COMPLETE)
        + "<br/>" +
        "Section C: " + getlabel(SECTION_C_COMPLETE_VS1, SECTION_C_COMPLETE)
        + "<br/>" +
        "Section D: " + getlabel(SECTION_D_COMPLETE_VS1, SECTION_D_COMPLETE)
        + "<br/>" +
        "Section E: " + getlabel(SECTION_E_COMPLETE_VS1, SECTION_E_COMPLETE);

end;
```

Note that "
" in question text will add a newline.

The question text will be:

> *You must the complete the following sections of the questionnaire:*
>
> %getSectionStatus()%
>
> *Choose a section to go to*

Now every time we enter the navigation field, we see the status of each of the questionnaire sections.

Note that we did not put any of the questionnaire status variables onto a form. If we had put them onto the form for each section we would have to use **visualvalue()** to read their values to build the question text. When a variable is not on a form, you never have to use **visualvalue**() since the variable can't be skipped or be ahead of the current field.

Group Exercise

Whenever the interviewer runs the GoToNavigation() save the name of the current field for the section that they are currently. Then when they restart that section, use advance to move them back to the field they were on when they navigated away from that section. For example, if the user is on the field D02 in section and they tap "Navigate" on the userbar to work on another section and then use navigate again to go back to section D, the program should automatically put them directly into D02 again.

You will need variables in the dictionary to store the last field for each section: one variable for last field in section C, one for the last field in section D, and one for the last field in section E. You can get the name of the current field as a string using the function **getsymbol()** and you can get the name of the record that the variable is on using **getrecord(getsymbol())**. To move to the saved field, use advance to the saved field name in the preproc of the form for the section.

Flow Control with Multiple Applications

An alternative approach to flow control is to implement each section as a separate application and have the main application launch the section applications using **execpff**() the same way we launch the listing and household applications from a menu application. By making each section a separate application, each section has its own flow so no fields ever need to be skipped.

Let's start by moving section C (health) into a separate application. Create a new application named "Section C". Put it in a subfolder of the Household folder also named "Section C". Copy the id-items from the household dictionary but change the names so they won't conflict when both dictionaries are used in the same application e.g. (C_PROVINCE, C_DISTRICT…). Copy the health-related fields from the person record into the household dictionary and paste it into a new section C record in the new dictionary with the same number of occurrences as the person record. Create forms in the section C application containing the id-items and the section C record.

Copy the logic and question text for section C from the household application into the new application. In order for the logic to work correctly we need to access the variables in the household dictionary from the section C application. To do that, add the household dictionary as external dictionary in the section C application. In the postproc of C_HOUSEHOLD_NUMBER use loadcase to load the household roster:

```
PROC C_HOUSEHOLD_NUMBER

// Load the case from the main household file to get access to the household
roster
if not loadcase(HOUSEHOLD_DICT, C_PROVINCE, C_DISTRICT, C_CLUSTER,
                C_HOUSEHOLD_NUMBER) then
    errmsg("Invalid household identifiers");
    stop(1); // Back to menu application
endif;
```

You will need to fix errors and warnings in logic mainly due to the fact that since the health questions and the household roster items are no longer in the same record (or even the same dictionary), we need to use subscripts to correctly access the household roster variables. To make the question text fills work correctly, copy the getFullName() function from the logic of the household application and modify it to use curocc() as a subscript for FIRST_NAME and LAST_NAME.

To correctly size the roster in section C we need to add a new singly occurring record containing a variable for the number of household members, drag the variable onto the form before the roster and use it as the roster control field. Copy the value from the NUMBER_OF_HOUSEHOLD_MEMBERS variable in the household dictionary into this new variable.

The next step is to launch the section C application from the household application. Modify the pff for the section C application using the pff editor to set the start mode to add and set the OnExitPff to relaunch the household questionnaire when the section C program exits. Use the pff editor to generate the CSPro logic to write out the pff. Modify the pff generation code to add the values of the id-items as parameters in the pff file.

```
function LaunchSectionC()

    FILE pffFile;

    setfile(pffFile,"SectionC/SectionC.pff",create);

    filewrite(pffFile,"[Run Information]");
    filewrite(pffFile,"Version=CSPro 7.2");
    filewrite(pffFile,"AppType=Entry");

    filewrite(pffFile,"[DataEntryInit]");
    filewrite(pffFile,"StartMode=Add");
    filewrite(pffFile,"ShowInApplicationListing=Never");

    filewrite(pffFile,"[Files]");
    filewrite(pffFile,"Application=%s","./SectionC.ent");
    filewrite(pffFile,"InputData=%s",
                      "../../Data/HouseholdSectionC.csdb|CSPRODB");

    filewrite(pffFile,"[ExternalFiles]");
    filewrite(pffFile,"HOUSEHOLD_DICT=%s",
                      "../../Data/Household.csdb|CSPRODB");

    filewrite(pffFile,"[Parameters]");
    filewrite(pffFile,"PROVINCE=%v",PROVINCE);
    filewrite(pffFile,"DISTRICT=%v",DISTRICT);
    filewrite(pffFile,"CLUSTER=%v",CLUSTER);
    filewrite(pffFile,"HOUSEHOLD_NUMBER=%v",HOUSEHOLD_NUMBER);
    filewrite(pffFile,"OnExit=%s","../PopstanHouseholdSurvey.pff");

    close(pffFile);

    execpff(filename(pffFile),stop);

end;
```

Modify the postproc of NAVIGATION to launch the section C application:

```
postproc

// Go to selected section
if $ = 1 then
    reenter SECTION_B_HOUSEHOLD_ROSTER_FORM;
elseif $ = 2 then
    savepartial();
    launchSectionC();
elseif $ = 3 then
```

Note that we must call **savepartial**() before launching the section C application so we do not lose any data.

Back in the section C application, modify the preproc of the questionnaire for the section C application to read the parameters from the pff file and assign them to the id items in the section C.

```
PROC SECTIONC_QUEST_FORM

preproc
// Copy id-items passed to use from household application via pff file
C_PROVINCE = tonumber(sysparm("PROVINCE"));
C_DISTRICT = tonumber(sysparm("DISTRICT"));
C_CLUSTER = tonumber(sysparm("CLUSTER"));
C_HOUSEHOLD_NUMBER = tonumber(sysparm("HOUSEHOLD_NUMBER"));
```

Follow the same procedure to create separate data entry applications for sections D and E.

The postproc for NAVIGATION should now look like:

```
postproc

// Go to selected section
if $ = 1 then
    reenter SECTION_B_HOUSEHOLD_ROSTER_FORM;
elseif $ = 2 then
    savepartial();
    launchSectionC();
elseif $ = 3 then
    savepartial();
    launchSectionD();
elseif $ = 4 then
    savepartial();
    launchSectionE();
elseif $ = 9 then
    // do nothing which falls through to end of questionnaire
endif;
```

Note that we no longer need the **advance** to complete the questionnaire. Since NAVIGATION is now the last field in the household questionnaire application, we can simply allow the program to end by doing nothing. For all other selections, one of the section applications will be launched instead.

To update the completion status of the section we have to add the dictionary for the section application as an external dictionary to the household application and use loadcase to check if the case has been completed in that dictionary. This can be done in the onfocus of NAVIGATION since that is where the completion status is displayed.

```
onfocus

// Clear previous choice
$ = notappl;

// Update completion statuses for sections
if loadcase(SECTIONC_DICT, PROVINCE, DISTRICT, CLUSTER, HOUSEHOLD_NUMBER) and
not ispartial(SECTIONC_DICT) then
    SECTION_C_COMPLETE = 1;
else
    SECTION_C_COMPLETE = notappl;
endif;
```

By using multiple applications, we have a control flow that does not require skips that could erase data. The disadvantage of this approach is that our data is now spread out among multiple data files. We will see how we can combine these data files together next week using a batch edit application.

Group Exercise

Create a separate application for section D that is launched from the household data entry application. Make sure that the correct completion status is shown in the question text for the NAVIGATION field.

Paradata

Paradata is data about the data collection process itself. This includes the time taken to answer each question, number of validation errors as well as information about the device itself. This information can be analyzed to find ways to improve your data entry instrument, field procedures and training. Paradata is saved to a paradata log file with the extension .cslog.

Collecting paradata

You can choose which types of paradata to collect in the paradata options dialog in the CSPro designer. You can choose to collect all possible paradata events, no paradata at all or you can select which individual events to collect. Many paradata events occur very frequently so collecting every event can result in very large paradata log files. In addition, some events may result in confidential information being written to the log such as field values and GPS coordinates. In some surveys you want not want to collect that data for privacy reasons. For our application we will choose to collect all paradata events in order to have as much data as possible to analyze.

Before we can analyze any paradata, we need to collect some by doing a few interviews. Let's have everyone take a phone or tablet, load the latest version of the application on it and interview three made up households. So that we get interesting GPS data, please conduct each interview in a different part of the hotel grounds.

Concatenating paradata

After collecting the data, we can combine the paradata log files from each tablet together using the Paradata Concatenator tool. We copy the paradata log from each tablet via usb to a laptop and then add each one the paradata file to the concatenator and run the concatenation. The result is single paradata log containing the data from all the tablets.

Viewing paradata

Double clicking on a paradata log file opens it in the Paradata Viewer. The viewer contains a number of built-in reports as well the ability to create your own customized reports. To run a built-in report simply click on the name of the report in the list of reports on the left-hand side of the screen. The result is displayed on the right. Some reports can be shown as charts in addition to as tables.

The first report, "Field Entry Information", shows the number of times each field in the questionnaire was entered as well as the average amount of time spent on each field. This can tell you which questions take the longest to answer as well as which fields interviewers make the most mistakes entering and are forced to reenter.

Too look at interview times see the "Sessions by Case ID" report which shows the total time in minutes spent on each case. You can use this data to figure out the average interview time.

Another interesting report is the "Displayed Message Frequency" which shows you how frequently each of the error messages in the application is shown. It indicates how often each consistency check is triggered.

Any report can be filtered using the filters on the left-hand side of the screen. You can filter by a date range, by user, by device, by application name etc… Selecting any of these filters limits the data to generate the report to the data that meets the filter criterion.

Custom queries

In addition to the built-in reports, you can view the raw data itself. From the view menu, choose "Table Browser". The paradata is stored in a relational database and this lets you see the tables in that database. If you click on the device_info table, for example you can see the raw data for information for all the devices used. Many of these tables are not useful by themselves. The tables are normalized, meaning that they need to be joined with other tables to get usable data. For example, the field_movement_event table has a column field_movement_instance, which refers to a row in the field_movement_instance table. In addition, all of the event tables are linked by id to the base event table which contains data common to all the different types of events. To include all the data from the linked tables, choose "Show Fully Linked Tables" from the "Options" menu. You can also see detailed information about each table by choosing "View"→"Table metadata". In this mode you can see information each of the columns in each table as well as the linkages between tables. At the bottom of the screen you can see the SQL queries used to select from and join the different tables.

To execute your own SQL queries, you can choose "Query Constructor" from the "View" menu. You can then write and execute SQL queries. For example, if we want to see how many devices of each brand were used, we can execute the following SQL:

```
SELECT device_brand, count(*) as num_devices
from device_info
group by device_brand
```

You can add your own queries into the list of reports by editing the file C:\Program Files (x86)\CSPro 7.2\Reports\CSPro Reports.csrs. This file contains the SQL and other information for all of the built-in reports. You will need administrative privileges on your computer to modify this file.

Plugins

Since we have collected GPS data in our paradata log we can also view the GPS points on a map. To do that, choose "Location Mapper" from the "View" menu. This will prompt you to download the location mapper plugin. Currently this is the only plugin for data viewer but if you know the C# programming language and how to use the SQLite database it is possible to write your own plugins and add them to the Paradata Viewer.

Custom events

Finally, if the built-in paradata events are not sufficient, you can log your own events to the paradata file using the **logtext** command. Logtext works like errmsg or trace except that message is written to the paradata file. For example, if we want to see how often interviewers navigate to different sections of the questionnaire using the navigation field, we can add calls to **logtext** in the postproc of NAVIGATION:

```
postproc

// Go to selected section
if $ = 1 then
    logtext("Navigate to section B");
    reenter SECTION_B_HOUSEHOLD_ROSTER_FORM;
elseif $ = 2 then
    logtext("Navigate to section C");
    skip to SECTION_C_HEALTH_FORM;
elseif $ = 3 then
    logtext("Navigate to section D");
    skip to SECTION_D_FOOD_SECURITY_FORM;
elseif $ = 4 then
    logtext("Navigate to section E");
    skip to SECTION_E_CONSUMPTION_FORM;
elseif $ = 9 then
    logtext("Validate and complete");
    // Validate and complete
    inFinalValidation = 1;
    advance;
endif;
```

The individual messages we write using **logtext** can be viewed using the Logged Event Frequency report. If we want to see total calls to each message, we can use the following query:

```
SELECT
text.text, count(*) as num
FROM message_event
JOIN text ON message_event.message_text = text.id
WHERE message_event.source = 3
group by message_text
```

Syncing paradata

Currently the only way to synchronize paradata is to use **syncfile**() to send the whole paradata log file each time. Since the paradata log get can quite large, you may want to limit the events collected if you wish to sync the paradata. Another option to use the logic function **sqlquery()** to query only the data you need from the paradata file and copy the results into the data file to have it synced.

Exercises

Use the paradata viewer to answer the following questions:

- How many cases were reopened (opened in modify mode)?
- Which user took the most total time entering data?
- What is the lowest battery level of all devices used?
- Which device had the lowest average battery level?

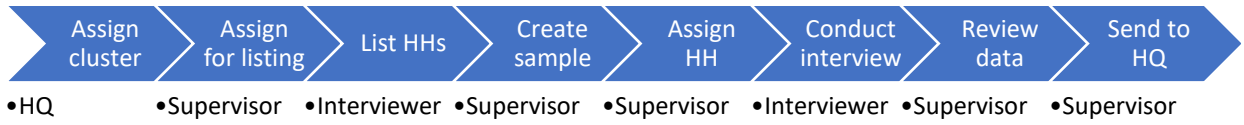
Session 3: Advanced Menu Programs

At the end of this lesson participants will be able to:

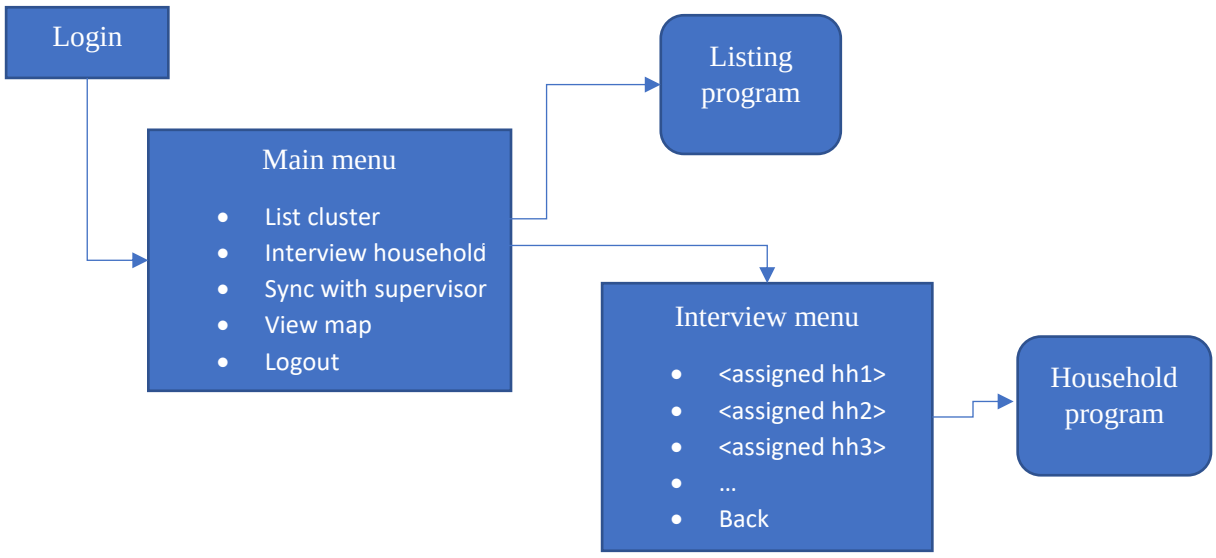
- Implement management of assignments in a menu program
- Create a sample of households in a menu program
- Manage status of clusters in a menu program

Structure of the Menu Program for the Example Survey

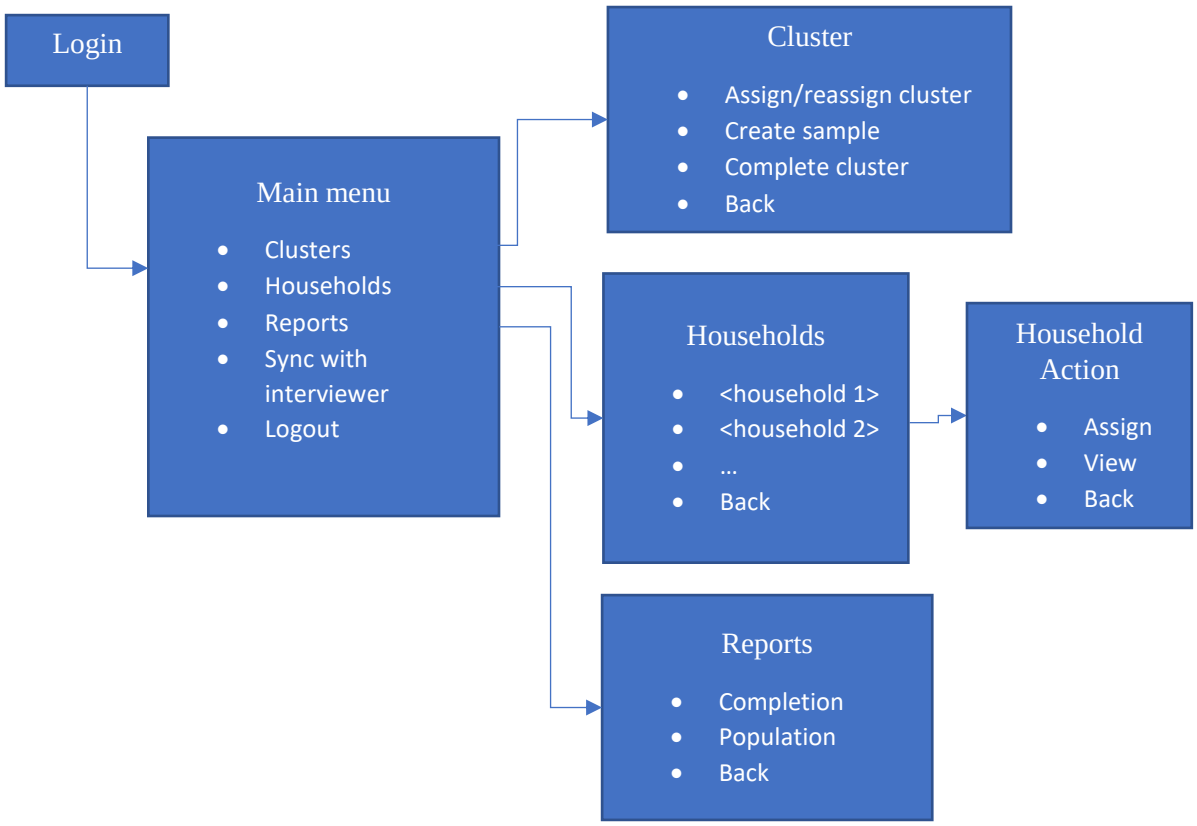
Over the next few days we will implement a system for survey management using CSPro menu programs. This will include not only menu programs on the tablet for the interviewer and supervisor but also a menu program to manage the supervisor assignments that will be run at headquarters. These menu programs will support the following survey workflow:



The interviewer menu program will have the following screens:



The supervisor menu will be similar:



Review of Menu Programs

As you remember, a menu a program is just a data entry program where each menu is a dictionary item with a value that contains the menu options. **Skip** and **reenter** are used to move from one menu to another and **execpff** is used to launch the other data entry programs.

Let's build the menu program together as an exercise. Divide into teams to work on the following tasks:

- Team 1: Login (including creating the staff dictionary and lookup file) and logout (in both supervisor and interviewer main menus). Save login using savesettings so you don't have to login when returning from listing/household programs.
- Team 2: Launch listing program from interviewer main menu
- Team 3: Interview household from interviewer main menu (show listed households and open household questionnaire program)
- Team 4: Implement synchronization between supervisor and interviewer using bluetooth. Synchronize household and listing data files. Update .pen and .pff files on interviewer tablet.

Managing Assignments

The supervisor needs to be able to assign clusters to interviewers for listing and households for interviews. In order to store the assignments in a data file we need to create data dictionaries. These assignment dictionaries will link a cluster or household to an interviewer. Let's start with the listing assignments. This dictionary will have the province, district and cluster as id-items and a single record containing the interviewer staff code. Later we will create the initial list of cluster assignments using the

headquarters management program but for now we can create an initial set of assignments in Excel and use Excel2CSPro to create the dictionary and data file. We will start with two clusters that are both unassigned.

Province	District	Cluster	Assigned To
1	1	1	
1	1	2	

In the onfocus of the assign cluster menu we will create a dynamic value set showing all the clusters and who they are currently assigned to. For each cluster we show, we will also display the name of the interviewer that it is currently assigned to. To do this we need to get the name of the interviewer from the interviewer code by looking it up in the staff data file.

```
PROC ASSIGN_CLUSTER
onfocus

// Build dynamic value set of clusters
numeric nextEntryValueSet = 1;
forcase LISTING_ASSIGNMENTS_DICT do
    codesString(nextEntryValueSet) = key(LISTING_ASSIGNMENTS_DICT);

    // Get current assignment
    string currentAssignment;
    if LA_ASSIGNED_INTERVIEWER = notappl then
        currentAssignment = "unassigned";
    else
        // Load staff name from staff data file
        loadcase(STAFF_DICT, LA_ASSIGNED_INTERVIEWER);
        currentAssignment = strip(STAFF_NAME);
    endif;

    labels(nextEntryValueSet) = maketext("%d-%d-%d (%s)",
                                    LA_PROVINCE, LA_DISTRICT, LA_CLUSTER,
                                    currentAssignment);
    nextEntryValueSet = nextEntryValueSet + 1;
endfor;


// Make sure there is at least one cluster to choose
if nextEntryValueSet = 1 then
    errmsg("No clusters available to assign");
    reenter SUPERVISOR_MAIN_MENU;
endif;

codesString(nextEntryValueSet) = "back";
labels(nextEntryValueSet) = "back";
nextEntryValueSet = nextEntryValueSet + 1;
codesString(nextEntryValueSet) = "";
setvalueset($, codesString, labels);
```

Once they have chosen a cluster to assign, we then need to show another menu asking who to assign it to. We can do that by adding a field called ASSIGN_CLUSTER_TO that shows the available staff to whom the interview can be assigned. This will be a length 4 numeric item to match the size of the staff code. If we add this new field directly after ASSIGN_CLUSTER and don't add a skip or reenter in the postproc, then by default after choosing the cluster we will go into our new field. However, if the user chooses "back" we need to reenter the main menu.

```
postproc
if $ = "back" then
    reenter SUPERVISOR_MAIN_MENU;
endif;
```

In the onfocus of ASSIGN_CLUSTER_TO we create a value set from the staff data file. In order to only show interviewers, we add a where clause to the forcase loop to only include staff with role of supervisor.

```
PROC ASSIGN_CLUSTER_TO
onfocus

// Create value set from staff data file
numeric nextEntryValueSet = 1;
forcase STAFF_DICT where STAFF_ROLE = 1 do
    codes(nextEntryValueSet) = STAFF_CODE;
    labels(nextEntryValueSet) = STAFF_NAME;
    nextEntryValueSet = nextEntryValueSet + 1;
endfor;

// Make sure there is at least one staff member to choose
if nextEntryValueSet = 1 then
    errmsg("No staff available for assignment");
    reenter ASSIGN_CLUSTER;
endif;

codes(nextEntryValueSet) = -1;
labels(nextEntryValueSet) = "back";
nextEntryValueSet = nextEntryValueSet + 1;
codes(nextEntryValueSet) = notappl;

setvalueset($, codes, labels);
```

Finally, in the postproc we update the assignment with the staff code chosen and then return to the previous menu.

```
postproc

if $ <> -1 then
    // Update assignment file
    loadcase(LISTING_ASSIGNMENTS_DICT, ASSIGN_CLUSTER);
    LA_ASSIGNED_INTERVIEWER = ASSIGN_CLUSTER_TO;
    writecase(LISTING_ASSIGNMENTS_DICT);
endif;

// Return to previous menu
reenter ASSIGN_CLUSTER;
```

Now we can modify the interviewer menu so that they are only able to list clusters that are assigned to them.

```
PROC LIST_CLUSTER
onfocus

// Clear previous choice
$ = "";

// Build dynamic value set of clusters
numeric nextEntryValueSet = 1;
forcase LISTING_ASSIGNMENTS_DICT where LA_ASSIGNED_INTERVIEWER = LOGIN do
    codesString(nextEntryValueSet) = itemlist(LA_PROVINCE, LA_DISTRICT,
                                              LA_CLUSTER);

    labels(nextEntryValueSet) = maketext("%d-%d-%d",
                                         LA_PROVINCE, LA_DISTRICT, LA_CLUSTER);
    nextEntryValueSet = nextEntryValueSet + 1;
endfor;


// Make sure there is at least one cluster to choose
if nextEntryValueSet = 1 then
    errmsg("No clusters have been assigned for listing");
    reenter INTERVIEWER_MAIN_MENU;
endif;

codesString(nextEntryValueSet) = "back";
labels(nextEntryValueSet) = "back";
nextEntryValueSet = nextEntryValueSet + 1;
codesString(nextEntryValueSet) = "";
setvalueset($, codesString, labels);


postproc

if $ = "back" then
    reenter INTERVIEWER_MAIN_MENU;
else
    launchListing($[1:1], $[2:2], $[4:3]);
endif;
```

We then modify the function launchListing to accept the province, district and cluster of the selected cluster. It then writes out the .pff file for the listing program and writes the province, district and cluster to the .pff file as system parameters and as the starting case ids for the startMode.

```
function launchListing(string prov, string dist, string clust)

    FILE pffFile;
    setfile(pffFile,"../Listing/Listing.pff",create);

    filewrite(pffFile,"[Run Information]");
    filewrite(pffFile,"Version=CSPro 7.2");
    filewrite(pffFile,"AppType=Entry");

    filewrite(pffFile,"[DataEntryInit]");
    filewrite(pffFile,"StartMode=Add;%s",concat(prov, dist, clust));
    filewrite(pffFile,"Lock=CaseListing");
    filewrite(pffFile,"ShowInApplicationListing=Hidden");

    filewrite(pffFile,"[Files]");
    filewrite(pffFile,"Application=%s","./Listing.ent");
    filewrite(pffFile,"InputData=%s","../Data/Listing.csdb|CSPRODB");

    filewrite(pffFile,"[ExternalFiles]");
    filewrite(pffFile,"ANN_2_DISTRICT_CODES_DICT=%s",

                      "../Lookup/District.csdb|CSPRODB");

    filewrite(pffFile,"[Parameters]");
    filewrite(pffFile,"PROVINCE=%s", prov);
    filewrite(pffFile,"CLUSTER=%s", clust);
    filewrite(pffFile,"DISTRICT=%s", dist);
    filewrite(pffFile,"OnExit=%s","../Menu/Menu.pff");

    close(pffFile);

    execpff(filename(pffFile), stop);
end;
```

Finally, in the listing program we add logic in the preproc of the questionnaire to pre-fill the id items with the parameters passed in through the pff file.

```
PROC LISTING_QUEST_FORM
preproc

if sysparm("PROVINCE") <> "" then
    LI_PROVINVE = tonumber(sysparm("PROVINCE"));
    LI_DISTRICT = tonumber(sysparm("DISTRICT"));
    LI_CLUSTER = tonumber(sysparm("CLUSTER"));

    setproperty(LI_PROVINVE, "Protected", "Yes");
    setproperty(LI_DISTRICT, "Protected", "Yes");
    setproperty(LI_CLUSTER, "Protected", "Yes");
endif;
```

Group Exercise

Implement household assignments in the supervisor menu. This will be similar to the listing assignments. You will need to create a new dictionary for household assignments that contains the identifiers of the household and the staff code of the interviewer that the household is assigned to. Add this dictionary to the menu application. Add a menu to choose the household the household to assign, which shows the households along with the name of the staff currently assigned. Implement a menu to assign a household to an interviewer that shows the list of interviewers and when on is picked updates the household assignments file with the staff code of the interviewer chosen. Unlike with the listing assignments, when showing the available households to assign in addition to displaying households that are already in the assignments file, you will also need to display the households in the listing data file that are not yet in the assignments file. These will be shown as unassigned. If one of these unassigned households is chosen you will add it to the assignments file. You can use **writecase** to do this as long as you first copy the id-items from the listing file entry into the household assignments file id-items.

Creating a Sample

The next step in the menu program is allow the supervisor to automatically take a random sample of the listed households in a cluster. Only the households selected in the sample will be able to be assigned to interviewers.

The first step in creating the sample is to add a new menu for the supervisor to choose which cluster to sample. Let's call it CHOOSE_CLUSTER_TO_SAMPLE. When the supervisor selects "Create sample" from the clusters menu, we will skip to this menu:

```
PROC CLUSTERS_MENU
onfocus
$ = notappl;

postproc

if $ = 1 then
    // Assign/reassign
    skip to CHOOSE_CLUSTER_TO_ASSIGN;
elseif $ = 2 then
    // Create sample
    skip to CHOOSE_CLUSTER_TO_SAMPLE;
elseif $ = 9 then
    //Back
    reenter SUPERVISOR_MAIN_MENU;
endif;
```

In the onfocus of CHOOSE_CLUSTER_TO_SAMPLE we will show a list of clusters using a dynamic value set. We should only show clusters that have been listed. We can tell if listing for a cluster is complete if there is a case for that cluster that exists in the listing data file and if that case is not partially saved.

```
PROC CHOOSE_CLUSTER_TO_SAMPLE
onfocus
$ = "";

// Create dynamic value set from listing
// assignment file
numeric nextEntryValueSet = 1;
forcase LISTINGASSIGNMENTS_DICT do

    string clusterIds = key(LISTINGASSIGNMENTS_DICT);

    // Only include clusters that have been completely listed
    if loadcase(LISTING_DICT, clusterIds)
       and not ispartial(LISTING_DICT) then

        labels(nextEntryValueSet) = maketext("%v-%v-%v",
                                        LA_PROVINCE,
                                        LA_DISTRICT,
                                        LA_CLUSTER);
        codesString(nextEntryValueSet) = clusterIds;

        nextEntryValueSet = nextEntryValueSet + 1;
    endif;
endfor;

if nextEntryValueSet = 1 then
    errmsg("There are no completed clusters to sample");
    reenter CLUSTERS_MENU;
endif;

labels(nextEntryValueSet) = "Back";
codesString(nextEntryValueSet) = "back";
nextEntryValueSet = nextEntryValueSet + 1;

codesString(nextEntryValueSet) = "";
setvalueset($, codesString, labels);
```

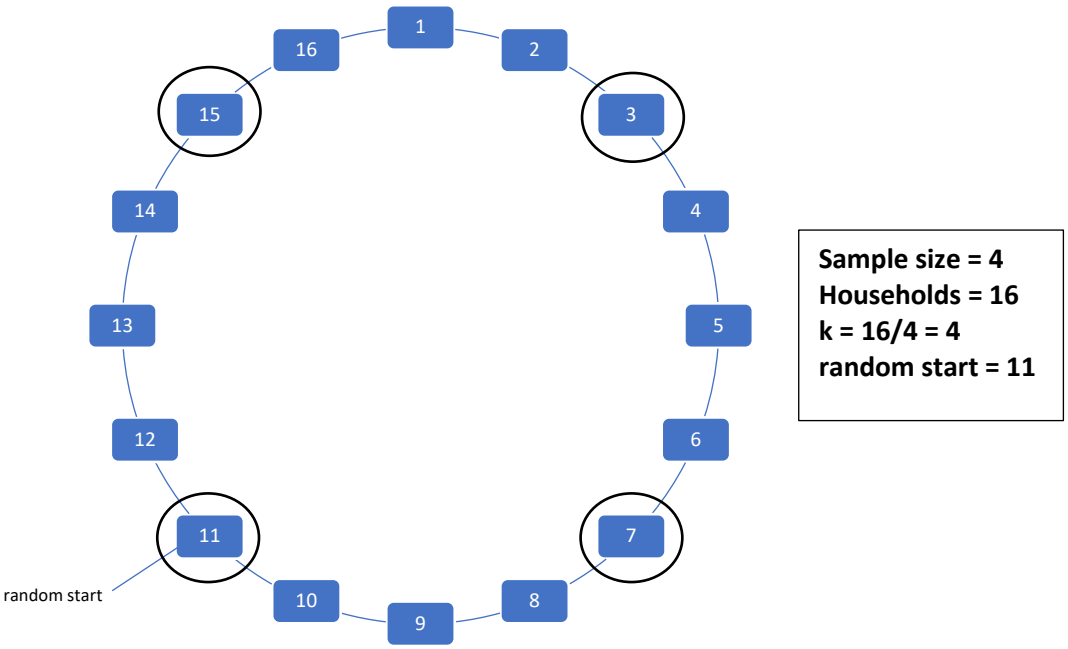
Each value set entry in our value set contains the province, district, and cluster number of the cluster to sample. In the postproc we check if they chose "back" and if so, return to the menu. Otherwise we extract the province, district and cluster from the value the user picked and call a function named CreateSample that we will write.

```
postproc

if $ = "back" then
    reenter CLUSTERS_MENU;
endif;

numeric prov = tonumber($[1:1]);
numeric dist = tonumber($[2:2]);
numeric clust = tonumber($[4:3]);
numeric sampleSize = 5;

if CreateSample(prov, dist, clust, sampleSize) then
    errmsg("Created sample of %d households for cluster %v-%v-%v",
        sampleSize, prov, dist, clust);
endif;

reenter CLUSTERS_MENU;
```

For testing, we chose a sample size of 5. In a real survey this number might be larger. We will write our CreateSample function to be able to use any sample size we choose.

The function CreateSample will create a random sample of the listed households in a cluster. The sampled households can then be assigned to an interviewer. For each of the sampled households we will write a new case to the household assignments file. These cases will initially be unassigned until the supervisor assigns them to interviewers. This way only sampled households will be in the household assignments file.

To create the sample, we will use systematic circular sampling as described in the UN Statistical Division publication *Designing Household Survey Samples: Practical Guidelines*. For this algorithm we first compute a sampling interval equal to the total population divided by the sample size. Then we select a first household for the sample at random from all of the households. We then pick the next household by adding the sampling interval to the first household number and we continue to add the sampling interval to select households for the sample until we enough households for the sample. Specifically, if the first household is number n, and the sampling interval is k we pick households n, n + k, n + 2k, n + 3k… If the next household number to select is greater than the total number of households, we wrap back around to the start of list of households as if the households were aligned along a circle with the first household directly following the last.



The first step in creating the sample is to load the cluster that the supervisor selected from the listing file so that we can read the listed households.

```
function CreateSample(prov, dist, clust, sampleSize)

    // Load the cluster from listing
    LI_PROVINVE = prov;
    LI_DISTRICT = dist;
    LI_CLUSTER = clust;
    loadcase(LISTING_DICT, LI_PROVINVE, LI_DISTRICT, LI_CLUSTER);
```

Next, we will make sure that there are enough eligible listed households in the cluster to create the sample.

```
    // first count the number of eligible households
    numeric numEligibleHouseholds =
                          count(LISTING_REC where LI_PARTICIPATION = 1);

    // make sure there are enough households
    if numEligibleHouseholds < sampleSize then
        errmsg("Not enough eligible households to sample");
        CreateSample = 0;
        exit;
    endif;
```

We only want to sample households from the listing file that are eligible for sampling. These are households where the respondent agreed to take part in the survey in question L07. To simplify the process of selecting the sample from only the eligible households, we will first copy just the eligible households from the listing file into an array. The array will contain the household numbers of the eligible households. This way instead of dealing directly with the household numbers, some of which

represent ineligible households, we can sample the positions in the array. Once we choose an index in the array, we can then get the household number stored at that index.

```
// Copy the eligible household numbers to an array to make
// the sampling easier
array eligibleHouseholds(10000); // Big enough to handle a lot of
                                 // households
numeric nextEligibleIndex = 1;
do numeric i = 1 while i <= count(LISTING_REC)
    if LI_PARTICIPATION(i) = 1 then
        eligibleHouseholds(nextEligibleIndex) = LI_HOUSEHOLD_NUMBER(i);
        nextEligibleIndex = nextEligibleIndex + 1;
    endif;
enddo;
```

Now that we have the eligible households in an array, we use the circular systematic algorithm described above. First, we compute the sampling interval:

```
// compute the sampling interval - space between sampled households
numeric samplingInterval = numEligibleHouseholds / sampleSize;
```

Then we choose a random starting point between 1 and the number of eligible households. To make the sequence of random numbers different each time the program is run we seed the random number generator with the current time.

```
// Pick a random starting point within the list of eligible households
seed(systime());
numeric startingPoint = random(1, numEligibleHouseholds);
```

Then we loop from 1 to the sample size, adding households to the sample. To add a household to the sample we create a new case in the household assignments file for the selected household. Since the array contains the household numbers of the eligible households, we use nextSample as a subscript in the array to get the household number. If the number of eligible households is not evenly divisible by the sample size, the sampling interval will not be a whole number so we need to truncate nextSample before we can use it as a subscript.

```
// These are needed for writecase and will be same for all sampled
// households in the cluster
HA_PROVINCE = LI_PROVINVE;
HA_DISTRICT = LI_DISTRICT;
HA_CLUSTER = LI_CLUSTER;
HA_ASSIGNED_TO = notappl;

numeric nextSample = startingPoint;

do numeric i = 1 while i <= sampleSize

    // Add sampled to household to assignments file
    HA_HOUSEHOLD_NUMBER = eligibleHouseholds(int(nextSample));
    writecase(HOUSEHOLDASSIGNMENTS_DICT);
```

```
        nextSample = nextSample + samplingInterval;

        // If we pass end, wrap around to start again
        if nextSample > numEligibleHouseholds then
            nextSample = nextSample - numEligibleHouseholds + 1;
        endif;
    enddo;

    CreateSample = 1;

end;
```

Now that the sampled households have been written to the household assignments file, we no longer need to show the households from the listing file to the supervisor when they assign households. The CHOOSE_HOUSEHOLD_TO_ASSIGN proc therefore becomes simpler. All it needs to do now is create a dynamic value set from the assignments file.

```
PROC CHOOSE_HOUSEHOLD_TO_ASSIGN
onfocus
$ = "";

// Create value set from list of households in assignments file
numeric nextEntryValueSet = 1;
forcase HOUSEHOLDASSIGNMENTS_DICT do

    // Lookup staff name from staff file
    string assignedInterviewer;
    if loadcase(STAFF_DICT, HA_ASSIGNED_TO) then
        assignedInterviewer = STAFF_NAME;
    else
        assignedInterviewer = "unassigned";
    endif;

    codesString(nextEntryValueSet) = key(HOUSEHOLDASSIGNMENTS_DICT);
    labels(nextEntryValueSet) = maketext("%v-%v-%v-%v : %s",
            HA_PROVINCE, HA_DISTRICT, HA_CLUSTER, HA_HOUSEHOLD_NUMBER,
            assignedInterviewer);
    nextEntryValueSet = nextEntryValueSet + 1;

endfor;

if nextEntryValueSet = 1 then
    errmsg("There are no households to assign. Please create a sample.");
    reenter SUPERVISOR_MAIN_MENU;
endif;

labels(nextEntryValueSet) = "Back";
codesString(nextEntryValueSet) = "back";
nextEntryValueSet = nextEntryValueSet + 1;

codesString(nextEntryValueSet) = "";
setvalueset($, codesString, labels);
```

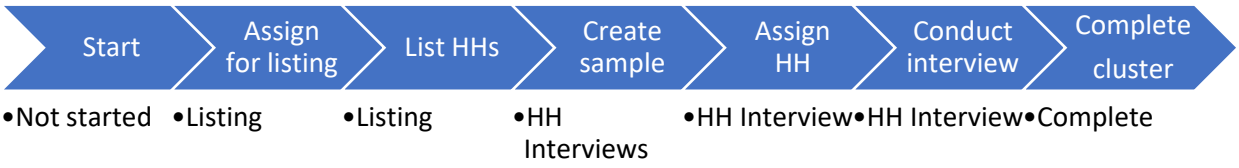
Cluster status

At this point, our menu program has a nearly complete workflow. We can assign clusters, list the cluster, sample the listed households, assign the sampled households and interview the sampled households. However, as we list more clusters, the number of households and clusters shown in our dynamic value sets will continue to grow until the lists become unmanageable. When a cluster is has moved from one step of the workflow to another, we should no longer show it or the households in it in the choices for the prior workflow steps. For example, once a cluster has been sampled, it should no longer be possible to list it. We will accomplish this by adding a variable to keep track of the state of the cluster.

We need a place to save the status of the cluster. We can do this in the listing assignments dictionary. Add a new length 1 item named CLUSTER_STATUS to the only record in that dictionary. This will store what stage of the workflow the cluster is in. We will use the following value set:

Not started	1
Listing	2
Household interviews	3
Complete	4

We will update the status of the cluster as we go through survey workflow. When the supervisor assigns the cluster for listing, creates the sample and marks the cluster as complete we will update the cluster status.



We need to update the listing assignments data file to include this status column. We can do this by adding a status column to the clusters spreadsheet, filling it with "1" (not started) and recreating the listing assignments data file using Excel2CSPro.

Next, we need to update the ASSIGN_CLUSTER_TO postproc to change the status of the cluster from "Not started" to "Listing".

```
postproc

if $ = -1 then
    reenter CHOOSE_CLUSTER_TO_ASSIGN;
else
    // Assign chosen cluster to chosen interviewer

    // Load assignment for cluster
    loadcase(LISTINGASSIGNMENTS_DICT, CHOOSE_CLUSTER_TO_ASSIGN);

    // Update the assignment with interviewer picked
    LA_ASSIGNED_INTERVIEWER = $;
    LA_CLUSTER_STATUS = 2; // Set status to listing
    writecase(LISTINGASSIGNMENTS_DICT);

    reenter CLUSTERS_MENU;
endif;
```

Then we need to update the CHOOSE_CLUSTER_TO_SAMPLE postproc to change the status of the cluster from "Listing" to "Household interviews".

```
if $ = "back" then
    reenter CLUSTERS_MENU;
endif;

numeric prov = tonumber($[1:1]);
numeric dist = tonumber($[2:2]);
numeric clust = tonumber($[4:3]);
numeric sampleSize = 5;

if CreateSample(prov, dist, clust, sampleSize) then

    // Mark cluster as sampled
    loadcase(LISTINGASSIGNMENTS_DICT, $);
    LA_CLUSTER_STATUS = 3;
    writecase(LISTINGASSIGNMENTS_DICT);

    errmsg("Created sample of %d households for cluster %v-%v-%v",
        sampleSize, prov, dist, clust);
endif;

reenter CLUSTERS_MENU;
```

Now we can implement the "complete cluster" option on the clusters menu which will change the status from "Household interviews" to "Completed". The first step will be to choose which cluster to complete so let's add a new menu named CHOOSE_CLUSTER_TO_COMPLETE. This will be a length six alpha item in the dictionary like the other menu items for choosing clusters.

The onfocus of this menu will list all the clusters that are not yet complete but are ready to be completed. These are clusters whose status is "Household interviews". A cluster that has not yet been listed or sampled may not be completed.

```
PROC CHOOSE_CLUSTER_TO_COMPLETE
onfocus
$ = "";

// Dynamic value set of clusters
numeric nextEntryValueSet = 1;
forcase LISTINGASSIGNMENTS_DICT where LA_CLUSTER_STATUS = 3 do

    codesString(nextEntryValueSet) = key(LISTINGASSIGNMENTS_DICT);
    labels(nextEntryValueSet) = maketext("%v-%v-%v",
                                         LA_PROVINCE,
                                         LA_DISTRICT,
                                         LA_CLUSTER);
    nextEntryValueSet = nextEntryValueSet + 1;

endfor;

// Check that there is at least one cluster to choose from
if nextEntryValueSet = 1 then
    errmsg("No clusters are ready to complete. Clusters must be listed,
sampled and household interviews done before being completed.");
    reenter CLUSTERS_MENU;
endif;

codesString(nextEntryValueSet) = "back";
labels(nextEntryValueSet) = "Back";
nextEntryValueSet = nextEntryValueSet + 1;

codesString(nextEntryValueSet) = "";

setvalueset($, codesString, labels);
```

In the postproc, we will prevent the supervisor from completing the cluster if there are still household interviews to be completed. We do this by comparing the number of household interviews in the household data file that are complete to the number of households in the sample from the household assignments file.

```
postproc

if $ = "back" then
    reenter INTERVIEWER_MAIN_MENU;
else

    numeric prov = tonumber($[1:1]);
    numeric dist = tonumber($[2:2]);
    numeric clust = tonumber($[4:3]);

    // Verify that the number of household interviews completed matches the
    // the number of households in the sample

    // Count number of completed cases in household file
    numeric completedCases = 0;
    forcase HOUSEHOLD_DICT where PROVINCE = prov and DISTRICT = dist and
                                 CLUSTER = clust do
        if not ispartial(HOUSEHOLD_DICT) then
            completedCases = completedCases + 1;
        endif;
    enddo;

    // Count sampled cases from assignments file
    numeric sampledCases = countcases(HOUSEHOLDASSIGNMENTS_DICT
        where HA_PROVINCE = prov and HA_DISTRICT = dist
              and HA_CLUSTER = clust);

    if completedCases < sampledCases then
        errmsg("Only %d of the %d sampled households in this cluster have
been interviewed. All interviews must be completed.",
               completedCases, sampledCases);
    else
        // Mark the cluster as complete
        loadcase(LISTINGASSIGNMENTS_DICT, $);
        LA_CLUSTER_STATUS = 4;
        writecase(LISTINGASSIGNMENTS_DICT);
    endif;

    reenter CLUSTERS_MENU;
endif;
```

Now it is possible to take a cluster through the entire workflow. Before we are done, however, we have to use the cluster status to show which clusters and households we show in the different menus where pick clusters and households.

In CHOOSE_CLUSTER_TO_ASSIGN we only want to include clusters that are "not started" or currently assigned ("listing").

```
PROC CHOOSE_CLUSTER_TO_ASSIGN
onfocus
$ = "";

// Create dynamic value set from listing
// assignment file
numeric nextEntryValueSet = 1;
forcase LISTINGASSIGNMENTS_DICT where LA_CLUSTER_STATUS in 1:2 do

    string assignedInterviewerName;

    if LA_ASSIGNED_INTERVIEWER = notappl then
        assignedInterviewerName = "unassigned";
    else
        loadcase(STAFF_DICT, LA_ASSIGNED_INTERVIEWER);
        assignedInterviewerName = strip(STAFF_NAME);
    endif;

    labels(nextEntryValueSet) = maketext("%v-%v-%v : %s",
                                    LA_PROVINCE,
                                    LA_DISTRICT,
                                    LA_CLUSTER,
                                    assignedInterviewerName);
    codesString(nextEntryValueSet) = key(LISTINGASSIGNMENTS_DICT);

    nextEntryValueSet = nextEntryValueSet + 1;
endfor;

labels(nextEntryValueSet) = "Back";
codesString(nextEntryValueSet) = "back";
nextEntryValueSet = nextEntryValueSet + 1;

codesString(nextEntryValueSet) = "";
setvalueset($, codesString, labels);
```

In the interviewers CHOOSE_CLUSTER_TO_LIST menu we only include clusters with status "listing".

```
PROC CHOOSE_CLUSTER_TO_LIST
onfocus
$ = "";

// Dynamic value set of clusters assigned
// to interviewer
numeric nextEntryValueSet = 1;
forcase LISTINGASSIGNMENTS_DICT where LA_ASSIGNED_INTERVIEWER = LOGIN
                                    and LA_CLUSTER_STATUS = 2 do

    codesString(nextEntryValueSet) = key(LISTINGASSIGNMENTS_DICT);
    labels(nextEntryValueSet) = maketext("%v-%v-%v",
                                    LA_PROVINCE,
                                    LA_DISTRICT,
                                    LA_CLUSTER);
    nextEntryValueSet = nextEntryValueSet + 1;

endfor;
```

Similarly, in the supervisor CHOOSE_CLUSTER_SAMPLE menu we will only include clusters with status "listing".

```
PROC CHOOSE_CLUSTER_TO_SAMPLE
onfocus
$ = "";

// Create dynamic value set from listing
// assignment file
numeric nextEntryValueSet = 1;
forcase LISTINGASSIGNMENTS_DICT where LA_CLUSTER_STATUS = 2 do

    string clusterIds = key(LISTINGASSIGNMENTS_DICT);

    // Only include clusters that have been completely listed
    if loadcase(LISTING_DICT, clusterIds)
        and not ispartial(LISTING_DICT) then

        labels(nextEntryValueSet) = maketext("%v-%v-%v",
                                        LA_PROVINCE,
                                        LA_DISTRICT,
                                        LA_CLUSTER);
        codesString(nextEntryValueSet) = clusterIds;

        nextEntryValueSet = nextEntryValueSet + 1;
    endif;
endfor;
```

In the interviewers INTERVIEW_HOUSEHOLD menu we only include households in clusters with status "Household interviews".

```
PROC INTERVIEW_HOUSEHOLD
onfocus

numeric nextEntryValueSet = 1;
forcase HOUSEHOLDASSIGNMENTS_DICT where HA_ASSIGNED_TO = LOGIN do

    // Skip households in completed clusters
    loadcase(LISTINGASSIGNMENTS_DICT, HA_PROVINCE,
                HA_DISTRICT, HA_CLUSTER);
    if LA_CLUSTER_STATUS <> 3 then
        next;
    endif;
```

Similarly, in the supervisors CHOOSE_HOUSEHOLD_TO_ASSIGN proc we include only households with status "Interview households".

```
PROC CHOOSE_HOUSEHOLD_TO_ASSIGN
onfocus
$ = "";

// Create value set from list of households in assignments file
numeric nextEntryValueSet = 1;
forcase HOUSEHOLDASSIGNMENTS_DICT do

    // Skip households in completed clusters
    loadcase(LISTINGASSIGNMENTS_DICT, HA_PROVINCE,
             HA_DISTRICT, HA_CLUSTER);
    if LA_CLUSTER_STATUS <> 3 then
        next;
    endif;
```

Now after a cluster is sampled it can no longer be listed or sampled and after a cluster is complete the households in that cluster can no longer be assigned or interviewed.

Finally, we need to make sure to sync the household assignments and listing assignments files from the supervisor to the interviewer.

```
//connect to the supervisor's device using Bluetooth
function syncToSupervisor()
    if syncconnect(Bluetooth) then
         //getting updates from supervisor's device to interviewer's device
         syncfile(GET,"../Household/PopstanHouseholdSurvey.pff",
                      "../Household/");
         syncfile(GET,"../Household/PopstanHouseholdSurvey.pen",
                      "../Household/");
         syncfile(GET,"../Listing/Listing.pff", "../Listing/");
         syncfile(GET,"../Listing/Listing.pen", "../Listing/");
         syncfile(GET,"Menu.pen");
         syncfile(GET,"Menu.pff");
         syncdata(GET, STAFF_DICT);
         syncdata(GET, LISTINGASSIGNMENTS_DICT);
         syncdata(GET, HOUSEHOLDASSIGNMENTS_DICT);

         //putting data from interviewer's device to supervisor'device
         syncdata(PUT,HOUSEHOLD_DICT);
         syncdata(PUT,LISTING_DICT);

         syncdisconnect();
    endif;
end;
```

Session 4: Advanced Menu Programs Continued

At the end of this lesson participants will be able to:

- Implement automatic synchronization
- Create dashboards
- Create rich reports using HTML templates

Automatic synchronization

We know how to use both simple synchronization and the synchronization logic functions to implement synchronization initiated by the interviewer or supervisor. In some cases, we would prefer to have the synchronization with headquarters happen automatically. We can do this using logic by synchronizing every time the supervisor comes to the main menu. Since the supervisor will very frequently come to the main menu this will force the supervisor to send data often.

To synchronize every time the supervisor reaches the menu, we will write a new function checkForSync that we will call from the onfocus of the supervisor main menu.

```
PROC SUPERVISOR_MAIN_MENU
onfocus
// Clear previous choice
$ = notappl;

// Automatically sync
checkForSync();
```

To avoid annoying the supervisor with overly frequent synchronizations we can save the last time we synced using **savesetting** and then only sync if a certain amount of time has passed since the last sync.

```
function checkForSync()

    // Time to allow between syncs to prevent too many syncs
    numeric minutesBetweenSyncs = 3 * 60; // 3 hours
    numeric lastSyncTime = tonumber(loadsetting("lastSyncTime"));
    numeric minutesSinceLastSync = (timestamp() - lastSyncTime)/60;
    if minutesSinceLastSync > minutesBetweenSyncs

       or lastSyncTime = default then
        if syncToHeadquarters() then
                savesetting("lastSyncTime", maketext("%d", timestamp()));
        endif;
    endif;
end;
```

Here we use the **timestamp()** function which returns the time in seconds. When computing differences between times it is simpler to use **timestamp()** rather than **systime()** since you don't have to worry about the difference between times on different days.

We can be a bit more clever and also not sync if the device is not connected to the internet by using the **connection()** function.

```
function checkForSync()

    // Time to allow between syncs to prevent too many syncs
    numeric minutesBetweenSyncs = 3 * 60; // 3 hours

    numeric lastSyncTime = tonumber(loadsetting("lastSyncTime"));
    numeric minutesSinceLastSync = (timestamp() - lastSyncTime)/60;
    if minutesSinceLastSync > minutesBetweenSyncs
       or lastSyncTime = default then
        // Only sync if there is an internet connection
        if connection() then
            if syncToHeadquarters() then
                savesetting("lastSyncTime", maketext("%d", timestamp()));
            endif;
        endif;
    endif;
end;
```

The syncToHeadquarters() uses the usual logic for synchronizing based on syncconnect(), syncdata() and syncfile().

```
function syncToHeadquarters()
    if syncconnect(csweb, "http://csweb.teleyah.com/api") then

         // Upload listing and household data from supervisor to HQ
         syncdata(PUT,HOUSEHOLD_DICT);
         syncdata(PUT,LISTING_DICT);

         // Download latest staff file
         syncdata(GET, STAFF_DICT);

         // Sync the listing assignments both ways
         syncdata(BOTH, LISTINGASSIGNMENTS_DICT);

         // Get latest programs from HQ
         syncfile(GET,"PHS/Household/PopstanHouseholdSurvey.pff",
                      "../Household/");
         syncfile(GET,"PHS/Household/PopstanHouseholdSurvey.pen",
                      "../Household/");
         syncfile(GET,"PHS/Listing/Listing.pff",
                      "../Listing/");
         syncfile(GET,"PHS/Listing/Listing.pen", "../Listing/");
         syncfile(GET,"PopstanHouseholdSurvey/Menu/Menu.pen");
         syncfile(GET,"PopstanHouseholdSurvey/Menu/Menu.pff");

         syncdisconnect();

         syncToHeadquarters = 1;
    else
        syncToHeadquarters = 0;
    endif;
end;
```

Dashboards

To give the interviewer a snapshot of their work remaining we can create a simple dashboard using the question text of the interviewer main menu. We will show the following text:

> Logged in as interviewer %currentLoginName%
>
> Current work:
>
> Listing: **%clustersCompletedByInterviewer%** completed of **%clustersAssignedToInterviewer%** assigned clusters
>
> Households: **%householdsCompletedByInterviewer%** completed of **%householdsAssignedToInterviewer%** assigned

We will use the following global variables to store the fills:

```
string currentLoginName;
numeric clustersAssignedToInterviewer;
numeric clustersCompletedByInterviewer;
numeric householdsAssignedToInterviewer;
numeric householdsCompletedByInterviewer;
```

We could also use functions, however, counting the number of completed and assigned households involves looping through the entire assignments file which could get slow after a lot of households have been assigned. Instead we can calculate this information once when the interviewer logs in. The only actions that can cause it to change are listing, interviewing and syncing with the supervisor. After listing or interviewing the interviewer will pass through the login screen anyway so we only need to update these variables during login and after synchronization with the supervisor.

```
PROC LOGIN
preproc

// Check for saved login
if loadsetting("login") <> "" then
    LOGIN = tonumber(loadsetting("login"));
    noinput;
endif;

postproc
// verify that staff code exists
// in lookup file
if loadcase(STAFF_DICT, LOGIN) = 0 then
    errmsg("Invalid login. Try again.");
    reenter;
else
    // Loadcase worked so login code is valid
    currentLoginName = STAFF_NAME;

    // Store login in settings so we can use it
    // on next login
    savesetting("login", maketext("%d",LOGIN));

    if STAFF_ROLE = 1 then
        // Interviewer
        updateInterviewerDashboard();
        skip to INTERVIEWER_MAIN_MENU;
    else
        // Supervisor
        skip to SUPERVISOR_MAIN_MENU;
    endif;
endif;
```

The function updateInterviewerDashboard will loop through the listing and household assignments files and count the number of assigned and completed clusters and households.

```
function updateInterviewerDashboard()
    clustersAssignedToInterviewer = 0;
    clustersCompletedByInterviewer = 0;

    forcase LISTINGASSIGNMENTS_DICT where LA_ASSIGNED_INTERVIEWER = LOGIN
                                          and LA_CLUSTER_STATUS = 2 do
        inc(clustersAssignedToInterviewer);
        if isClusterComplete(LA_PROVINCE, LA_DISTRICT, LA_CLUSTER) then
            inc(clustersCompletedByInterviewer);
        endif;
    endfor;

    householdsAssignedToInterviewer = 0;
    householdsCompletedByInterviewer = 0;

    forcase HOUSEHOLDASSIGNMENTS_DICT where HA_ASSIGNED_TO = LOGIN
                  and getClusterStatus(HA_PROVINCE, HA_DISTRICT,
                                       HA_CLUSTER) = 3 do
        inc(householdsAssignedToInterviewer);

        if isHouseholdComplete(HA_PROVINCE, HA_DISTRICT, HA_CLUSTER,
                               HA_HOUSEHOLD_NUMBER) then
            inc(householdsCompletedByInterviewer);
        endif;
    endfor;
end;
```

This function uses the following helper functions to determine the statuses of households and clusters:

```
function isClusterComplete(prov, dist, clust)
    LI_PROVINVE = prov;
    LI_DISTRICT = dist;
    LI_CLUSTER = clust;
    if loadcase(LISTING_DICT, LI_PROVINVE, LI_DISTRICT, LI_CLUSTER)
        and not ispartial(LISTING_DICT) then
        isClusterComplete = 1;
    else
        isClusterComplete = 0;
    endif;
end;

function getClusterStatus(prov, dist, clust)
    LI_PROVINVE = prov;
    LI_DISTRICT = dist;
    LI_CLUSTER = clust;
    loadcase(LISTINGASSIGNMENTS_DICT, LI_PROVINVE,
             LI_DISTRICT, LI_CLUSTER);
    getClusterStatus = LA_CLUSTER_STATUS;
end;
```

```
function isHouseholdComplete(prov, dist, clust, hhnum)
    PROVINCE = prov;
    DISTRICT = dist;
    CLUSTER = clust;
    HOUSEHOLD_NUMBER = hhnum;
    if loadcase(HOUSEHOLD_DICT, PROVINCE, DISTRICT, CLUSTER,
                HOUSEHOLD_NUMBER) and
        not ispartial(HOUSEHOLD_DICT) then
        isHouseholdComplete = 1;
    else
        isHouseholdComplete = 0;
    endif;
end;
```

Group Exercise

Create a dashboard for the supervisor. Use the question text for the supervisor main menu to show the following:

> Logged in as supervisor %currentLoginName%
>
> Current work:
>
> Listing: **%clustersCompletedByAllInterviewers%** completed of **%clustersAssignedToAllInterviewers%** assigned clusters
>
> Households: **%householdsCompletedByAllInterviewers%** completed of **%householdsAssignedToAllInterviewers%** assigned

Create the appropriate global variables for the fills in the question text and a function updateSupervisorDashboard(). Call your function when the supervisor logs in, after syncing with an interviewer and after modifying cluster or household assignments. The function should calculate the assigned and interviewed clusters and households the same way that the updateInterviewerDashboard does except that it should count households and clusters assigned to **all** interviewers rather than the only to the one logged in.

Reports

CSPro supports generating rich reports using HTML templates that can be run on mobile or on desktop. As an example, lets create the progress report.

To launch the report from CSPro we need to implement the reports menu from the supervisor menu. Add a new item to the menu dictionary named REPORTS with value set Completion – 1, Household Status – 2, Back – 9. Add the item to the form and skip to it from the supervisor main menu when the supervisor picks reports. In the postproc for REPORTS we will call a function that we will create named progressReport().

```
PROC REPORTS
onfocus
$ = notappl;

postproc

if $ = 1 then
    progressReport();
    reenter;
elseif $ = 9 then
    reenter SUPERVISOR_MAIN_MENU;
endif;
```

The progress report will show the number of households assigned, number complete, number remaining and number partially complete. We will display it in a table:

Assigned	Complete	Partial	Remaining

The HTML for the table would look like:

```
<table>
    <thead>
        <th>Assigned</th>
        <th>Complete</th>
        <th>Partial</th>
        <th>Remaining</th>
    </thead>
    <tbody>
      <tr>
        <td></td>
        <td></td>
        <td></td>
        <td></td>
      </tr>
    </tbody>
</table>
```

We need to calculate the numbers of assigned, complete, partial and remaining cases and insert those values into the table data (<td>) nodes. To support inserting values into an HTML document, CSPro uses templates. A template is an HTML document that contains variables in it that can be replaced by values from CSPro logic. CSPro uses a templating system called Mustache, which is used by many websites and is well documented online. Any variable to be replaced by CSPro logic must be surrounded by two curly brackets e.g. **{{**variable**}}**. For our table we would use:

```
<table>
    <thead>
        <th>Assigned</th>
        <th>Complete</th>
        <th>Partial</th>
        <th>Remaining</th>
    </thead>
    <tbody>
      <tr>
        <td>{{assigned}}</td>
        <td>{{complete}}</td>
        <td>{{partial}}</td>
        <td>{{remaining}}</td>
      </tr>
    </tbody>
</table>
```

In addition, the entire template must be wrapped in a script tag of type "application/vnd.cspro.report-template" with id "cspro_report_template". This script tag should be part of an HTML document that contains an empty div node with id "cspro_report". When the report is displayed, this div node will be replaced with the content of the template script tag that has all the variables replaced with values from CSPro logic. Using this structure, our HTML file would look like:

```
<!doctype html>
<html lang="en">
  <head>
    <title>Progress Report</title>
</head>
<body>
    <h1>Progress report</h1>

    <div id="cspro_report"><!-- replaced by expanded template --></div>

    <!-- report template -->
    <script type="application/vnd.cspro.report-template" id="cspro_report_template">
        <table>
        <thead>
            <th>Assigned</th>
            <th>Complete</th>
            <th>Partial</th>
            <th>Remaining</th>
        </thead>
        <tbody>
            <tr>
                <td>{{assigned}}</td>
```

```
                <td>{{complete}}</td>
                <td>{{partial}}</td>
                <td>{{remaining}}</td>
            </tr>
        </tbody>
    </table>
    </script>
</body>
</html>
```

To make the page look nicer you can add css and Javascript to the page just as you would for any web page. For example, if we add Bootstrap css and Javascript our page it would look like:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <link rel="stylesheet" href="css/bootstrap.min.css">
    <title>Progress Report</title>
</head>
<body>
    <h1>Progress report</h1>
    <div id="cspro_report"><!-- replaced by expanded template --></div>
    <!-- report template -->
    <script type="application/vnd.cspro.report-template" id="cspro_report_template">
        <table class="table table-bordered">
        <thead>
            <th>Assigned</th>
            <th>Complete</th>
            <th>Partial</th>
            <th>Remaining</th>
        </thead>
        <tbody>
            <tr>
                <td>{{assigned}}</td>
                <td>{{complete}}</td>
                <td>{{partial}}</td>
                <td>{{remaining}}</td>
            </tr>
        </tbody>
    </table>
    </script>
    <script src="js/jquery-3.3.1.slim.min.js"></script>
    <script src="js/popper.min.js"></script>
```

```
    <script src="js/bootstrap.min.js"></script>
</body>
</html>
```

Note that if you want your report to work offline, you will need to include css and Javascript that is stored on the device that the report is running on. If you have no internet you will not be able to use css and Javascript hosted on a CDN or other server. We will download the css and Javascript files we need and put them in subfolders of the folder containing our reports named css and js.

Now that we have our HTML file, we can implement the function progressReport to show the report. The first step is to calculate the values for the table cells by looping through the household assignments file.

```
function progressReport()

    // Compute data for report
    numeric assignedHouseholds = 0;
    numeric completeHouseholds = 0;
    numeric partialHouseholds = 0;

    forcase HOUSEHOLDASSIGNMENTS_DICT where getClusterStatus(HA_PROVINCE,
                                            HA_DISTRICT, HA_CLUSTER) = 3 do

        // Count every household as assigned
        inc(assignedHouseholds);

        if loadcase(HOUSEHOLD_DICT, PROVINCE, DISTRICT, CLUSTER,
                    HOUSEHOLD_NUMBER) then
            // Households with case in household data file
            // are either partial or complete
            if ispartial(HOUSEHOLD_DICT) then
                inc(partialHouseholds);
            else
                inc(completeHouseholds);
            endif;
        endif;
    endfor;

    // Remaining households are assigned households that are not complete
    numeric remainingHouseholds = assignedHouseholds - completeHouseholds;
```

The next step is to assign these calculated values to corresponding parameters in the report template. This is done using the command **setreportdata()**. This function takes the name of the parameter, i.e. the variable in the HTML file surrounded by {{}}, and the value to replace that variable with. The value must be a string so we use **maketext**() to convert our numeric variables to strings.

```
    // Set data to be replaced in report
    setreportdata("assigned", maketext("%d", assignedHouseholds));
    setreportdata("complete", maketext("%d", completeHouseholds));
    setreportdata("partial", maketext("%d", partialHouseholds));
    setreportdata("remaining", maketext("%d", remainingHouseholds));
```

Finally, we run and display the report with the command **report**. We pass it the path to the HTML file. If we place our HTML file in the *Reports* subfolder of the menu folder then we would call:

```
    // create and view the report
    report("Reports/progress.html");

end;
```

This will display the HTML page with the parameters in the template replaced by the values passed to **setreportdata()**.

Group Exercise
Modify the table template and the logic in the progressReport function to show the percent complete. The percent complete column should be on the far right of the table and should be computed as the percentage of complete cases to assigned cases.

Let's create a better version of the report that breaks out the data for each interviewer.

Progress report for supervisor

Interviewer	Assigned	Complete	Partial	Remaining
Andrew Benninger	0	0	0	0
Angelica Swenson	2	0	0	2
Zelma Hawke	2	0	0	2
Willis Catron	1	0	0	1
Total	**0**	**0**	**0**	**0**

We could implement this report with 25 parameters in the template – one for each cell in the 5 x 5 table. However, this will not work if the number of interviewers was to change. We need a way to allow the number of rows to be variable depending on the number of interviewers in the staff file. The **setreportdata** command can be passed a repeating record in the dictionary and generate a row in the table for each occurrence of the record. Unfortunately, we don't have a repeating record in our dictionary for the interviewers. The staff dictionary has a single record for each staff member so we can't just pass the staff record to **setreportdata**. However, we can create a new multiply occurring record, copy the data from the staff dictionary to the repeating record and pass the repeating record to **setreportdata**.

We could create a new record in the menu program for this, however it is common to use a working storage dictionary for this purpose. A working storage dictionary is a dictionary that is not associated with any file. It contains temporary data that only exists while the application is running. To add a working storage dictionary to the application, choose "Add files…" from the files menu and check the box next to "Working storage dictionary" at the bottom of the dialog box.

In the working storage dictionary create a new record named WS_PROGRESS_REC with up to 10 occurrences. Add items in this record for each column of the report: WS_PROGRESS_INTERVIEWER (alpha length 25), WS_PROGRESS_ASSIGNED (numeric length 4), WS_PROGRESS_COMPLETE (numeric length 4), WS_PROGRESS_PARTIAL (numeric length 4) and WS_PROGRESS_REMAINING (numeric length 4).

In the progessReport() function we will fill in one occurrence of WS_PROGRESS_REC for each interviewer. Before we can do that, we need to set the correct number of record occurrences. By default, a case in an external or working storage dictionary is created with all of its occurrences i.e. the number of max occurrences specified in the dictionary. Using the max occurrences would cause the table we generate to have more rows than there are interviewers. To prevent this, we first delete all of the default occurrences of the record using the **delete** command and then we add back just the number of occurrences we need using the **insert** command.

```
function progressReport()

    // Compute data for report

    clear(WS_DICT);

    // By default, a working storage dictionary has all occurrences but
    // that will create too big a table in report so we delete all
    // occurrences and add back the ones we need
    do varying numeric i = count(WS_PROGRESS_REC) until i <= 0 by (-1)
        delete (WS_PROGRESS_REC(i));
    enddo;

    numeric i = 1;
    forcase STAFF_DICT where STAFF_ROLE = 1 do
        insert(WS_PROGRESS_REC(i));
```

Instead of calculating the number of assigned, completed, partial and remaining households once, we need to calculate it for each of the interviewers. First, we loop through the staff data file to get each interviewer. Inside that loop we need to compute the number of assigned, completed, partial remaining cases for just that interviewer. The natural way to do this would be to use a **forcase** inside another **forcase**, however it is not possible to have nested **forcase** loops so we need to make one of the loops use the old locate/while loadcase technique.

```
    numeric i = 1;
    forcase STAFF_DICT where STAFF_ROLE = 1 do
        insert(WS_PROGRESS_REC(i));
        WS_PROGRESS_INTERVIEWER(i) = STAFF_NAME;
        WS_PROGRESS_ASSIGNED(i) = 0;
        WS_PROGRESS_COMPLETED(i) = 0;
        WS_PROGRESS_PARTIAL(i) = 0;

        locate(HOUSEHOLDASSIGNMENTS_DICT, >=, "");
        while loadcase(HOUSEHOLDASSIGNMENTS_DICT) do
            if getClusterStatus(HA_PROVINCE, HA_DISTRICT, HA_CLUSTER) = 3
                and HA_ASSIGNED_TO = STAFF_CODE then

                // Count every household as assigned
                inc(WS_PROGRESS_ASSIGNED(i));

                if loadcase(HOUSEHOLD_DICT, PROVINCE, DISTRICT, CLUSTER,
                            HOUSEHOLD_NUMBER) then
                    // Households with case in household data file are either
                    // partial or complete
                    if ispartial(HOUSEHOLD_DICT) then
                        inc(WS_PROGRESS_PARTIAL(i));
                    else
                        inc(WS_PROGRESS_COMPLETED(i));
                    endif;
                endif;
            endif;
        enddo;

        // Remaining households are assigned households that are not complete
        WS_PROGRESS_REMAINING(i) =
                          WS_PROGRESS_ASSIGNED(i) - WS_PROGRESS_COMPLETED(i);
        inc(i);
    endfor;
```

Now we can pass the WS_PROGRESS_REC directly to **setreportdata**. We will also pass the totals so that we can fill in the total row of the table.

```
    // Set data to be replaced in report
    setreportdata(clear);
    setreportdata(WS_PROGRESS_REC);

    numeric totalAssignedHouseholds = sum(WS_PROGRESS_ASSIGNED);
    numeric totalCompleteHouseholds = sum(WS_PROGRESS_COMPLETED);
    numeric totalPartialHouseholds = sum(WS_PROGRESS_PARTIAL);
    numeric totalRemainingHousehold = sum(WS_PROGRESS_REMAINING);
    setreportdata("assigned", maketext("%d", totalAssignedHouseholds));
    setreportdata("complete", maketext("%d", totalCompleteHouseholds));
    setreportdata("partial", maketext("%d", totalPartialHouseholds));
    setreportdata("remaining", maketext("%d", totalRemainingHousehold));

    // create and view the report
    report("Reports/progress.html");

end;
```

To support the progress record in our template we will use a special syntax for repeating a block of HTML. If we put the name of a repeating record in {{}} preceded by a # sign, that tells the system to repeat the HTML that follows for each occurrence of the record. To mark the end of the repeated HTML use the record name preceded by / in {{}}. In our report we can make a row of the table repeat once for each occurrence of WS_PROGRESS_REC with the following:

```
{{#WS_PROGRESS_REC}}
<tr>
    <td>{{WS_PROGRESS_INTERVIEWER}}</td>
    <td>{{WS_PROGRESS_ASSIGNED}}</td>
    <td>{{WS_PROGRESS_COMPLETED}}</td>
    <td>{{WS_PROGRESS_PARTIAL}}</td>
    <td>{{WS_PROGRESS_REMAINING}}</td>
</tr>
{{/WS_PROGRESS_REC}}
```

Using this syntax, any names of items in the repeating record in {{}} will be replaced with the value of that item for each occurrence of the record.

After adding the repeating row and the total row at the bottom of the table our table template will be:

```
<table class="table table-bordered">
        <thead>
            <tr>
                <th>Interviewer</th>
                <th>Assigned</th>
                <th>Complete</th>
                <th>Partial</th>
                <th>Remaining</th>
            </tr>
        </thead>
        <tbody>
            {{#WS_PROGRESS_REC}}
            <tr>
                <td>{{WS_PROGRESS_INTERVIEWER}}</td>
                <td>{{WS_PROGRESS_ASSIGNED}}</td>
                <td>{{WS_PROGRESS_COMPLETED}}</td>
                <td>{{WS_PROGRESS_PARTIAL}}</td>
                <td>{{WS_PROGRESS_REMAINING}}</td>
            </tr>
            {{/WS_PROGRESS_REC}}
        </tbody>
        <tfoot>
            <tr>
                <th>Total</th>
                <th>{{assigned}}</th>
```

```
                <th>{{complete}}</th>
                <th>{{partial}}</th>
                <th>{{remaining}}</th>
            </tr>
        </tfoot>
    </table>
```

This will give us both the totals and the data for each interviewer.

Group Exercise

Implement the household status report. It will be a table that shows all the households in all clusters with status "household interviews". For each household, display the household number, name of the interviewer it is assigned to and the status (not started, partial or complete) of the household.

Household number	Assigned to	Status
1	Willis Catron	Not started
2	Zelma Hawke	Partial
3	Zelma Hawke	Complete

Create a new record in the working storage dictionary to hold the rows of the table. Loop through the household assignments file and add a new occurrence to the record for each household in a cluster with status "Interview households". You can use the getClusterStatus() helper function to check the cluster status. Look up the name of the interviewer from the staff file and compute the status based on whether or not there is a corresponding case in the household data file and whether or not that case is partially saved.

Session 5: Headquarters Management Programs

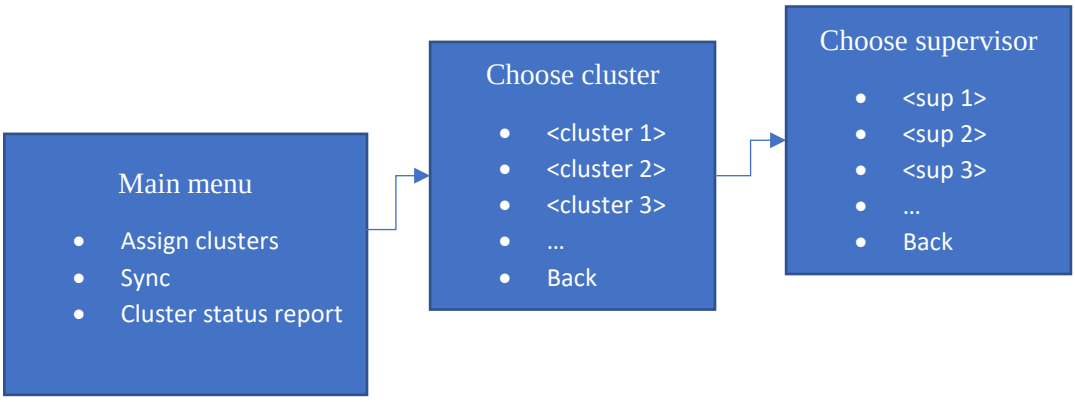
At the end of this lesson participants will be able to:

- Implement a basic survey management system to run at headquarters

Structure of the HQ Menu Program

In addition to the menu programs for the supervisor and the interviewer, we will also implement a menu program for managing the survey from the central office. This menu program will allow central office staff to manage assignments of clusters to supervisors, download data and view reports. It could be run on a desktop computer or a mobile device.

This menu program will have the following screens:



As with our other menu programs, we create a new CAPI application with a new dictionary and add an entry in the dictionary for each menu. In this program we will have three items in the dictionary: *Assign clusters*, *Choose cluster* and *Choose supervisor*. Sync and cluster status reports will be implemented as user functions. After dragging the three items from the dictionary onto the form, we implement the main menu as follows:

```
PROC MAIN_MENU
onfocus
$ = notappl;

if $ = 1 then
    // Assign clusters
    skip to CHOOSE_CLUSTER_TO_ASSIGN;
elseif $ = 2 then
    // Sync
    synchronize();
elseif $ = 3 then
    // Cluster status report
    clusterStatusReport();
endif;

reenter;
```

Assigning Clusters

A major function of our HQ menu program will be to assign clusters to supervisors. In order to implement this, we will add a new variable to the listing assignments dictionary to store the staff code of the assigned supervisor. We will name it LA_ASSIGNED_SUPERVISOR. Initially it will be blank but will be filled in with the supervisor staff code once the cluster is assigned to a supervisor.

The CHOOSE_CLUSTER_TO_ASSIGN onfocus will show a dynamic value set containing the list of clusters that are in the "not started" state. These are the clusters that can be assigned to supervisors.

```
PROC CHOOSE_CLUSTER_TO_ASSIGN
onfocus
$ = "";

// Create value set from all clusters with state "not stared"
numeric nextEntryValueSet = 1;
forcase LISTINGASSIGNMENTS_DICT where LA_CLUSTER_STATUS = 1 do

    // Get name of currently assigned supervisor from staff file
    string assignedSupervisorName;
    if LA_ASSIGNED_SUPERVISOR = notappl then
        assignedSupervisorName = "unassigned";
    else
        loadcase(STAFF_DICT, LA_ASSIGNED_SUPERVISOR);
        assignedSupervisorName = STAFF_NAME;
    endif;

    codesString(nextEntryValueSet) = key(LISTINGASSIGNMENTS_DICT);
    labels(nextEntryValueSet) = maketext("%v-%v-%v : %s",
        LA_PROVINCE, LA_DISTRICT, LA_CLUSTER, assignedSupervisorName);
    nextEntryValueSet = nextEntryValueSet + 1;
enddo;

codesString(nextEntryValueSet) = "back";
labels(nextEntryValueSet) = "Back";
nextEntryValueSet = nextEntryValueSet + 1;
```

```
codesString(nextEntryValueSet) = "";

setvalueset($, codesString, labels);
```

In the postproc we handle the back option or continue to the next field on the form, ASSIGN_CLUSTER_TO.

```
postproc

if $ = "back" then
    reenter MAIN_MENU;
endif;
```

In the onfocus of ASSIGN_CLUSTER_TO we show the list of supervisors that the cluster may be assigned to by looping through the staff file.

```
PROC ASSIGN_CLUSTER_TO
onfocus
$ = notappl;

// Create value set of supervisors
numeric nextEntryValueSet = 1;
forcase STAFF_DICT where STAFF_ROLE = 2 do

    codes(nextEntryValueSet) = STAFF_CODE;
    labels(nextEntryValueSet) = STAFF_NAME;
    nextEntryValueSet = nextEntryValueSet + 1;
enddo;

codes(nextEntryValueSet) = -1;
labels(nextEntryValueSet) = "Back";
nextEntryValueSet = nextEntryValueSet + 1;

codes(nextEntryValueSet) = notappl;

setvalueset($, codes, labels);
```

In the postproc, we set the assigned supervisor to the chosen staff member and write the case.

```
postproc

if $ = -1 then
    reenter CHOOSE_CLUSTER_TO_ASSIGN;
endif;

LA_PROVINCE = tonumber(CHOOSE_CLUSTER_TO_ASSIGN[1:1]);
LA_DISTRICT = tonumber(CHOOSE_CLUSTER_TO_ASSIGN[2:2]);
LA_CLUSTER = tonumber(CHOOSE_CLUSTER_TO_ASSIGN[4:3]);
loadcase(LISTINGASSIGNMENTS_DICT, LA_PROVINCE, LA_DISTRICT, LA_CLUSTER);
LA_ASSIGNED_SUPERVISOR = $;
LA_ASSIGNED_INTERVIEWER = notappl;
LA_CLUSTER_STATUS = 1;
writecase(LISTINGASSIGNMENTS_DICT);
reenter CHOOSE_CLUSTER_TO_ASSIGN;
```

All that is left now is to go back and modify the supervisor menu program so that the supervisor may only assign clusters for listing that are already assigned to her.

```
PROC CHOOSE_CLUSTER_TO_ASSIGN
onfocus
$ = "";

// Create dynamic value set from listing
// assignment file, only includes clusters that are
// not started or currently assigned for listing
numeric nextEntryValueSet = 1;
forcase LISTINGASSIGNMENTS_DICT where LA_CLUSTER_STATUS in 1:2
    and LA_ASSIGNED_SUPERVISOR = LOGIN do
```

Synchronizing

The key data to synchronize is the listing assignments file. It needs to be synced both ways in order to send the new assignments made at the central office to devices in the field as well as to receive the latest cluster statuses from the field. In addition to syncing the listing assignments file we will also download the latest household and listing data files for use in reports.

```
function synchronize()
    if syncconnect(CSWeb, "http://csweb.teleyah.com/api") then

        // Bi-directional sync of listing assignments
        syncdata(BOTH, LISTINGASSIGNMENTS_DICT);

        // Download data files
        syncdata(GET, HOUSEHOLD_DICT);
        syncdata(GET, LISTING_DICT);

    endif;
end;
```

Reports

The central office management menu will also include reports. Just like in the supervisor menu, we will use HTML templates to create reports. The cluster status report will show the progress for each cluster including the status, assigned supervisor, number of listed households and number of interviewed households.

Cluster Status Report

Province	District	Cluster	Status	Supervisor	Listed	Interviewed
1	1	3	Listing	Shemika Rothenberger	1	0
1	1	2	Listing	Shemika Rothenberger	7	0
1	1	1	Complete	Shemika Rothenberger	29	5

We will use a multiply occurring record in the working storage dictionary to store the data for the table. Each occurrence of the record will store data for one row in the table. Add a working storage dictionary to the application and in it add a record named WS_CLUSTER_STATUS_REC with 999 occurrences (enough to hold all the clusters in our survey). Add the following items to this record:

- WS_CLUSTER_STATUS_PROVINCE (numeric, length 1)
- WS_CLUSTER_STATUS_DISTRICT (numeric, length 2)
- WS_CLUSTER_STATUS_CLUSTER (numeric, length 3)
- WS_CLUSTER_STATUS_STATUS (alpha, length 30)
- WS_CLUSTER_STATUS_SUPERVISOR (alpha, length 30)
- WS_CLUSTER_STATUS_LISTED (numeric, length 3)
- WS_CLUSTER_STATUS_INTERVIEWED (numeric, length 3)

Our clusterStatusReport function will fill in the occurrences of this record and run the report. The first step is to clear the extra occurrences from the WS_CLUSTER_STATUS record so that we can add in the correct number.

```
function clusterStatusReport()

    setreportdata(clear);
    clear(WS_DICT);

    // Remove default occurrences of record
    do numeric i = count(WS_CLUSTER_STATUS_REC) until i <= 0 by (-1)
        delete (WS_CLUSTER_STATUS_REC(i));
    enddo;
```

Next, we loop through all the clusters in the listing assignments dictionary and add a new occurrence for each cluster.

```
    numeric i = 1;
    forcase LISTINGASSIGNMENTS_DICT do
        insert(WS_CLUSTER_STATUS_REC(i));
```

Then we copy the cluster ids from the listing assignments dictionary into the working storage record.

```
WS_CLUSTER_STATUS_PROVINCE(i) = LA_PROVINCE;
WS_CLUSTER_STATUS_DISTRICT(i) = LA_DISTRICT;
WS_CLUSTER_STATUS_CLUSTER(i) = LA_CLUSTER;
```

We can use **getlabel**() to get the a status of the cluster as a string instead of a number and add that to the working storage record.

```
WS_CLUSTER_STATUS_STATUS(i) =
                getlabel(LA_CLUSTER_STATUS_VS1, LA_CLUSTER_STATUS);
```

To get the supervisor name we use **loadcase** to look it up in the staff file based on the supervisor code in the listing assignments file.

```
if LA_ASSIGNED_SUPERVISOR <> notappl then
    loadcase(STAFF_DICT, LA_ASSIGNED_SUPERVISOR);
    WS_CLUSTER_STATUS_SUPERVISOR(i) = STAFF_NAME;
else
    WS_CLUSTER_STATUS_SUPERVISOR(i) = "";
endif;
```

The total number of listed households in the cluster is obtained by loading the cluster from the listing file and using **count** to get the number of rows in the roster.

```
if loadcase(LISTING_DICT, LA_PROVINCE, LA_DISTRICT, LA_CLUSTER) then
    WS_CLUSTER_STATUS_LISTED(i) = count(LISTING_REC);
else
    WS_CLUSTER_STATUS_LISTED(i) = 0;
endif;
```

We compute the number of interviewed households by looping through the household data file to get all the completed (not partial) cases in the cluster. Normally we would use **countcases**() to do this but **countcases**() cannot be used inside a **forcase** loop, nor can another **forcase** loop. To work around this, we use **locate** and **loadcase**.

```
WS_CLUSTER_STATUS_INTERVIEWED(i) = 0;
locate(HOUSEHOLD_DICT, >=, "");
while loadcase(HOUSEHOLD_DICT) do
    if PROVINCE = LA_PROVINCE and DISTRICT = LA_DISTRICT and
            CLUSTER = LA_CLUSTER and
            not ispartial(HOUSEHOLD_DICT) then
        inc(WS_CLUSTER_STATUS_INTERVIEWED(i));
    endif;
enddo;
```

Finally we end the loop, attach the working storage to the report using **setreportdata**() and run the report.

```
        inc(i);
    endfor;

    setreportdata(WS_CLUSTER_STATUS_REC);
    report("../Reports/clusterstatus.html");
end;
```

The template for our report is just and HTML table with columns matching the variables in the working storage record.

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <link rel="stylesheet" href="css/bootstrap.min.css">
    <title>Cluster Status Report</title>
</head>
<body>
    <h1>Cluster Status Report</h1>
    <br/>
    <div id="cspro_report"><!-- replaced by expanded template --></div>

    <!-- report template -->
    <script type="application/vnd.cspro.report-template" id="cspro_report_template">
        <table class="table table-bordered">
                <thead>
                    <tr>
                        <th>Province</th>
                        <th>District</th>
                        <th>Cluster</th>
                        <th>Status</th>
                        <th>Supervisor</th>
                        <th>Listed</th>
                        <th>Interviewed</th>
                    </tr>
                </thead>
                <tbody>
                    {{#WS_CLUSTER_STATUS_REC}}
                    <tr>
                        <td>{{WS_CLUSTER_STATUS_PROVINCE}}</td>
                        <td>{{WS_CLUSTER_STATUS_DISTRICT}}</td>
                        <td>{{WS_CLUSTER_STATUS_CLUSTER}}</td>
                        <td>{{WS_CLUSTER_STATUS_STATUS}}</td>
                        <td>{{WS_CLUSTER_STATUS_SUPERVISOR}}</td>
                        <td>{{WS_CLUSTER_STATUS_LISTED}}</td>
```

```
                    <td>{{WS_CLUSTER_STATUS_INTERVIEWED}}</td>
                </tr>
                {{/WS_CLUSTER_STATUS_REC}}
            </tbody>
        </table>
    </script>
    <script src="js/jquery-3.3.1.slim.min.js"></script>
    <script src="js/popper.min.js"></script>
    <script src="js/bootstrap.min.js"></script>
</body>
</html>
```

While our HQ menu program is now fully functional, it is far from complete. We should add additional reports as well functions to manage the staff data file by adding new staff members and assigning interviewers to supervisors. This is left as an exercise for the reader.

Session 6: Batch Edit

At the end of this lesson participants will be able to:

- Use batch edit to find errors in a data file
- Use batch edit to correct errors in a data file
- Use the hot deck technique to impute missing/inconsistent values
- Use batch edit to filter, split and merge data files
- Use batch edit to add new variables to a data file

Overview of Batch Edit

As we saw in the previous workshop, batch edit applications in CSPro allow you to run logic on a data file to produce an output data file and a report. Batch applications are similar to data entry applications in that they use a data dictionary and have PROCs that use most of the same logic functions. Where batch applications differ is that they run only on data that already exists and they run on the entire data file with no user intervention.

Since we have very little existing data for our Popstan Household Survey application, we will use a different dataset for our batch edit exercises. We will use the Popstan census data on the course website.

Checking for errors

One use for batch edit applications is to identify errors and inconsistencies in a data file. Whenever you call the **errmsg()** function in a batch application, rather than showing the message in a pop-up window, the message is written to the listing file.

For example, to check if the marital status variable has a valid value one could add the following logic in the proc for marital status.

```
PROC P05_MS

if not invalueset($) then
    errmsg("Marital status %d not in value set", $);
endif;
```

Here we use in the function **invalueset()** which returns true if the value of the variable falls within the first value set of the variable.

When we run this program the listing file is displayed. From that we can see, that there are 33 individuals in the data file that have a blank marital status.

```
CSPRO Process Summary

          +-------------------------------------------------+
          |   29143 Records Read ( 100% of input file)      |
          |       0 Ignored (       0 unknown,        0 erased) |
          |      33 Messages (      33 U,       0 W,       0 E) |
          +---------+--------------+--------------+-------------+
          |  Level  |  Input Case  |  Bad Struct  |  Level Post |
          +---------+--------------+--------------+-------------+
          |     1   |       4872   |          0   |       4872  |
          +---------+--------------+--------------+-------------+


Process Messages


*** Case [010212500511020261] has 1 messages (0 E / 0 W / 1U)
    U    -9 Marital status NOTAPPL not in value set

*** Case [010608700220730221] has 1 messages (0 E / 0 W / 1U)
    U    -9 Marital status NOTAPPL not in value set

*** Case [020411800121100151] has 1 messages (0 E / 0 W / 1U)
    U    -9 Marital status NOTAPPL not in value set


User unnumbered messages:


   Line    Freq  Pct.  Message text                                    Denom
   ----    ----  ----  ------------                                    -----
      9      33    -   Marital status %d not in value set                 -


CSPRO Executor Normal End
--------------------------------------------------------------------------------
```

The listing file displays the case id above the error messages for that case. We can view any of the individual cases that have errors by copying the case id from the listing file and pasting it into the search box in DataViewer.

We can provide a denominator to the **errmsg** function so that it can compute a percentage for the error. The denominator is a variable that is used to count the universe that error applies too. This universe differs depending on the error. If it is an error on a single record then the denominator will be the number of cases. If the error applies to all individuals then the denominator will be the total number of individuals. If the error applies only to women then the denominator will be the total number of women. To calculate the denominator, we increment a logic variable every time we encounter a case or individual in the universe of the error. At the end of the batch edit run, the final value of the variable will be used as the denominator. For example, for the marital status check we would use a denominator of total number of individuals. We declare a global variable named numIndividuals which we increment in the line number proc (although we could do this in the proc of any variable on the person record).

```
PROC GLOBAL
numeric numIndividuals = 0;

PROC CENSUS_DICTIONARY_FF

PROC LINE

inc(numIndividuals);

PROC P05_MS

if not invalueset($) then
    errmsg("Marital status %d not in value set", $) denom = numIndividuals;
endif;
```

With the denominator, we now get the percentage of individuals with invalid marital status in the listing file.

```
   Line    Freq  Pct.  Message text                                   Denom

   ----    ----  ----  ------------                                   -----

     13      33   0.1  Marital status %d not in value set            24271
```

Group Exercise

Use batch edit to identify cases in the data file where number of children born alive (P18_BORN) is inconsistent with age and/or sex. Specifically, print a message for all females under 12 with children born alive greater than zero and for all males of any age with children born alive greater than zero. Use **denom** in your errmsg to compute the percentages.

Correcting errors

To correct an error in the data file we can simply change the value of the variable by assigning a new value to it. For example, to set the marital status to "married" for all individuals with invalid marital status we assign it to 1.

```
PROC P05_MS

if not invalueset($) then
    errmsg("Marital status %d not in value set", $) denom = numIndividuals;

    $ = 1; // Set to married
endif;
```

When we run this program, we specify an output data file. Any changes our logic makes to the data will only be reflected in the output file. The input file is never modified in batch edit.

This fixes the error, although this may not be the most correct fix. We could instead come up with a set of rules to decide how to correct this error. For example:

1) If the individual is 10 years of old or under marital status = never married
2) If the relationship is spouse marital status = married
3) If the relationship is head and there is at least one spouse in the household then marital status = married
4) Otherwise pick a random value within the value set

We can implement this easily in CSPro logic.

```
PROC P05_MS

if not invalueset($) then
    errmsg("Marital status %d not in value set", $) denom = numIndividuals;

    if P04_AGE < 10 then
        $ = 5; // Child - never married
    elseif P02_REL = 2 then
        $ = 2; // Spouse - must be married
    elseif P02_REL = 1 and count(P02_REL = 2) > 0 then
        $ = 2 // Head with spouse - married
    else
        $ = random(1,5); // Choose randomly
    endif;

endif;
```

To be able to better see how we are changing the data, instead of assigning new values, we can use the **impute()** command. Impute modifies the variable and in addition it tracks all of the changes made and shows us a frequency distribution of the values that were imputed. Modifying our code to use impute gives the following:

```
PROC P05_MS

if not invalueset($) then
    errmsg("Marital status %d not in value set", $) denom = numIndividuals;

    if P04_AGE < 10 then
        impute($, 1); // Spouse - must be married
    elseif P02_REL = 1 and count(P02_REL = 2) > 0 then
        impute($, 1); // Head with spouse - married
    else
        impute($, random(1,5)); // Choose randomly
    endif;

endif;
```

When we run this, in addition to the listing file, we get an imputation frequency file that shows the number of times each value of marital status was imputed.

```
                          IMPUTE FREQUENCIES                          Page    1


_____________________________________________________________________________

         Imputed Item P05_MS: Marital status - all occurrences

                                  _____________________________ _____________
 Categories                             Frequency       CumFreq       %  Cum %
_________________________________ _____________________________ _____________
 1 Married                                     23            23    69.7   69.7
 2 Divorced                                     4            27    12.1   81.8
 3 Separated                                    2            29     6.1   87.9
 4 Widowed                                      1            30     3.0   90.9
 5 Never Married                                3            33     9.1  100.0
_________________________________ _____________________________ _____________
 TOTAL                                         33            33   100.0  100.0
```

From the imputation frequency file, we can see that our rules appear to make individuals married almost 70% of the time.

Another way to verify that our corrections are working and that we have not introduced new errors in the data is to run the batch application again using the output data file as the input data file. If our corrections have been done correctly, we should see no error messages when run on the output file.

Once we have added a lot of error messages, the listing file can get quite busy. You can limit the error messages so that only the final summary message is written to the listing file by adding the key word **summary** after your **errmsg** call.

```
errmsg("Marital status invalid") denom = numIndividuals summary;
```

When you add **summary**, you will no longer see the error messages on each individual case. It is common to leave out the summary when first editing a variable so that you can look at individual cases for patterns and then add summary once you have completed writing the edit.

Group Exercise

Use the following algorithm to correct the invalid values for literacy (P11_LITERACY):

- If age is less than 5, set literacy to blank (question does not apply for under 5)
- If highest grade completed (P10_HIGH_GR) is greater than or equal to 5 (primary school) set literacy to literate
- If highest grade completed is less than 5, set literacy to illiterate

Use the impute function and check how frequently each value of literacy is imputed.

Cold deck imputation

Often, rules for imputations like the ones above are difficult to come up with and may significantly change the distribution of the data. Another imputation technique is to use a matrix containing common valid values for a variable based on the values of other variables. For example, the following matrix shows the most common values of marital status for a given age and sex.

	Sex	
Age	**Male**	**Female**
< 12	Never married	Never married
12-19	Never married	Never married
20-29	Never married	Married
30-39	Married	Married
40-49	Married	Married
50-59	Married	Married
60+	Married	Widowed

We can use this matrix to replace invalid values in our data. If we find and individual with invalid marital status, we find the marital status in the table above that matches the age and sex of the individual with the invalid marital status. For example, if we find an individual whose age is 15 and sex is male then we find the cell in the table for age 12-19 and sex male which contains the value "never married". We then impute never married for the marital status. The cold deck technique can be implemented easily in CSPro by representing the table as an array, however, it will set all the invalid values for a particular age and sex to the same valid value. In the above example we see that we will never impute a value of divorced or separated.

Hot deck imputation

Hot deck imputation builds on the cold deck technique but instead of always using the same value for each cell of the table, it pulls valid values into the table from valid cases in the data file and uses those values to replace the invalid ones. As you loop through the data file checking for invalid values of a variable, every time you find a valid value for the variable, add it to the hot deck in the appropriate cell. Every time you find an invalid value, use the current value in the appropriate cell of the hot deck to replace the invalid one.

As an example, let's apply the hot deck technique for marital status using a table like the one for the cold deck above on the following population.

Person num	Age	Sex	Marital Status
1	40	M	Married
2	40	F	Divorced
3	10	M	Never married
4	42	F	blank
5	47	F	Married
6	41	F	blank

For the first person, we have a valid value for marital status so we store that value in the hot deck in the cell for age 40-49 and male.

	Sex	
Age	**Male**	**Female**
< 12		
12-19		
20-29		
30-39		
40-49	Married	
50-59		
60+		

The next person also has a valid value for marital status so we add divorced to the cell for female age 40-49

	Sex	
Age	**Male**	**Female**
< 12		
12-19		
20-29		
30-39		
40-49	Married	Divorced
50-59		
60+		

For the next person we add never married to the male under 12 cell.

	Sex	
Age	**Male**	**Female**
< 12	Never married	
12-19		
20-29		
30-39		
40-49	Married	Divorced
50-59		
60+		

Person number 4 has an invalid value for marital status so we get the value from the cell in the hot deck for female aged 40-49 which is Divorced. We impute divorced for the marital status of person number 4. We do not change the hot deck.

Person number 5 has a valid value for marital status so we replace the value in the hot deck for female 40-49 with married.

	Sex	
Age	**Male**	**Female**
< 12	Never married	
12-19		
20-29		
30-39		
40-49	Married	Married
50-59		
60+		

Person number 6 has an invalid marital status so we replace it with the value from the hot deck for female 40-49 which is now "married".

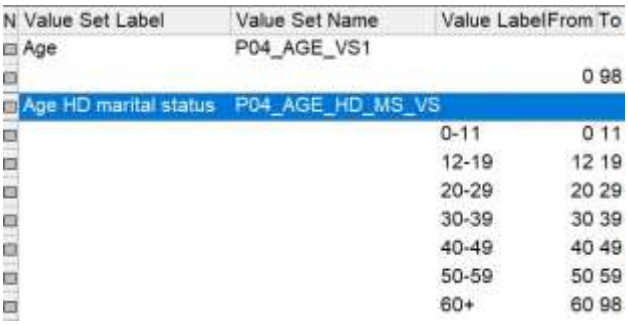
These imputations result in the following output data:

Person num	Age	Sex	Marital Status
1	40	M	Married
2	40	F	Divorced
3	10	M	Never married
4	42	F	Divorced
5	47	F	Married
6	41	F	Married

Unlike with the cold deck, the hot deck imputed two different values for women age 40-49.

One issue with using the hot deck technique is how to initialize the matrix. One option is to use a cold deck to determine the starting values. Another option is to run the edit application twice. After the first run you discard the output file and save the values written to the hot deck. The saved hot deck is then used to initialize the matrix for the second run which does the actual edits.

CSPro provides a number of features that makes it easy to work with hot decks. CSPro supports a special kind of array called a *deck array*. A deck array is declared like a normal array except that it is possible to put a value set as one or more of the dimensions. To create the hot deck with the dimensions above we would first add the appropriate value sets to the dictionary for the row and column groups of the table. We already have a value set for sex with the two vales male and female but we need to create a new value set for age that has the same groupings used in the hot deck.



N	Value Set Label	Value Set Name	Value Label	From To
	Age	P04_AGE_VS1		
				0 98
	Age HD marital status	P04_AGE_HD_MS_VS		
			0-11	0 11
			12-19	12 19
			20-29	20 29
			30-39	30 39
			40-49	40 49
			50-59	50 59
			60+	60 98

We can then use the names of our value sets as the dimensions of the array.

```
PROC GLOBAL
array hdMaritalStatusFromAgeSex(P04_AGE_HD_MS_VS, P03_SEX_VS1) save;
```

CSPro will look at the number of entries in the value sets used and create a 7x2 array. The `save` keyword will save the values in the array to a file after the application is run and reload them to initialize the array on the next run. This supports running the edit program twice to initialize the hot deck.

To retrieve a value from the hot deck we use the function **getdeck()** which takes the name of the hot deck array and the values to use to index the array. Usually these values are the variables for the value sets that were used to declare the array.

```
getdeck(hdMaritalStatusFromAgeSex, P04_AGE, P03_SEX)
```

To add or replace a value in the hot deck we use **putdeck()** which takes the name of the array, the value to insert and the indexes to use.

```
putdeck(hdMaritalStatusFromAgeSex, P05_MS, P04_AGE, P03_SEX)
```

Putting this all together, when we find a valid marital status we use **putdeck** to add it to the hot deck and when we find an invalid marital status, we use **getdeck** to retrieve the valid value and impute it.

```
PROC P05_MS

if invalueset($) then
    putdeck(hdMaritalStatusFromAgeSex, $, P04_AGE, P03_SEX);
else
    errmsg("Marital status %d not in value set", $) denom = numIndividuals;
    impute($, getdeck(hdMaritalStatusFromAgeSex, P04_AGE, P03_SEX));
endif;
```

We should run our edit twice, once to initialize the hot deck and a second time to do the imputations.

When using hot deck arrays, in addition to the imputation report, you can also view the saved hot deck file. It is saved in the same directory as the batch application and has the file extension ".sva" for "saved array". If you double click on this file, the saved array viewer starts and displays the values in the array from the end of the last run of the application. Any cells that have not been set with putdeck will have the value "default". These cells can be problematic. If you try to impute using the value retrieved from them you will introduce variables with value "default" in the output file. If you have such cells, you should adjust the value sets used to declare the array to combine the cells containing default with adjacent cells that have non-default values in them. The saved array viewer can also show you the number of putdeck and getdeck calls for each cell in the table which can also help you adjust the value sets used for the deck rows and columns.

Note that hot decks do not have be two-dimensional. Sometimes a single dimension or even a 3- or 4-dimension hot deck can be appropriate. Deck arrays in CSPro support any number of dimensions.

Group Exercise

Impute valid values for children born alive using the following rules:

1) For males, set children born alive to blank (notappl)

2) For females under 12, set children born alive to blank (notappl)

3) For all other invalid or unreported values use a hot deck to impute the value. The hot deck should be based on age and highest grade completed. Make appropriate value sets of both to use for the hot deck and verify that the hot deck is working correctly using saved array viewer.

When to use hot decks

Hot deck imputation works best when the percentage of invalid values in the data is low so that for every invalid value there are plenty of valid values to use to fill the hot deck. This is generally the case for population censuses and large surveys, however for small sample surveys hot deck may not be appropriate as even a small change in the distribution of responses in a small sample is magnified. In such situations it may be better to use a rule based approached or to simply set invalid values to "unknown" or "no response".

Using batch edit to remove cases

Sometimes it is useful to get a subset of a data file by filtering out certain cases. For example, removing empty households from the data file or even just taking a 10% sample to have a smaller file to work with. In batch edit, this can be done using the **skip case** command. When skip is run on any proc in a questionnaire, it stops processing that case and does not write it to the output file.

As an example, let's create a batch application to remove all the empty households from the data file. An empty household is one that contains no person records. We create a new batch application using the Popstan census dictionary and the following logic:

```
PROC QUEST
preproc
if count(PERSON) = 0  then
    errmsg("Removing empty household") summary;
    skip case;
endif;
```

After running this application, we see that 416 households were removed. Checking the size of the output file we see that it contains only 4,456 cases which is 416 less than the 4,872 cases in the original data.

Group Exercise

Write a batch edit program to create a data file containing only households headed by children under 18. How many households do you get in the output file?

We could use the same technique to create a sample file of the original data file if we wanted to work with a smaller dataset. However, CSPro has built-in function for creating a sample in the dictionary macros.

Splitting files

Let's say we wanted to split our data file into a single file for each province so that we can analyze each province separately. We can do this using the setoutput function. Setoutput allows you to change the output data file from inside your batch edit application. To split the data file into one output file per province we can set the name of the output file based on the province code.

```
setoutput(maketext("Prov%02d.csdb", PROVINCE));
```

After running our batch application, we end up with multiple files: Prov01.csdb, Prov02.csdb,…

Adding weights

You can use batch edit to add new variables to a data file. You might use this to add a weight variable to your data. Attaching the weight to each household in the data file will make it easier to produce weighted tables (we will see that tomorrow). Say we have weights for each district computed from a sample in an Excel Spreadsheet like this one:

PROV	DISTRICT	WGT
1	1	1.2345
1	2	3.4567
1	3	5.6789
1	4	7.8901
1	5	9.0123

We can add a new variable for the weight to the end of the household record. We add it to the end to avoid changing the start positions of the existing data.

Next, we convert the weights spreadsheet to a CSPro lookup file and dictionary using Excel2CSPro. We add the new dictionary to our batch application and now we can use **loadcase** to assign the weight.

```
PROC WEIGHT

// Load weight from lookup
if loadcase(WEIGHT_DICT, PROVINCE, DISTRICT) then
    $ = WGT;
else
    errmsg("Missing weight for %d %d", PROVINCE, DISTRICT);
endif;
```

Combining data files

In session 2, we implemented flow control by separating our data into multiple applications. The result is that we have separate data files for each section of the questionnaire. In order to analyze the data, we need to combine these separate sections into a single data file. We can do this using a batch edit application. Create a new batch edit application with the combined dictionary (the one containing all the sections) as the main dictionary. As the input file we use the main household roster data file. Since this is a subset of the combined dictionary, our batch application will read it and leave the records and variables for the other sections blank. We add the section dictionaries to our batch application as external dictionaries. For each case that is read in the main dictionary, we use loadcase to load the corresponding household in the external dictionary for each section. We then copy the values one by one from the section dictionary to the combined dictionary. Our output file will contain the data from all the section data files merged together.

Session 7: Tabulation

At the end of this lesson participants will be able to:

- Create cross tabulations between two variables
- Use universes in cross tabulations
- Apply formatting to tables
- Add percentages and summary statistics to tables
- Use tablogic to compute new variables in a tabulation application
- Use postcalc logic to add computed rows columns to a table
- Run tables by area
- Copy and export tables

Cross tabulation applications

To create cross tabulations in CSPro we create a new application of type *Tabulation*. Just like when you create a data entry or batch edit application you choose a data dictionary. You are initially presented with a blank table. On the left-hand side of the window you see your data dictionary. Dragging an item or a value set from the dictionary to the table adds that item to the table. Think of the table area on the right as a box, with an imaginary line running from the upper left to the lower right. If you drop a variable in the upper right it becomes a column, but if you drop it in the lower left it becomes a row. To create a cross tabulation, drop one item as a row and another as a column. Drag the variable **Sex** from the dictionary to the top of the table to make columns and drag and drop **Relationship** on the left to make rows. To run the table, click the traffic light and pick the data file. The result should look like:

Table 1. Relationship by Sex

Relationship	Sex		
	Total	Male	Female
Total	24,271	11,787	12,484
Head	4,456	3,470	986
Spouse	3,131	59	3,072
Child	12,110	6,126	5,984
Parent	249	51	198
Other relative	3,971	1,924	2,047
Nonrelative	354	157	197
Not Reported	-	-	-

You can add more tables to your table application by clicking the "Add table" button on the toolbar. Let's add a second table that crosses Age by Sex. Note that for the variable Age, we have multiple value

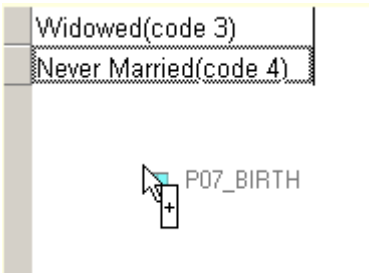
sets to choose from. We can drag and drop age in 5-year groups, age in 10-year groups, single year of age… We can always edit the dictionary to add additional value sets to use in our tables. Let's use 5-year groups for this example. When we run our table note that both tables are regenerated. Every time the application is run, all tables in the application are recomputed.

Table 2. Age in 5 year groups by Sex

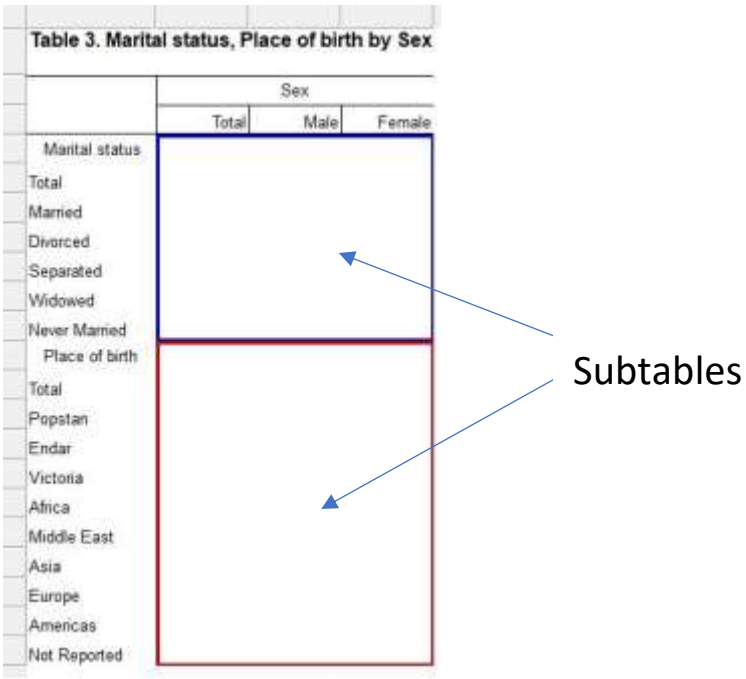
Age in 5 year groups	Sex		
	Total	Male	Female
Total	24,271	11,787	12,484
0 to 4 years	4,130	2,088	2,042
5 to 9 years	3,337	1,646	1,691
10 to 14 years	3,250	1,606	1,644
15 to 19 years	2,967	1,394	1,573
20 to 24 years	2,243	1,061	1,182
25 to 29 years	1,686	789	897
30 to 34 years	1,508	761	747
35 to 39 years	1,300	600	700
40 to 44 years	957	474	483
45 to 49 years	661	323	338
50 to 54 years	588	303	285
55 to 59 years	497	225	272
60 to 64 years	418	194	224
65 to 69 years	307	141	166
70 to 74 years	213	90	123
75 to 79 years	115	51	64
80 to 84 years	63	31	32
85 to 89 years	17	8	9
90 to 94 years	14	2	12
95 years and over	-	-	-

Subtables

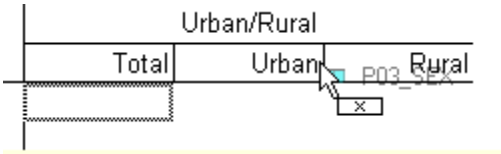
You can drag more than one variable onto the rows or columns. Create a new table and drop **Marital status** on the rows and **Sex** on the columns. Now drop Place of Birth on the left underneath Marital status. You should see a little plus sign if you are dropping it in the correct place.



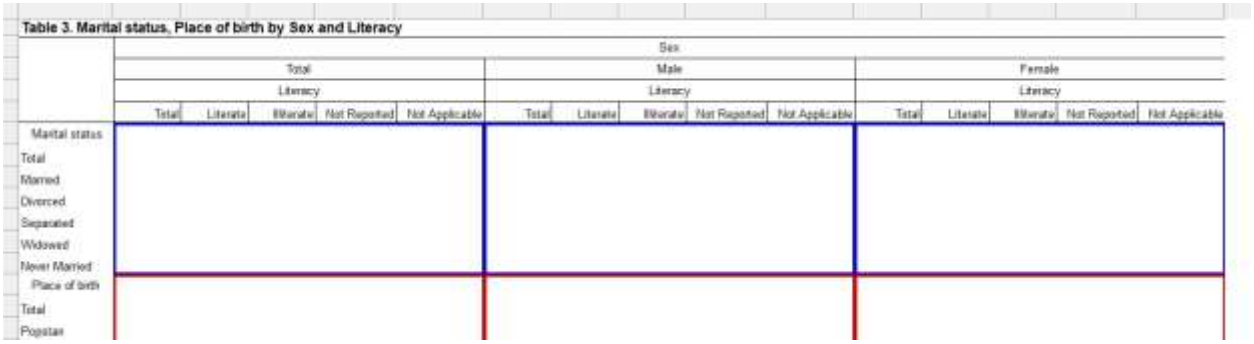
This adds rows for place of birth below the rows for marital status. You should now see a red box and a blue box on your table. These boxes show the *subtables* that have been created. Subtables are individual cross-tabulations that are combined to make the full table.



Instead of dropping the second variable below or to the right of an already dropped variable, you can also drop the new item on top of the existing rows or columns to create 'crossed' subgroupings. When you do this, you will see an "x" instead of the plus sign as you drag the item.



Let's create a crossed subgrouping with Sex and Literacy by dropping literacy from the dictionary on top of sex. The table now has all the columns for literacy repeated for each value of sex.



Universe

You can add a universe to a table by right clicking on the table and choosing "Tally Attributes Table". A universe is an expression used to filter the records that are used in the table. For example, to limit a

table to only individuals 5 and under you could use a universe of *P04_AGE <=5*. As an example, let's make a table of age by literacy for only women. We drop the age and literacy on the table and then use tally attributes table to enter the universe of P03_SEX=2. After running the table with and without the universe we can see that the totals drop by about half. Note that when we add a universe, we probably want to modify the table title to indicate what it is. We can do that by simply double clicking on the title and editing it. You can customize the text in any part of the table this way.

Table 5. Age in 10 year groups by Literacy

Age in 10 year groups	Literacy				
	Total	Literate	Illiterate	Not Reported	Not Applicable
Total	24,271	13,908	2,896	3,337	4,130
0 to 9 years	7,467	-	-	3,337	4,130
10 to 19 years	6,217	5,137	1,080	-	-
20 to 29 years	3,929	3,407	522	-	-
30 to 39 years	2,808	2,483	325	-	-
40 to 49 years	1,618	1,403	215	-	-
50 to 59 years	1,085	774	311	-	-
60 to 69 years	725	450	275	-	-
70 to 79 years	328	200	128	-	-
80 to 89 years	80	51	29	-	-
90 years and over	14	3	11	-	-

Table 5. Age in 10 year groups by Literacy for females

Age in 10 year groups	Literacy				
	Total	Literate	Illiterate	Not Reported	Not Applicable
Total	12,484	7,063	1,688	1,851	2,042
0 to 9 years	3,733	-	-	1,691	2,042
10 to 19 years	3,217	2,857	360	-	-
20 to 29 years	2,079	1,777	302	-	-
30 to 39 years	1,447	1,263	184	-	-
40 to 49 years	821	681	140	-	-
50 to 59 years	557	345	212	-	-
60 to 69 years	398	210	188	-	-
70 to 79 years	187	104	83	-	-
80 to 89 years	41	23	18	-	-
90 years and over	12	3	9	-	-

Looking at this table, all the values for age 0 to 9 years are either not reported or not applicable. It would appear that the universe for literacy is age 10 and above. We should also limit the universe of the table to age 10 and above. We can modify the universe to be P03_SEX=12and P04_AGE >= 10. With this universe the not reported and not applicable columns are now all zero, however it would be nice to get rid of those rows entirely as well as the 0 to 9 years columns. We can do this by making new value sets in the dictionary that don't contain those values and dropping them onto the table. Alternatively, we can stick with the current value set and hide the rows and columns we don’t want. To hide a column, right click on the column header, choose "Format (Column Head)" and check "Hide". For the row, right click on the row header, choose "Format (Stub)" and check "Hide". Another option is to check the box "Hide when all cells are zero" which will only hide the row or column if it contains no cells with non-zero counts.

Group Exercise

Create the following tables in CSPro using the Popstan Census dictionary and data.

Table 1. Mother living by Sex

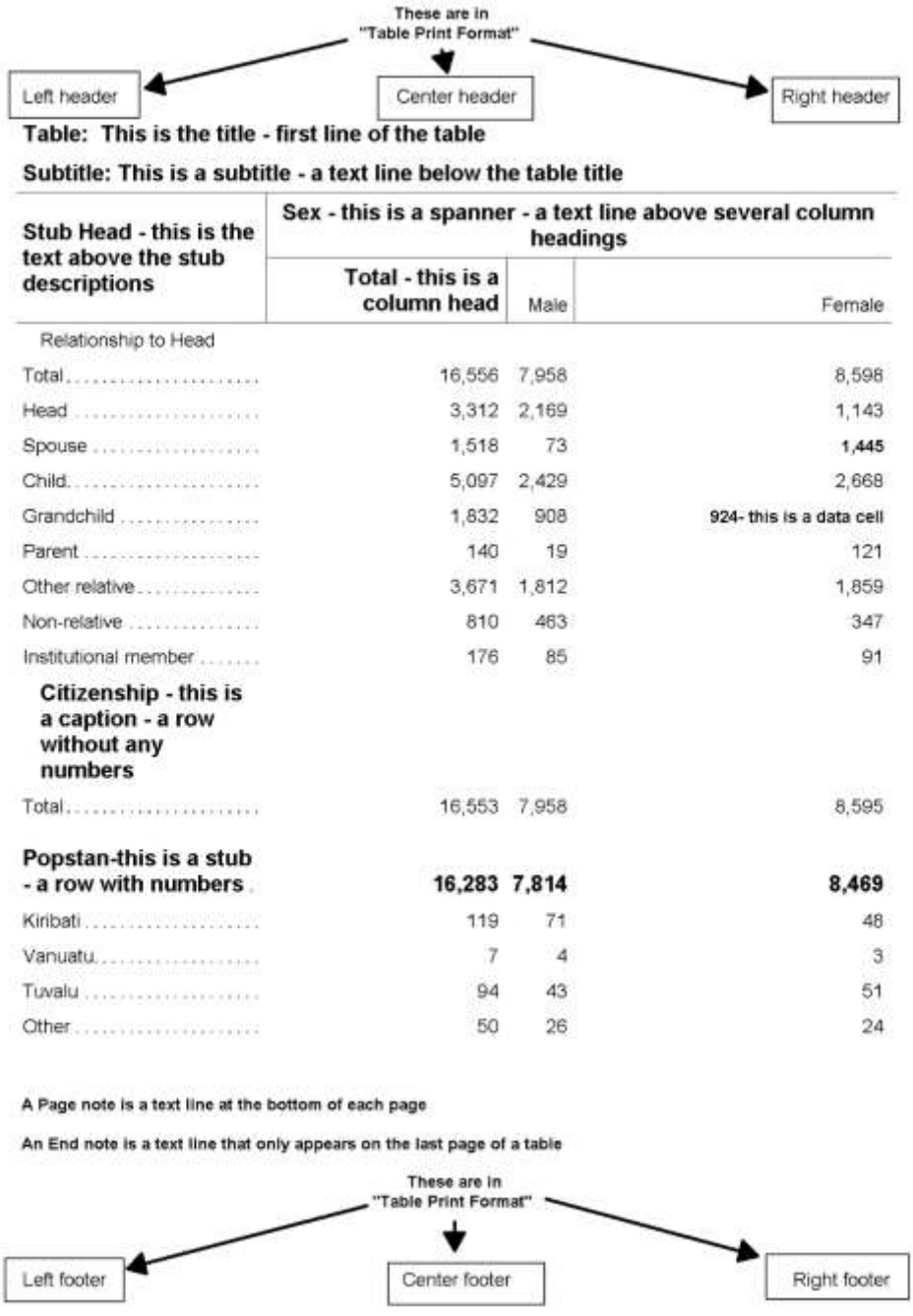
Mother living	Sex		
	Total	Male	Female
Total	24,271	11,787	12,484
Yes	19,239	9,436	9,803
No	4,655	2,191	2,464
Don't know	377	160	217

Table 2. Age by Attending school and Sex for school age children

Age	Attending school								
	Total			Yes			No		
	Total	Male	Female	Total	Male	Female	Total	Male	Female
Total	9,369	4,534	4,835	4,656	2,283	2,373	4,713	2,251	2,462
6	614	290	324	292	133	159	322	157	165
7	643	309	334	494	193	211	239	116	123
8	767	355	352	462	233	229	245	122	123
9	642	310	332	448	212	236	194	98	96
10	645	337	308	447	237	210	198	100	98
11	656	329	327	448	227	221	208	102	106
12	612	311	301	372	181	191	240	130	110
13	669	306	363	367	179	188	302	127	175
14	668	323	345	325	159	166	343	164	179
15	661	297	364	262	119	143	399	178	221
16	636	299	337	250	132	118	388	167	213
17	612	292	320	183	90	93	429	202	227
18	537	235	302	167	76	91	370	159	211
19	527	271	256	128	61	67	399	210	189
20	546	270	276	101	51	50	445	219	226

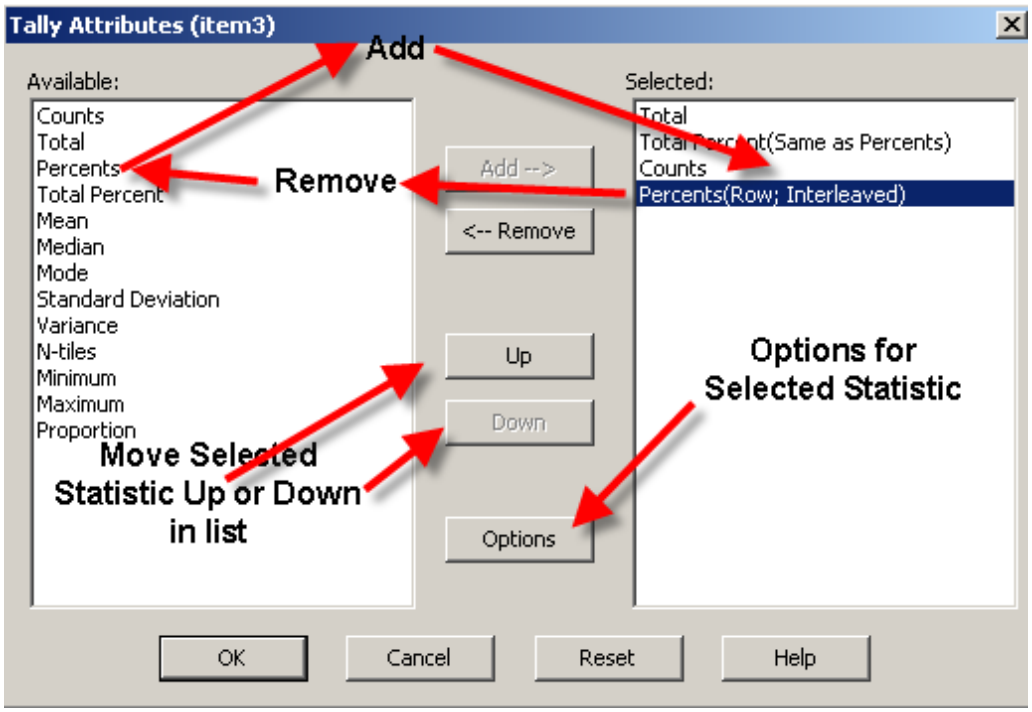
Table Formatting

You may have noticed that in addition to the "Hide" option, the format dialog for the stub and column head also allows for a number of formatting options such as changing fonts, colors, borders and alignment. Column head and stub are two parts of a table in CSPro that can be formatted but there are many others. The following diagram shows the various parts of the table and the terms used to describe them.



Percentages and statistics

In addition to the "Tally Attributes Table" there are tally attributes for each variable used in the table. You can modify these tally attributes to add percentages and statistics such as mean and median.



By default, every variable added to the table will include the counts and the total. You can remove either or both of these and add any of the above additional attributes from the tally attributes dialog. You can also reorder the attributes for each variable.

Let's add percents to the relationship in the relationship by sex table. Percents can be interleaved with the counts or separate depending on the options chosen.

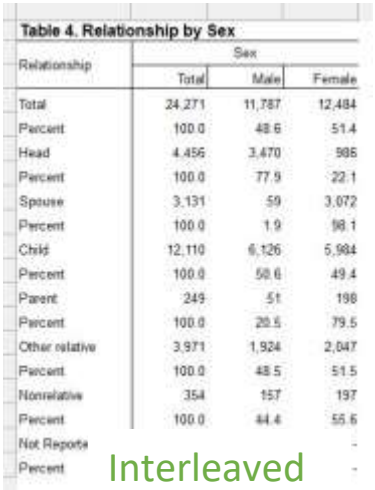


Table 4. Relationship by Sex

Relationship	Sex		
	Total	Male	Female
Total	24,271	11,787	12,484
Percent	100.0	48.6	51.4
Head	4,456	3,470	986
Percent	100.0	77.9	22.1
Spouse	3,131	59	3,072
Percent	100.0	1.9	98.1
Child	12,110	6,126	5,984
Percent	100.0	50.6	49.4
Parent	249	51	198
Percent	100.0	20.5	79.5
Other relative	3,971	1,924	2,047
Percent	100.0	48.5	51.5
Nonrelative	354	157	197
Percent	100.0	44.4	55.6
Not Reporte			
Percent			

Interleaved

Table 4. Relationship by Sex

Relationship	Sex		
	Total	Male	Female
Total	24,271	11,787	12,484
% Total	100.0	48.6	51.4
Head	4,456	3,470	986
Spouse	3,131	59	3,072
Child	12,110	6,126	5,984
Parent	249	51	198
Other relative	3,971	1,924	2,047
Nonrelative	354	157	197
Not Reported	-	-	-
% Head	100.0	77.9	22.1
% Spouse	100.0	1.9	98.1
% Child	100.0	50.6	49.4
% Parent	100.0	20.5	79.5
% Other relative	100.0	48.5	51.5
% Nonrelati			55.6
% Not Repo			-

Separate

It is also possible to remove the counts and show just the percents. The percent total can also be included or excluded separately from the percents themselves.

Group exercise

Create the following table in CSPro using the Popstan Census dictionary and data.

Table 5. Number of children still living by Marital status for women age 12 and above

Number of children still living	Marital status					
	Total	Married	Divorced	Separated	Widowed	Never Married
None	3,629	495	95	-	103	2,936
1	844	484	113	-	67	180
2	733	515	93	-	50	75
3	601	426	79	-	60	36
4	559	378	60	-	58	63
5	385	278	46	-	52	9
6	413	292	39	-	68	14
7	260	194	21	-	44	1
8	243	171	20	-	43	9
9	166	120	8	-	37	1
10+	283	192	25	-	57	9
Median	1.5	3.7	3.0	-	4.7	0.6
Mean	2.4	3.8	3.2	-	4.5	0.3

Weights

If you have a variable in the data file representing the weight for each case you can use that variable to weight the tables. In the "Tally Attributes Table" dialog enter the name of the weight variable. You can also use the "Apply to all tables" button to copy this weight variable to all the tables in your application if you want to weight all of them.

Tablogic

Sometimes we need to tabulate a value that doesn't directly exist in our data file but can be calculated from the existing variables. This can be done using tablogic. Tablogic allows you to add CSPro logic that is executed during tabulation. This is mainly used to add your own "recoded" variables to tables instead of using existing variables from your dictionary. This is useful when the categories you want in your table must be tallied based on the values of more than one of the variables in the dictionary, in other words, when the categories you want must be computed based on the values of multiple variables. In such a case, you can create a new variable with the categories you want and write logic to set its value based on the values of existing variables.

For example, if you want to tabulate houses that have "complete plumbing," meaning that they have piped water, bathing facilities and a toilet inside the housing unit. The categories for "complete plumbing" are:

- Complete – piped water inside the unit, and a private toilet inside the unit and bathing facilities inside the unit.
- Some but not all –one or more of the three conditions above, but not all three.
- None – none of the above conditions.

The variable "complete plumbing" does not exist in the main dictionary. It can be however, be determined based on the values of the following three variables that are in the dictionary:

N	Value Set Label	Value Set Name	Value Label	From
☐	Source of water	H10_WATER_VS1		
☐			Inside piped	1
☐			Outside piped	2
☐			Bottled or canned	3
☐			Closed well or close	4
☐			Open well or spring	5
☐			River/Stream/Other	6

N	Value Set Label	Value Set Name	Value Label	From
☐	Type of bathing fac	H09_BATH_VS1		
☐			Exclusive	1
☐			Shared	2
☐			Hand basin	3
☐			Portable tub or basir	4
☐			Other	5
☐			None	6

N	Value Set Label	Value Set Name	Value Label	From
☐	Type of toilet faciliti	H08_TOILET_VS1		
☐			Private toilet	1
☐			Shared toilet	2
☐			Outhouse	3
☐			Pit	4
☐			Other	5
☐			None	6

Rather than adding new variables to your existing dictionary, you can add new variables to the working storage dictionary. The working storage dictionary is automatically added when you create a tabulation application.

In this example, we will add the new "complete plumbing variable" to the working storage dictionary. The working storage dictionary appears just below the main dictionary in the dictionary tree. Adding a new variable to the working storage dictionary is the same as adding a variable to any dictionary. Right-click on the "Working Storage Record" under the working storage dictionary in the dictionary tree and choose "Add Item". Add a value set with: Complete – 1, Some but not all – 2, None – 3.

Drag the new item onto the rows of a table and drag urban/rural from the id-items onto the columns to make the table Complete plumbing by Urban/Rural.

Finally add the tablogic to set the value of complete plumbing based on the values of source of water, toilet facilities and bathing facilities. Bring up the Tally Attributes (Table) dialog and enter the following code in the tab logic edit box:

```
if H08_TOILET = 1 and H09_BATH = 1 and H10_WATER = 1 then
    COMPLETE_PLUMBING = 1; // complete
elseif H08_TOILET = 1 or H09_BATH = 1 or H10_WATER = 1 then
    COMPLETE_PLUMBING = 2; // some
else
    COMPLETE_PLUMBING = 3; // none;
endif;
```

The final table should look like:

Table 1. Complete Plumbing by Urban/Rural

Complete Plumbing	Urban/Rural		
	Total	Urban	Rural
Total	4,872	2,821	2,051
Complete	106	106	-
Some but not all	361	354	7
None	4,405	2,361	2,044

Group exercise

Create the following table:

Table 1. Children deceased since birth by Marital status for women age 12 and above

Children deceased since birth	Marital status					
	Total	Married	Divorced	Separated	Widowed	Never Married
Total	8,116	3,545	599	-	639	3,333
0	6,513	2,488	411	-	352	3,262
1	752	524	86	-	89	53
2	395	257	58	-	69	11
3	213	139	24	-	48	2
4	111	66	12	-	31	2
5+	132	71	8	-	50	3

Create a new variable in the working storage dictionary for children deceased since birth. Compute this variable by subtracting P19_LIVING from P18_BORN. Make sure to use the universe to include only women 12 and above.

Postcalc logic

Sometimes you want to add additional calculations to the table that need to occur *after* the tabulation run when the counts are complete. For example, lets add the following table that shows the sex ratio (males divided by females) for the population.

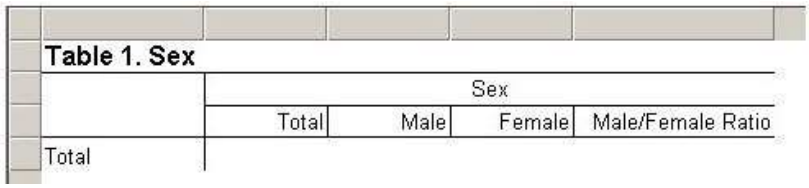


Table 1. Sex				
	Sex			
	Total	Male	Female	Male/Female Ratio
Total				

In order to add a column to hold the sex ratio we need to create a new value set for the sex variable that includes the additional value "Male/Female Ratio". We can make it code 3 although the code does not really matter as we will overwrite the numbers in this column later. Drag the new value set for sex onto the table and your table should have the column for sex ratio. Rather than add a new value set we could also add a new variable to the working storage dictionary with a value set that has one entry and drag that onto the table. The difference is that the column header for the column would be separate from the column headers for sex.

The calculation of the sex ratio can only be done AFTER the male and female columns have been calculated. For any calculations after the tabulation is run, we add postcalc logic in the Tally Attributes Table dialog. Postcalc logic lets you do calculations and assignments on cells, rows and columns in the completed table. The table is treated as a two-dimensional array where the row and column numbering start with zero.

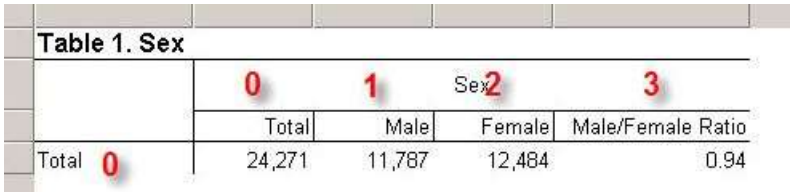
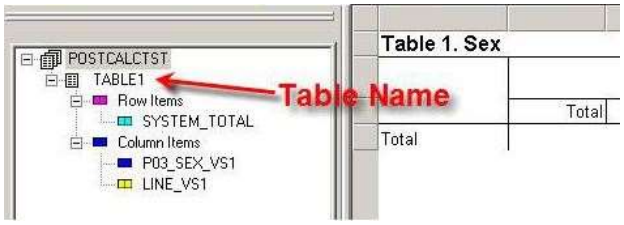


Table 1. Sex				
	0	1	Sex 2	3
	Total	Male	Female	Male/Female Ratio
Total 0	24,271	11,787	12,484	0.94

The name of the array is the same as the name of the table which you can see from the tables tree on the left side of the screen.



In our case the table name is "TABLE1". We can reference any cell in the table using parenthesis to provide the subscripts for row and column i.e. TABLE1(row, column). So TABLE1(0,0) is the total cell, TABLE1(0, 1) is the male cell etc… To compute the sex ratio, we simply divide the male cell by the female cell and put the result in the sex ratio cell:

```
TABLE1(0,3) = TABLE1(0,1) / TABLE1(0,2);
```

We enter this code into the postcalc section of the Tally Attributes Table dialog and run the table to see the results.

What if we want to compute the sex ratio crossed with age as in the following table?

Table 1. Age in 5 year groups by Sex

Age in 5 year groups	0	1	Sex 2	3
	Total	Male	Female	Male/Female Ratio
Total	24,271	11,787	12,484	0.94
0 to 4 years	4,130	2,088	2,042	1.02
5 to 9 years	3,337	1,646	1,691	0.97
10 to 14 years	3,250	1,606	1,644	0.98
15 to 19 years	2,967	1,394	1,573	0.89
20 to 24 years	2,243	1,061	1,182	0.90
25 to 29 years	1,686	789	897	0.88
30 to 34 years	1,508	761	747	1.02
35 to 39 years	1,300	600	700	0.86
40 to 44 years	957	474	483	0.98
45 to 49 years	661	323	338	0.96
50 to 54 years	588	303	285	1.06
55 to 59 years	497	225	272	0.83
60 to 64 years	418	194	224	0.87
65 to 69 years	307	141	166	0.85
70 to 74 years	213	90	123	0.73
75 to 79 years	115	51	64	0.80
80 to 84 years	63	31	32	0.97
85 to 89 years	17	8	9	0.89
90 to 94 years	14	2	12	0.17
95 years and over	-	-	-	-

To get the table shell we can simply drag age in 5 years groups onto the table we just created. However, the postcalc logic now needs to compute the sex ratio for all the rows instead of just the first one. We could set the sex ratio for each cell individually but it would require a lot of repeated code:

```
TABLE1(0,3) = TABLE1(0,1) / TABLE1(0,2);

TABLE1(1,3) = TABLE1(1,1) / TABLE1(1,2);

…

TABLE1(2,3) = TABLE1(2,1) / TABLE1(2,2);

TABLE1(20,3) = TABLE1(20,1) / TABLE1(20,2);
```

Instead we can use square brackets which represent ranges of cells in the table. For example, the following gets the range of cells in column 1, rows 0 through 20 and divides every cell in that range with the corresponding cell in the range column 2, rows 0 through 20. The cells in that are then copied into

the range column 3 rows 0 through 20. In other words, divide row 1 by row 2 and put the results in row 3.

```
TABLE1[0:20,3] = TABLE1[0:20,1] / TABLE1[0:20,2];
```

We an also use the wildcard * in either dimensions to copy an entire row or column without specifying the other dimension.

```
TABLE1[*,3] = TABLE1[*,1] / TABLE1[*,2];
```

When we first run the table note that the sex ratio column is rounded to the nearest whole number. We can change that by setting the number of decimal places for the column to two in the "Format (Column Head)" dialog.

Group Exercise

Create the following table in CSPro.

Table 1. Age by marital status, married/unmarried ratio for individuals 10 and above

Age	Marital status						Married/unmarried ratio
	Total	Married	Divorced	Separated	Widowed	Never Married	
Total	16,804	6,985	799	-	788	8,232	0.71
10 to 19 years	6,217	270	25	-	7	5,915	0.05
20 to 29 years	3,929	1,925	165	-	22	1,817	0.96
30 to 39 years	2,808	2,177	220	-	74	337	3.45
40 to 49 years	1,618	1,283	143	-	125	67	3.83
50 to 59 years	1,085	771	114	-	171	29	2.46
60 to 69 years	725	404	87	-	201	33	1.26
70 to 79 years	328	124	35	-	141	28	0.61
80 to 89 years	80	28	9	-	38	5	0.54
90 years and over	14	3	1	-	9	1	0.27

The married/unmarried ratio is defined as Married/(Divorced + Separated + Widowed + Never Married)

Copying and exporting tables

You can copy and paste a table from CSPro and paste it into Excel, Word or a text document. When pasting into Word, formatting is preserved.

You can also export tables to a few formats by choosing "Save tables" from the file menu. Tables can be exported to:

- CSPro table viewer (tbw) file – tables can be viewed by but not edited by anyone with CSPro
- HTML file – for use on a web page
- Rich text format (rtf) – can be opened and viewed in Microsoft Word and other word processing programs
- Tab delimited – can be imported into spreadheets and databases

Tables by area

An important feature of CSPro tabulation is support for area processing. Area processing generates the same table at different levels of geography that you specify like province, district… To enable area processing choose "Area" () from the toolbar. This brings up the area dialog where you can select the levels of geography to use. Usually these will come from the id-items in your dictionary.

After enabling area processing you will see an additional stub in your table to display the area.

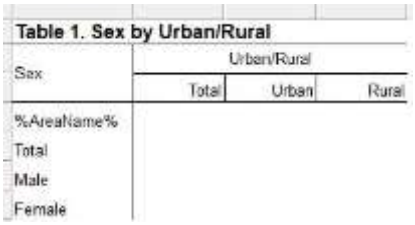


Table 1. Sex by Urban/Rural			
Sex	Urban/Rural		
	Total	Urban	Rural
%AreaName%			
Total			
Male			
Female			

When you run the table, the table is repeated for each geographic area and the stub is replaced with the codes for the area.

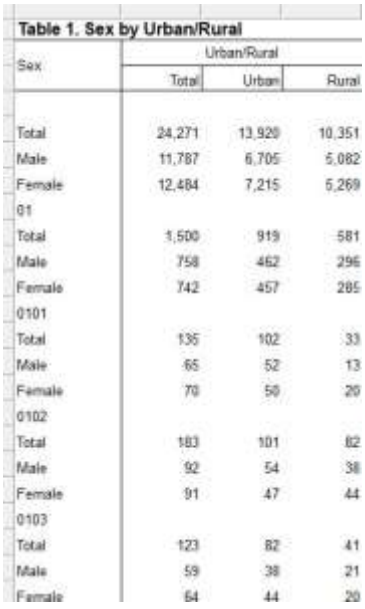
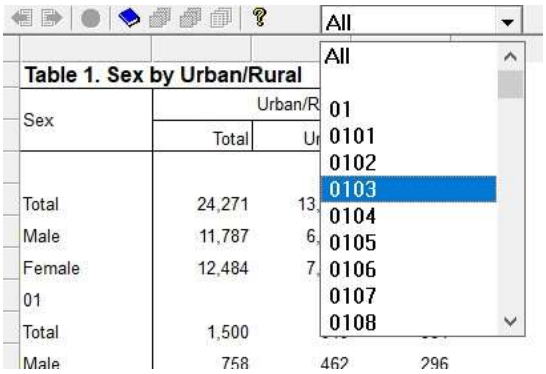


Table 1. Sex by Urban/Rural			
Sex	Urban/Rural		
	Total	Urban	Rural
Total	24,271	13,920	10,351
Male	11,787	6,705	5,082
Female	12,484	7,215	5,269
01			
Total	1,500	919	581
Male	758	462	296
Female	742	457	285
0101			
Total	135	102	33
Male	65	52	13
Female	70	50	20
0102			
Total	183	101	82
Male	92	54	38
Female	91	47	44
0103			
Total	123	82	41
Male	59	38	21
Female	64	44	20

On the toolbar there is a dropdown that allows you to choose to see the table for just one geographic area or for all geographic areas.



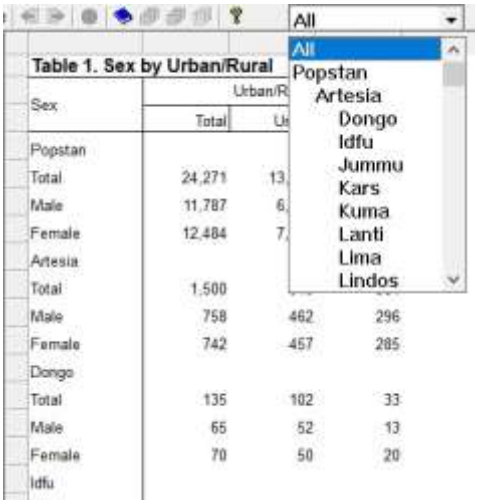
Note that when you add area processing it applies to the entire file, not just the current table.

Working with the raw geographic codes is not very friendly. When we run the tables, we can supply CSPro with an area names file and it will replace the codes with the names of each area.

An area names file lists the codes of each region on the left and the names of the regions on the right. For higher level regions, use "X" for the lower level code. For example, for province 1 the codes are "1" for province 1 and "X" for the district. The code "X X" represents the entire country. The example file below just has 2 levels of hierarchy: province and district. It is possible to have deeper hierarchies to go down to lower levels of geography.

```
[Area Names]
Version=CSPro 7.2

[Levels]
Name=Province
Name=District

[Areas]
X     X     =     Popstan
1     X     =     Artesia
1     1     =     Dongo
1     2     =     Idfu
1     3     =     Jummu
```

Once you provide the area names file to CSPro, your tables will use the names in place of the codes.



Session 8: Export and Production Runner

At the end of this lesson participants will be able to:

- Use the export command in a batch application
- Use production runner to run multiple applications together

Export from logic

In the previous workshop we used the Export Data tool to export data from CSPro to other packages such as Excel, Stata, SPSS and R. This tool can handle most export cases but particularly complex scenarios, it is possible to write your own export program by creating a batch application that calls the **export** command in logic.

As an example, lets export the Popstan census data into 3 files:

1) Household records only
2) Fertility section, including line number, only for women 12 and over
3) Person record without fertility items

We could not do this with a single run of the Export Data tool but it can be done using the export command in a batch application. We start by creating a batch edit application using the Popstan Census Dictionary. To get the initial code for our export, the easiest way is to run the Export tool, setup the export settings we want and then choose "Copy logic to clipboard" from the Options menu. This puts logic for an export with the parameters chosen onto the clipboard. If you paste this logic into your batch edit application, your batch application will now do the export using the settings you selected.

We can then modify the generated logic to export fields from the person record twice: once for the fertility items and once for the remaining items from the population record.

```
PROC GLOBAL

NUMERIC rec_occ;

FILE file_PERSON;
FILE file_FERTILITY;
FILE file_HOUSING;

PROC POPSTANCENSUS_FF

PROC QUEST
PreProc
    set behavior() export (CommaDelim, ItemOnly, ANSI);

    For rec_occ in RECORD PERSON do
        EXPORT TO file_PERSON
        CASE_ID(PROVINCE, DISTRICT, VILLAGE, EA, UR, BUILDING, HU, HH)
        LINE, P02_REL, P03_SEX, P04_AGE, P05_MS, P06_MOTHER, P07_BIRTH,
P08_RES95, P09_ATTEND, P10_HIGH_GR,
P11_LITERACY, P12_WORKING, P13_LOOKING, P14_WHY_NOT, P15_OCC, P15A_OCC,
P16_IND, P16A_IND, P17_WK_STATUS,
ECON_ACTIVE;

        if P03_SEX = 2 and P04_AGE >= 12 then
            EXPORT TO file_FERTILITY
            CASE_ID(PROVINCE, DISTRICT, VILLAGE, EA, UR, BUILDING, HU, HH)
            LINE, P18_BORN, P19_LIVING, P20_BORN12, H04_FLOOR;
        endif;
    Enddo;

    EXPORT TO file_HOUSING
    CASE_ID(PROVINCE, DISTRICT, VILLAGE, EA, UR, BUILDING, HU, HH)
    H01_TYPE, H02_WALL, H03_ROOF, H04_FLOOR, H05_ROOMS, H06_TENURE, H07_RENT,
H08_TOILET, H09_BATH, H10_WATER,
H11_LIGHT, H12_FUEL;
```

You can also use export from batch to export calculated and recoded variables. For example, if wanted to export the complete plumbing variable that we used in our tables we could add a working storage dictionary to our batch application, add the complete plumbing variable to the working storage dictionary and add logic to set the variable before we export it.

Group Exercise

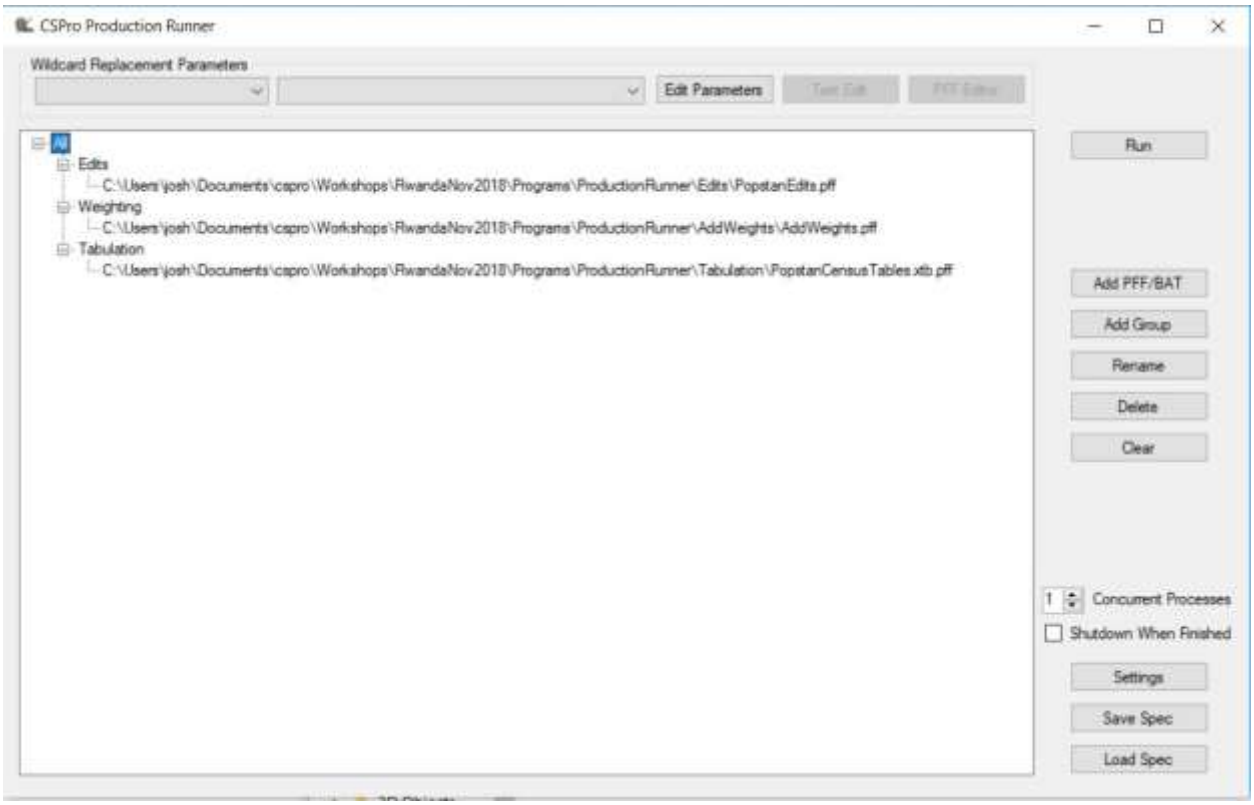
Create a batch application that exports the person record from the Popstan census data into two files: one containing only women and one containing only men.

Production runner

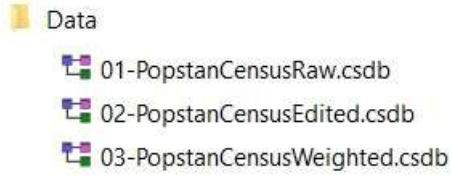
Production runner is a tool that lets you run multiple applications together in sequence.

For example, to produce tables for the Popstan census from the raw data file you now have to first run the edits (twice to initialize the hot decks), then run a second batch application to add weights and then run the tabulation application to generate the tables. Production runner allows us to combine all of these steps together so that they can all be run with a single click.

Start production runner from the tools menu in CSPro. To create a new sequence of applications to run, click on the "Add Group" button. We will create a group named "All". Then we will create child groups under "All" for each of the steps in our production. Then we drag and drop the pff files for each of the steps onto the group for that step.



Before we can launch our production run, we first have to make sure that the program at each step is configured to use the data file from the output of the previous step. To make this easier we can put all the data files in a single data folder and name them for the step that produces them.



To launch the production run, select the group "All" and click the "Run" button. This runs each of the steps in order. This will run all of the batch applications and the tabulation application to generate tables from the edited and weighted data.

We can save our production runner specification so that we can easily run it again later by clicking on the "Save Spec" button. This allows us to rerun the entire process as often as we want.