**Coding sample description**:

To build an analytical data set for drug comparators, we must create “abuse” variables given specific drugs classified. If one of the drugs belonging to the comparator has an abuse response (i.e., equal to one), then the drug comparator group will take on a value of one.

|  |  |  |
| --- | --- | --- |
| opioids\_adf\_label | Reformulated OxyContin | OP1F |
| Xtampza ER | OP1M |
| Hysingla ER | OP3I |
| EMBEDA | OP8H |
| Oxaydo | OP16G |
| Arymo ER | OP8I |
| Morphabond | OP8J |

The logic would be:

If OP1F\_use=1 or OP1M\_use=1 or OP3I\_use=1 or OP8H\_use=1 or OP16G\_use=1 or OP8I\_use=1 or OP8J\_use=1 **THEN** opioids\_adf\_label\_abuse=1; else opioids\_adf\_label\_abuse=0;

On tab A of the specs sheet we see the requested comparator groupings for which we must create the comparator abuse variables.

Continuing to tab B of the specs sheet, we also see must create route of administration variables (how the patient reported he/her abused the drug).

Below are the suffixes corresponding to the ROA abuse variables needed to be created:

* + Swallow whole: R1(name variable: <DRUGname>\_swallow)
  + Snort: R2 (name variable: <DRUGname>\_snort)
  + Smoke: R3 (name variable: <DRUGname>\_smoke)
  + NonIV inject: R4 (name variable: <DRUGname>\_inject\_nonIV)
  + IV inject: R5 (name variable: <DRUGname>\_inject\_IV)
  + Chew: R6 (name variable: <DRUGname>\_chew)
  + Other: R7 (name variable: <DRUGname>\_other)
  + Dissolve: R8 (name variable: <DRUGname>\_dissolve)
  + Drank: R9 (name variable: <DRUGname>\_drank)

For the swallow variable the logic would be:

If OP1FR1=1 or OP1MR1=1 or OP3IR1=1 or OP8HR1=1 or OP16GR1=1 or OP8IR1=1 or OP8JR1=1 **THEN** opioids\_adf\_label\_swallow=1; else opioids\_adf\_label\_swallow=0;

The next component is to create required grouping variables based on above:

Groupings of ROA are sometimes requested by clients. Groupings are as follows:

* + Any oral: R1, R6, R8, R9, R11, R12, R13, R14 (name variabe:<DRUGname>\_anyoral)
  + Any inject: R4, R5 (name variabe:<DRUGname)\_anyinject)

Logic for the “anyinject” grouping would be:

If OP1FR4=1 or OP1MR4=1 or OP3IR4=1 or OP8HR4=1 or OP16GR4=1 or OP8IR4=1 or OP8JR4=1

Or

OP1FR5=1 or OP1MR5=1 or OP3IR5=1 or OP8HR5=1 or OP16GR5=1 or OP8IR5=1 or OP8JR5=1

**THEN** opioids\_adf\_label\_anyinject=1; else opioids\_adf\_label\_anyinject=0;

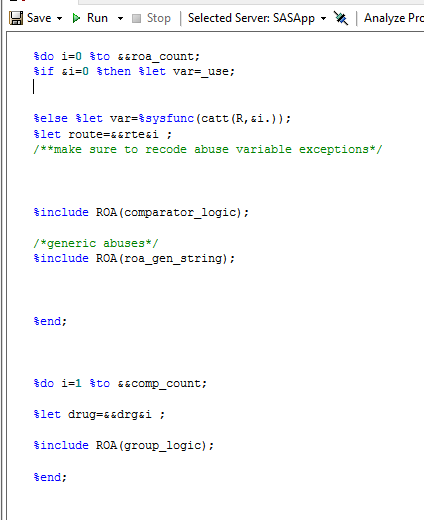
**Objective**: The analytic file must create all the “abuse” and ROA (14 route of administration types) and ROA grouping variables for the comparator groups (in this case two) requested. Since this sample request has 53 comparators requested, we need to create **901** variables. This is a major challenge in terms of the tediousness of the task. In fact, the prior solution used by other analysts was to manually code these variables in their own separate programs (see attached \_priormethod.sas codes with methadone example).

This is not only an extremely tedious task, but it is a headache to make changes to the code if the comparator composition changes (for example, OP8J is not considered part of opioids\_adf\_label anymore).

**Solution**: Because the naming of the opioid abuse variables (e.ge. OP8J) followed a structured suffix pattern (i.e OP8J\_abuse, OP8JR1, OP8JR2…., OP8JR14), I looked to exploit this structure to automate the creation of the variables. I implemented the idea of creating “code strings” using macrotization rather than individually coding each variable. The code would loop through the suffixes (abuse, R1,R2,…R14) to create all the needed variables.

Basically, I decided to take the inputs in from Excel of the comparators (see comparator\_input\_sheet.xlsx), ROA, and requested groups to make the solution more “front end”.

l essentially have the code “create code” in order to actually create the 901 “if/else” statements (in this project). See “comparator\_logic.sas” and “group\_logic.sas” to see the string code outputted by the engine that is later “run” via %include to create the 901 variables. Below shows a snippet of the code involved looping of the variable creation using the codes created by the string engine:



**Validation:**

Even sometimes when code does not error, data can be generated with unintended errors. That is why it is important to implement a validation (preferably independently run by a peer). I created a validation engine that peers use which can check the analytic file without creating all the abuse variables (the 901 variables above). Since the analytic file requires creating binary variable of the comparator variables, I can use PROC SQL “case then” statements to check the total of a variable without creating the variable.

For example, I can check the total of **opioids\_adf\_label\_abuse** with the following SQL logic:

SELECT SUM(CASE WHEN OP1F\_use=1 or OP1M\_use=1 or OP3I\_use=1 or OP8H\_use=1 or OP16G\_use=1 or OP8I\_use=1 or OP8J\_use=1 **THEN 1 ELSE 0 END)** as opioids\_adf\_label\_abuse\_total.

Therefore, I wrote a validation engine that writes Case/when statement strings to check the variable totals and see if those totals equal that of the analytic file. The code **sample\_validation\_check.sas** shows this engine and the output string files **qc\_queries.sas** and **qc\_queriesROA.sas** show the created strings that are later called via %Include.