PharmaSUG China 2024 - Paper CC-10011

# Metadata-Driven Programming Using R: A Case Study in BDS

Qian SU and Michael WANG, Sanofi

## ABSTRACT

In clinical trials, the creation of Analysis Data Models (ADaM) datasets demands considerable effort from SAS programmers. Despite the widespread use of modularized macro libraries and metadata repositories, which are designed to standardize common transformations and manage variations across studies through a parameter-driven approach, the challenge of repetitive tasks in file management and code generation seems intractable.

In recent years, Metadata-Mriven programming has emerged as a dynamic solution to address diverse needs. This approach greatly streamlines the rules among multiple studies through metadata construction and simplifies user inputs through metadata values thereby enabling execution automation. Furthermore, it facilitates a streamlined documentation process, adhering to a single-source-of-truth principle, which minimizes the need for repetitive revisions and inconsistencies.

A case study will be displayed to explore the application of a metadata-driven approach in Basic Data Structure (BDS) using R programming language, with implementation of {metacore} and {metatools} packages and customized programming- metadata to allow sufficient flexibility and automation. We still use “parameters” instead of “arguments” for convenience in the subsequent sections.

## INTRODUCTION

In clinical trials, a vital piece of the statistical programmers’ working landscape involves fine-tuning codes for generating Analysis Data Models (ADaM) datasets together with creating and maintaining study files, which include metadata (MD), algorithm notes and other supportive files, all in alignment with Study Analysis Plan (SAP) and company standards. In a traditional workflow shown in Figure 1, department-level macro libraries are used to process standard data algorithms and reduce custom programming at the study level for only some level of automation.

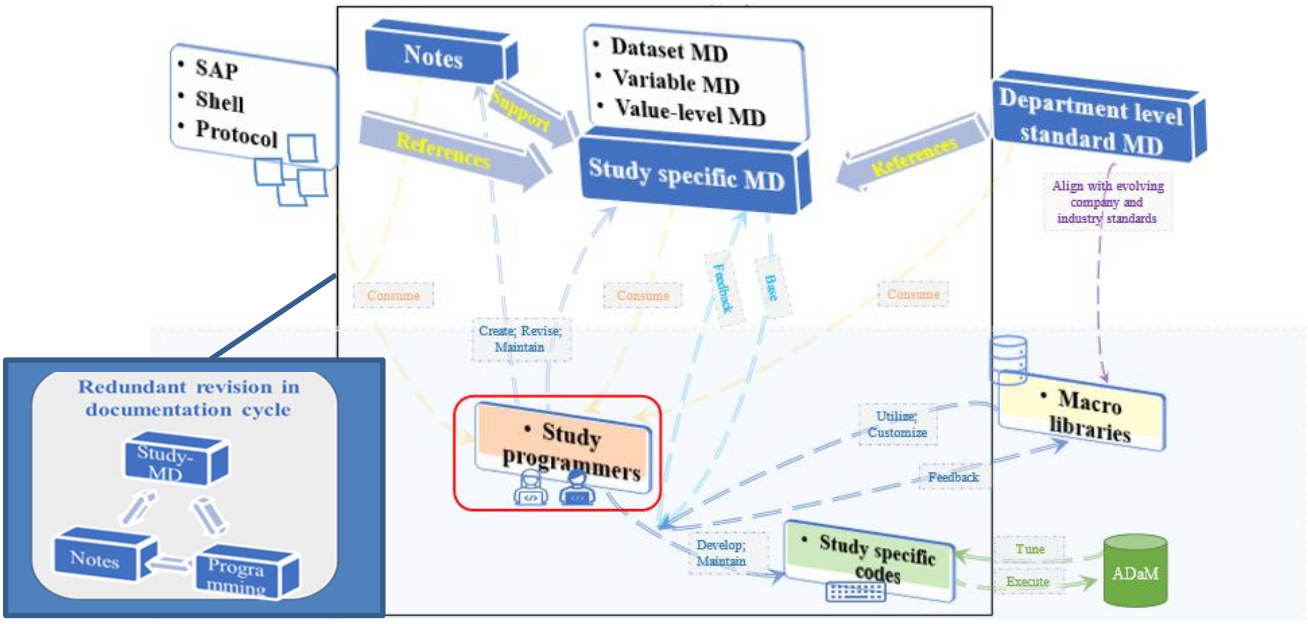


**Figure 1. The common workflow to generate ADaM datasets**

However, this traditional approach is encountering several limitations:

- Macros becomes less user-friendly or even impracticable when unforeseen discrepancies exceed their predefined operational scope.
- Constant modifications in programming and an expansion of macro libraries are anticipated to accommodate to evolving company and industry standards and diverse requirements.
- The current documentation cycle isolates study MD and algorithm notes from programming and all of them are handled in different environments in an asynchronous manner, resulting in redundant revisions in file-code cycle and potential inconsistencies across studies, particularly in specific therapeutic areas where standards are consistently high. This issue can also arise between files and codes within an individual study.

In fact, MD, algorithm notes and programming codes are so closely related that MD can be considered as the single-source-of-truth, energizing the others by interconversions. This approach is known as Metadata-Driven (MDD) design, enabling the dynamic and automatic generations in a timesaving and streamlined processes.

We will present a case study to demonstrate a MDD design to achieve high efficiency and execution automation in BDS based on a progressed metadata as the single-source-of-truth.

## PREREQUISITES

### CLARIFYING STUDY-METADATA STRUCTURE

- **Dataset level tab**: It houses comprehensive information about all ADaM datasets, such as dataset name, the primary keys, and additional dataset attributes.
- **Variable level tab**: It contains all essential information required for the creation and interpretation of variables in each ADaM dataset, such as variable name, attributes, derivation rules.

| Variable Origin | Where to retrieve/Source | How to |
|---|---|---|
| Predecessor | SDTM /other ADaMs | {metacore}/{metatools} |
| Assigned | Codelist in separate Controlled Terminology tab | {metacore}/{metatools} |
| Derived | Derivation Rules | Programming-Metadata, in a horizontal way |

**Table 1. Variable Origin and Source**

As shown in Table 1, variables to be retrieved for ADaM datasets are primarily categorized as 3 types based on their origin: Predecessor, Assigned and Derived variables.

1. **Predecessors**: These are often directly inherited from source datasets, such as the Standard Data Tabulation Model (SDTM) or other existing ADaM datasets, without any changes to their values or structure.
2. **Assigned Variables**: These variables are usually assigned values based on predefined criteria or code lists. A significant portion of these can be retrieved from code lists within a separate Controlled Terminology tab, which houses standardized code lists.
3. **Derived Variables**: The creation and processing of derived variables are typically the most time-consuming and critical aspects of ADaM generation. Derivation can involve simple transformations or more complex algorithms, and they often require programming logic to be

implemented.

- **Value level tab**: A special tab for BDS datasets that contains assigned values for parameters and derivation methods for parameter value dependent variables.

| Parameter/ Parameter Value | Where to retrieve | How to |
|---|---|---|
| PARAMCD;<br>AVAL/AVALC | Derivation rules | Programming-Metadata, in a horizontal / vertical way |

**Table 2. Parameters and parameter value**

The derivation of variables is typically the most time-consuming and critical aspect of creating ADaM datasets. The following section will focus on the methods and techniques used to manage the derivation of these variables within a metadata-driven framework.

## A FIT-FOR-PURPOSE PROGRAMMING-METADATA CONSTRUCTION

The key for derivation within metadata framework is a fit-for-purpose MD. The following section will delve into the construction of a progressed programming-MD used in R for derived variables. The initial step, where derivation rules in study MD are required to be re-described in human expressiveness using a predefined grammar, is paramount. Then the metadata file will be imported in R to be constructed into a progressed programming-MD which is readable by programming language, allowing for refinement into an executable format.

The following are some considerations for programming-MD construction:

- **Microsoft Excel worksheets**: The most common way is to construct metadata in one or more separate tabs. An Excel sheet with rows and columns can be easily accessed by programming and easy to maintain and review. All Excel worksheets along with complete information can be easily accessed and used with {metacore} and {metatools} packages.
- **Human expressiveness using a predefined grammar in Derivation Rules**: The column “Derivation” should be modified by using predefined grammar and can be explicitly categorized regarding purpose and process. Conventions of the column must be consistent in all rows. For example (see Table 3), all variable names should be in all-caps and in a “DATASET.VIRIABLE” format, using single quote ‘’ instead of double quotes “”.

The primary types of expressiveness with our experience are as following:

**Dataset/Variable selection**: DATASET.VARIABLE. This notation specifies the source of the data and the variable to be manipulated.

**Assignment**: Set to “value”. “value” represents the specific data point or constant to be assigned.

**Calculation**: Calculated as algorithm/DATASET.VARIABLE. Calculations are performed based on mathematical or logical operation or another variable.

**Conversion**: Date portion of DTC converted into date. The transformation of data types

**Condition**: Set to “value”/Calculated as algorithm/DATASET.VARIABLE when/where condition. The implementation of conditional logic, allowing for assignments or calculations to be contingent upon certain “condition(s)”.

**Control flow**: Set to “value”/Calculated as algorithm/DATASET.VARIABLE if condition; “value”/algorithm/DATASET.VARIABLE if condition; else “value”/algorithm/DATASET.VARIABLE. Control flow mechanisms, manage operations based on conditions.

**Processing**: The procedural steps involved in preparing and manipulating data.

- **Program-executable format**: As metadata eventually becomes part of program logic, “Derivation” column needs to be interpreted in an executable format and be reconstructed by extracting key

components for program logic and execution. For example (see Table 4), “rules” column is a list that contains multiple components, including datasets and variables used in logic, processing component, condition-value mapping, “cond” column in rules can be a valid code segment to be parsed, like “DIAUSC=='Y' | DIAHOSP=='Y' | DIAEMR=='Y'”.

- **Variables and datasets dependencies**: The order of variables can be used to control the dependencies, which requires executions done in a sequential process. Additional dependencies check should be added in the execution cycle by fully implementing the whole metadata information.
- **Automate formats assignment**: Column Where ID and Value Label in Table 5 can be directly used to assign formats instead of creating through look-up tables (similar with PROC FORMAT in SAS).

### Programming metadata

***Scenario 1 - Derive new variables in a horizontal way.***

| Dataset Name | Relative Order | Variable Name | Variable Label | Type | Derivation | Modified Derivation |
|---|---|---|---|---|---|---|
| ADD | 1032 | ADURN | Duration (N) | float | AENDT-ASTDT+1 | Calculated as AENDT-ASTDT+1 |
| ADD | 1033 | ADURU | Duration Units | text | DAYS | Set to 'DAYS' |
| ADD | 1040 | DIAEOUT | Outcome | text | CE.CEOUT, merged by USUBJID and CEREFID | Calculated as CE.CEOUT; merged by USUBJID and CE.CEREFID |
| ADD | 1041 | DIAUSC | Use of XX | text | FAAE.FAORRES when FAOBJ = 'USE OF XX' | Calculated as FAAE.FAORRES when FAAE.FAOBJ = 'USE OF XX' |
| ADD | 1044 | EXACERB | Severe Exacerbation event | text | Set to 'Y' if DIAUSC='Y' or DIAHOSP='Y' or DIAEMR='Y', otherwise null | Set to 'Y' if DIAUSC='Y' or DIAHOSP='Y' or DIAEMR='Y'; else null |
| ADD | 1052 | ERONLY | Emergency Room Only | text | Set to 'Y' when DIADEMR >0 and (DIADGMU <=0 and DIADSMU <=0 and DIADICU <=0 and DIADSIU <=0 and DIADOHU <=0) | Set to 'Y' when DIADEMR >0 and (DIADGMU <=0 and DIADSMU <=0 and DIADICU <=0 and DIADSIU <=0 and DIADOHU <=0) |
| ADD | 1070 | ONTRTFL | On Treatment Record Flag | text | Set to 'Y' if ASTDT >= min(ADPSL.RFSTDT, ADPSL.ENROLDT) and ASTDT <= ADPSL.RFENDT, null otherwise | Set to 'Y' if ASTDT >= min(ADPSL.RFSTDT, ADPSL.ENROLDT) and ASTDT <= ADPSL.RFENDT; else null |

**Study-MD**

**(Variable-level tab)**

**Table 3. Modified derivations using predefined grammar (partial)**

Programming-MD derives from variable level tab and derivation rules are reconstructed into computer interpretable formats and program-executable expressions are values in cells. New variable derivations and values setting can be completed automatically once the metadata is read in as input along with essential datasets.

| dataset | variable | derivation_id | derivation | type | rules | cond | value |
|---|---|---|---|---|---|---|---|
| ADD | ADURN | ADD.ADURN | Calculated as AENDT-ASTDT+1 | Calculation | list( ) | | AENDT-ASTDT+1 |
| ADD | EXACERB | ADD.EXACERB | Set to 'Y' if DIAUSC=='Y' or DIAHOSP='Y' or DIAEMR = 'Y', else null | Control flow | list( ) | DIAUSC=='Y' \| DIAHOSP=='Y' \| DIAEMR=='Y' | 'Y' |
| | | | | | | TRUE | NA |
| ADD | DIADSIU | ADD.DIADSIU | Calculated as numeric format of FAAE.FAORRES when FAAE.FAOBJ="DAYS OF SURGICAL/INVASIVE UNIT" | Condition | list( ) | FAOBJ == 'DAYS OF SURGICAL/INVASIVE UNIT' | FAORRES |
| ADD | ADURU | ADD.ADURU | Set to 'DAYS' | Assignment | list( ) | | 'DAYS' |

**Programming-MD**

**(Key components in a structured format)**

**Table 4. Computer readable and program-executable format derivation constructure (partial)**

### *Scenario 2 - Derive PARAMCD in a vertical way.*

Programming MD derives from value level tab and algorithm notes. In this case, variables and values cannot be directly retrieved from Finding Domains in SDTM. Although structure of value-level MD for parameterized variables is different from variable level, derivation can be reconstructed using the same method in vertical way.

| Dataset Name | Where ID | Where Variable | Check Value | Value Label | Modified Derivation | condition level | cond | value |
|---|---|---|---|---|---|---|---|---|
| ADD | 1 | PARAMCD | EXACERB | Asthma Exacerbation Event | Set to 'Y' when DIAHOSP = 'Y' | 1 | DIAHOSP == 'Y' | 'Y' |
| ADD | 2 | PARAMCD | LOAC | Loss of Asthma Control | Set to 'Y' when DIAPUF = 'Y' or DIADFEV = 'Y' or DIAICSI = 'Y' or DIASCSIL = 'Y' or DIADPEF = 'Y' or DIAUSCS = 'Y' or DIAHOSP = 'Y' | 1 | DIAPUF == 'Y' \| DIADFEV == "Y" \| DIAICSI == "Y" \| DIASCSIL == "Y" \| DIADPEF == "Y" \| DIAUSCS == "Y" \| DIAHOSP == "Y" | 'Y' |
| ADD | 3 | PARAMCD | HOSPORER | Hospitalization or Emergency Room Visit Require Corticosteroids | Set to 'Y' when DIASEV = 'Y' | 1 | DIASEV == "Y" | 'Y' |

**Study-MD**

**(Value-level tab)**

**Programming-MD**

**(Key components in a structured format)**

**Table 5. Programming-MD for PARAMCD derivation in a vertical way (partial)**

## PROGRAMMING FLOW

Once Programming-MD prepared, the MD can be utilized in a following flow.

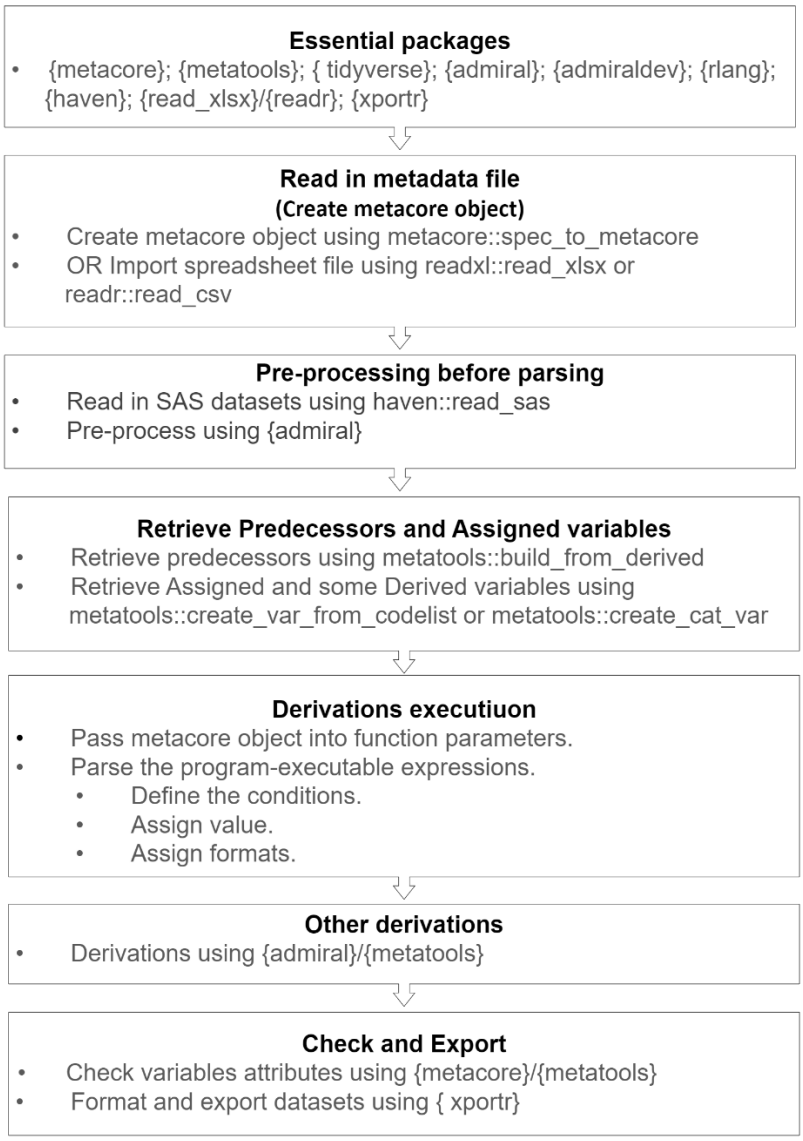


**Figure 2. Programming flow in MD-driven method**

A simplified workflow can be obtained through Metadata-Driven programming (Figure 3), and MDD design is more useful and possible to:

- Streamline and simplify the file management within individual studies or across multiple studies.
- Allow users to flexibly control predictable differences between studies through metadata.
- Synchronizes the programming process with the documentation flow, ensuring alignment and reducing discrepancies.
- Enhance the standardization of programming by integrating a wide range of study-specific inputs into the programming framework.
- Avoid constant modifications in MD file-programming code cycle.
- Ensure that the logic behind the variable derivations is transparent and can be easily audited or modified as needed.

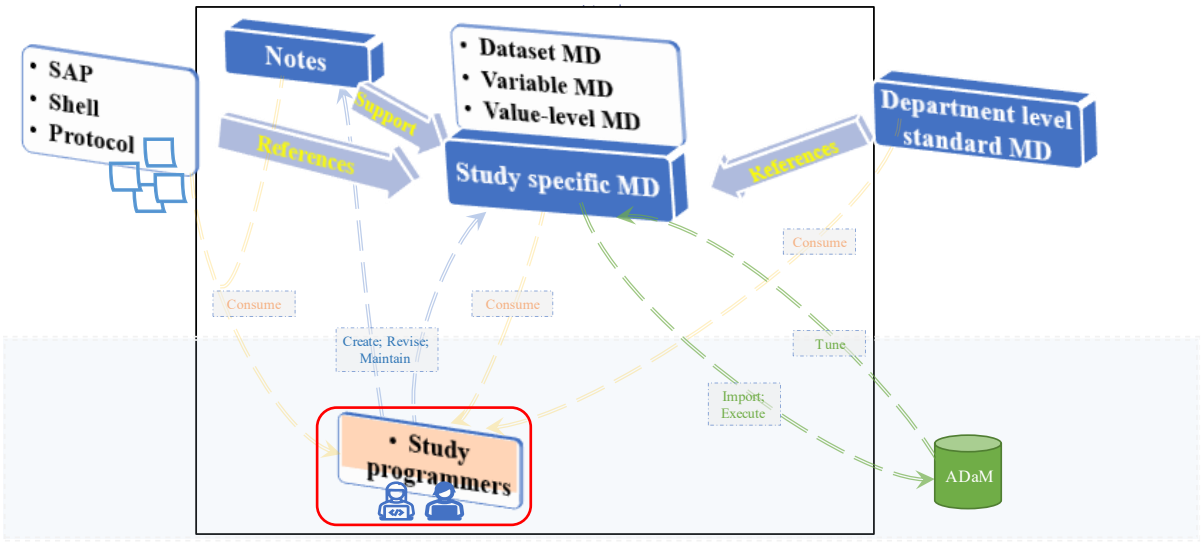


**Figure 3. A simplified workflow to generate ADaMs with MDD design**

## CONCLUSION

Metadata-Driven Programming method in R can serve as a dynamic solution to chase high efficiency and consistency in programming. A simpler workflow can make our daily work clearer without redundant works.

As we are focusing on variable derivations along with values setting, pre-processing of input data is not deeply discussed. Not all data processing and variable derivation are good candidates for MD design in the current framework. A useful application of the technique is for candidates where data set-specific algorithms are specified explicitly.

How to organize human expressiveness in derivations and how to convert them into programing languages and readable structure are issues with extensive demand that require further explorations and more innovative technological support. What's more, interconversions in documentation flow and larger scope of programming-MD can be explored in future.

## REFERENCES

metatools: Enable the Use of 'metacore' to Help Create and Check Dataset (r-project.org)

metacore: A Centralized Metadata Object Focus on Clinical Trial Data Programming Workflows (r-project.org)

Bo, Yu and Chen, Zhiheng and Zhao, Kang. 2023. "Improving CDISC Basic Data Structure Requirements on PARAMCD for ADaM Automation." PharmaSUG China

Kang Zhao. 2023. "Domain Specific Language for ADaM Dataset Generation in Regulatory Clinical Research". PharmaSUG China

Ellen Lin, Aditya Tella, Yeshashwini Chenna, and Michiel Hagendoorn. 2020. "Metadata-driven Modular Macro Design for SDTM and ADaM". PharmaSUG

## ACKNOWLEGEMENTS

The authors would like to thank Sanofi and my team for facilitating and supporting this paper.

## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Qian SU, qian.su@sanofi.com
Michael WANG, michael3.wang@sanofi.com



## APPENDIXIES

### Read in metadata file and create metacore object (partial)

```
# read in metadata file
doc <- metacore::read_all_sheets(paste0(MISC,"/MD4TEST.xlsx"))

# spec_to_metacore(paste0(MISC,"/MD4TEST.xlsx"))

# create ds_spec
ds_spec <- spec_type_to_ds_spec(
  doc,
  cols = c("dataset" = "[N|n]ame|[D|d]ataset [N|n]ame|[D|d]omain [N|n]ame",
           "structure" = "[D|d]ataset [S|s]tructure",
           "label" = "[[D|d]ataset L|l]abel|[D|d]ataset [D|d]escription"),
   sheet = "[D|d]ataset|[D|d]omain"
)

# create ds_vars
ds_vars <- spec_type_to_ds_vars(
  doc,
  cols = c("dataset" = "[D|d]ataset [N|n]ame|[D|d]omain [N|n]ame",
         "variable" = "[V|v]ariable [N|n]ame",
         "order" = "[R|r]elative [O|o]rder",
  key_seq_cols = c("dataset" = "[D|d]ataset [N|n]am|[D|d]omain [N|n]ame",
                   "key_seq" = "[K|k]ey"),
   sheet = "[V|v]ariable Metadata|[D|d]ataset|Domains"
)

# create var_spec
var_spec <- spec_type_to_var_spec(
  doc,
  cols = c("variable" = "[V|v]ariable [N|n]ame",
           "length" = "[S|s]tandard [L|l]ength",
           "label" = "[V|v]ariable [L|l]abel",
           "dataset" = "[D|d]ataset [N|n]ame|[D|d]omain [N|n]ame",
           "format" = "[D|d]ispaly |[F|f]ormat",
           "common" = "[C|c]ommon"),
 sheet = "[V|v]ariable Metadata"
)

# create value_spec
value_spec <- spec_type_to_value_spec(
  doc,
  cols = c("dataset" = "[D|d]ataset [N|n]ame|[D|d]omain [N|n]ame",
           "variable" = "[V|v]ariable [N|n]ame",
           "code_id" = "[C|c]ontrolled",
           "where" = "[S|s]ource", # "[A|a]ssigned"
          "predecessor" = "[S|s]ource"),
  sheet = "[V|v]ariable Metadata",
  where_sep_sheet = FALSE,
   var_sheet = "[V|v]ariable Metadata")
```

```
# create codelist
codelist <- spec_type_to_codelist(
  doc,
  codelist_cols = c("code_id" = "[C|c]ontrolled",
                    "name" = "[C|c]ontrolled",
                    "code" = "Coded Value",
                    "decode" = "Decoded Value"),
   sheets = "[C|c]ontrolled",
   simplify = FALSE,
   dict_cols = NULL)

# make the metacore object
metacore <- quietly(metacore)(ds_spec, ds_vars, var_spec, value_spec,
derivation_var_spec, codelist)$result
```

**Retrieve Predecessors and Assigned variables based on metacore object**

```
### read in dataset

### ADSL
### retrieve Predecessor through `derivation`
spec <- metacore %>% select_dataset("ADSL")
dv <- spec$derivations %>%
        filter(str_detect(derivation_id, "pred."))
ds_list <- list(DM = dm, ADPSL = adpsl, SUPPDM = dm))
adsl_pred <- build_from_derived(spec, ds_list, predecessor_only = FALSE,
keep=TRUE)

### retrieve assigned/derived variables through code list
### like `input` `put` in SAS
adsl_assign <- adsl_pred %>%
create_cat_var(metacore = spec, AGE, AGEGR1, AGEGR1N)
# adsl_assign <- adsl_pred %>% create_cat_var(metacore = spec, BMIBL,
BMIGR1, BMIGR1N)

get_control_term(spec, TRT01PN)$decode[1:3] == unique(adsl_pred$TRT01P)[2:4]
adsl_assign <- adsl_assign %>% create_var_from_codelist(metacore = spec,
input_var = TRT01P, out_var = TRT01PN, decode_to_code = TRUE) %>%
create_var_from_codelist(metacore = spec, input_var = TRT01A, out_var =
TRT01AN, decode_to_code = TRUE)
# create_var_from_codelist(metacore = spec, input_var = DCSREAS, out_var =
DCSREASN, decode_to_code = TRUE)
# create_var_from_codelist(metacore = spec, input_var = DCTREAS, out_var =
DCTREASN, decode_to_code = TRUE)

### BDS
spec <- metacore %>% select_dataset("BDS")
dv <- spec$derivations

### retrieve Predecessor through `derivation`
ds_list <- list(DM = dm), ADPSL = adpsl, SUPPDM = dm, FAAE = faae)
bds_pred <- build_from_derived(spec, ds_list, predecessor_only = FALSE,
keep=TRUE)
```

**Scenario 1: New variables generation in a horizontal way (partial)**

```
### retrieve derived variables using programming metadata
### New variables
derive_var_horizontal <- function(dsin = dsin,
   ds_list = list(ADPSL = adpsl, CE = ce, FAAE= faae),
   metacore = spec,
   cols = c("dataset" = "[D|d]ataset [N|n]ame|[D|d]omain [N|n]ame",
            "variable" = "[V|v]ariable [N|n]ame",
            "variable_type" = "[T|t]ype",
            "derivation_id" = "[V|v]ariable [N|n]ame",
            "derivation" = "[D|d]erivation"),
   recid = exprs(USUBJID, FAREFID, FASEQ, FASPID)
                                  ) {
        assert_vars(recid)
        # Create derivation reference table
        ref <- metacore$derivations %>%
                filter(!is.na(derivation) & str detect(derivation id, "pred.") ==
FALSE)
        # Check needed variable existence, displayed in format of `dataset.variable`
        # Retrieve data set and variables from derivations
        dat var <- unique(unlist(str extract all(ref$derivation, "[A-Za-z]+\\.[A-Za-
z]+")))
        dat <- unique(str_extract(dat_var, "^\\w*(?=\\.)")) # data set name
        var <- unique(str_extract(dat_var, "(?<=\\.).*")) # variable name
        # Check if all needed variables exist in input data set
        # Extract key information
        # Convert human expressiveness in derivation rules into executable format
        list of rules <<- map2 df(condition$condition, condition$variable, ~
extract_rules(.x, .y)) %>%
                right_join(condition, by = c("variable", "condition"))
        # Get new variables list that will be processed in `Mutate`
        list_of_vars <<- list_of_rules %>%
                distinct(variable, type)
     # Do Loop: Key portion of the function
     for (i in seq_along(list_of_vars$variable)) {
     # Get sub-list of rules from metadata for the certain new variable
     list_of_sub_rules <<- tibble((list_of_rules %>%
               dplyr::filter(variable == list_of_vars$variable[i]) %>%
               select(rules) %>%
                 reduce(bind_rows))[[1]])
       if (nrow(list_of_sub_rules) >0) {
             if (str detect(str to lower(type), "[A|a]ssignment") == FALSE &
str_detect(str_to_lower(type), "[C|c]alculation") == FALSE) {
               if (str_detect(str_to_lower(type), "[C|c]ontrol") == TRUE) {
                 parse_this_rule <- list_of_sub_rules %>%
                                   str_glue_data("{miss} {cond} ~ {value};") %>%
                                   str_flatten()
                    exprs <- parse_exprs(parse_this_rule)
                       # Parse the rule in `case_when`
```

```
                        be_parsed <- to_be_input %>%
                            rowwise() %>%
                            dplyr::mutate(
                                  "{new_var}" := case_when(
                                                !!!exprs)
                                        ) %>%
                                  select(!!!recid, !!new_var) %>%
                                  filter(!is.na(.data[[new_var]])) %>%
                                  distinct()

                } else if (str_detect(str_to_lower(type), "[C|c]ondition") == TRUE) {
                    parse_this_rule <- list_of_sub_rules$cond
                    parse_this_value <- list_of_sub_rules$value
                    exprs_cond <- parse_exprs(parse_this_rule)
                    exprs_value <- parse_exprs(parse_this_value)
                        # Parse the rule in `if_else`
                    be_parsed <- to_be_input %>%
                            rowwise() %>%
                            dplyr::mutate(
                            "{new_var}" := ifelse(!!!exprs_cond, !!!exprs_value,
NA)) %>%
                            select(!!!recid, !!new_var) %>%
                            filter(!is.na(.data[[new_var]])) %>%
                            distinct()

                                                }

                    } else {
                       parse_this_rule <- list_of_sub_rules$value
                       exprs <- parse_exprs(parse_this_rule)
                       be_parsed <- to_be_input %>%
                          rowwise() %>%
                              mutate(
                              "{new_var}" := !!!exprs
                                    )
                        new_name <- names(chk %>% select(last_col()))
                        names(new_name) <-  new_var
                        be_parsed <- be_parsed %>%
                              rename(any_of(new_name)) %>%
                              select(!!!recid, !!new_var) %>%
                              filter(!is.na(.data[[new_var]])) %>%
                              distinct()
                          }
            to_be_input <- to_be_input %>% left_join(be_parsed, by =
vars2chr(recid))
              } else {cat(green("No rules selected to be parsed.\n\n\n"))}
        }
         return(to_be_input)
}
```

**Scenario 2: PARAMCD derivation in a vertical way**

```
### BDS: parameters and parameter value
derive_vars_bds <- function(dsin = dsin,
   doc  = doc,
   cols = c("dataset" = "[D|d]ataset [N|n]ame|[D|d]omain [N|n]ame",
           "analysis_val" = "[V|v]ariable [N|n]ame",
           "variable_type" = "[T|t]ype",
           "derivation_id" = "Check Value",
           "derivation" = "[D|d]erivation",
           "label" = "[V|v]alue [L|l]abel",
           "order" = "[W|w]here [ID|id]"),
   recid = exprs(USUBJID, FAREFID, FASEQ, FASPID)) {
   assert_vars(recid)
   # Create derivation reference table
   ref <- create_tbl(doc, cols) %>%
     mutate(paramcd = derivation_id, derivation_id = paste0(dataset, ".",
derivation_id)) %>%
     filter(!is.na(dataset) & !is.na(analysis_val) & dataset == adam_dataset)
   # Convert human expressiveness in derivation rules into executable format
   list_of_rules <<- map2_df(ref$derivation, ref$paramcd, ~
extract_rules(.x, .y)) %>%
     right_join(ref, by = c("variable" = "paramcd", "condition" =
"derivation"))
   # Get new variables list that will be processed in `Mutate`
   list_of_vars <<- list_of_rules %>%
     distinct(analysis_val, variable, variable_type, type, label, order)
       # Do Loop: Key portion of the function
        for (i in seq_along(list_of_vars$variable)) {
           analy_var <- list_of_vars$analysis_val[i]
           paramcd <- list_of_vars$variable[i]
           type <<- list_of_vars$type[i]
           var_type <<- list_of_vars$variable_type[i]
           # Get sub-list of rules from metadata for the certain new variable
           list_of_sub_rules <<- tibble((list_of_rules %>%
             dplyr::filter(variable == paramcd) %>%
             select(rules) %>%
             reduce(bind_rows))[[1]])
           if (str_detect(str_to_lower(type), "[C|c]ontrol") == TRUE) {
```

```
                parse_this_rule <- list_of_sub_rules %>%
                  str_glue_data("{cond} ~ {value};") %>%
                  str_flatten()
                exprs <- parse_exprs(parse_this_rule)
                # Parse the rule in `case_when`
                be_parsed <- to_be_input %>%
                  rowwise() %>%
                    dplyr::mutate("{analy_var}" := case_when(!!!exprs),
                      PARAMCD := if_else(!is.na(!!sym(analy_var)),
                          list_of_vars$variable[i], NA_character_),
                      PARAM = if_else(!is.na(!!sym(analy_var)),
                          list_of_vars$label[i], NA_character_),
                      PARAMN = if_else(!is.na(!!sym(analy_var)),
                          list_of_vars$order[i], NA_integer_)) %>%
                    filter(!is.na(!!sym(analy_var)))
                 } else if (str_detect(str_to_lower(type), "[C|c]ondition") ==
TRUE) {
                parse_this_rule <- list_of_sub_rules$cond
                parse_this_value <- list_of_sub_rules$value
                exprs_cond <- parse_exprs(parse_this_rule)
                exprs_value <- parse_exprs(parse_this_value)
                be_parsed <- to_be_input %>%
                  mutate("{analy_var}" :=
                        ifelse(!!!exprs_cond, !!!exprs_value, NA),
                    PARAMCD := if_else(!is.na(!!sym(analy_var)),
                          list_of_vars$variable[i], NA_character_),
                    PARAM = if_else(!is.na(!!sym(analy_var)),
                          list_of_vars$label[i], NA_character_),
                    PARAMN = if_else(!is.na(!!sym(analy_var)),
                        as.numeric(list_of_vars$order[i]), NA_integer_)) %>%
                  filter(!is.na(!!sym(analy_var)))
                 }
        # Append parsed data
        if(i==1){
           out_data <- be_parsed
            }else{
            out_data <- bind_rows(out_data, be_parsed) }
         }
         return(out_data)
}
```