SCSUG 2017

From Readability to Responsible Risk Management: Facilitating the Automatic Identification and Aggregation of Software Technical Debt within an Organization Through Standardized Commenting in SAS® Program Files and SAS Enterprise Guide Project Files

Troy Martin Hughes

ABSTRACT

Software readability is greatly improved when programs include descriptive comments in a predictable, standardized format. Program headers that describe software requirements, author, creation date, versioning history, caveats, and other metadata are a common method to facilitate a greater understanding of software objectives, strengths, weaknesses, and prerequisites. Moreover, when program headers are standardized, they are not only more readable to developers but also to parsing algorithms that can automatically extract metadata for analysis or archival. Comments throughout software can also improve its readability and, when constructed in a standardized format, can be parsed automatically and saved in control tables. This text introduces a standardized commenting methodology that enables both qualitative and quantitative comments to be parsed from SAS® software headers and body. A configuration file defines comment formatting and content and provides a flexible, scalable, reusable, data-driven SAS macro-based solution. This text demonstrates one use case for this methodology in which software technical debt and risk are assessed via both qualitative (e.g., risk description, proposed risk resolution) and quantitative (e.g., risk severity, risk probability, likelihood of risk discovery, ease of risk mitigation) metadata and metrics included within SAS comments. The comment interpreter dynamically identifies and parses all SAS program files and SAS Enterprise Guide project files (including imbedded SAS programs therein) within one or more folders to produce a comprehensive, quantifiable risk register. This data-driven documentation, generated with push-button simplicity, enables SAS practitioners to better understand and make decisions about technical debt and risk, including at the program, project, developer, team, and organizational levels.

INTRODUCTION

Software readability is defined as "the ease with which a system's source code can be read and understood, especially at the detailed, statement level."[i] Comments improve software readability by describing program-specific metadata (e.g., program intent, author, creation date, versioning), by describing SAS macros and their respective parameters, by describing complex software actions, and by describing limitations or risk within software, each of which increases a developer's ability to understand, use, reuse, repurpose, and maintain software. To the extent that software comments can be included, and their format standardized by a team or organization, software readability is improved because SAS practitioners can be assured of consistent, complete metadata and descriptive information.

A secondary benefit of clear, consistent, complete commenting is the ease with which not only SAS practitioners but also automated parsing algorithms can extract metadata from SAS programs. For example, SAS literature is replete with solutions that demonstrate automatically extracting software metadata from program headers once standardized headers have been implemented. A SAS program—an *interpreter*—is used to parse other SAS programs to extract and aggregate metadata from comments and produces a resultant data set or report. Interpreters represent a reusable documentation methodology that improves readability while reinforcing standardization of program commenting; their use, however, has often been unnecessarily limited in functionality and scope.

One limitation in SAS literature has been the ability of interpreter programs to parse only SAS program files (which are text files) while excluding Enterprise Guide project files (which are zipped files). A second limitation has been the focus on extracting only program header comments while excluding program body comments. A third limitation has been the inclusion of business rules *inside* the interpreter—limiting its flexibility and reuse—rather than deriving logic from an *external* control file that can be modified independently. A fourth limitation has been the extraction of only qualitative—not quantitative—comments and metadata, with the rare exception of creation/modification dates found in program headers. The author overcomes the first three limitations in his 2016 text: *Your Local Fire Engine Has an Apparatus Inventory Sheet and So Should Your Software: Automatically Generating Software Use and Reuse Libraries and*

Catalogs from Standardized SAS® Code.[ii] The current text additionally overcomes the fourth limitation by empowering SAS practitioners to define both character and numeric metadata that can be extracted from code.

This text reintroduces and refactors several macros from the author's previously referenced text. The PARSEDIR macro iteratively parses one or more folders (and optionally its subdirectories) to create a list of all SAS program files and SAS Enterprise Guide project files contained therein. The READXML macro unzips project files and extracts all linked and imbedded SAS program files therein. The READCODE macro parses and interprets the SAS program files utilizing business rules contained in a program-specific configuration file (that is parsed with the READCONFIG macro). Thus, with only subtle modifications to the code base, macros originally created to produce a reuse catalog have been extensibly refactored to produce a risk register that qualitatively and quantitatively describes individual and cumulative technical debt and risk for SAS programs within a team or organization.

In standardizing how software risk information is assessed and documented within an organization, this bottom-up approach ensures that developers can quickly document elements of risk within software to describe actual or potential functional or performance deficiencies. For example, a SAS developer might assess that a given data transformation process poses risk because it may not scale efficiently as the data volume increases. To document this risk, the developer could create a standardized comment that denotes that risk severity is high (10) yet risk probability is low (1) because data volume is not expected to increase to unmanageable levels for several months. With similar risk-related comments interspersed throughout all programs within an environment, an organization can gain a quantifiable understanding of its risk and technical debt and, moreover, can assess the quality of its software and prioritize preventative, corrective, and adaptive maintenance necessary to ensure reliable software operation.

TECHNICAL DEBT

Technical debt is found wherever a gap exists between software functional requirements and actual functionality, between software performance requirements and actual performance, or between software project management requirements and actual project implementation. When development work (including all phases of the software development life cycle, or SDLC) remains to be completed (that already *should* have been completed), technical debt has been incurred. Within Agile[1] development environments, technical debt typically does not denote planned work that has not yet been prioritized or scheduled into future iterations—because that work is not yet expected to have been finished. Technical debt is commonly incurred when software meets functional but not performance requirements and when developers (or other stakeholders) subsequently decide to pursue completing additional functional requirements rather than remedying the incomplete performance requirements; the software may function—but it lacks quality.

In some cases, technical debt can be incurred when phases of the SDLC are underrepresented or missing. For example, if a developer writes SAS software and immediately releases it into a production environment with insufficient testing or validation, the software might contain latent defects that degrade its quality. When discovered, while previously unknown to stakeholders, these defects increment technical debt because they must be resolved. Thus, when developers engage in risky behavior such as failing to test or document software, or to follow other software development best practices, technical debt can be *implied* (even before it is observed) because later work will be required to ensure or validate that necessary software quality has been achieved. Technical debt can also be incurred when software requirements are unclear, ambiguous, misconstrued, or otherwise not followed. In addition to software maintenance or modification, elimination of technical debt might also require updating software requirements, documentation, or other software product artifacts.

To provide one example, a team of SAS practitioners is tasked with building an analytic program that transforms data to produce a recurring report. The program is intended to be run hourly so the service level agreement (SLA) states that it must complete in less than 45 minutes. Moreover, the customer has requested that a very specific color scheme be implemented in the report to match the company's new marketing directive. During software testing, it's revealed that 1) the program takes 65 minutes to execute and 2) the developers haven't fully implemented the custom color scheme. However, because the report is (believed to be) numerically accurate and denotes significant business value,

[1] For an introduction to Agile methodologies, please consult the author's text: *When Software Development is Your Means Not Your End: Abstracting Agile Methodologies for End-User Development and Analytic Application*, available at https://www.lexjansen.com/wuss/2014/70_Final_Paper_PDF.pdf.

the customer nevertheless requests that the software be put into production on its scheduled release date and fixed later. Thus, technical debt has been incurred because undone work exists that should have been completed.

Once the software is in production, however, developers realize that malformed data can occasionally cause the software to produce invalid reports with spurious results. A root cause analysis (RCA) reveals that insufficient software requirements documentation facilitated the failure because not all data quality controls had been specified. These latent defects—the insufficient requirements (i.e., inaccurate documentation) and missing quality controls (i.e., inaccurate code)—each represent technical debt because they must be resolved to align the software with its requirements (and intended quality). At this point, stakeholders must prioritize what work should be completed next. Should developers update the quality controls to improve the accuracy of the reports, thereby improving functionality? Should they improve the color scheme or execution time? Or should they choose to modify the requirements documentation, thereby reducing expectations of the software? Of course, all these options could be ignored in favor of calling the software "done" and moving on to other priorities that provide greater business value than software modification or maintenance. Thus, especially in Agile environments in which rapid development is espoused through incremental, iterative software releases, as technical debt is identified throughout the SDLC, it must be prioritized against further development work; developers often must effectively choose between completing old work (i.e., eliminating technical debt) or beginning new work prioritized for future software releases.

But to understand opportunity costs and responsibly decide whether and which technical debt to resolve, not only technical *debt* but also its *interest* must be considered. Just as most financial debt incurs interest, technical debt too can incur technical interest. While technical debt typically describes undone work that should be completed (only once), technical interest describes the additional (and often recurring) costs associated with operating poor quality software. In the previous example, technical debt exists in part due to incomplete software requirements and incomplete data quality controls, and until this debt is resolved, interest is incurred each time an invalid report is created. This interest might include the maintenance time required to stop and restart the offending program, a period of unavailability of reports for key stakeholders, stopgap maintenance solutions required to remove the invalid data manually, or emails or other communiques that attempt to mitigate the cost of the failed reports.

It's more difficult to quantify technical interest than technical debt, in part because *interest* is often accrued during emergency maintenance and other operationally chaotic periods, whereas *debt* often describes development work defined within software requirements. Some technical interest cannot be quantified, such as the business value lost when developers ignore the color scheme carefully crafted by a marketing department and instead release calico-colored reports that resemble Hotdog on a Stick signage. However, even incalculable interest contravenes the development and maintenance activities that aim to eliminate technical debt. In a worst-case scenario, the development of poor quality software can lead to spiraling distress in which SAS practitioners have no time to improve software proactively (through technical debt reduction) because they are reactively fighting fires and investing all resources in suppressing technical interest. This failure is akin to a country's national debt that rises to the point at which only interest—but not principal—can be paid. And just as insurmountable financial debt can lead to a national default and crisis, insurmountable technical debt can harbinger the failure of a software project or product.

A common method to quantify and compare technical debt is to understand that all debt incurs risk and to measure and evaluate that risk. The risk of slow motion software could be a reduction in its popularity and violation of SLAs. The risk of an incorrect color scheme could be an enraged marketing department and CEO. Yet these risks might pale in comparison to the risk of spurious reports having invalid data (caused by a lack of data quality controls) that could annihilate customer confidence. Each of these scenarios, however, describes only the severity of risk while ignoring other dimensions such as the probability of the risk occurring, how easily the risk could be detected, or the level of effort required to mitigate the risk. Risk assessments are made richer by incorporating these multidimensional aspects of risk, especially when those dimensions can be quantified to facilitate better data-driven decision making.

In some cases, all technical debt must be resolved prior to software release—whether that is an incremental Agile release or ultimate Waterfall release—so the only decision is what debt to fix first. In other cases, however, software is released with known technical debt with the intent that the software will be upgraded in the future to eliminate all or some of this debt. Responsible risk management requires that risks associated with existing debt at least be documented and evaluated, even if the decision is made not to eliminate the debt. Risk registers, introduced in the next section, capture multidimensional aspects of risk and provide a proven method to document, evaluate, aggregate, and

quantify risks and to prioritize their respective resolution. Risk, risk registers, and risk management are discussed extensively in the author's text: *SAS Data Analytic Development: Dimensions of Software Quality*.[iii]

RISK MANAGEMENT

While most technical debt does denote risk, not all risk denotes technical debt—that is, risk often exists because of software or environmental factors that lie outside the scope of software requirements or remediation. For example, if SAS software is intended to be run only within a Windows environment, it might be worthwhile to document that the software has been designed and tested only for Windows and that it might not port successfully to UNIX. However, this risk would not constitute technical debt if software requirements neither specify nor imply operating system (OS) portability to UNIX. Thus, risk-related comments within software (or external documentation) may convey development work that should be completed (i.e., technical debt) but also can describe additional conditions, caveats, limitations, threats, or vulnerabilities that cannot or should not be corrected or mitigated because of the nature or intent of the software.

Risk-related commenting is often sparse within software, as demonstrated by the following PARSE macro:

DO NOT USE—FOR DEMONSTRATION ONLY

```
* written by Troy Martin Hughes, SCSUG 2017;
* this macro parses multiple folders in search of .SAS, .EGP, or other files;
* and creates a data set that contains the folder, file name, and create date;
%macro parse(dir= /* asterisk-delimited list of folders to parse */,
      dsn= /* data set in LIB.DSN format into which to put directory info */,
      subdir=YES /* YES to parse subdirectories, NO for current DIR only */,
      filetype=SAS EGP /* space-delimited list of file extensions */);
%if %upcase(&subdir)=YES %then %let subdirs=/s;
%else %let subdirs=;
filename filelist pipe "dir &subdirs &dir";
data &dsn (keep=dir_name file_name crdate);
   length dir_name $200 file_name $200 crdate 8 inp $400;
      format crdate datetime17.;
      infile filelist truncover;
      input inp $400.;
      if length(inp)>9 and substr(strip(inp),1,9)='Directory'
         then dir_name=strip(substr(strip(inp),13));
      else if lengthc(strip(inp))>0 and find(inp,'<DIR>')=0 and
         not(find(inp,'Volume in drive')>0 and find(inp,'has no label')>0)
         and not(find(inp,'File(s)')>0 and find(inp,'bytes')>0) and
         not(find(inp,'Dir(s)')>0 and find(inp,'bytes')>0) then do;
            if lowcase(scan(inp,-1,'.')) in('sas','egp') then do;
                  crdate=input(substr(inp,1,20),mdyampm20.);
                  file_name=strip(lowcase(substr(inp,39)));
                  output;
                  end;
            end;
      retain dir_name;
run;
%mend;

%parse(dir=d:\sas, dsn=myfiles, subdir=YES, filetype=SAS EGP);
```

If the first two comment lines can be construed to represent the macro intent (i.e., the software requirements), then the macro has apparent deficiencies (i.e., risks) that should at least be documented, if not also remediated:

1. PARSE was intended to parse one or multiple folders, but as coded, accepts only a single folder into the DIR parameter, so the macro doesn't deliver the functionality stated within the software requirements. In fact, the macro will fail if an asterisk-delimited DIR parameter is specified.

2. The FILETYPE parameter is also functionless, as the “in('sas','egp')” logic hardcodes these two file types. Perhaps the developer *intended* to add the stated flexibility and scalability (but was interrupted), but as written, file names are only added to the Myfiles data set when the file type is either SAS or EGP.

3. A coworker attempts to run the macro in a UNIX environment and it fails because the DIR command is not recognized within UNIX and will need to be replaced with an equivalent LS command. This should only be considered a defect if the macro was intended to be portable across both Windows and UNIX environments. In this scenario, because the macro was designed for only Windows environments, this *deficit* poses risk but does not constitute a *defect* (or technical debt) because UNIX portability was neither specified nor implied in the software requirements. Notwithstanding, the risk (i.e., lack of portability) can still be documented within the code to alert subsequent developers who may encounter this failure pattern on a UNIX system.

Despite these risks, the macro still might be released into production because it provides partial functionality (i.e., business value) despite its shortcomings. Whether or not specific risks are intended to be mitigated, risk-related statements within code are beneficial because they convey critical information to developers who must run, modify, and otherwise maintain the software. The revised macro headers now convey each of the identified risks:

```
* written by Troy Martin Hughes, SCSUG 2017;
* this macro parses multiple folders for SAS, EGP, or other files;
* and creates a data set that contains the folder, file name, and create date;
*** as written, only one folder can be parsed per invocation;
*** the FILETYPE parameter is dead thus only SAS and EGP files are identified;
*** software designed for Windows OS only (and will fail in UNIX);
%macro parse(dir= /* asterisk-delimited list of folders to parse */,
      dsn= /* data set in LIB.DSN format into which to put directory info */,
      subdir=YES /* YES to parse subdirectories, NO for current DIR only */,
      filetype=SAS EGP /* space-delimited list of file extensions */);
```

Note that the software intent (i.e., requirements) has not been modified, demonstrating that that first two deficits do represent technical debt that should be eliminated. Alternatively, the header comment expressing that “multiple folders” are parsed and the DIR parameter comment expressing “list of folders” could have been amended to restate that only one folder would be accepted in the parameter. This alternative solution would be an example of eliminating technical debt by revising software requirements, rather than by modifying software itself. Regardless of the chosen risk resolution strategy, the triple-asterisk (***) distinction improves software readability and risk identification by providing a standardized symbology that sets risk-related comments apart from more general comments. For example, this enables developers to locate known issues within software easily by searching for ***.

Unfortunately, informal risk-related comments (like those prefaced with ***) often lack standardization—in formatting and content—so one developer may focus on software threats, a second on software risks, a third on software vulnerabilities, and a fourth on risk resolution strategies. Moreover, from examination of the code alone (in isolation from software requirements), it may be unclear which risks should be mitigated and which should be accepted. For these and other reasons, when risk management is implemented within development environments, risks are typically documented through external artifacts such as risk registers rather than in (or in addition to) the code itself. This added documentation—and the requirement to maintain congruence between software and its respective risk register—unfortunately contribute to the paucity of software projects that implement formalized risk management.

A *risk register* is defined as a “record of information about identified risks.”[iv] Risk registers include qualitative information about risks, threats, and vulnerabilities such as a description of the risk, a description of the vulnerability, or a description of how the threat or vulnerability should be resolved (i.e., risk acceptance, avoidance, mitigation, transfer). Risk registers also typically include quantitative information that can be used to prioritize the severity of risks, prioritize which risks should be resolved first, or assess the cumulative risk for a software developer, software product, team, or organization. For example, Table 1 demonstrates an abbreviated risk register that qualitatively describes the risks identified in the PARSE macro.

Program	Line	Vulnerability	Risk	Resolution
parse.sas	6	only one folder can be parsed per invocation	if multiple folders need to be parsed, then the macro would need to be run multiple times and the results joined	enable PARSE to read multiple folders, parse them iteratively, and aggregate the results with PROC APPEND
parse.sas	9	the SAS and EGP file types are hardcoded	if either SAS or EGP files are not desired, or if PARSE is used for TXT or other file types, the results will be incorrect	enable PARSE to dynamically interpret a list of file names as originally conceptualized in software requirements
parse.sas	12	the FILENAME DIR command functions only within Windows	causes runtime error in UNIX or other environments	1. accept and document risk 2. write exception handling that only runs PARSEDIR_OLD in Windows 3. write LS statement for UNIX OS

Table 1. Abbreviated Risk Register (Qualitative Attributes Only)

Note that the third risk includes three resolution strategies, likely indicating that further discussion is warranted to determine which course of action should be taken. Risk registers provide a standardized format to collect and evaluate risk information and to prioritize risk mitigation efforts, but risk management and risk-driven decision making first must be valued by stakeholders to ensure that risk registers are not only initiated but also enforced and maintained. The only thing worse than not having a risk register is failing to maintain a risk register, which can lead to invalid assessments of risk and poor decision making.

For example, a week after implementation of the previous PARSE macro, an additional fourth deficit—a latent defect—is discovered:

1. When a folder name in the DIR parameter contains a space (e.g., D:\Folder with spaces), the macro fails. This is considered a latent defect because the macro should have accommodated folder names containing spaces—a common folder-naming convention—but the developers failed to imagine this use case.

Upon discovery, the SAS practitioner immediately updates the PARSE macro to document the new defect:

```
* written by Troy Martin Hughes, SCSUG 2017;
* this macro parses multiple folders for SAS, EGP, or other files;
* and creates a data set that contains the folder, file name, and create date;
*** as written, only one folder can be parsed per invocation;
*** the FILETYPE parameter is dead thus only SAS and EGP files are identified;
*** software designed for Windows OS only;
*** macro fails if DIR includes a folder name with spaces--NEED TO FIX!!!;
%macro parse(dir= /* asterisk-delimited list of folders to parse */,
      dsn= /* data set in LIB.DSN format into which to put directory info */,
      subdir=YES /* YES to parse subdirectories, NO for current DIR only */,
      filetype=SAS EGP /* space-delimited list of file extensions */);
```

It's great that the defect was detected early and even better that it was documented within the code, because this will alert SAS practitioners to the risk of failure. However, if an external risk register is maintained (such as the one demonstrated in Table 1), the newly discovered risk should also be documented within the register because the risk reflects development work (i.e., technical debt) that will require resources to remedy. Without documentation in a risk register, project managers, product owners, and other key stakeholders (who aren't waist-deep in the code) might be unaware of the risk and fail to prioritize its remediation against other development or operational activities.

This redundant documentation, however, intimates why formal risk registers are often not maintained or even initiated within software development environments. Of course, in this example, the vulnerability is resolved merely by placing double quotes around the &DIR macro variable (i.e., "&DIR"); more realistic defects might take hours or days to design and develop a resolution, more clearly justifying their inclusion within a risk register. But other aspects of risk registers can also become outdated. For example, the additional comment (describing the newly discovered defect) incremented

all code by one line, so all referenced line numbers (statically listed in the risk register) must be incremented—if not abandoned. As risks are mitigated, their respective comments must be deleted from software and from risk register entries, again duplicating effort and reducing viability in fast-paced development environments.

One solution that eliminates redundant risk-related documentation is the adoption of standardized code commenting that can facilitate automated documentation (i.e., data-driven documentation) through code parsing algorithms. By standardizing the format and content of risk-related comments, metadata can be identified and extracted from code and inserted into a risk register. This eliminates the need to maintain separate versions of documentation—one inside and one outside of the code—and encourages developers to document risk because more stakeholders benefit from this effort. In the next sections, standardized qualitative and quantitative risk-related comments are introduced that facilitate the automated production of risk registers.

RISK DATA MODEL

The first step in analyzing risk, consistent with domain-driven design principles, is to ensure a consistent risk model is understood, implemented, and utilized. Once the domain (i.e., risk) is understood, it can be modeled within the configuration file and analyzed programmatically. The following configuration file can be saved as D:\sas\risk\riskconfig.txt and includes program-specific tags (<AUTH> and <DESC>) as well as risk-related tags:

```
* this risk-related data model should be saved as D:\sas\risk\riskconfig.txt
<desc/len=$500>Program Description
<auth/len=$50>Program Author

[<risk>work.risk
<threatdesc/len=$200>Threat Description
<vulndesc/len=$200>Vulnerability Description
<riskdesc/len=$200>Risk Description
<riskres/len=$200>Recommended Resolution
<risksev/len=8>Risk Severity (1 to 10)
<riskprob/len=8>Risk Probability (1 to 10)
<riskdisc/len=8>Likelihood of Discovery (1 to 10)
<riskease/len=8>Ease of Risk Resolution (1 to 10)]
```

This configuration file demonstrates both singular tags and grouped tags (described in the next two sections). In general, singular tags are associated with program-specific information (e.g., one author per program) whereas grouped tags can be repeated throughout a single program (e.g., a program can have multiple risks). The sample RISK data model within the configuration file includes the following components:

- THREATDESC – A description of a specific threat to software, which might also specify one or more specific software vulnerabilities that the threat could exploit.
- VULNDESC – A description of the software vulnerability.
- RISKDESC – A description of the risk to software, which might describe all the bad things that can happen if the associated threat and/or vulnerability are not resolved.
- RISKRES – The risk resolution provides the description of the course of action to be taken regarding the risk (i.e., avoidance, mitigation, transfer/sharing, or acceptance/retention). Many risk models describe the specific method through which a risk is, was, or should be resolved (e.g., “can be mitigated by adding exception handling to the code that can detect if a data set is missing”).
- RISKSEV – Risk severity is the numeric representation of the loss that would be incurred were the risk realized. In this model, 1 represents a risk that would cause little to no damage while 10 represents a risk that would cause grave damage were it to occur.
- RISKPROB – Risk probability is the numeric representation of the probability that the specific risk will occur. In this model, 1 represents a risk that is unlikely to be occur while 10 represents a risk that will likely occur.

- RISKDISC – Likelihood of discovery is the numeric representation of the likelihood that a given risk will be discovered if it occurs. In this model, 1 represents a risk that is easily discovered whereas 10 represents a risk that would be difficult to discover were it to occur. For example, a failure in software that causes no warning or runtime error might be dangerous because it's more difficult to detect than failures that do produce warnings or runtime errors that can be programmatically detected or observed within log files.

- RISKEASE – Ease of risk resolution is the numeric representation of the ease with which a specific risk can be resolved. In this model, 1 represents a risk that can be resolved with little effort while 10 represents a risk that would be very difficult to resolve.

Some risks are resolved because they can cause the most harm if they occur while other risks are resolved because they can be fixed quickly without much effort. Formalizing a quantitative risk model and tracking software risks, threats, and vulnerabilities is important because it allows the multidimensional nature of risks to be aggregated, analyzed, and prioritized for resolution. For example, two risks might each be expressed by a severity of 7, but if one risk occurs often (risk probability of 9) and the other infrequently (risk probability of 2), the former risk should likely be mitigated first because of the expected risk frequency.

For example, one quantitative analysis of risk might add the severity, probability, likelihood of discovery, and ease of resolution to generate a composite risk score for each specific vulnerability (having a range of between 4 and 40). Within this model (using the previous 1 to 10 ratings), a score of four would indicate a risk that 1) is not severe, 2) has a low probability of occurrence, 3) can be detected easily, and 4) is easy to resolve or mitigate while a score of 40 would indicate a risk that 1) is severe, 2) has a high probability of occurrence, 3) cannot be easily detected, and 4) cannot be easily mitigated or resolved. Other quantitative analyses might instead multiply the values or perform various mathematical and statistical calculations to generate risk models that most accurately depict risk. Whatever analytic method is chosen, the benefit of quantitatively modeling risk is that it allows individual risks to be compared with each other and to be aggregated to form composite risk scores.

SINGULAR TAGS/COMMENTS

Software metadata—including risk-related information—often appear in code as raw, unstandardized comments. For example, the previous PARSE macro lists the program author and includes some risk-related information, but the freeform comments decrease readability—especially the ability to parse the code automatically. One method to standardize comments is to preface them with user-defined tags such as <AUTH> (i.e., author), <CONF> (i.e., SAS user conference for which the program was written), or <DESC> (i.e., software description) that indicate specific metadata. For example, the previous PARSE header can be recreated with standardized comments that more precisely delimit the metadata:

```
*<auth>Troy Martin Hughes;
*<conf>SCSUG 2017;
*<desc>this macro parses multiple folders for SAS, EGP, and other files and
creates a data set that contains the folder, file name, and create date;
```

With this comment tagging paradigm in place, a separate interpreter program can now be used to parse SAS programs that include the designated tags <AUTH>, <CONF>, or <DESC>. For example, if the previous three lines of code are saved to the SAS program D:\sas\risk\header.sas, the following conditional logic parses the program, identifies the <AUTH>, <CONF>, and <DESC> tags, extracts their accompanying text, and writes the metadata to the D:\sas\risk\metadata.sas7bdat data set:

```
library risk 'd:\sas\risk';
filename cfgfile 'd:\sas\risk\header.sas';
data risk.metadata (drop=line);
   infile cfgfile truncover end=eof;
   length line $1000 auth $50 conf $50 desc $500;
   label auth='Author' conf='Conference' desc='Description';
   input line $1000.;
   if find(lowcase(compress(line)),'*<auth>') then do;
```

```
      auth=strip(substr(line,find(lowcase(line),'<auth>')+6));
      auth=substr(auth,1,length(auth)-1);
      end;
   if find(lowcase(compress(line)),'*<conf>') then do;
      conf=strip(substr(line,find(lowcase(line),'<conf>')+6));
      conf=substr(conf,1,length(conf)-1);
      end;
   if find(lowcase(compress(line)),'*<desc>') then do;
      desc=strip(substr(line,find(lowcase(line),'<desc>')+6));
      desc=substr(desc,1,length(desc)-1);
      end;
   if eof then output;
   retain auth conf desc;
run;
```

The resultant Metadata data set is demonstrated in Table 2.

Author	Conference	Description
Troy Martin Hughes	SCSUG 2017	this macro parses multiple folders for SAS, EGP, and other files and creates a data set that contains the folder, file name, and create date

Table 2. Metadata Extracted from Standardized Comments

Although effective at extracting the standardized comments, this hardcoded interpreter is neither flexible nor scalable because it requires modification each time user-defined metadata tags need to be added or changed. Thus, were an additional <VER> tag desired to specify the minimum SAS version required to run a specific program, the interpreter would need to be modified to include this logic. A more flexible and reusable methodology defines tags externally in a configuration file that can be ingested by the interpreter program, enabling the interpreter to remain stable while only the configuration file is modified. This data-driven technique enables non-developers to modify external configuration files to achieve dynamic results without the necessity to modify the underlying software. Rather than relying on hardcoded business rules to *imply* a data model, this modular solution instead *supplies* a data model (i.e., the tag definitions within the configuration file) that is abstracted by the interpreter to produce business rules dynamically.

To demonstrate the increased flexibility that data-driven configuration files provide (over hardcoded methods) to implement business rules and program logic, the following text can be saved within a text file as D:\sas\risk\config_header.txt:

```
* This is a comment within the configuration file

<auth/len=$50>Author
<conf/len=$50>Conference
<desc/len=$500>Description
<ver/len=$5>SAS Version Required
```

Note that comments have been arbitrarily defined as statements beginning with an asterisk, and that semicolons are not required to terminate lines (because the configuration file represents instructions for SAS code, but is not SAS code itself). In this example, the third line defines a metadata tag <AUTH> that captures the program author in a 50-character TAG_auth variable. Within the **text configuration file**, each singular tag must conform to the following specifications:

- Each tag must appear on a separate line, but blank lines can be inserted to improve readability.
- Semicolons are not required to terminate lines.
- Comments must start with an asterisk.
- Singular tags cannot contain brackets ([ ]), as this would cause them to be interpreted (erroneously) as grouped tags.

- The tag name (e.g., AUTH, CONF, DESC) must conform to SAS variable naming conventions with one exception—because “TAG_” is appended to the user-defined tag name to create the variable name, the tag name must be between 1 and 28 characters in length (rather than between 1 and 32 characters).v

- The character length of a singular comment can vary and must be specified by the respective LEN= token in the associated configuration file.

- The tag description (e.g., Author, Conference, Description) must conform to SAS variable label naming conventions (between 1 and 256 characters). For example, the AUTH tag will create the variable TAG_auth having a variable label “Author”.

In this example, once the previous configuration file (D:\sas\risk\config_header.txt) is modified and saved, the following sample SAS program (D:\sas\risk\header.sas) should be saved:

```
*<auth>Troy Martin Hughes;
*<conf>SCSUG 2017;
*<desc>this macro parses multiple folders for SAS, EGP, or other files;
*<desc>creates a data set that contains the folder, file name, and create date;
*<ver>9.4;

data mydata; * <undefined> only parsed if this tag is added to the config file;
   set somedata;
run;
```

Note that the program author and conference are now represented by the <AUTH> and <CONF> tags, allowing these values to be placed more easily into a data set such as the one demonstrated in Table 1. The description has been distributed across two lines—both of which include the <DESC> tag to improve readability, and which are terminated by semicolons. Within a **SAS program file**, singular comments must conform to the following specifications:

- A singular comment begins with an asterisk, followed by a tag (defined in the configuration file), and concludes with the tag description. The comment can occur at the start of a new line or can follow code at the end of a line. The tag can immediately follow the asterisk, or they can be separated with spaces.

- Only one singular comment can be written per code line, and the comment must exist on a single code line.

- Singular comments are always character but can have varying lengths, as defined in the associated configuration file by the respective LEN tokens.

- Comments having the same tag (within a single program) are aggregated. Thus, the two consecutive <DESC> tags will create a single observation in the Metadata data set that includes the concatenation of both <DESC> comments. This allows tag content to span multiple lines as well as to be interspersed throughout a program, as there is no requirement that identical tags must be consecutive. However, the total length of the variable remains the length specified in the LEN token (within the configuration file). For example, if a program description is limited to 160 characters but the <DESC> tags are distributed over three lines of approximately 80 characters each, the last 80 characters (i.e., the final <DESC> tag and its comment) will be truncated once the 160-character limit is reached.

- Despite the number of tags that are included within a SAS program, each program generates only one observation in the Metadata data set, having one variable for each tag represented in the associated configuration file.

- Tags included within a configuration file do not have to appear within SAS programs that are analyzed by SCAVENGER. However, SCAVENGER will identify and export only those tags that have been defined in the associated configuration file. For example, if a SAS program is analyzed by SCAVENGER using this Config_header configuration file, the code being analyzed does not need to include any specific comment tags. However, when the <UNDEFINED> tag is encountered by SCAVENGER in the Header program that is

being analyzed, the associated comment on that code line will not be included in the Metadata data set because <UNDEFINED> is not defined within the associated configuration file.

Because each SAS program that is analyzed by SCAVENGER will generate only one observation in the output Metadata data set, singular comments are ideal for expressing and documenting program-specific metadata like author or creation date. The aggregate nature of singular comments also lends them to support simple versioning documentation. For example, if the Config_header configuration file is again modified to the following text, a new <UPDATE> tag is defined:

```
* This is a comment within the updated configuration file

<auth/len=$50>Author
<conf/len=$50>Conference
<desc/len=$500>Description
<ver/len=$5>SAS Version Required
<update/len=$2000>Program Updates and Versioning
```

Given the updated configuration file, SCAVENGER can be used to parse the following program Test.sas:

```
*<update>JUL 2017 – this DATA step added;
*<ver> 9.3;
*<desc>This sample program demonstrates software versioning;

*<update>AUG 2017 – removed DROP statement;
data mytest;
   length var1 8 var2 8 var3 8;
   var1=10;
   retain var1 var2; *<update>SEP 2017 – added var2 to RETAIN;
run;
*<donothing> blah blah blah;
```

When SCAVENGER is invoked and encounters the Header.sas program, it produces the Metadata data set that includes the following variables:

- TAG_auth
- TAG_conf
- TAG_desc
- TAG_ver
- TAG_update

Table 3, an abridged depiction of the Metadata data set, includes the values of the last three tags as extracted from the Test program. Note that the <UPDATE> tag is captured in the TAG_Update variable (i.e., Program Updates and Versioning) and includes the concatenation of all <UPDATE> comments in the order in which they appear in the program. If the aggregated text from all <UPDATE> tags had exceeded the 2,000-character limit (i.e., the character length defined within the configuration file), only the first 2,000 characters would have been saved to the TAG_Update variable in the Metadata data set.

Version Required	Description	Program Updates and Versioning
requires SAS 9.3 or higher	This sample program demonstrates software versioning	JUL 2017 – this DATA step added AUG 2017 – removed DROP statement SEP 2017 – added var2 to RETAIN

Table 3. Abridged Metadata Data Set Demonstrating Comment Aggregation Across Multiple Tags

Note that the <DONOTHING> tag appears in the Test program but is not added to the Metadata data set because the <DONOTHING> tag does not appear in the associated configuration file. However, were the configuration file modified to include <DONOTHING>—or the SCAVENGER macro invoked with a different configuration file that did include <DONOTHING>—then the <DONOTHING> comments within Test would be parsed and extracted. This demonstrates the tremendous versatility of SCAVENGER, as it can be invoked with an endless number of configuration files, enabling it to meet the changing demands of diverse stakeholders across disparate industries and organizations. This also enables data models to develop and evolve over time to include additional metadata tags (through inclusion within configuration files) without the need to modify the interpreter program SCAVENGER.

GROUPED TAGS/COMMENTS

Grouped tags/comments provide a second method to parse and extract metadata from SAS comments, including the option to define both character and numeric data. Grouped tags can be included in the same configuration file as singular tags, as demonstrated in the following sample configuration file Config_grouped.txt:

```
* the following tags (DESC AND AUTH) are singular tags
<desc/len=$500>Program Description
<auth/len=$50>Program Author

* the following grouped tags (RISK_DESC and RISK_SEV) are in the RISK tag group
[<risk>work.risk
<riskdesc/len=$200>Risk Description
<risksev/len=8>Risk Severity]
```

The configuration file includes two singular tags (DESC and AUTH) that will be created (as TAG_desc and TAG_auth) in the Metadata data set, as well as two grouped tags (RISKDESC and RISKSEV) that will be created (as GRP_riskdesc and GRP_risksev) in the WORK.Risk data set. In this example, only one set of grouped tags is created (RISK), but additional grouped tags could have been defined within the same configuration file. This flexibility and extensibility is demonstrated later in this text.

Within the **text configuration file**, each grouped tag must conform to the following specifications:

- Configuration file comments must start with an asterisk but do not require a terminal semicolon.
- Each tag must appear on a separate line, but blank lines can be inserted to improve readability.
- A grouped tag begins with an open/left bracket ([), followed by the grouped tag name (e.g., <RISK>), followed by the data set (in LIBRARY.DataSetName format) in which all comments associated with the tag will be generated. This single line comprises the grouped tag definition.
- Subsequent lines (that follow a grouped tag definition) include the tag (e.g., RISKDESC), followed by the length definition (e.g., len=$200), followed by the SAS variable label that will be applied to the metadata variable that is created.
- The tag name (e.g., AUTH, CONF, DESC) must conform to SAS variable naming conventions with one exception—because “GRP_” is appended to the user-defined tag name to create the variable name, the tag must be between 1 and 28 characters in length (rather than between 1 and 32 characters).
- Grouped comments can be either character ($ precedes the LEN value) or numeric (no $ preceding the LEN value) type. In the sample configuration file, RISKDESC is character with length 200 and RISKSEV is numeric with length 8.
- The tag description (e.g., Risk Description, Risk Severity) must conform to SAS variable label naming conventions. For example, the RISKDESC tag will create the variable GRP_riskdesc having a variable label “Risk Description”.
- A grouped tag concludes with a closed/right bracket (]) that terminates the final tag definition.

Given the same configuration file Config_grouped.txt, the following abridged SAS program (header only) can be saved as D:\sas\risk\sample_grouped.sas and parsed by the SCAVENGER macro:

```
*<auth> Troy Martin Hughes;
*<desc> sample program with singular and grouped comments;

*<riskdesc>as written, only one folder can be parsed per invocation <risksev>3;
*<riskdesc>FILETYPE is dead, only SAS and EGP files are identified <risksev>7;
*<riskdesc>software designed for Windows OS only <risksev>1;
*<riskdesc>macro fails if DIR includes a folder name with spaces <risksev>7;

%macro parse(dir= /* asterisk-delimited list of folders to parse */,
      dsn= /* data set in LIB.DSN format into which to put directory info */,
      subdir=YES /* YES to parse subdirectories, NO for current DIR only */,
      filetype=SAS EGP /* space-delimited list of file extensions */);
```

This updated program header displays the author and program description using <AUTH> and <DESC> tags, as demonstrated previously. Additionally, the risk-related information—previously indicated with triple asterisks (***)—is now defined within multiple grouped comments that qualitatively describe the risk description and quantitatively describe the risk severity. This allows multiple dimensions of risk (or any other domain) to be described within software and extracted by a parser.

Grouped comments differ from singular comments primarily in that multiple grouped comments can occur on one line of code, whereas only one singular comment can occur on a one line of code. Grouped comments can also represent either character or numeric data, whereas singular comments are always defined as character. Finally, grouped comments across multiple lines of code are not aggregated, unlike singular comments which are concatenated as demonstrated in Table 3. Within a **SAS program file**, grouped comments must conform to the following specifications:

- A grouped comment begins with an asterisk, followed by a tag (defined in the configuration file), and concludes with the tag description. The comment can occur at the start of a new line or can follow code at the end of a line. The tag can immediately follow the asterisk, or it can be separated with spaces. For example, the following two lines create identical entries in the Risk data set:

```
*<riskdesc> risky line <risksev> 3;
data temp; *  <riskdesc> risky line <risksev> 3;
```

- Multiple grouped comments can occur on a single line of code. For example, because the configuration file defines both RISKDESC and RISKSEV within a single group (RISK), both grouped tags can occur on a single line of code. Grouped comments are treated independently, however, and not all tags within a group must be present on each line. Thus, a risk-related comment might include only the <RISKDESC> tag or only the <RISKSEV> tag or might include both tags; this versatility allows for additional grouped comments to be added over time as more information is gathered about a risk. For example, the following three standardized comments are each valid, but missing tags will be represented by missing data in the Risk data set:

```
* <riskdesc> risky line <risksev> 3;
* <riskdesc> risky line;
* <risksev> 3;
```

- Grouped comments can occur in any order within a line. For example, the following two lines create identical entries in the Risk data set:

```
* <riskdesc> risky line <risksev> 3;
* <risksev> 3 <riskdesc> risky line;
```

- Grouped comments can be either character or numeric type, as defined by the LEN statement in the associated configuration file.

- Grouped comments cannot contain greater than (>) or less than (<) symbols, as these can confuse the parser, which could interpret them as subsequent comment tags.

- Each line containing one or multiple grouped comments creates one observation in the respective data set—WORK.Risk in this example. Grouped comments do not create observations in the Metadata data set.

SCAVENGER MACRO SETUP AND COMPONENTS

The SCAVENGER macro suite (Appendix A) should be downloaded and saved. The following examples demonstrate downloading SCAVENGER to D:\sas\risk\scavenger.sas but the program can reside anywhere. The program contains several subordinate macros that are defined and described in this section.

The SCAVENGER macro definition follows:

```
%macro scavenger(dir= /* asterisk delimited list of folders to include */,
   cfg= /* folder and name of configuration file */,
   dsnout= /* metadata data set in LIB.DSN format (which is first deleted) */,
   subdir=YES /* YES to parse subdirectories, NO to not include */,
   filetype=SAS EGP /* space-delimited list of all file extensions to find */);
```

SCAVENGER defines the following parameters:

- DIR – This asterisk-delimited list of one or more folder paths will be parsed by SCAVENGER.

- CFG – SCAVENGER requires an associated configuration file—essentially, the data model containing metadata tags that will be used to parse comments within all SAS programs. In the following examples, the configuration file in the “Risk Data Model” section should be saved as D:\sas\risk\riskconfig.txt.

- DSNOUT – A metadata data set (in LIB.DSN format) is created in which program-specific metadata (extracted from comments) are saved. Each SAS program that is encountered and parsed by SCAVENGER will create one observation in the metadata data set; SAS Enterprise Guide project files will create one observation for each imbedded (but not linked) SAS program inside each project file.

- SUBDIR – This parameter specifies whether all subdirectories will be recursively parsed (YES) or ignored (NO). For example, if the DIR parameter specified is C: and the SUBDIR parameter is YES then all SAS program files on the C: drive will be searched. The default parameter setting is YES.

- FILETYPE – This space-delimited parameter specifies the file extensions that will be searched. The default parameter setting is SAS EGP, indicating that both SAS program files and SAS Enterprise Guide project files will be identified and parsed.

The following code loads the SCAVENGER macro suite and invokes SCAVENGER, searching the D:\sas folder (and subordinate folders) and using the Riskconfig configuration file as its data model:

```
%include 'd:\sas\risk\scavenger.sas';
%scavenger(dir=d:\sas, cfg= d:\sas\risk\riskconfig.txt, dsnout=metadata);
```

This invocation of SCAVENGER creates three primary output data sets: Filelist, Metadata, and Risk. The Filelist data set includes a list of all SAS program and project files identified by PARSEDIR. The Metadata data set includes program- and project-specific metadata, such as macros that are defined, global variables that are created, lines of code per program, and program files that are referenced with %INCLUDE statements. The Metadata data set also includes all singular comments that conform to the associated data model tags (within the Riskconfig configuration file) extracted from identified SAS program and project files. Similarly, the Risk data set contains all grouped comments extracted from the identified programs. These data sets are demonstrated in later sections.

The SCAVENGER macro calls subordinate macros, including PARSEDIR, READXML, READCODE, and READCONFIG, all of which are included in Appendix A. The macros are not invoked manually but are introduced in this section to provide a behind-the-scenes view of SCAVENGER operations:

- **PARSEDIR** — This macro iteratively parses a directory (and optionally its subdirectories) to create a data set (DSN parameter) that includes the folder location, file name, and create date for each identified file that matches a space-delimited list of acceptable file extensions. For example, the default FILETYPE parameter is SAS EGP, representing that PARSEDIR will identify and analyze only SAS program and project files. The macro is called only once with the following definition:

```
%macro parsedir(dir= /* asterisk-delimited list of folders to parse */,
   dsn= /* data set in LIB.DSN format into which to put directory info */,
   subdir=YES /* YES to parse subdirectories, NO to only parse current DIR */,
   filetype=SAS EGP /* space-delimited list of file extensions to find */);
```

- **READXML** — This macro is dynamically invoked whenever SCAVENGER encounters a SAS project file. All SAS Enterprise Guide project files are compressed zip files, each of which contains the file Project.xml in its root structure. Thus, by peering into the zipped project file (with the ZIP option of the FILENAME statement), READXML interrogates the Project.xml file and extracts information, including the names of all imbedded SAS programs, the names of all linked SAS programs, and a SAS project note (only one note) if it exists within the project. With this information retrieved, READCODE iteratively opens and parses each imbedded SAS program (skipping the linked programs). For each SAS Enterprise Guide project file that is encountered, READXML is called only once with the following definition:

```
%macro readxml(egpfile= /* folder and name of EGP file */,
   xmlfile=project.xml /* name of XML inside EGP file */);
```

- **READCODE** — This macro ingests and parses SAS programs, including both externally saved SAS programs and imbedded SAS programs that are saved within SAS Enterprise Guide project files. The FILENAME statement, called dynamically from SCAVENGER, enables this flexibility. When READCODE encounters the first program file it reads the configuration file to retrieve a list of metadata tags that will be searched for within SAS programs. For example, including the <AUTH> tag in a configuration file will cause READCODE to search for this tag within all SAS programs and to extract the associated metadata that follows that tag. In addition to parsing tags dynamically identified within the configuration file, READCODE also extracts other metadata, including all macros that are defined, all user-created SAS global macro variables, and the number of lines of code per program. The macro definition follows:

```
%macro readcode(cfg= /* folder and name of configuration file */,
   infilename= /* file reference to the SAS program file */,
   dsn= /* metadata data set in LIB.DSN format */);
```

- **READCONFIG** — This macro is invoked only when READCODE is called for the first time, when encountering the first SAS program file. A sample configuration file is demonstrated previously, and the macro definition follows:

```
%macro readconfig(cfg= /* folder and name of configuration file */);
```

The modular nature of these macros supports the maintainability and extensibility of SCAVENGER. For example, while SCAVENGER is intended to be run on Windows systems, a simple modification to the PARSEDIR macro could make SCAVENGER portable to UNIX environments without the necessity to modify other SCAVENGER modules. Similarly, if a SAS practitioner wanted to extract additional metadata from the Projects.xml file within SAS Enterprise Guide project files, he could do so by modifying the parsing logic within the READXML macro without having to modify other aspects of the software.

This modularity also facilitates software reusability. For example, the PARSEDIR macro can be used in separate programs that require a directory listing of files—even for non-SAS files. PARSEDIR could be used to search for and

find all text or XML files on one or multiple drives. The READXML macro also can be used to parse SAS Enterprise Guide project files for other purposes. The structure of SAS Enterprise Guide project files and the methods through which READXML parses them is described in the next section.

STRUCTURE OF SAS ENTERPRISE GUIDE PROJECT FILES

Because the parsing and analysis of SAS Enterprise Guide project files has represented an obstacle to SAS practitioners in the past, some background on the structure of project files is warranted and will facilitate the reuse and generalizability of the READXML and READCODE macros (included in Appendix A) that read and parse the project file structure and imbedded programs, respectively. Figure 1 demonstrates the typical "Process Flow" view of a sample SAS Enterprise Guide project file. In this example, the Initialize program executes first, after which branches bifurcate into two programs, Macros and Create_data. The project note describes this prerequisite.

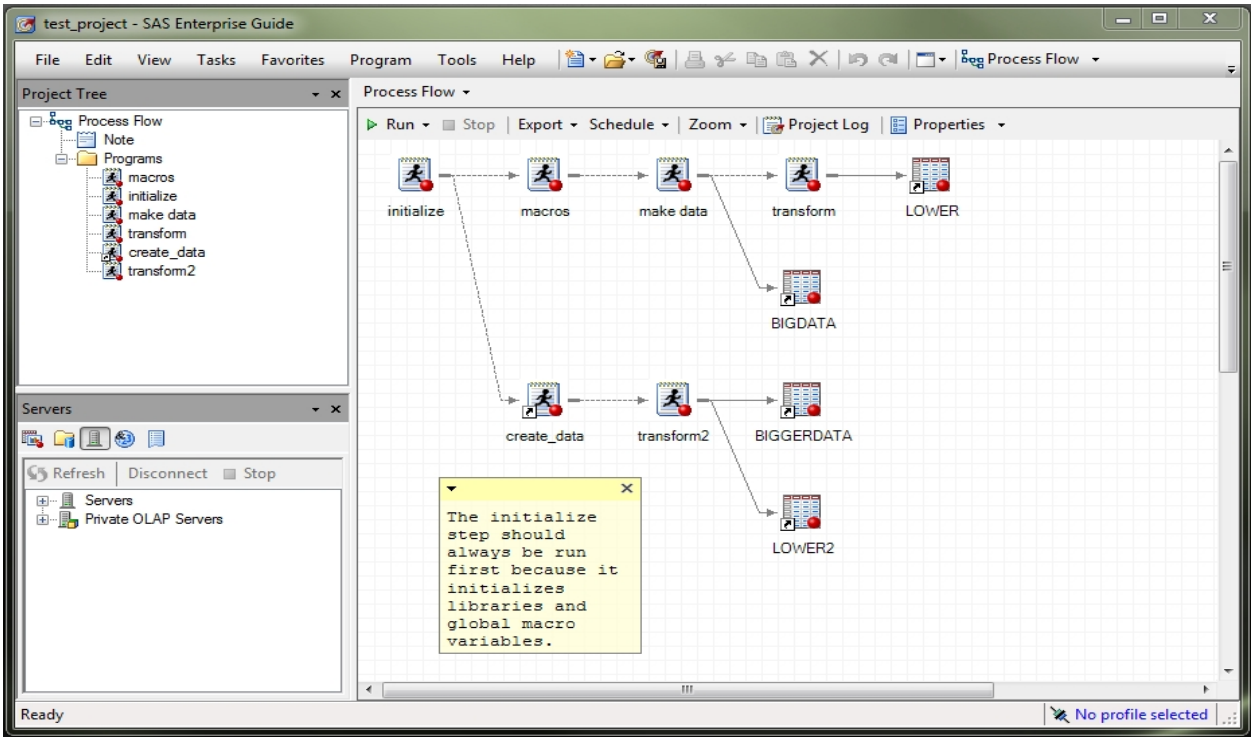

Figure 1. Process Flow View of a SAS Enterprise Guide Project File

The Initialize and Macros programs are both imbedded programs, in that they exist only inside the project file itself. A program can be imbedded by right clicking the workspace and selecting "New Program." However, because imbedded programs are saved in the project file, they are not available for external use or application. The Create_data program conversely represents a linked program, in which the code lies outside of the SAS project file (indicated by the tiny arrow in the icon). In this example, the Create_data program resides in the Perm folder as C:\perm\create_data.sas. Linked programs can be created simply by dragging a SAS program file into the SAS Enterprise Guide workspace or by selecting "Open Program" and choosing a specific program to import into the project.

Reusability is greatly facilitated through linked programs because external SAS programs can be modified in only one location and affect all subsequent uses of the code. For example, The Macros program is currently an imbedded program that defines macros used in this project. However, if external SAS programs or projects could benefit from the same macros, the code must be inefficiently copied or imported from the current project, thwarting reusability. Thus, a preferred design to promote software reuse would incorporate an external SAS program (e.g., C:\perm\macros.sas) that could be linked to this and other programs or projects. Moreover, software maintainability is improved because the external program can be updated once to affect all downstream uses of its code.

A potential downside of centralized software reuse can be decreased security. Were Macros.sas maintained in a single, external SAS program and linked to (rather than imbedded in) this project (or otherwise referenced via the %INCLUDE statement or SAS Autocall Macro Facility), control over that code might be wrested from the developer's grasp. Rather than being owned by a single individual and utilized in a stovepipe, the program would be owned by the team or organization and possibly controlled via change control policies and procedures. Thus, where code reuse is supported in production software, measures must be taken to ensure that linked or referenced modules are not accidentally modified or modified intentionally with unintended consequences. Reuse libraries and catalogs facilitate this objective by documenting and organizing software assets, including those intended for reuse.

While Figure 1 depicts the view of project files that most users see, by renaming the file from an EGP to a ZIP extension, the underlying structure of the project file can be understood and interrogated. Figure 2 demonstrates the same project file after it has been renamed and opened with a zip application such as WinZip or WinRAR.

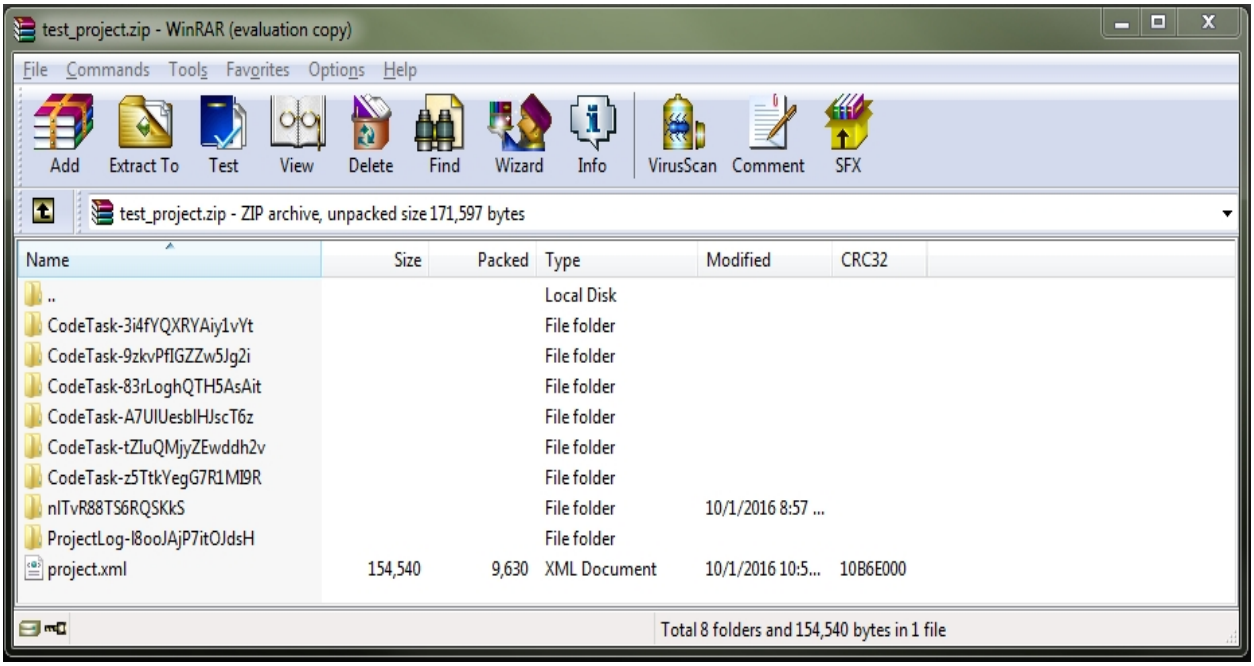

Figure 2. View of a SAS Enterprise Guide Project File as a Zipped Folder Structure

Within each project file, Project.xml is the hub of information. Metadata about project content, linked or imbedded programs, SAS data sets, process flow relationships, notes, last project execution date, and other information is stored in this file. The READXML macro exploits the structure of the XML file to identify all imbedded and linked programs and to ingest all SAS notes. For example, all SAS programs are identified by the <Element Type="SAS.EG.ProjectElements.CodeTask"> tag while the corresponding <Label> tag represents the program name and the <ID> tag represents the program ID within the XML structure. Thus, the following XML snipped demonstrates that that imbedded SAS program Initialize can be found in the zipped folder CodeTask-9zkvPfIGZZw5Jg2i:

```
<Element Type="SAS.EG.ProjectElements.CodeTask">
  <Element>
    <Label>initialize</Label>
    <Type>TASK</Type>
    <Container>PFD-Ti6KbIpksie4aNSb</Container>
    <ID>CodeTask-9zkvPfIGZZw5Jg2i</ID>
```

Figure 3 demonstrates the XML snippet in which the imbedded program Initialize is referenced. Note that other metadata (not described or utilized in this text) such as create date and modify date are also maintained within the SAS Enterprise Guide metadata structure. Thus, if a team wanted to extract additional information contained within

Project.xml files, they would only need to modify the READXML macro to enable it to recognize and extract the additional metadata and content.

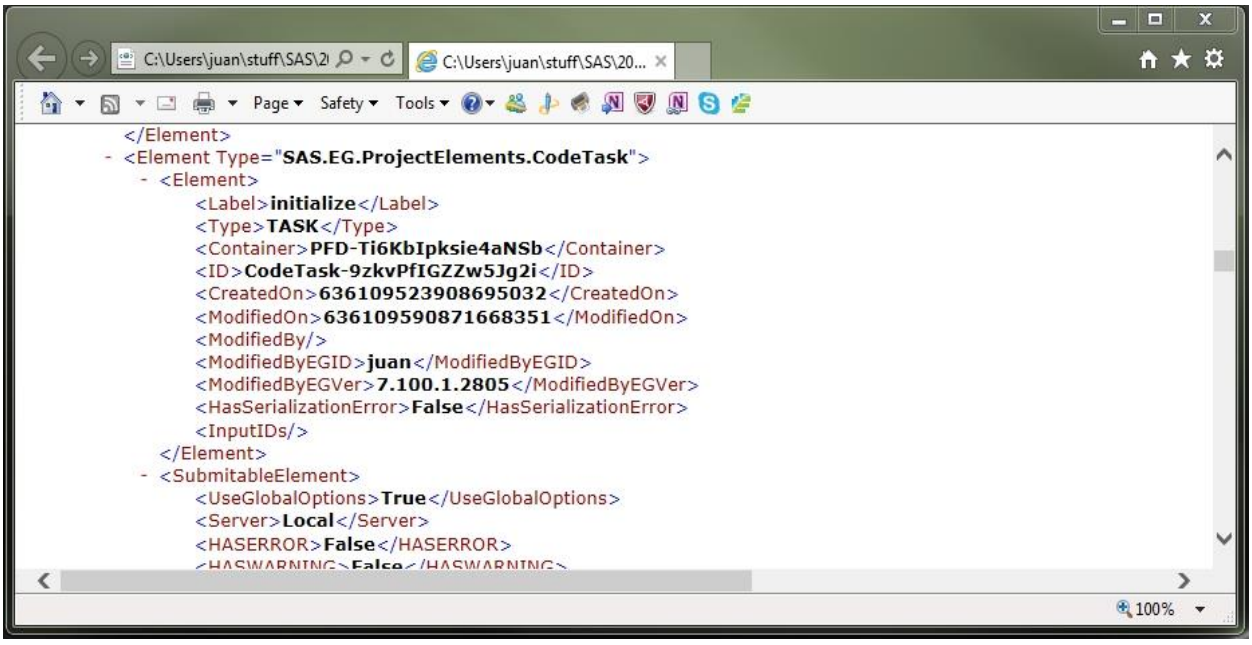

Figure 3. View of a SAS Enterprise Guide Project.xml File (Imbedded SAS Program)

With the zipped folder now identified in which Initialize resides (i.e., CodeTask-9zkvPfIGZZw5Jg2i), the user (or, in this case, the READCODE macro) can navigate to that folder and extract the contents of the Initialize program. Figure 4 demonstrates the zipped view of CodeTask-9zkvPfIGZZw5Jg2i folder with Initialize renamed as code.sas. Within the SAS Enterprise Guide compressed folder structure, all programs are saved as code.sas, thus the program ID is required to identify actual program names and their respective folder locations.

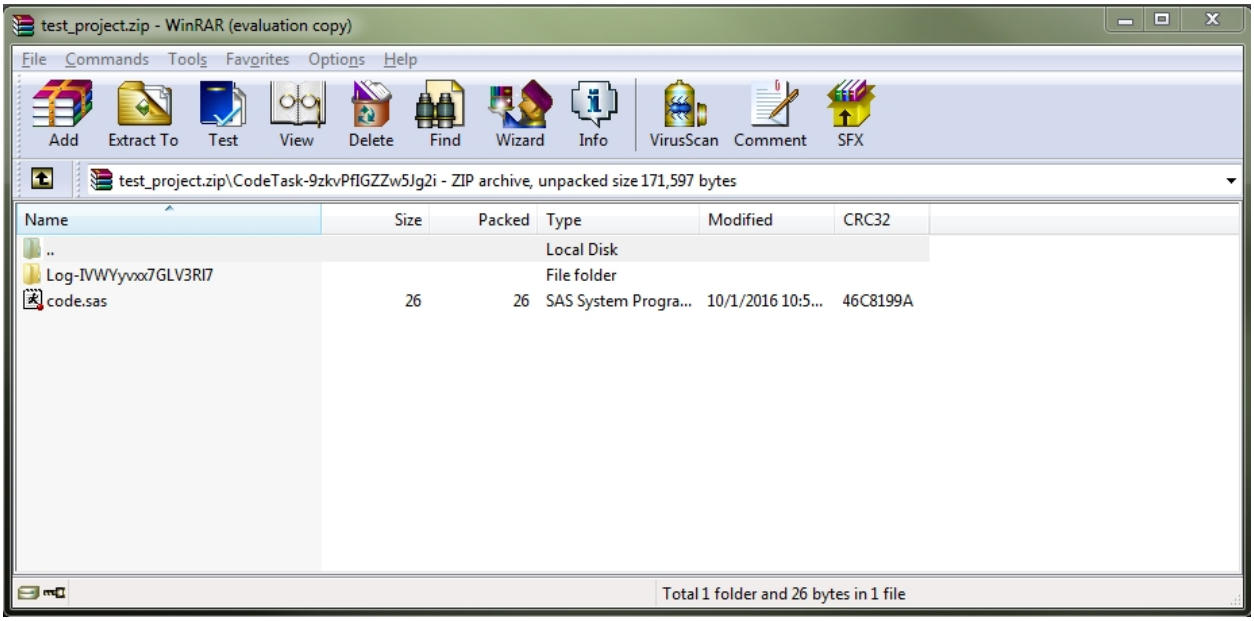

Figure 4. View of Initialize Program Within Compressed Folder Structure

To identify linked programs, the <DNA> tag (subsumed under <CodeTask>) can be utilized to determine the location of the linked program. For example, further interrogation of the Project.xml file demonstrates that the Create_data program has an ID of CodeTask-A7UlUesblHJscT6z as shown in Figure 5.

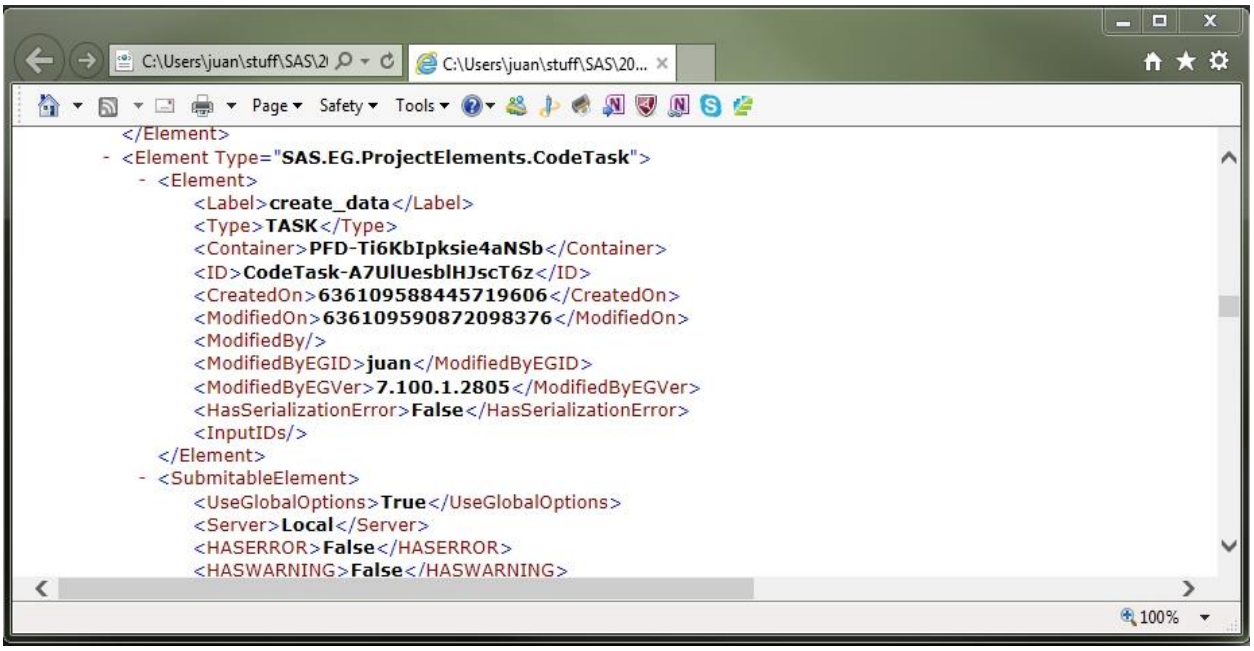

Figure 5. View of a SAS Enterprise Guide Project.xml File (Linked SAS Program)

However, because the SAS program is linked—not imbedded—no Code.sas program file resides within the corresponding zipped folder. The external code instead must first be located within the <DNA> tag, which is demonstrated in Figure 6 and shows the program is in fact C:\perm\create_data.sas.

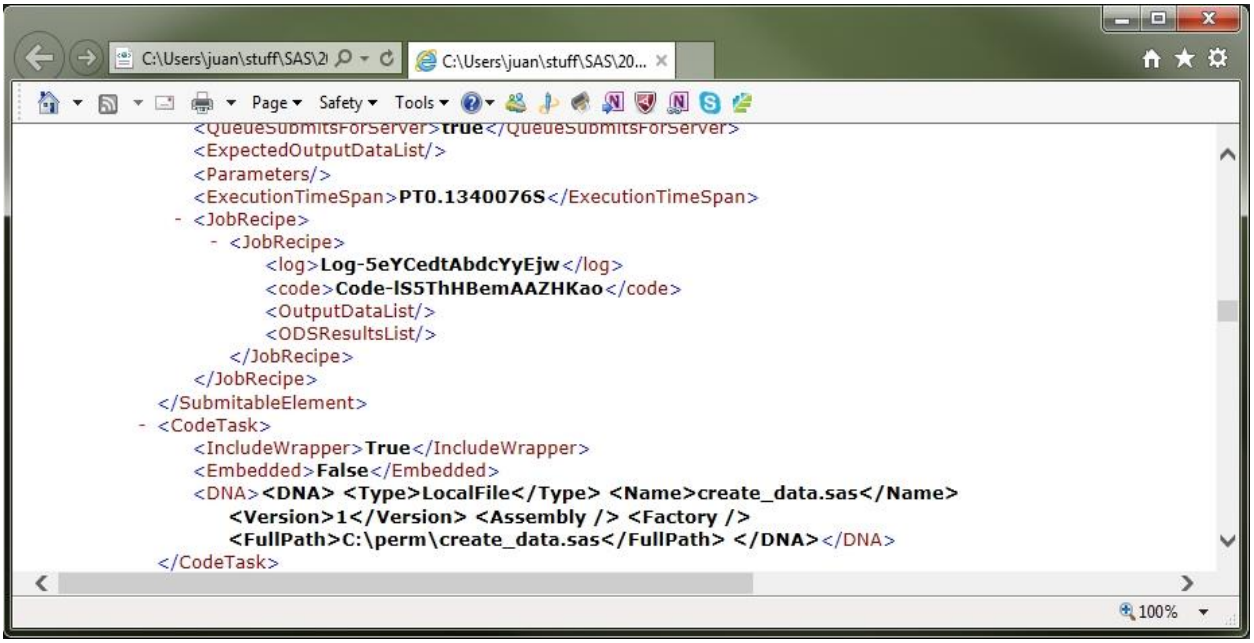

Figure 6. View of a SAS Enterprise Guide Project.xml File (Linked SAS Program)

Both SAS programs imbedded within SAS Enterprise Guide project files and externally saved SAS programs are parsed with the same READCODE macro. Linked programs (like Create_data.sas in this example) are not parsed with READCODE because this could create redundancy if multiple project files referenced the same imbedded program. Thus, to parse and extract comments from an imbedded program, SCAVENGER should be pointed (via the DIR parameter) at the folder in which the linked program resides (outside of the project file).

METADATA DATA SET

Regardless of whether READCODE is invoked on imbedded SAS program files (within project files) or externally saved SAS program files, one observation is created for each program file encountered and incrementally appended to the Metadata data set. The Metadata data set contains the following variables:

- **SoftwareName** - This variable is the full path and name (with extension) of the SAS project or program file. Because the path is included, this variable can serve as the unique key when joining this table for other purposes.

- **ProgramName** - This variable is the abbreviated name of the program, thus without the path or file extension. For SAS program files (that are saved externally in a folder), the PROGRAMNAME will always be identical to the SOFTWARENAME, with only the path and extension removed. However, for imbedded SAS programs (inside SAS Enterprise Guide project files), PROGRAMNAME will be the name of imbedded SAS program. For linked SAS programs (also inside project files), PROGRAMNAME will be blank, representing that the program was not parsed, since it exists externally and may have already been analyzed separately.

- **EGPNote** - The SAS Enterprise Guide Note contains the first 500 characters of a project note. Because these notes are endemic to SAS Enterprise Guide, this variable is blank for all externally observed SAS programs. Moreover, because each note references the project file itself (and not an individual SAS program therein), EGPNOTE is ascribed to all program files (imbedded and linked) found within a specific project file.

- **SaveDate** - The date (in DATETIME17 format) that the file was saved is generated through the FILENAME statement within the PARSEDIR macro. Externally saved SAS program files will have individual date-time stamps while SAS programs within SAS Enterprise Guide project files will inherit the date-time stamp from the project file itself.

- **MacroList** - This space-delimited variable represents a list of all macro definitions that appear in a program. With little effort, it can be used to demonstrate redundancy, dependency, conflicts, or other interaction among macros. For example, if an identically named macro is defined in multiple programs, this could represent redundant code that should be ingested into a shared repository like a reuse library. However, if the redundantly named macros perform disparate functions, this could cause conflict (and failure) were the wrong macro ever incorrectly referenced due to this ambiguity.

- **IncludeList** - This asterisk-delimited field represents a list of all program dependencies in which the %INCLUDE statement is utilized to reference external code. This field can be used to detect dependencies among programs as well as to ensure that reuse of software (via %INCLUDE) is pointing at the most complete, correct, and recent version of some SAS program or saved macro. Moreover, when a macro or other reusable code module must be modified, examination of the INCLUDELIST field will immediately identify all software products that rely on a specific external program or macro. With this information, backward compatibility and regression testing are facilitated.

- **GlobalList** - This space-delimited field represents all user-defined global macro variables that are explicitly defined with the %GLOBAL statement, including both those that are defined singly and in series. Because global macro variables persist between SAS macros and even SAS programs running in the same SAS session, they represent a tremendous security risk when not implemented with consistency and care. For example, if two macros each define the global macro variable &MYMACROVAR, each macro will overwrite the macro variable value of the other. To help identify macro variable conflicts that could unsuspectingly topple software, a comparison of all global macro variables within a SAS infrastructure or environment (not demonstrated) could substantially mitigate this risk.

- **Codelines** - This variable represents the number of lines of code in each externally saved or internally imbedded SAS program. Whereas this is an easy feat (aided by the <CTRL> and <END> keys) in any SAS program, the girth of SAS project files is sometimes more difficult to calculate because it can involve opening dozens of separate program files. Thus, by summing the lines of code across all imbedded SAS programs within a SAS project, an accurate picture of the size of the project can be immediately ascertained.

- **ENDLESS USER-SPECIFIED VARIABLES** - Although the previous variables are produced in all metadata data sets, additional variables can be dynamically added, enabling search and documentation of code to be highly customized. For example, given the tags and descriptions provided in the D:\sas\risk\riskconfig.txt Risk configuration file, the following additional user-defined variables would be created: TAG_auth and TAG_desc.

MACROLIST, INCLUDELIST, AND GLOBALLIST METADATA

When SCAVENGER is invoked and references an empty configuration file (that contains neither singular nor grouped tabs), the Metadata data set will still be produced with the metadata variables created in the previous section. For example, even without a data model, the READCODE interpreter will still extract all %INCLUDE statements, globally defined macro variables, and macros that are defined in a program, and count the lines of code in the program.

Although this text encourages the standardization of SAS comments to support readability and automated parsing, substantial variation still might be found in how the %MACRO, %INCLUDE, and %GLOBAL statements are implemented by different developers or across teams and organizations. The READCODE interpreter attempts to overcome style variability to ensure the widest possible array of metadata are captured. For example, the following SAS program can be saved as D:\sas\risk\macro_include_global.sas and demonstrates variability in horizontal spacing, vertical spacing, quotations, and commenting:

```
* sample program to test INCLUDE, GLOBAL, and MACRO statement parsing;

%include 'some file1';
%include "SOME FILE2";
%include "&path\somefile3.ext";
   %INCLUDE 'some file4'; * preceding tab;
%include 'some file5'; * with endline comment;
%include 'some file6' /* with inline comment */;
%include   'some file7';

%global var1;
%GLOBAL VAR2;
   %global var3; * preceding tab;
%global var4 var5 var6;
%global var7; * with endline comment;
%global var8 /* with inline comment */;
%global   var9;

%macro mac1;
%MACRO MAC2;
   %macro mac3; * preceding tab;
%macro mac4();
%macro mac5(par1=, par2=);
%macro mac6(par1=,
   par2=);
%macro mac7(par1=); * with endline comment;
%macro mac8; * with endline comment;
%macro mac9(par1= /* inline parameter comment */);
%macro mac10(par1=) /* with inline comment */;
%macro mac11 /* with inline comment */;
%macro   mac12;
```

This nonsensical code will never be executed, but it can be parsed using the following SCAVENGER invocation:

```
%scavenger(dir=d:\sas\risk, cfg=d:\sas\risk\riskconfig.txt, dsnout=metadata);
```

If the Macro_include_global program is the only SAS program (or project) in the D:\sas\risk folder, the Metadata data set will be created with one observation that describes the program. Table 4 demonstrates three variables created within Metadata—MacroList, IncludeList, and GlobalList.

Macros Defined	INCLUDE Files	Global Macro Variables Defined
mac1 MAC2 mac3 mac4 mac5 mac6 mac7 mac8 mac9 mac10 mac11 mac12	some file1 * SOME FILE2 * &path\somefile3.ext * some file4 * some file5 * some file6 * some file7	var1 VAR2 var3 var4 var5 var6 var7 var8 var9

Table 4. Versatility of SCAVENGER in Reading %MACRO, %INCLUDE, and %GLOBAL Statements

As Table 4 demonstrates, SCAVENGER successfully parses each of the variations included in the Macro_include_global program. Note that the IncludeList variable is asterisk-delimited because folder names and file names can contain spaces. Also note that capitalization is maintained, which might need to be standardized before usage to ensure entity resolution. For example, before comparing uses of the MAC1 macro across an organization, capitalization would need to be standardized to ensure that MAC, Mac, and mac did not erroneously appear to be different macros, since the Base SAS language does not differentiate capitalization.

Also note the effect of macro variables when they are encountered by SCAVENGER. When the &PATH global macro variable is parsed by SCAVENGER, &PATH is interpreted literally and thus appears as &PATH in the Metadata data set rather than the value of &PATH that would have been assigned during macro compilation. This dynamism is useful in building SAS programs because it makes SAS software more flexible. For example, in a development environment, &PATH could refer to a development folder while the subsequent production environment, &PATH could refer to a separate production folder. However, this dynamism also limits the ability of SCAVENGER to interpret code, as there is no way to identify the precise location of &path\somefile3.ext when found in the Metadata data set.

Similarly, not all global macro variables will be captured by SCAVENGER. For example, the following code creates three global macro variables (&GLOBALVAR, &NUM, &VAR443, and &VAR) yet only &NUM is correctly interpreted by SCAVENGER and included in the Metadata data set:

```
%let globalvar=something;

%global num;
%let num=443;
%global var&num;
%let var&num=hello;

data _null_;
   length var $20;
   var='macrovalue';
   call symput('var',var);
   run;
```

The &GLOBALVAR macro variable is not identified by SCAVENGER because %GLOBAL is implied but not stated. The &VAR443 macro variable is not identified because SCAVENGER reads the literal macro variable VAR&NUM within the code. Finally, global macro variables can also be defined in DATA steps with the CALL SYMPUT routine, so &VAR is not identified by SCAVENGER. This demonstrates that while SCAVENGER is useful is identifying global macro variable uses and potential conflicts, it should not be used exclusively because of these known limitations.

ASSESSING PROGRAM, PROGRAMMER, AND ORGANIZATIONAL RISK

The power of SCAVENGER lies not just in its ability to parse a single SAS program but also in its ability to parse all SAS programs and project files within an entire team or organization. When SCAVENGER is invoked with the

Riskconfig configuration file (demonstrated previously in the “Risk Data Model” section), this enables risk to be analyzed and aggregated by program. Moreover, because the Metadata data set contains program-specific information such as the program author or team (if specified in the associated configuration file), these metadata can be joined with the risk metadata to augment risk information and present a fuller understanding of overall risk.

To illustrate quantitative risk analysis with SCAVENGER, three nonsensical programs written by two developers—Ricky and Diane—are demonstrated and should be saved as Pgm1, Pgm2, and Pgm3. Only quantitative risk tags are included, so no additional information like risk description or mitigation strategy are described:

```
* this code saved as d:\sas\risk\pgm1.sas;
*<auth> Ricky;

*<risksev> 7 <riskprob> 2 <riskdisc> 4 <riskease> 5;
data temp1;
run;
*<risksev> 3 <riskprob> 8 <riskdisc> 8 <riskease> 7;
data temp2;
run;
*<risksev> 2 <riskprob> 3 <riskdisc> 6 <riskease> 2;
data temp3;


* this code saved as d:\sas\risk\pgm2.sas;
*<auth> Ricky;

*<risksev> 7 <riskprob> 4 <riskdisc> 3 <riskease> 2;
data temp1;
run;
*<risksev> 8 <riskprob> 6 <riskdisc> 5 <riskease> 7;
data temp2;
run;

* this code saved as d:\sas\risk\pgm3.sas;
*<auth> Diane;

*<risksev> 2 <riskprob> 2 <riskdisc> 3 <riskease> 2;
data temp1;
run;
```

The following SCAVENGER invocation specifies the RIskconfig configuration file as its data model, searches for program and project files only within the D:\sas\risk folder, and creates the Metadata and Risk data sets that contain metadata extracted from standardized comments within the program files that are encountered:

```
%scavenger(dir=d:\sas\risk, cfg=d:\sas\risk\riskconfig.txt, dsnout=metadata);
```

The default data set Metadata is created, and an abridged version is demonstrated in Table 5, indicating the folder, program name, author, and lines of code per program.

Project of Program Name	Program Name	Code Lines	Program Author
d:\sas\risk\test\pgm1.sas	pgm1	11	Ricky
d:\sas\risk\test\pgm2.sas	pgm2	9	Ricky
d:\sas\risk\test\pgm3.sas	pgm3	6	Diane

Table 5. Abridged Metadata Data Set

The Riskconfig configuration file specifies that the Risk data set be created with its risk-related grouped tags. Table 6 demonstrates an abridged version of Risk, showing only the quantitative risk-related comments that were extracted.

Project or Program Name	Program Name	Risksev	Riskprob	Riskdisc	Riskease
d:\sas\risk\test\pgm1.sas	pgm1	7	2	4	5
d:\sas\risk\test\pgm1.sas	pgm1	3	8	8	7
d:\sas\risk\test\pgm1.sas	pgm1	2	3	6	2
d:\sas\risk\test\pgm2.sas	pgm2	7	4	3	2
d:\sas\risk\test\pgm2.sas	pgm2	8	6	5	7
d:\sas\risk\test\pgm3.sas	pgm3	2	2	3	2

Table 6. Abridged Risk Data Set

A straightforward way to aggregate the various risk metrics is to add the severity, probability, likelihood of discovery, and ease of resolution for each risk-related comment. Note that this is only one of an infinite number of ways to model risk, and that this simplistic model doesn't account for missing values. Notwithstanding, the following code creates an aggregate risk value for each comment by program name:

```
data riskanal;
   set risk;
   length risktot 8;
   risktot=grp_risksev+grp_riskprob+grp_riskdisc+grp_riskease;
run;

proc means data=riskanal sum;
   class softwarename;
   var risktot;
run;
```

The SAS output from the MEANS procedure, demonstrated in Table 7, now shows the number of risk-related issues (N Obs) and cumulative risk (Sum) per program:

Analysis Variable : risktot		
Project or Program Name	**N Obs**	**Sum**
d:\sas\risk\test\pgm1.sas	3	57
d:\sas\risk\test\pgm2.sas	2	42
d:\sas\risk\test\pgm3.sas	1	9

Table 7. Cumulative Risk Per SAS Program

Armed with this cumulative risk information at the program level, developers and other stakeholders can now determine which programs have the most risk. This information can be used to identify existing technical debt, to assess the quality of software, or to prioritize refactoring efforts that reduce or eliminate risk. Additional analyses can be performed by analyzing risk-related data with respect to program-specific data found in the Metadata data set. For example, the following code joins Metadata with Risk and assesses cumulative risk by developer:

```
proc sort data=riskanal;
   by softwarename programname;
run;

proc sort data=metadata;
   by softwarename programname;
run;

data riskjoin;
   merge riskanal metadata;
```

```
   by softwarename programname;
run;

proc means data=riskjoin sum;
   class TAG_auth;
   var risktot;
run;
```

The SAS output from the MEANS procedure, demonstrated in Table 8, shows the number of risk-related issues (N Obs) and cumulative risk (Sum) per author, indicating that Ricky has incurred substantially more risk than Diane:

Analysis Variable : risktot		
Program Author	**N Obs**	**Sum**
Diane	1	9
Ricky	5	99

Table 8. Cumulative Risk Per SAS Program

Note that the Sum doesn't indicate the level of effort to remediate these issues, but rather the aggregate risk as previously modeled by adding the four quantitative risk metrics. The total risk to a team or organization can be assessed simply by removing the CLASS statement from the MEANS procedure. Or to gain a sense of the level of effort required to resolve the risk-related issues, a stakeholder could instead analyze Riskease in a similar fashion. By taking risk-related snapshots over time (by program, programmer, team, organization, or any other categorical attribute), stakeholders can responsibly monitor risk and assess whether it is trending in the desired direction. There are endless ways in which risk-related quantitative metrics can be analyzed and used by an organization to improve software development practices and the quality of the software developed.

CONCLUSION

This text introduces the concept of technical debt, which can often be operationalized within a risk model that qualifies and quantifies known issues or undone work within software. Documenting programmatic risks within code is a common practice, but by standardizing software comments across a team or organization, an interpreter program (like SCAVENGER) can parse all programs—including those saved inside SAS Enterprise Guide project files—to extract metadata (like risk metrics) defined and documented within code. SCAVENGER is a modular solution that encompasses five SAS macros, and which relies on dynamic data models that are captured in configuration files. This separation of duties ensures that data models can be flexibly adapted or modified to suit any purpose. Moreover, these data models can be maintained and modified by non-developers without the necessity to modify SCAVENGER. By running simple analyses on the output data sets created by SCAVENGER, stakeholders can not only assess technical debt and risk within an organization, but also differentiate and aggregate these issues by program, project, developer, team, or any other metric that is specified in the data model and captured through standardized SAS comments. Technical debt does not always need to be resolved, nor must risk always be mitigated within software, but responsible developers should always document and quantify known issues within their software; SCAVENGER facilitates and automates this objective.

REFERENCES

i ISO/IEC/IEEE 24765:2010. *Systems and software engineering—Vocabulary*. Geneva, Switzerland: International Organization for Standardization, International Electrotechnical Commission, and Institute of Electrical and Electronics Engineers.

ii Troy Martin Hughes. 2016. Your Local Fire Engine Has an Apparatus Inventory Sheet and So Should Your Software: Automatically Generating Software Use and Reuse Libraries and Catalogs from Standardized SAS® Code. *Midwest SAS Users Group (MWSUG)*. Retrieved from https://www.mwsug.org/proceedings/2016/TT/MWSUG-2016-TT09.pdf.

[iii] Troy Martin Hughes. 2016. *SAS® Data Analytic Development: Dimensions of Software Quality*. John Wiley and Sons, Inc. Hoboken, NJ.

[iv] ISO/Guide 73:2009. *Risk management—Vocabulary*. Geneva, Switzerland: International Organization for Standardization.

[v] *SAS® 9.4 Language Reference: Concepts, Sixth Edition*. Names in the SAS Language. SAS Institute. Retrieved from http://documentation.sas.com/?docsetId=lrcon&docsetTarget=p18cdcs4v5wd2dn1q0x296d3qek6.htm&docsetVersion=9.4&locale=en.

CONTACT INFORMATION

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

Name: Troy Martin Hughes
E-mail: troymartinhughes@gmail.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

APPENDIX A. THE SCAVENGER MACRO SUITE

```
%macro parsedir(dir= /* asterisk-delimited list of folders to parse */,
   dsn= /* data set in LIB.DSN format into which to put directory info */,
   subdir=YES /* YES to parse subdirectories, NO to only parse current DIR */,
   filetype=SAS EGP /* space-delimited list of file extensions to find */);
%let syscc=0;
%global parsedirRC;
%let parsedirRC=GLOBAL FAILURE;
%local subdirs i j ext;
%if %upcase(&subdir)=YES %then %let subdirs=/s;
%else %let subdirs=;
%let j=1;
%do %while(%length(%scan(&dir,&j,*))>1);
   %let disdir=%scan(&dir,&j,*);
   %if &j=1 %then %do;
       data &dsn;
           length dir_name $300 file_name $300 crdate 8;
           format crdate datetime17.;
           if ^missing(dir_name);
       run;
       %end;
   filename filelist pipe "dir &subdirs ""&disdir""";
   data temp&j (keep=dir_name file_name crdate);
       length dir_name $300 file_name $300 crdate 8 inp $600;
       format crdate datetime17.;
       infile filelist truncover;
       input inp $600.;
       if length(inp)>9 and substr(strip(inp),1,9)='Directory'
           then dir_name=strip(substr(strip(inp),13));
       else if lengthc(strip(inp))>0 and find(inp,'<DIR>')=0
           and not(find(inp,'Volume in drive')>0 and find(inp,'has no label')>0)
            and not(find(inp,'File(s)')>0 and find(inp,'bytes')>0)
           and not(find(inp,'Dir(s)')>0 and find(inp,'bytes')>0) then do;
                if
                %if %length(&filetype)>=1 %then %do;
                      %let i=1;
                      %do %while(%length(%scan(&filetype,&i,,S))>1);
                            %let ext=%scan(&filetype,&i,,S);
                            %if &i>1 %then or;
                            lowcase(scan(inp,-1,'.'))="%lowcase(&ext)"
                            %let i=%eval(&i+1);
                            %end;
                      %end;
                 %else %do;
                      1=1
                      %end;
               then do;
                   crdate=input(substr(inp,1,20),mdyampm20.);
                   file_name=strip(lowcase(substr(inp,39)));
                   output;
                   end;
               end;
       retain dir_name;
   run;
   proc append base=&dsn data=temp&j;
   run;
   %let j=%eval(&j+1);
   %end;
```

```
%if &syscc=0 %then %let parsedirRC=;
%mend;

%macro readxml(egpfile= /* folder and name of EGP file */,
   xmlfile=project.xml /* name of XML inside EGP file */);
%global totName totID totRefLoc;
%let totName=;
%let totID=;
%let totRefLoc=;
%let EGPnote=;
filename projzip zip "&egpfile" member="&xmlfile";
data _null_;
   infile projzip truncover encoding='utf-16be';
   input line $400.;
   length foundCode 8 foundElement 8 foundCodeTask 8 foundDNA 8
       programName $100 programID $40 programRefLoc $200
       totName $1000 totID $1000 totRefLoc $1000 sep $2
       EGPnote $1000;
   retain foundCode foundElement foundCodeTask foundDNA foundNote re
       programName programID programRefLoc totName totID totRefLoc EGPnote;
   if _n_=1 then do;
       foundCode=0;
       foundElement=0;
       foundCodeTask=0;
       foundDNA=0;
       foundNote=0;
       programName='';
       programID='';
       programRefLoc='';
       totName='';
       totID='';
       totRefLoc='';
       re=prxparse("s/<.*?>//");
       end;
   line=strip(line);
   if line=:'<Element Type="SAS.EG.ProjectElements.CodeTask">' then foundCode=1; *starts
element;
   else if foundCode and line=:'<Element>' then foundElement=1;
   else if foundCode and foundElement then do;
       if line=:'<Label>' then call prxchange(re,-1,line,programName);
       else if line=:'<ID>' then call prxchange(re,-1,line,programID);
       else if line=:'</Element>' then foundElement=0;
       end;
   else if foundCode and not foundElement then do;
       if not foundCodeTask then do;
           if line=:'<CodeTask>' then foundCodeTask=1;
           else if line=:'</Element>' then do; * terminates element;
              sep=ifc(length(totName)=1,'','*');
              totName=strip(totName) || sep || strip(programName);
              totID=strip(totID) || sep || strip(programID);
              totRefLoc=strip(totRefLoc) || sep || strip(programRefLoc);
              call symput('totName',strip(totName));
              call symput('totID',strip(totID));
              call symput('totRefLoc',strip(totRefLoc));
              foundCode=0;
              output;
              programName='';
              programID='';
              programRefLoc='';
              end;
```

```
            end;
        else if foundCodeTask then do;
            if line=:'<DNA>' then foundDNA=1;
            else if foundDNA and line=:'&lt;FullPath&gt;' then do;
                programRefLoc=substr(line,17,length(line)-34);
               end;
            else if line=:'</CodeTask>' then do;
                foundCodeTask=0;
                foundDNA=0;
               end;
           end;
       end;
   if line=:'<Element Type="SAS.EG.ProjectElements.Note">' then foundNote=1;
   else if foundNote and line=:'<Text>' then do;
       call prxchange(re,-1,line,EGPnote);
       call symput('EGPnote',strip(EGPnote));
       foundNote=0;
       end;
run;
%mend;

%macro readconfig(cfg= /* folder and name of configuration file */);
%global taglist lenslist desclist bracket bracketfile brackettags
   bracketlens bracketdesc;
%local i;
   filename cfgfile "&cfg";
   %if &sysfilrc=0 %then %do;
       data _null_;
           infile cfgfile truncover end=eof;
           length tags $2000 lens $2000 desc $2000 bracket $2000
               bracketfile $2000 brackettags $2000 bracketlens $2000
               bracketdesc $2000 inbracket 3 bracketcnt 3;
           input line $1000.;
           if _n_=1 then do;
               tags='';
               lens='';
               desc='';
               bracket='';
               bracketfile='';
               brackettags='';
               bracketlens='';
               bracketdesc='';
               inbracket=0;
               end;
           if lengthn(line)>0 then do;
               if substr(line,1,1)^='*' then do;
                   if strip(line)=:'[' then do;
                           inbracket=1;
                           bracketcnt=0;
                           bracket=catx('*',bracket,scan(substr(line,find(line,'<')+1)
                                   ,1,'>'));
                           bracketfile=catx('*',bracketfile,
                                   scan(substr(line,find(line,'<')+1),2,'>'));
                           end;
                   else if find(line,'<')>0 and find(line,'>')>0 and line=:'<' then do;
                           if missing(bracket) then do;
                                   tags=catx(' ',tags,'TAG_' || substr(line,2,find(line,'/')-
2));
                                   lens=catx(' ',lens,scan(substr(line,
                                           find(lowcase(line),'/len')+4),1,'=/>'));
```

```
                                desc=catx('*',desc,scan(substr(line,
                                        find(lowcase(line),'/len')),2,'/>]'));
                                end;
                            else do;
                                bracketcnt=bracketcnt+1;
                                brackettags=catx(' ',brackettags,'GRP_' || substr(line,2,
                                        find(line,'/')-2));
                                bracketlens=catx(' ',bracketlens,scan(substr(line,
                                        find(lowcase(line),'/len')+4),1,'=/>'));
                                bracketdesc=catx('*',bracketdesc,scan(substr(line,
                                        find(lowcase(line),'/len')),2,'/>]'));
                                end;
                            end;
                        if lengthn(line)>0 and strip(substr(line,length(line),1))=']'
                                then do;
                            inbracket=0;
                            bracket=catx('*',bracket,strip(put(bracketcnt,8.)));
                            end;
                        end;
                    end;
                if eof then do;
                    call symput('taglist',strip(tags));
                    call symput('lenslist',strip(lens));
                    call symput('desclist',strip(desc));
                    call symput('bracket',strip(bracket));
                    call symput('bracketfile',strip(bracketfile));
                    call symput('brackettags',strip(brackettags));
                    call symput('bracketlens',strip(bracketlens));
                    call symput('bracketdesc',strip(bracketdesc));
                    end;
                retain tags lens desc bracket bracketfile brackettags bracketlens
                    bracketdesc inbracket bracketcnt;
            run;
        * delete previous bracketed metadata if they exist from previous runs;
            %let i=1;
            %do %while(%length(%scan(&bracketfile,&i,*))>1);
                %if %sysfunc(exist(%scan(&bracketfile,&i,*))) %then %do;
                    proc delete data=%scan(&bracketfile,&i,*);
                    run;
                    %end;
                %let i=%eval(&i+1);
                %end;
            %end;
        %mend;

        %macro readcode(cfg= /* folder and name of configuration file */,
           infilename= /* file reference to the SAS program file */,
           dsn= /* metadata data set in LIB.DSN format */);
        %global taglist macrolist includelist globallist codelines;
        %local i j jtot var brack bracknum;
        * configuration file is read only the first time READCODE is called;
        %if &needtoreadconfig=YES %then %do;
           %readconfig(cfg=&cfg);
           %let needtoreadconfig=NO;
           %end;
        data &dsn.temp (keep=softwarename programname EGPnote savedate macrolist
            includelist globallist codelines
            %if %symexist(taglist) and %symexist(desclist) %then %do;
                %let i=1;
                %do %while(%length(%scan(&taglist,&i,,S))>0);
```

```
            %scan(&taglist,&i,,S)
            %let i=%eval(&i+1);
            %end;
        %end;
    )
%if %symexist(bracket) %then %do;
    %let i=1;
    %let jtot=1;
    %do %while(%length(%scan(&bracket,%sysevalf((&i*2)-1),*))>1);
        btag&i (keep=softwarename programname codeline
        %let brack=%scan(&bracket,&i,*);
        %let bracknum=%scan(&bracket,%sysevalf(&i*2),*);
        %do j=1 %to &bracknum;
            %scan(&brackettags,%eval(&jtot),,S)
            %let jtot=%eval(&jtot+1);
            %end;
        )
        %let i=%eval(&i+1);
        %end;
    %end;
    ;
infile &infilename truncover end=eof;
length i 8 notmiss 8 line $1000 softwarename $200 programname $200 EGPnote $1000
        savedate 8 macrolist $500 includelist $2000 globallist $500
        codelines 8 codeline 8 bracketcomment $500 leftgroup 8 templine $500
    %if %symexist(taglist) and %symexist(desclist) %then %do;
        %let i=1;
        %do %while(%length(%scan(&taglist,&i,,S))>0);
            %scan(&taglist,&i,,S) %scan(&lenslist,&i,,S)
            %let i=%eval(&i+1);
            %end;
        %end;
    %if %symexist(bracket) %then %do;
        %let i=1;
        %do %while(%length(%scan(&brackettags,&i,,S))>1);
            %scan(&brackettags,&i,,S) %scan(&bracketlens,&i,,S)
            %let i=%eval(&i+1);
            %end;
        %end;
    ;
format savedate datetime17.;
label softwarename='Project or Program Name' programname='Program Name'
        EGPnote='EG Note' savedate='Save Date' macrolist='Macros Defined'
        globallist='Global Macro Variables Defined'
        includelist='INCLUDE files' codelines='Code Lines' codeline 'Program Line'
    %if %symexist(bracket) %then %do;
        %let i=1;
        %do %while(%length(%scan(&brackettags,&i,,S))>1);
            %scan(&brackettags,&i,,S)="%scan(&bracketdesc,&i,*)"
            %let i=%eval(&i+1);
            %end;
        %end;
    ;
%if %length(&sasfilename)>0 %then %do; * ingest the SAS program;
    input line $1000.;
    if _n_=1 then do;
        macrolist='';
        includelist='';
        globallist='';
        %if %symexist(taglist) and %symexist(desclist) %then %do;
```

```
            %let i=1;
            %do %while(%length(%scan(&taglist,&i,,S))>0);
                %scan(&taglist,&i,,S)='';
                %let i=%eval(&i+1);
                %end;
            %end;
        * because 3 characters precede the first line in EG imbedded programs;
        if substr(line,1,1)=byte(239) and substr(line,2,1)=byte(187) and
            substr(line,3,1)=byte(191) then line=substr(line,4);
        end;
    codeline=_n_; * current line of code;
    * macro definitions;
    if lowcase(strip(compress(line,,'H')))=:'%macro' then do; * compress required to remove
preceding tabs;
        macrolist=catx(' ',macrolist,scan(strip(compress(line,,'H')),2));
        end;
    * macro inclusion;
    if lowcase(strip(compress(line,,'H')))=:'%include' then do;
        templine=scan(substr(strip(compress(line,,'H')),9),1,';/(');
        includelist=catx(' * ',includelist,dequote(strip(templine)));
        end;
    * global macro variables;
    if lowcase(strip(compress(line,,'H')))=:'%global' and find(line,';')>0 then do;
        templine=scan(substr(strip(compress(line,,'H')),8),1,';/');
        i=1;
        do while(length(scan(templine,i,,'S'))>1);
            globallist=catx(' ',globallist,scan(templine,i,,'S'));
            i=i+1;
            end;
        end;
    * configuration file singular tags;
    %if %symexist(taglist) and %symexist(desclist)
            and %length(&taglist)>0 and %length(&desclist)>0 %then %do;
        %let i=1;
        %do %while(%length(%scan(&taglist,&i,,S))>0);
            %let var=%substr(%scan(&taglist,&i,,S),5);
            if find(lowcase(compress(line)),"*<&var>") then do;
                TAG_&var=catx(' ',TAG_&var,reverse(substr(reverse(strip(substr(
                    line,find(lowcase(line),"<&var>")+length("&var")+2))),2)));
                end;
            %let i=%eval(&i+1);
            %end;
        %end;
    retain macrolist includelist globallist codelines
    %if %symexist(taglist) and %symexist(desclist)
            and %length(&taglist)>0 and %length(&desclist)>0 %then %do;
        %let i=1;
        %do %while(%length(%scan(&taglist,&i,,S))>0);
            %scan(&taglist,&i,,S)
            %let i=%eval(&i+1);
            %end;
        %end;;
    %if %symexist(taglist) and %symexist(desclist)
            and %length(&taglist)>0 and %length(&desclist)>0 %then %do;
        label
        %let i=1;
        %do %while(%length(%scan(&taglist,&i,,S))>0);
            %scan(&taglist,&i,,S)="%scan(&desclist,&i,*)"
            %let i=%eval(&i+1);
            %end;
```

```
      %end;;
   * configuration file grouped tags (in brackets);
   %if %symexist(brackettags) and %symexist(bracketdesc)
           and %length(&brackettags)>0 and %length(&bracketdesc)>0 %then %do;
      softwarename="&pathfile";
      programname="&sasfilename";
      savedate=&fileupdate;
      if lowcase(programname)^='scavenger' then do; * it should not recursively parse
itself;
         i=1;
         leftgroup=.; *position of the leftmost tag if one exists;
         do while(length(scan("&brackettags",i,,'S'))>1);
            if find(lowcase(line),'<' || substr(lowcase(scan("&brackettags",i,,'S')),5) ||
'>')>0 then do;
               if missing(leftgroup) or find(lowcase(line),'<' ||
substr(lowcase(scan("&brackettags",i,,'S')),5) || '>') < leftgroup
                     then leftgroup=find(lowcase(line),'<' ||
substr(lowcase(scan("&brackettags",i,,'S')),5) || '>');
               end;
            i=i+1;
            end;
         if ^missing(leftgroup) then do;
            bracketcomment=strip(compress(substr(line,leftgroup),';'));
            i=1;
            do while(length(scan(bracketcomment,i,'<>'))>1);
               %let i=1;
               %do %while(%length(%scan(&brackettags,&i,,S))>1);
                     if
lowcase(scan(bracketcomment,i,'<>'))="%substr(%lowcase(%scan(&brackettags,&i,,S)),5)" then
do;
                           %if %substr(%scan(&bracketlens,&i,,S),1,1)=$ %then %do;

   %scan(&brackettags,&i,,S)=strip(scan(bracketcomment,i+1,'<>'));
                           %end;
                           %else %do;

   %scan(&brackettags,&i,,S)=input(scan(bracketcomment,i+1,'<>'),$8.);
                           %end;
                           end;
                     %let i=%eval(&i+1);
                     %end;
               i=i+2;
               end;
            %let i=1;
            %let jtot=1;
            %do %while(%length(%scan(&bracket,%sysevalf((&i*2)-1),*))>1);
               notmiss=0;
               %let brack=%scan(&bracket,&i,*);
               %let bracknum=%scan(&bracket,%sysevalf(&i*2),*);
               %do j=1 %to &bracknum;
                     if ^missing(%scan(&brackettags,%eval(&jtot),,S)) then notmiss=1;
                     %let jtot=%eval(&jtot+1);
                     %end;
               * output when metadata are relevant to metadata file;
               if notmiss=1 then output btag&i;
               %let i=%eval(&i+1);
               %end;
            end;
         end;
      %end;
```

```
   * output metadata;
   if eof then do;
   %end; * create the following variables even for linked EGP programs;
       softwarename="&pathfile";
       programname="&sasfilename";
       savedate=&fileupdate;
       EGPnote="&EGPnote";
       codelines=_n_;
       %if %length(&sasfilename)=0 %then %do;
           includelist="&linkedfile";
           %end;
       output &dsn.temp;
   %if %length(&sasfilename)>0 %then %do;
       end;
       %end;
run;
* append temporary data set to primary data set;
%if %sysfunc(exist(&dsn))=0 %then %do;
   data &dsn;
       set &dsn.temp;
   run;
   %end;
%else %do;
   proc append base=&dsn data=&dsn.temp;
   run;
   %end;
%let i=1;
%do %while(%length(%scan(&bracketfile,&i,*))>1);
   %let brack=%scan(&bracketfile,&i,*);
   %if %sysfunc(exist(&brack))=0 %then %do;
       data &brack;
           set btag&i;
       run;
       %end;
   %else %do;
       proc append base=&brack data=btag&i;
       run;
       %end;
   %let i=%eval(&i+1);
   %end;
%mend;

%macro scavenger(dir= /* folder to be parsed */,
   cfg= /* folder and name of configuration file */,
   dsnout= /* metadata data set in LIB.DSN format (which is first deleted) */,
   subdir=YES /* YES to parse subdirectories, NO to not */,
   filetype=SAS EGP /* space-delimited list of all file extensions to find */);
%global sasfilename fileupdate pathfile linkedfile EGPnote needtoreadconfig;
%let sasfilename=;
%let fileupdate=;
%let linkedfile=;
%let needtoreadconfig=YES;
%local i j dsid fil closed;
%if %sysfunc(exist(&dsnout)) %then %do;
   proc delete data=&dsnout;
   run;
   %end;
%parsedir(dir=&dir, dsn=filelist, subdir=&subdir, filetype=&filetype);
%let dsid=%sysfunc(open(filelist,i));
%if &dsid>0 %then %do;
```

```
   %let i=1;
   %do %while(%sysfunc(fetchobs(&dsid,&i))=0);
       %let path=%sysfunc(strip(%sysfunc(getvarc(&dsid,1))));
       %let fil=%sysfunc(strip(%sysfunc(getvarc(&dsid,2))));
       %let fileupdate=%sysfunc(strip(%sysfunc(getvarn(&dsid,3))));
       %let pathfile=&path\&fil;
       %if %lowcase(%scan(&pathfile,-1,.))=egp %then %do;
           %readxml(egpfile=&pathfile, xmlfile=project.xml);
           %let j=1;
           %do %while(%length(%scan(&totName,&j,*))>1);
               %let sasfilename=%scan(&totName,&j,*);
               %if %length(%scan(&totRefLoc,&j,*,m))=0 %then %do;
                   *read imbedded SAS program;
                   filename sascode zip "&pathfile"
member="%sysfunc(strip(%scan(&totID,&j,*,m)))/code.sas";
                   %readcode(cfg=&cfg, infilename=sascode, dsn=&dsnout);
                   %end;
               %else %do; * linked SAS programs are not read;
                   %let sasfilename=;
                   %let linkedfile=%scan(&totRefLoc,&j,*,m);
                   %readcode(cfg=&cfg, infilename=sascode, dsn=&dsnout);
                   %end;
               %let j=%eval(&j+1);
               %end;
           %end;
       %else %if %lowcase(%scan(&pathfile,-1,.))=sas %then %do;
           %let sasfilename=%substr(&fil,1,%length(&fil)-4);
           %let EGPnote=;
           filename sascode "&pathfile";
           %readcode(cfg=&cfg, infilename=sascode, dsn=&dsnout);
           %end;
       %let i=%eval(&i+1);
       %end;
   %let closed=%sysfunc(close(&dsid));
   %end;
%mend;
```