Contents

0.1 __call.cpp

```
#include "__call.h"
#include "Exception.h"
#include "MString.h"
#include <ostream>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

namespace jet {

   __call::__call(coreutils::ZString &in, coreutils::MString &parentOut, Global
       &global, Tag *parent, Tag *local) : Tag(in, parentOut, global, parent,
       local) {
      if(hasContainer)
        throw coreutils::Exception("call␣tag␣cannot␣have␣a␣container.");
      if(!keywordDefined("pgm"))
        throw coreutils::Exception("pgm␣keyword␣must␣be␣specified.");
      for(ix = 0; ix <= 50; ++ix)
        argv[ix] = NULL;
      argv[0] = resolveKeyword("pgm").c_str(); // TODO: Need to peel off the
          program name only and pass as argv[0].
      for(ix = 1; ix <= 50; ++ix) {
         coreutils::MString arg("arg");
         arg << ix;
         if(keywordDefined(arg)) {
           argv[ix] = resolveKeyword(arg).c_str();
         } else
           break;
      }
      pipe(fdo);
      pid = fork();
      if(pid == 0) {
         close(fdo[0]);
         dup2(fdo[1], 1);
         if(keywordDefined("input")) {
            coreutils::MString input(resolveKeyword("input"));
            pipe(fdi);
            if(fork() == 0) {
               close(fdi[0]);
               write(fdi[1], input.getData(), input.getLength());
               close(fdi[1]);
               exit(0);
            }
            close(fdi[1]);
            dup2(fdi[0], 0);
         }
         rc = execvpe(resolveKeyword("pgm").c_str(), argv, global.envp);
         close(fdo[1]);
         exit(errno);
      }
      close(fdo[1]);
      if(keywordDefined("name")) {
         coreutils::MString value;
         value.read(fdo[0]);
```

```
            storeVariable(resolveKeyword("name"), value, resolveKeyword("scope"));
        } else
            out.read(fdo[0]);
        waitpid(pid, &status, 0);
        if(keywordDefined("error")) {
            global.variables[resolveKeyword("error")] = (status >> 8 & 255);
        }
    }

}
```

0.2 __call.h

```
#ifndef ____call_h__
#define ____call_h__

#include "Tag.h"

namespace jet {

   class __call : public Tag {

    public:
      __call(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

    private:
      int pid;
      int status;
      int ix;
      int fdi[2];
      int fdo[2];
      int rc;
      char *argv[50];

   };

}

#endif
```

0.3 __comment.cpp

```
#include "__comment.h"
#include "Exception.h"

namespace jet {

   __comment::__comment(coreutils::ZString &in, coreutils::MString &parentOut,
       Global &global, Tag *parent, Tag *local) : Tag(in, parentOut, global,
       parent, this) {
      if(!hasContainer)
        throw coreutils::Exception("comment must have a container.");
      output = false;
   }

}
```

0.4 __comment.h

```
#ifndef ____comment_h__
#define ____comment_h__

#include "Tag.h"

namespace jet {

   class __comment : public Tag {

    public:
      __comment(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

   };

}

#endif
```

0.5 __dotag.cpp

```
#include "__dotag.h"
#include "Exception.h"

namespace jet {

   __dotag::__dotag(coreutils::ZString &in, coreutils::MString &parentOut,
       Global &global, Tag *parent, Tag *local) : Tag(in, parentOut, global,
       parent, local) {
      coreutils::ZString container3 = global.tags[name];
      processContainer(container3, container);
   }
}
```

0.6 __dotag.h

```
#ifndef ____dotag_h__
#define ____dotag_h__

#include "Tag.h"
#include "ZString.h"

namespace jet {

   class __dotag : public Tag {

    public:
      __dotag(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

   };

}

#endif
```

0.7 __dump.cpp

```
#include "__dump.h"
#include "Exception.h"
#include <iostream>
#include <fstream>

namespace jet {

   __dump::__dump(coreutils::ZString &in, coreutils::MString &parentOut, Global
       &global, Tag *parent, Tag *local) : Tag(in, parentOut, global, parent,
       local) {
      if(!keywordDefined("file"))
        throw coreutils::Exception("file must be sppecified for dump tag.");

      std::ofstream outFile(keywords["file"].str());

      if(global.cgi) {
         outFile << "*** CGI VARIABLES ***" << std::endl;
         for (auto i = global.cgiVariables.begin(); i != global.cgiVariables.end
             (); i++)
           outFile << i->first << "=[" << i->second << "]" << std::endl;
      }

      outFile << "*** GLOBAL VARIABLES ***" << std::endl;
      for (auto i = global.variables.begin(); i != global.variables.end(); i++)
        outFile << i->first << "=[" << i->second << "]" << std::endl;

      outFile << "*** LOCAL VARIABLES ***" << std::endl;
      for (auto i = local->variables.begin(); i != local->variables.end(); i++)
        outFile << i->first << "=[" << i->second << "]" << std::endl;

      outFile << "*** KEYWORD VALUES ***" << std::endl;
      for (auto i = parent->keywords.begin(); i != parent->keywords.end(); i++)
        outFile << i->first << "=[" << i->second << "]" << std::endl;

      outFile << "*** COOKIES ***" << std::endl;
      for (auto i = global.cookies.data.begin(); i != global.cookies.data.end();
           i++)
        outFile << i->first << "=[" << i->second << "]" << std::endl;

      outFile.close();

   }

}
```

0.8 __dump.h

```
#ifndef ____dump_h__
#define ____dump_h__

#include "Tag.h"
#include "MString.h"
#include "Global.h"

namespace jet {

   class __dump : public Tag {

    public:
      __dump(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

   };

}

#endif
```

0.9 __for.cpp

```
#include "__for.h"
#include "Exception.h"
#include <iostream>

namespace jet {

   __for::__for(coreutils::ZString &in, coreutils::MString &parentOut, Global &
       global, Tag *parent, Tag *local) : Tag(in, parentOut, global, parent, this
       ) {
      double counter = 0.0f;
      bool nameDefined = keywordDefined("name");
      if(keywordDefined("start")) {
         counter = resolveKeyword("start").asDouble();
         keywords["start"].reset();
      }
      if(!keywordDefined("end"))
        throw coreutils::Exception("for␣tag␣requires␣end␣keyword.");
      if(!keywordDefined("step"))
        throw coreutils::Exception("for␣tag␣requires␣step␣keyword.");
      for(double ix = counter; ix <= resolveKeyword("end").asDouble(); ix +=
          resolveKeyword("step").asDouble()) {
         resolveKeyword("end").reset();
         resolveKeyword("step").reset();
         if(nameDefined) {
            if(!keywordDefined("scope") || (resolveKeyword("scope") == "global")
                )
              global.variables[resolveKeyword("name")] = ix;
            else if(resolveKeyword("scope") == "local")
              this->local->variables[resolveKeyword("name")] = ix;
            else if(resolveKeyword("scope") == "parent")
              parent->local->variables[resolveKeyword("name")] = ix;
            else
              throw coreutils::Exception("scope␣value␣is␣not␣valid.");
         }
         processContainer(container);
         container.reset();
      }
   }

}
```

0.10 __for.cpp

```
#include "__for.h"
#include "Exception.h"
#include <iostream>

namespace jet {

   __for::__for(coreutils::ZString &in, coreutils::MString &parentOut, Global &
       global, Tag *parent, Tag *local) : Tag(in, parentOut, global, parent, this
       ) {
      double counter = 0.0f;
      bool nameDefined = keywordDefined("name");
      if(keywordDefined("start")) {
         counter = resolveKeyword("start").asDouble();
         keywords["start"].reset();
      }
      if(!keywordDefined("end"))
        throw coreutils::Exception("for␣tag␣requires␣end␣keyword.");
      if(!keywordDefined("step"))
        throw coreutils::Exception("for␣tag␣requires␣step␣keyword.");
      for(double ix = counter; ix <= resolveKeyword("end").asDouble(); ix +=
          resolveKeyword("step").asDouble()) {
         resolveKeyword("end").reset();
         resolveKeyword("step").reset();
         if(nameDefined) {
            if(!keywordDefined("scope") || (resolveKeyword("scope") == "global")
                )
              global.variables[resolveKeyword("name")] = ix;
            else if(resolveKeyword("scope") == "local")
              this->local->variables[resolveKeyword("name")] = ix;
            else if(resolveKeyword("scope") == "parent")
              parent->local->variables[resolveKeyword("name")] = ix;
            else
              throw coreutils::Exception("scope␣value␣is␣not␣valid.");
         }
         processContainer(container);
         container.reset();
      }
   }

}
```

0.11 __for.h

```
#ifndef ____for_h__
#define ____for_h__

#include "Tag.h"
#include <sstream>

namespace jet {

   class __for : public Tag {

    public:
      __for(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

   };

}

#endif
```

0.12 Global.cpp

```
#include "Global.h"
#include "Exception.h"
#include "__mysql.h"
#include <iostream>
#include <stdlib.h>

namespace jet {

   Global::Global(char **envp) : envp(envp) {

   }

   Global::~Global() {

   }

   void Global::dump() {
      for (auto i = variables.begin(); i != variables.end(); i++)
        std::cout << i->first << "=[" << i->second << "]" << std::endl;
   }

   bool Global::sessionExists(coreutils::MString sessionId) {
      return sessions.find(sessionId) != sessions.end();
   }

   void Global::addSession(coreutils::MString sessionId, __mysql *mysql) {
      if(sessionExists(sessionId))
        coreutils::Exception("sessionid␣already␣exists.");
      sessions[sessionId] = mysql;
   }

   void Global::removeSession(coreutils::MString sessionId) {
      sessions.erase(sessionId);
   }

   __mysql * Global::getSession(coreutils::MString sessionId) {
      if(sessions.find(sessionId) == sessions.end())
        throw coreutils::Exception("requested␣session␣is␣not␣available.");
      return sessions[sessionId];
   }

   coreutils::ZString Global::getSessionVariable(coreutils::MString &splitName)
       {
      if(sessions.find(splitName[0]) == sessions.end())
        throw coreutils::Exception("requested␣session␣is␣not␣available␣in␣
            variable.");
      return sessions[splitName[0]]->getColumnValue(splitName[1]);
   }

   void Global::outputHeaders() {
      if(headers.size() > 0) {
         for(auto header = headers.begin();
             header != headers.end();
             ++header) {
            std::cout << header->first << ":␣" << header->second << std::endl;
```

```
      }
      std::cout << std::endl;
   }
}

void Global::setupFormData(coreutils::ZString &formdata) {
   coreutils::ZString boundary = formdata.goeol();
   while(!formdata.eod()) {
      if(formdata.ifNext("Content-Disposition:␣form-data;")) {
         formdata.skipWhitespace();
         if(formdata.ifNext("name=\"")) {
            coreutils::ZString name = formdata.getTokenInclude("
                ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
                ._-");
            if(formdata.ifNext("\"")) {
               formdata.goeol();
               formdata.goeol();
               coreutils::ZString data = formdata.getTokenExclude("-"); //
                   TODO: Fix this parsing. Need a string exclusion method to
                   check for 'boundary'.
               data.trimCRLF();
               formdata.ifNext(boundary);
               int index = 0;
               coreutils::MString namex;
               do {
                  namex = "";
                  namex << name << ":" << index++;
               } while(cgiVariables.count(namex) != 0);
               cgiVariables[namex] = data;
               if(formdata.ifNext("--"))
                 break;
               formdata.goeol();
            }
            else
              throw coreutils::Exception("expecting␣closing␣double␣quote␣on␣
                  variable␣name␣in␣received␣CGI␣data.");
         } else
           throw coreutils::Exception("expecting␣name␣subfield␣in␣received␣
               CGI␣data.");
      } else
        throw coreutils::Exception("expecting␣Content-Disposition␣header␣in␣
            received␣CGI␣data.");
   }
}

void Global::setupFormURLEncoded(coreutils::ZString &formdata) {
   while(!formdata.eod()) {
      coreutils::ZString name = formdata.getTokenInclude("
          ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789._-");
      if(formdata.ifNext("=")) {
         coreutils::MString data = formdata.getTokenExclude("&");
         formdata.ifNext("&");
         int index = 0;
         coreutils::MString namex;
         do {
            namex = "";
            namex << name << ":" << index++;
```

```
            } while(cgiVariables.count(namex) != 0);
            cgiVariables[namex] = data.fromCGI();
          } else
            throw coreutils::Exception("expecting␣=␣after␣name␣in␣received␣CGI␣
                data.");
        }
    }

}
```

0.13 Global.h

```
#ifndef __Global_h__
# define __Global_h__

# include "MString.h"
# include "CGIFormattedData.h"
# include <map>

namespace  jet {

   class __mysql;

   class Global {

    public:
      Global(char **envp);
      virtual ~Global();

      void dump();
      bool sessionExists(coreutils::MString sessionId);
      void addSession(coreutils::MString sessionId, __mysql *mysql);
      void removeSession(coreutils::MString sessionId);
      __mysql * getSession(coreutils::MString sessionId);
      coreutils::ZString getSessionVariable(coreutils::MString &splitName);
      void outputHeaders();
      void setupFormData(coreutils::ZString &formdata);
      void setupFormURLEncoded(coreutils::ZString &formdata);
      char *errorCursor = NULL;

      coreutils::CGIFormattedData cookies;
      std::map<coreutils::MString, coreutils::MString> variables;
      std::map<coreutils::MString, coreutils::MString> cgiVariables;
      std::map<coreutils::MString, __mysql *> sessions;
      std::map<coreutils::MString, coreutils::MString> headers;
      std::map<coreutils::MString, coreutils::MString> tags;
      char **envp;
      bool cgi = false;
      bool session = false;
      coreutils::MString sessionId;

   };

}

#endif
```

0.14 __header.cpp

```
#include "__header.h"
#include "Exception.h"
#include "Operand.h"
#include <iostream>

namespace jet {

   __header::__header(coreutils::ZString &in, coreutils::MString &parentOut,
       Global &global, Tag *parent, Tag *local) : Tag(in, parentOut, global,
       parent, local) {
      output = false;
      if(!global.cgi)
        throw coreutils::Exception("must enable cgi mode on jet tag to use
            header tag.");
      if(!keywordDefined("name"))
        throw coreutils::Exception("header tag must have name defined.");
      if(!keywordDefined("expr") && keywordDefined("value") && hasContainer)
        throw coreutils::Exception("header tag cannot have both value and a
            container.");
      if(keywordDefined("expr") && !keywordDefined("value") && hasContainer)
        throw coreutils::Exception("header tag cannot have both expr and a
            container.");
      if(keywordDefined("expr") && keywordDefined("value") && !hasContainer)
        throw coreutils::Exception("header tag cannot have both expr and value."
            );
      if(!keywordDefined("expr") && !keywordDefined("value") && !hasContainer)
        throw coreutils::Exception("header tag must have a value, expr or a
            container.");
      if(keywordDefined("expr")) {
         if(keywordDefined("eval"))
           throw coreutils::Exception("Cannot use eval with expr.");
         global.headers[resolveKeyword("name")] = Operand(keywords["expr"], *
             this).string;
      } else if(hasContainer) {
         processContainer(container);
         if(evaluate) {
            global.headers[resolveKeyword("name")] = out;
         } else {
            global.headers[resolveKeyword("name")] = container;
         }
      } else
        global.headers[resolveKeyword("name")] = resolveKeyword("value");

   }
}
```

0.15 __header.h

```
#ifndef ____header_h__
#define ____header_h__

#include "Tag.h"
#include "ZString.h"
#include "MString.h"
#include <sstream>

namespace jet {

   class __header : public Tag {

    public:
      __header(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

    protected:

   };

}

#endif
```

0.16 __if.cpp

```
#include "__if.h"
#include "Exception.h"
#include <iostream>
#include "Operand.h"

namespace jet {

   __if::__if(coreutils::ZString &in, coreutils::MString &parentOut, Global &
       global, Tag *parent, Tag *local) : Tag(in, parentOut, global, parent, this
       , "else") {
      coreutils::MString result;
      bool booleanResult = false;
      if(keywordDefined("value1")) {
         if(keywordDefined("expr"))
           throw coreutils::Exception("Either␣value1␣or␣expr␣can␣be␣specified␣
               but␣not␣both.");
         if(keywordDefined("value2")) {
            if(!keywordDefined("type"))
              throw coreutils::Exception("type␣expected␣if␣value1␣and␣value2␣
                  specified.");
         } else
           throw coreutils::Exception("value2␣required␣if␣value1␣specified.");
         coreutils::MString type = resolveKeyword("type");
         if((type != "eq") &&
            (type != "ne") &&
            (type != "lt") &&
            (type != "le") &&
            (type != "gt") &&
            (type != "ge"))
           throw coreutils::Exception("type␣value␣must␣be␣'eq','ne','lt','le','
               gt','ge'.");
         int rc = resolveKeyword("value1").compare(resolveKeyword("value2"));
//       std::cout << "if: " << resolveKeyword("value1") << " " << type << " "
    << resolveKeyword("value2") << ":" << rc << std::endl;
         if(((type == "eq") && (rc == 0)) ||
            ((type == "ne") && (rc != 0)) ||
            ((type == "lt") && (rc == -1)) ||
            ((type == "le") && (rc != 1)) ||
            ((type == "gt") && (rc == 1)) ||
            ((type == "ge") && (rc != -1)))
           booleanResult = true;
         else
           booleanResult = false;
      } else if(keywordDefined("expr")) {
         if(keywordDefined("value2"))
           throw coreutils::Exception("value2␣should␣not␣be␣specified␣with␣expr.
               ");
         if(keywordDefined("type"))
           throw coreutils::Exception("type␣should␣not␣be␣specified␣with␣expr.")
               ;
         booleanResult = Operand(keywords["expr"], *this).boolean;
      }
      if(booleanResult)
        processContainer(container);
      else if(hasContainer2)
```

```
        processContainer(container2);
    }

}
```

0.17 __if.h

```
#ifndef ____if_h__
#define ____if_h__

#include "Tag.h"
#include "ZString.h"
#include "MString.h"
#include <sstream>

namespace jet {

   class __if : public Tag {

    public:
      __if(coreutils::ZString &in, coreutils::MString &parentOut, Global &global
          , Tag *parent, Tag *local);

   };

}

#endif
```

0.18 __ifrow.cpp

```
#include "__ifrow.h"
#include "Exception.h"
#include "MString.h"
#include "__mysql.h"
#include <stdlib.h>
#include <unistd.h>

namespace jet {

   __ifrow::__ifrow(coreutils::ZString &in, coreutils::MString &parentOut,
       Global &global, Tag *parent, Tag *local) : Tag(in, parentOut, global,
       parent, this, "else") {
      output = false;
      if(!hasContainer)
        throw coreutils::Exception("ifrow tag must have a container.");
      if(!global.sessionExists(keywords[resolveKeyword("sessionid")]))
        throw coreutils::Exception("sessionid does not exist.");
      if(global.getSession(keywords[resolveKeyword("sessionid")])->hasRow())
        processContainer(container);
      else
        processContainer(container2);
   }

}
```

0.19 __ifrow.h

```
#ifndef ____ifrow_h__
#define ____ifrow_h__

#include "Tag.h"

namespace jet {

   class __ifrow : public Tag {

    public:
      __ifrow(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

   };

}

#endif
```

0.20 __include.cpp

```
#include "__include.h"
#include "Exception.h"
#include "File.h"

namespace jet {

   __include::__include(coreutils::ZString &in, coreutils::MString &parentOut,
       Global &global, Tag *parent, Tag *local) : Tag(in, parentOut, global,
       parent, local) {
      if(!keywordDefined("file"))
        throw coreutils::Exception("file␣keyword␣must␣be␣specified.");
      if(hasContainer)
        throw coreutils::Exception("include␣tag␣should␣not␣have␣a␣container.");
      hasContainer = true;
      coreutils::File file(resolveKeyword("file"));
      file.read();
      container = file.asZString();
      try {
         processContainer(container);
      }
      catch(coreutils::Exception e) {
         container.setCursor(global.errorCursor);
         container.moveBackToLineStart();
         std::cout << "----------------------------" << std::endl;
         std::cout << "Error␣in␣included␣script␣'" << variables["file"] << "'␣at
             ␣line␣" << container.getLineNumberAtCursor() << std::endl;
         std::cout << "Error␣text:␣" << e.text << std::endl;
         std::cout << "----------------------------" << std::endl;
         std::cout << container.parsed() << std::endl;
         std::cout << "********␣Error␣caught:␣" << e.text << std::endl;
         std::cout << container.unparsed() << std::endl;
         global.errorCursor = in.getCursor();
         throw coreutils::Exception("error␣in␣included␣file.");
      }

   }

}
```

0.21 __include.h

```
#ifndef ____include_h__
#define ____include_h__

#include "Tag.h"

namespace jet {

   class __include : public Tag {

    public:
      __include(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

   };

}

#endif
```

0.22 jet-2.0.cpp

```
#include <iostream>
#include <sstream>
#include "File.h"
#include "Global.h"
#include "Exception.h"
#include "__jet.h"

int main(int argc, char **argv, char **envp) {

   coreutils::File script(argv[1]);
   script.read();
   coreutils::ZString data = script.asZString();
   data.goeol();

   jet::Global global(envp);

   try {
      coreutils::MString out;
      global.errorCursor = data.getCursor();
      jet::__jet *jet = new jet::__jet(data, out, global, NULL, NULL);
      delete jet;
      global.outputHeaders();
      std::cout << out;
   }
   catch(coreutils::Exception e) {
      data.setCursor(global.errorCursor);
      data.moveBackToLineStart();
      std::cout << "------------------------------" << std::endl;
      std::cout << "Error␣in␣jet␣script␣'" << argv[1] << "'␣at␣line␣" << data.
          getLineNumberAtCursor() << std::endl;
      std::cout << "Error␣text:␣" << e.text << std::endl;
      std::cout << "------------------------------" << std::endl;
      std::cout << data.parsed() << std::endl;
      std::cout << "********␣Error␣caught:␣" << e.text << std::endl;
      std::cout << data.unparsed() << std::endl;
      global.dump();
   }

}
```

0.23 __jet.cpp

```
#include "__jet.h"
#include "Exception.h"
#include "Global.h"
#include "SessionId.h"
#include <iostream>
#include <fstream>

namespace jet {

   __jet::__jet(coreutils::ZString &in, coreutils::MString &parentOut, Global &
       global, Tag *parent, Tag *local) : Tag(in, parentOut, global, parent, this
       ) {
      char *cookies;
      if(keywordDefined("cgi") && (resolveKeyword("cgi") == "true")) {
         global.cgi = true;

         if(keywordDefined("sessiondir")) {
            global.session = true;
            global.cookies = getenv("HTTP_COOKIE");

//             if request_has_cookie then
//               pull sessionfile from sessiondir.
//               if last activity time is expired then ignore.
//               follow sessiontimeoutredirecturl.
//             else
               SessionId sessionId;
               global.headers["Set-Cookie"] << "session=" << sessionId;
               if(keywordDefined("sessiontimeout")) {
                  time_t timeout = time(0) + keywords["sessiontimeout"].asInteger()
                      ;
               }

//                   also save last activity time in session file.

         }

         coreutils::ZString requestMethod(getenv("REQUEST_METHOD"));
         if(requestMethod == "POST") {
            coreutils::ZString contentLength(getenv("CONTENT_LENGTH"));
            coreutils::ZString contentType(getenv("CONTENT_TYPE"));

            if(contentType == "multipart/form-data")
               coreutils::IMFMultipart postdata(0, "");

//             else if(contentType == "application/x-www-form-urlencoded")
//               global.setupFormURLEncoded(postdata);
         }
      }
      processContainer(container, NULL, true);
   }
}
```

0.24 __jet.h

```
#ifndef ____jet_h__
#define ____jet_h__

#include "Tag.h"
#include "ZString.h"
#include "IMFRequest.h"
#include "IMFMessage.h"
#include <sstream>

namespace jet {

   class __jet : public Tag {

    public:
      __jet(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

   };

}

#endif
```

0.25 KeywordValue.cpp

0.26 KeywordValue.h

0.27 Modifiers.cpp

0.28 Modifiers.h

0.29 __mysql.cpp

```
#include "__mysql.h"
#include "Exception.h"
#include <iostream>

namespace jet {

   __mysql::__mysql(coreutils::ZString &in, coreutils::MString &parentOut,
       Global &global, Tag *parent, Tag *local) : Tag(in, parentOut, global,
       parent, this) {

      if(!keywordDefined("host"))
        throw coreutils::Exception("host must be specified for mysql tag.");
      if(!keywordDefined("database"))
        throw coreutils::Exception("database must be specified for mysql tag.");
      if(!keywordDefined("user"))
        throw coreutils::Exception("user must be specified for mysql tag.");
      if(!keywordDefined("password"))
        throw coreutils::Exception("password must be specified for mysql tag.");

      sessionId = keywords[resolveKeyword("sessionid")];

      global.addSession(sessionId, this);

      mysql = mysql_init(NULL);
      mysql = mysql_real_connect(mysql,
                                 keywords[resolveKeyword("host")].c_str(),
                                 keywords[resolveKeyword("user")].c_str(),
                                 keywords[resolveKeyword("password")].c_str(),
                                 keywords[resolveKeyword("database")].c_str(),
                                     0, NULL, 0);

      if(!mysql)
        throw coreutils::Exception("database and host parameters are not valid."
            );

      processContainer(container);

   }

   __mysql::~__mysql() {
      global.removeSession(sessionId);
      mysql_free_result(result);
      mysql_close(mysql);
   }

   void __mysql::query(coreutils::MString query) {
      int rc = mysql_real_query(mysql, query.getData(), query.getLength());
      result = mysql_store_result(mysql);
      if(result) {
         row = mysql_fetch_row(result);
         fieldLength = mysql_fetch_lengths(result);
         qFields = mysql_num_fields(result);
      }
   }
```

```
   void __mysql::nextRow() {
      row = mysql_fetch_row(result);
      fieldLength = mysql_fetch_lengths(result);
   }

   bool __mysql::hasRow() {
      return row != NULL;
   }

   coreutils::ZString __mysql::getColumnValue(coreutils::ZString column) {
      MYSQL_FIELD *field;
      if(column == "?#") {
         nbrOfColumns = (int)qFields;
         return nbrOfColumns;
      } else if(column.ifNext("#")) {
         if(column.eod()) {
            nbrOfRows = (int)mysql_num_rows(result);
            return nbrOfRows;
         } else {
            std::cout << "[" << column.unparsed() << "]" << std::endl;
            int index = column.asInteger();
            std::cout << "integer: " << index << std::endl;
            field = mysql_fetch_field_direct(result, index - 1);
            return coreutils::ZString(field->name);
         }
      }

      for(int ix = 0; ix < qFields; ++ix) {
         field = mysql_fetch_field_direct(result, ix);
         if(column.equals((char *)field->name)) {
            return coreutils::ZString(row[ix], fieldLength[ix]);
         }
      }
      throw coreutils::Exception("column does not exist in session result.");
   }

}
```

0.30 __mysql.h

```
#ifndef ____mysql_h__
#define ____mysql_h__

#include "Tag.h"
#include "ZString.h"
#include "MString.h"
#include <sstream>
#include <mysql/mysql.h>

namespace jet {

   class __mysql : public Tag {

    public:
      __mysql(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);
      ~__mysql();

      void query(coreutils::MString query);
      void nextRow();
      bool hasRow();
      coreutils::ZString getColumnValue(coreutils::ZString column);

    private:
      MYSQL *mysql;
      MYSQL_RES *result;
      MYSQL_ROW row;
      unsigned long *fieldLength;
      unsigned int qFields;
      coreutils::MString sessionId;

      coreutils::MString nbrOfRows = "0";
      coreutils::MString nbrOfColumns = "0";

   };

}

#endif
```

0.31 Operand.cpp

```
#include "Operand.h"
#include "Exception.h"
#include <format>
#include <iostream>
#include <time.h>

namespace jet {

   Operand::Operand(coreutils::ZString &in, Tag &tag) {

      doubleValue = 0;

      in.skipWhitespace();

      if(in.startsWith("$[") || in.startsWith("#[")) {
         string = tag.getVariable(in);
         doubleValue = string.asDouble();
         isNumber = string.eod();
         string.reset();
         if((string == "false") || (string == "true"))
           boolean = true;
      } else if(in.ifNext("(")) {
         Operand op(in, tag);
         string = op.string;
         doubleValue = op.doubleValue;
         isNumber = op.isNumber;
         if(!in.ifNext(")"))
           throw coreutils::Exception("expected␣)␣in␣expression.");
      } else if(in.ifNextIgnoreCase("SUBSTRING")) {
         if(!in.ifNext("("))
           throw coreutils::Exception("Expecting␣(␣for␣SUBSTRING␣parameters.");
         Operand parm1(in, tag);
         if(!in.ifNext(","))
           throw coreutils::Exception("Expecting␣,␣in␣SUBSTRING␣expression.");
         Operand parm2(in, tag);
         if(in.ifNext(")")) {
            string = parm1.string.substring(parm2.string.asInteger());
         } else if(!in.ifNext(","))
           throw coreutils::Exception("Expecting␣,␣in␣SUBSTRING␣expression.");
         Operand parm3(in, tag);
         if(in.ifNext(")")) {
            string = parm1.string.substring(parm2.string.asInteger(), parm3.
                string.asInteger());
         } else
           throw coreutils::Exception("Expecting␣)␣at␣end␣of␣substring␣
               expression.");
      } else if(in.ifNextIgnoreCase("LEFT")) {
         if(!in.ifNext("("))
           throw coreutils::Exception("Expecting␣(␣for␣LEFT␣parameters.");
         Operand parm1(in, tag);
         if(!in.ifNext(","))
           throw coreutils::Exception("Expecting␣,␣in␣LEFT␣expression.");
         Operand parm2(in, tag);
         if(in.ifNext(")")) {
            string = parm1.string.substring(0, parm2.string.asInteger());
```

```
    } else
      throw coreutils::Exception("Expecting␣)␣at␣end␣of␣LEFT␣expression.");

} else if(in.ifNextIgnoreCase("EXPR")) {
    if(!in.ifNext("("))
      throw coreutils::Exception("Expecting␣(␣for␣EXPR␣parameters.");
    Operand parm1(in, tag);
    if(in.ifNext(")")) {
       Operand op(parm1.string, tag);
       string = op.string;
       isNumber = op.isNumber;
       boolean = op.boolean;
    } else
      throw coreutils::Exception("Expecting␣)␣at␣end␣of␣EXPR␣expression.");

} else if(in.ifNextIgnoreCase("RIGHT")) {
    if(!in.ifNext("("))
      throw coreutils::Exception("Expecting␣(␣for␣RIGHT␣parameters.");
    Operand parm1(in, tag);
    if(!in.ifNext(","))
      throw coreutils::Exception("Expecting␣,␣in␣RIGHT␣expression.");
    Operand parm2(in, tag);
    if(in.ifNext(")")) {
       int len = parm1.string.getLength();
       int size = parm2.string.asInteger();
       int start = len - size;
       string = parm1.string.substring(start, size);
    } else
      throw coreutils::Exception("Expecting␣)␣at␣end␣of␣RIGHT␣expression.")
          ;
} else if(in.ifNextIgnoreCase("TRIM")) {
    if(!in.ifNext("("))
      throw coreutils::Exception("Expecting␣(␣for␣TRIM␣parameters.");
    Operand parm1(in, tag);
    if(in.ifNext(")")) {
       string = parm1.string.trim();
    } else
      throw coreutils::Exception("Expecting␣)␣at␣end␣of␣TRIM␣expression.");

} else if(in.ifNextIgnoreCase("TOUPPER")) {
    if(!in.ifNext("("))
      throw coreutils::Exception("Expecting␣(␣for␣TOUPPER␣parameters.");
    Operand parm1(in, tag);
    if(in.ifNext(")")) {
       string = parm1.string.toUpper();
    } else
      throw coreutils::Exception("Expecting␣)␣at␣end␣of␣TOUPPER␣expression.
          ");
} else if(in.ifNextIgnoreCase("TOLOWER")) {
    if(!in.ifNext("("))
      throw coreutils::Exception("Expecting␣(␣for␣TOLOWER␣parameters.");
    Operand parm1(in, tag);
    if(in.ifNext(")")) {
       string = parm1.string.toLower();
    } else
      throw coreutils::Exception("Expecting␣)␣at␣end␣of␣TOLOWER␣expression.
          ");
```

```
} else if(in.ifNextIgnoreCase("REVERSE")) {

} else if(in.ifNextIgnoreCase("CONCAT")) {

} else if(in.ifNextIgnoreCase("INTEGER")) {
   if(!in.ifNext("("))
     throw coreutils::Exception("Expecting␣(␣for␣INTEGER␣parameters.");
   Operand parm1(in, tag);
   if(in.ifNext(")")) {
      string = parm1.string.asInteger();
   } else
     throw coreutils::Exception("Expecting␣)␣at␣end␣of␣INTEGER␣expression.
         ");
} else if(in.ifNextIgnoreCase("ROUND")) {

} else if(in.ifNextIgnoreCase("RANDOM")) {
   unsigned int seed = (unsigned int)clock();
   doubleValue = (double) rand_r(&seed) / (RAND_MAX + 1.0);
   isNumber = true;
   string = std::format("{:.12f}", doubleValue);
   string.removeTrailingZeros();
} else if(in.ifNextIgnoreCase("true")) {
   boolean = true;
   string = "true";
} else if(in.ifNextIgnoreCase("false")) {
   boolean = false;
   string = "false";
} else if(in.startsWithNumber()) {
   doubleValue = in.asDouble();
   string = std::format("{:.12f}", doubleValue);
   isNumber = true;
} else if(in.ifNext("'")) {
   string = in.getTokenExclude("'");
   in.ifNext("'");
   isNumber = false;
} else
  throw coreutils::Exception("operand␣is␣not␣valid.");

in.skipWhitespace();

if(in.ifNext("!=") || in.ifNext("<>")) {
   Operand op(in, tag);
   if(isNumber && op.isNumber) {
      if(doubleValue != op.doubleValue) {
         boolean = true;
         isNumber = false;
         string = "true";
      } else {
         boolean = false;
         isNumber = false;
         string = "false";
      }
   } else if(!isNumber && !op.isNumber) {
      if(string != op.string) {
         boolean = true;
         isNumber = false;
         string = "true";
```

```
        } else {
            boolean = false;
            isNumber = false;
            string = "false";
        }
    }
}
if(in.ifNext("<=")) {
    Operand op(in, tag);
    if(isNumber && op.isNumber) {
        if(doubleValue <= op.doubleValue) {
            boolean = true;
            isNumber = false;
            string = "true";
        } else {
            boolean = false;
            isNumber = false;
            string = "false";
        }
    } else if(!isNumber && !op.isNumber) {
        if(string <= op.string) {
            boolean = true;
            isNumber = false;
            string = "true";
        } else {
            boolean = false;
            isNumber = false;
            string = "false";
        }
    }
}
if(in.ifNext(">=")) {
    Operand op(in, tag);
    if(isNumber && op.isNumber) {
        if(doubleValue >= op.doubleValue) {
            boolean = true;
            isNumber = false;
            string = "true";
        } else {
            boolean = false;
            isNumber = false;
            string = "false";
        }
    } else if(!isNumber && !op.isNumber) {
        if(string >= op.string) {
            boolean = true;
            isNumber = false;
            string = "true";
        } else {
            boolean = false;
            isNumber = false;
            string = "false";
        }
    }
}
if(in.ifNext("=")) {
    Operand op(in, tag);
```

```
   if(isNumber && op.isNumber) {
      if(doubleValue == op.doubleValue) {
         boolean = true;
         isNumber = false;
         string = "true";
      } else {
         boolean = false;
         isNumber = false;
         string = "false";
      }
   } else if(!isNumber && !op.isNumber) {
      if(string == op.string) {
         boolean = true;
         isNumber = false;
         string = "true";
      } else {
         boolean = false;
         isNumber = false;
         string = "false";
      }
   }
}
if(in.ifNext("<")) {
   Operand op(in, tag);
   if(isNumber && op.isNumber) {
      if(doubleValue < op.doubleValue) {
         boolean = true;
         isNumber = false;
         string = "true";
      } else {
         boolean = false;
         isNumber = false;
         string = "false";
      }
   } else if(!isNumber && !op.isNumber) {
      if(string < op.string) {
         boolean = true;
         isNumber = false;
         string = "true";
      } else {
         boolean = false;
         isNumber = false;
         string = "false";
      }
   }
}
if(in.ifNext(">")) {
   Operand op(in, tag);
   if(isNumber && op.isNumber) {
      if(doubleValue > op.doubleValue) {
         boolean = true;
         isNumber = false;
         string = "true";
      } else {
         boolean = false;
         isNumber = false;
         string = "false";
```

```
      }
   } else if(!isNumber && !op.isNumber) {
      if(string > op.string) {
         boolean = true;
         isNumber = false;
         string = "true";
      } else {
         boolean = false;
         isNumber = false;
         string = "false";
      }
   }
}
if(in.ifNext("+")) {
   if(isNumber) {
      Operand op(in, tag);
      if(op.isNumber) {
         doubleValue += op.doubleValue;
         string = std::format("{:.12f}", doubleValue);
         string.removeTrailingZeros();
      } else
        throw coreutils::Exception("operand␣is␣not␣a␣number.");
   } else
     throw coreutils::Exception("operand␣is␣not␣a␣number.");
} else if(in.ifNext("-")) {
   if(isNumber) {
      Operand op(in, tag);
      if(op.isNumber) {
         doubleValue -= op.doubleValue;
         string = std::format("{:.12f}", doubleValue);
         string.removeTrailingZeros();
      } else
        throw coreutils::Exception("operand␣is␣not␣a␣number.");
   } else
     throw coreutils::Exception("operand␣is␣not␣a␣number.");
} else if(in.ifNext("*")) {
   if(isNumber) {
      Operand op(in, tag);
      if(op.isNumber) {
         doubleValue *= op.doubleValue;
         string = std::format("{:.12f}", doubleValue);
         string.removeTrailingZeros();
      } else
        throw coreutils::Exception("operand␣is␣not␣a␣number.");
   } else
     throw coreutils::Exception("operand␣is␣not␣a␣number.");
} else if(in.ifNext("/")) {
   if(isNumber) {
      Operand op(in, tag);
      if(op.isNumber) {
         doubleValue /= op.doubleValue;
         string = std::format("{:.12f}", doubleValue);
         string.removeTrailingZeros();
      } else
        throw coreutils::Exception("operand␣is␣not␣a␣number.");
   } else
     throw coreutils::Exception("operand␣is␣not␣a␣number.");
```

```
      } else
        return;
    }

}
```

0.32 Operand.h

```
#ifndef __Operand_h__
#define __Operand_h__

#include "MString.h"
#include "Tag.h"
#include "Global.h"

namespace jet {

   class Operand {

    public:
      Operand(coreutils::ZString &in, Tag &tag);

      bool isNumber;

      ///
      /// boolean is set by internal processes to return the boolean
      /// equivilent value.
      ///

      bool boolean;
      coreutils::MString string = "";

      double doubleValue;


   };
}

#endif
```

0.33 __read.cpp

```
#include "__read.h"
#include "Exception.h"
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

namespace jet {

   __read::__read(coreutils::ZString &in, coreutils::MString &parentOut, Global
       &global, Tag *parent, Tag *local) : Tag(in, parentOut, global, parent,
       this) {
      if(!keywordDefined("file"))
        throw coreutils::Exception("file␣keyword␣must␣be␣specified.");
      if(!keywordDefined("name"))
        throw coreutils::Exception("name␣keyword␣must␣be␣specified.");
      if(hasContainer)
        throw coreutils::Exception("read␣tag␣does␣not␣have␣a␣container.");
      fd = open(resolveKeyword("file").c_str(), O_RDONLY);
      if(fd < 0)
        throw coreutils::Exception("file␣name␣is␣not␣found.");
      global.variables[resolveKeyword("name")].read(fd);
      close(fd);
   }

}
```

0.34 __read.h

```
#ifndef ____read_h__
#define ____read_h__

#include "Tag.h"

namespace jet {

   class __read : public Tag {

    public:
      __read(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

    private:
      int fd;
      int len;
      char buffer[4096];

   };

}

#endif
```

0.35 __set.cpp

```
#include "__set.h"
#include "Exception.h"
#include <iostream>

namespace jet {

   __set::__set(coreutils::ZString &in, coreutils::MString &parentOut, Global &
       global, Tag *parent, Tag *local) : Tag(in, parentOut, global, parent,
       local) {
      output = false;
      if(!keywordDefined("name"))
        throw coreutils::Exception("set tag must have name defined.");
      if(!keywordDefined("expr") && keywordDefined("value") && hasContainer)
        throw coreutils::Exception("set tag cannot have both value and a
            container.");
      if(keywordDefined("expr") && !keywordDefined("value") && hasContainer)
        throw coreutils::Exception("set tag cannot have both expr and a
            container.");
      if(keywordDefined("expr") && keywordDefined("value") && !hasContainer)
        throw coreutils::Exception("set tag cannot have both expr and value.");
      if(!keywordDefined("expr") && !keywordDefined("value") && !hasContainer)
        throw coreutils::Exception("set tag must have a value, expr or a
            container.");
      if(keywordDefined("expr") && keywordDefined("eval"))
        throw coreutils::Exception("Cannot use eval with expr.");

      storeVariable(resolveKeyword("name"));

   }

}
```

0.36 __set.h

```
#ifndef ____set_h__
#define ____set_h__

#include "Tag.h"
#include "ZString.h"
#include "MString.h"
#include <sstream>

namespace jet {

   class __set : public Tag {

    public:
      __set(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

    protected:

   };

}

#endif
```

0.37 __sql.cpp

```
#include "__sql.h"
#include "Exception.h"
#include "MString.h"
#include "__mysql.h"
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

namespace jet {

   __sql::__sql(coreutils::ZString &in, coreutils::MString &parentOut, Global &
       global, Tag *parent, Tag *local) : Tag(in, parentOut, global, parent,
       local) {
      output = false;
      if(!hasContainer)
        throw coreutils::Exception("sql tag must have a container.");
      if(!global.sessionExists(keywords["sessionid"]))
        throw coreutils::Exception("sessionid does not exist.");
      processContainer(container);
      global.getSession(keywords[resolveKeyword("sessionid")])->query(out);
   }

}
```

0.38 __sql.h

```
#ifndef ____sql_h__
#define ____sql_h__

#include "Tag.h"

namespace jet {

   class __sql : public Tag {

    public:
      __sql(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

   };

}

#endif
```

0.39 __stream.cpp

```
#include "__stream.h"
#include "Exception.h"
#include <unistd.h>
#include <fcntl.h>

namespace jet {

   __stream::__stream(coreutils::ZString &in, coreutils::MString &parentOut,
       Global &global, Tag *parent, Tag *local) : Tag(in, parentOut, global,
       parent, this) {
      output = false;
      if(!global.cgi)
        throw coreutils::Exception("must␣enable␣cgi␣mode␣on␣jet␣tag␣to␣use␣
            stream␣tag.");
      if(!keywordDefined("name"))
        throw coreutils::Exception("stream␣tag␣must␣have␣a␣file␣name␣to␣stream."
            );
      global.outputHeaders();
      int len;
      char buffer[1024];
      int fd = open(keywords["name"].c_str(), O_RDONLY);
      do {
         len = read(fd, &buffer, 1024);
         std::cout << buffer;
      } while (len > 0);
   }
}
```

0.40 __stream.h

```
#ifndef ____stream_h__
#define ____stream_h__

#include "Tag.h"
#include "ZString.h"

namespace jet {

   class __stream : public Tag {

    public:
      __stream(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

   };

}

#endif
```

0.41 __system.cpp

```
#include "__system.h"
#include "Exception.h"
#include "MString.h"
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

namespace jet {

   __system::__system(coreutils::ZString &in, coreutils::MString &parentOut,
       Global &global, Tag *parent, Tag *local) : Tag(in, parentOut, global,
       parent, local) {
      if(hasContainer)
        throw coreutils::Exception("system tag cannot have a container.");
      if(!keywordDefined(coreutils::ZString("cmd")))
        throw coreutils::Exception("cmd keyword must be specified.");
      pipe(fdo);
      pid = fork();
      if(pid == 0) {
         close(fdo[0]);
         dup2(fdo[1], 1);
         if(keywordDefined("input")) {
            resolveKeyword("input");
            coreutils::ZString input(keywords[resolveKeyword("input")]);
            pipe(fdi);
            if(fork() == 0) {
               close(fdi[0]);
               write(fdi[1], input.getData(), input.getLength());
               close(fdi[1]);
               exit(0);
            }
            close(fdi[1]);
            dup2(fdi[0], 0);
         }
         system(keywords[resolveKeyword("cmd")].c_str());
         close(fdo[1]);
         exit(errno);
      }
      close(fdo[1]);
      if(keywordDefined("name"))
        global.variables[keywords[resolveKeyword("name")]].read(fdo[0]);
      else
        out.read(fdo[0]);
      waitpid(pid, &status, 0);
   }

}
```

0.42 __system.h

```
#ifndef ____system_h__
#define ____system_h__

#include "Tag.h"

namespace jet {

   class __system : public Tag {

    public:
      __system(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

    private:
      int pid;
      int status;
      int ix;
      int fdi[2];
      int fdo[2];
      int rc;
      char *argv[50];

   };

}

#endif
```

0.43 __tag.cpp

```
#include "__tag.h"
#include "Exception.h"

namespace jet {

   __tag::__tag(coreutils::ZString &in, coreutils::MString &parentOut, Global &
       global, Tag *parent, Tag *local) : Tag(in, parentOut, global, this, this)
       {
      evaluate = false;
      output = false;
      if(!keywordDefined("name"))
        throw coreutils::Exception("tag must have a name.");
      if(!hasContainer)
        throw coreutils::Exception("tag requires a container to process.");
      global.tags[keywords["name"]] = container;
   }
}
```

0.44 Tag.cpp

```
#include "Tag.h"
#include "Exception.h"
#include "Global.h"
#include "__mysql.h"
#include "__sql.h"
#include "__whilerow.h"
#include "__comment.h"
#include "__exclude.h"
#include "__for.h"
#include "__if.h"
#include "__ifrow.h"
#include "__include.h"
#include "__read.h"
#include "__write.h"
#include "__set.h"
#include "__call.h"
#include "__system.h"
#include "__jet.h"
#include "__while.h"
#include "__until.h"
#include "__header.h"
#include "__cookie.h"
#include "__whiledir.h"
#include "__tag.h"
#include "__dotag.h"
#include "__stream.h"
#include "__dump.h"
#include "Operand.h"
#include <iostream>

namespace jet {

   Tag::Tag(coreutils::ZString &in, coreutils::MString &parentOut, Global &
       global, Tag *parent, Tag *local, coreutils::ZString splitTagName)
     : ZString(in), parentOut(parentOut), global(global), parent(parent), local(
         local) {
       this->splitTagName = splitTagName;
       global.errorCursor = in.getCursor();
       if(parent && in.ifNext("<")) {
          name = in.getTokenInclude("
              ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_!"
              );
          if(in.startsWith("␣") || in.startsWith("/") || in.startsWith(">")) {
             bool finished = false;
             while(!finished) {
                in.skipWhitespace();
                if(in.ifNext(">")) {
                   hasContainer = true;
                   hasContainer2 = false;
                   finished = true;
                   break;
                } else if(in.ifNext("/>")) {
                   hasContainer = false;
                   hasContainer2 = false;
                   finished = true;
```

```
            break;
         }
         if(!finished) {
            coreutils::ZString keywordName = in.getTokenInclude("
                ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789
                -_");
            if(in.ifNext("=\"")) {
               if(keywords.count(keywordName) == 0)
                 keywords[keywordName] = in.getTokenExclude("\"");
               else
                 throw coreutils::Exception("keyword␣name␣must␣be␣unique
                     ␣for␣tag.");
            }
            if(!in.ifNext("\"")) {}
         }
      }
      if(keywordDefined("filterblanklines")) {
         filterBlankLines = keywords["filterblanklines"] == "true" ?
             true: false;
      }
      if(keywordDefined("trimlines")) {
         trimLines = keywords["trimlines"] == "true" ? true: false;
      }
      if(hasContainer) {
         bool hasSplitTag = splitTagName == "" ? false: true;
         char *start = in.getCursor();
         while(!in.eod()) {
            char *end = in.getCursor();
            if(ifEndTagName(in)) {
               if(hasContainer2)
                 container2 = coreutils::ZString(start, end - start);
               else
                 container = coreutils::ZString(start, end - start);
               break;
            } else if(hasSplitTag && ifSplitTagName(in)) {
               hasContainer2 = true;
               container = coreutils::ZString(start, end - start);
               in.ifNext("<else>");
               start = in.getCursor();
            } else if(ifNested(in)) {
            } else
              in.nextChar();
         }
      }
      setZString(in.parsed());
      if(keywordDefined("eval")) {
         if(keywords["eval"] == "yes") {
            evaluate = true;
         } else if(keywords["eval"] == "no") {
            evaluate = false;
         } else
            throw coreutils::Exception("keyword␣'eval'␣must␣be␣'yes'␣or␣
                'no'.");
      }
   }
} else
  parseContainer(in, out);
```

```
  }

Tag::~Tag() {
   if(evaluate)
     if(output)
       copyContainer(out, parentOut);
   else if(output)
     copyContainer(container, parentOut);
}

coreutils::MString Tag::resolveKeyword(coreutils::ZString keyword) {
   coreutils::MString resolved;
   keywords[keyword].reset();
   while(!keywords[keyword].eod()) {
      if(keywords[keyword].startsWith("$[") || keywords[keyword].startsWith("
          #[")) {
         resolved.write(getVariable(keywords[keyword]));
      } else {
         resolved.write(keywords[keyword].charAt(0));
         keywords[keyword].nextChar();
      }
   }
   keywords[keyword].reset();
   return resolved;
}

void Tag::processContainer(coreutils::ZString &container, coreutils::ZString
    container2, bool topLevel) {
   if(hasContainer && evaluate)
     parseContainer(container, out, container2, topLevel);
}

void Tag::parseContainer(coreutils::ZString &in, coreutils::MString &out,
    coreutils::ZString container2, bool topLevel) {
   coreutils::ZString tag;
   char *start = in.getCursor();
   while(!in.eod()) {
      if(in.ifNext("|>")) {
         if(!topLevel)
           out.write("|>");
         char ch = in.nextChar();
         while(!in.ifNext("<|")) {
           out.write(ch);
           ch = in.nextChar();
         }
         out.write(ch);
         if(!topLevel)
           out.write("<|");
         continue;
      } else if(in.startsWith("<")) {
         if(ifTagName(in, "mysql")) {
            __mysql _mysql(in, out, global, this, local);
            continue;
         } else if(ifTagName(in, "comment")) {
            __comment _comment(in, out, global, this, local);
            continue;
         } else if(ifTagName(in, "sql")) {
```

```
    __sql _sql(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "whilerow")) {
    __whilerow _whilerow(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "for")) {
    __for _for(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "if")) {
    __if _if(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "ifrow")) {
    __ifrow _ifrow(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "include")) {
    __include _include(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "exclude")) {
    __exclude _exclude(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "jet")) {
    __jet _jet(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "read")) {
    __read _read(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "write")) {
    __write _write(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "set")) {
    __set _set(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "call")) {
    __call _call(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "system")) {
    __system _system(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "while")) {
    __while _while(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "until")) {
    __until _until(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "header")) {
    __header _header(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "cookie")) {
    __cookie _cookie(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "whiledir")) {
    __whiledir _whiledir(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "stream")) {
    __stream _stream(in, out, global, this, local);
    continue;
} else if(ifTagName(in, "dump")) {
```

```
            __dump _dump(in, out, global, this, local);
            continue;
         } else if(ifTagName(in, "tag")) {
            __tag _tag(in, out, global, this, local);
            continue;
         } else if(ifTagDefined(in, tag)) {
            __dotag _dotag(in, out, global, this, local);
            continue;
         } else if(ifTagName(in, "container")) {
            processContainer(container2);
            in.ifNext("<container");
            in.skipWhitespace();
            in.ifNext("/>");
            continue;
         } else {
            out.write(in.nextChar());
            continue;
         }
      } else if(in.startsWith("$[") || in.startsWith("#[")) {
         global.errorCursor = in.getCursor();
         out.write(getVariable(in, true));
      } else {
         out.write(in.nextChar());
      }
   }
}

void Tag::scanContainer(coreutils::ZString &in) {
   while(!in.eod()) {
      if(ifEndTagName(in))
        return;
      else if(ifNested(in)) {}
      else in.nextChar();
   }
}

bool Tag::ifTagName(coreutils::ZString &in, const char *tag) {
   in.push();
   if(in.ifNext("<"))
     if(in.getTokenInclude("
         ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_").
         equals(tag)) {
        if(in.ifNext("␣")) {
           in.pop();
           return true;
        } else if(in.ifNext("/>")) {
           in.pop();
           return true;
        } else if(in.ifNext(">")) {
           in.pop();
           return true;
        }
     }
   in.pop();
   return false;
}
```

```
bool Tag::ifTagName(coreutils::ZString &in) {
   in.push();
   if(in.ifNext("<"))
     if(in.getTokenInclude("
         ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_").
         equals(name)) {
        in.push();
        if(in.ifNext("␣")) {
           in.pop();
           return true;
        } else if(in.ifNext("/>")) {
           in.pop();
           return true;
        } else if(in.ifNext(">")) {
           in.pop();
           return true;
        }
        in.pop();
     }
   in.pop();
   return false;
}

bool Tag::ifTagDefined(coreutils::ZString &in, coreutils::ZString &tag) {
   in.push();
   if(in.ifNext("<")) {
      tag = in.getTokenInclude("
          ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_");
      if(global.tags.count(tag)) {
         if(in.ifNext("␣")) {
            in.pop();
            return true;
         } else if(in.ifNext("/>")) {
            in.pop();
            return true;
         } else if(in.ifNext(">")) {
            in.pop();
            return true;
         }
         in.pop();
      }
   }
   in.pop();
   return false;
}

bool Tag::ifEndTagName(coreutils::ZString &in) {
   in.push();
   if(in.ifNext("</"))
     if(in.getTokenInclude("
         ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_").
         equals(name)) {
        if(in.ifNext(">"))
          return true;
     }
   in.pop();
   return false;
```

```
}

bool Tag::ifSplitTagName(coreutils::ZString &in) {
   in.push();
   if(in.ifNext("<"))
     if(in.getTokenInclude("
         ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_").
         equals(splitTagName)) {
        if(in.ifNext(">"))
          in.pop();
          return true;
     }
   in.pop();
   return false;
}

int Tag::skipBlankLine(coreutils::ZString in) {
   ZString temp = in.getTokenInclude("␣\t");
   if(ifNext("\n"))
     return temp.getLength() + 1;
   return 0;
}

void Tag::copyContainer(coreutils::ZString &in, coreutils::MString &out) {
   while(!in.eod()) {
      if(filterBlankLines) {
         if(!in.lineIsWhitespace()) {
            if(trimLines)
              out.write(in.goeol().trim());
            else
              out.write(in.goeol());
            out.write('\n');
         }
         else {
            in.goeol();
         }
      }
      else {
         out.write(in.charAt(0));
         in.nextChar();
      }
   }
}

bool Tag::keywordDefined(coreutils::ZString keyword) {
   return keywords.find(keyword) != keywords.end();
}

bool Tag::ifNested(coreutils::ZString &in) {
   bool hasContainer = false;
   if(ifTagName(in)) {
      while(!in.eod()) {
         in.skipWhitespace();
         if(in.ifNext(">")) {
            hasContainer = true;
            break;
         } else if(in.ifNext("/>")) {
```

```
            hasContainer = false;
            return true;
         } else if(in.getTokenExclude("=").getLength() != 0) {
            if(in.ifNext("=\"")) {
               while(1) {
                  if(in.ifNext("\"")) {
                     break;
                  }
                  in.nextChar();
               }
            }
         }
      }
      if(hasContainer)
        scanContainer(in);
   }
   return false;
}

coreutils::MString Tag::getVariable(coreutils::ZString &variable, bool
    inContainer) {
   if(variable.ifNext("$[")) {
      coreutils::MString name;
      coreutils::MString modifier;
      if(variable.ifNext("!")) {
         renderVariableName(variable, name, modifier);
         return global.variables[name];
      } else if(variable.ifNext("%")) {
         renderVariableName(variable, name, modifier);
         if(inContainer)
           return keywords[name];
         else
           return parent->keywords[name];
      } else if(variable.ifNext(":")) {
         renderVariableName(variable, name, modifier);
         if(name.find(":") == -1) {
            name << ":0";
         }
         return processModifier(global.cgiVariables[name], modifier);
      } else if(variable.ifNext("@")) {
         // TODO: should only allow session variables. Allow substitution.
      } else if(variable.ifNext("%")) {
         renderVariableName(variable, name, modifier);
         return getenv(name.c_str());
      } else if(variable.ifNext("^")) {
         renderVariableName(variable, name, modifier);
         return global.cookies.data[variable];
      } else {
         renderVariableName(variable, name, modifier);
         name.split(".");
         if(name.getList().size() == 1) {
            if(global.variables.find(name[0]) == global.variables.end())
              throw coreutils::Exception("global␣variable␣is␣not␣initialized.
                  ");
            return processModifier(global.variables[name[0]], modifier);
         }
         return global.getSessionVariable(name);
```

```
      }
      throw coreutils::Exception("expected␣variable␣name␣or␣type␣designator."
          );
   } else if(variable.ifNext("#[")) {
      coreutils::MString name;
      coreutils::MString modifier;
      renderVariableName(variable, name, modifier);
      if(local->variables.find(name) == local->variables.end())
        throw coreutils::Exception("local␣variable␣is␣not␣initialized.");
      return local->variables[name];
   }
   throw coreutils::Exception("Expecting␣a␣variable␣initializer␣('$['␣or␣
       '#[').");
}

void Tag::renderVariableName(coreutils::ZString &variable, coreutils::MString
     &name, coreutils::MString &modifier) {
   while(!variable.ifNext("]")) {
      name << variable.getTokenInclude("?
          ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789._-");
      if(variable.ifNext(";")) {
         renderVariableName(variable, modifier, modifier);
         return;
      } else if(variable.ifNext(":")) {
         name << ":";
      } else if(variable.startsWith("$[") || variable.startsWith("#[")) {
         name << getVariable(variable);
      } else if(variable.ifNext("]"))
           return;
      else if(!variable.ifNextInclude("?
          ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789._-"))
        throw coreutils::Exception("invalid␣variable␣name.");
   }
   return;
}

void Tag::storeVariable(coreutils::ZString variable, coreutils::MString value
    , coreutils::ZString scope) {
   if((scope == "global") || (scope == ""))
     global.variables[variable] = value;
   else if(scope == "local")
     local->variables[variable] = value;
   else if(scope == "parent")
     local->parent->variables[variable] = value;
}

void Tag::storeVariable(coreutils::ZString variable) {
   if(keywordDefined("expr")) {
      if(!keywordDefined("scope") || (resolveKeyword("scope") == "global"))
        global.variables[variable] = Operand(keywords["expr"], *this).string;
      else if(resolveKeyword("scope") == "local")
        local->variables[variable] = Operand(keywords["expr"], *this).string;
      else if(resolveKeyword("scope") == "parent")
        local->parent->variables[variable] = Operand(keywords["expr"], *this)
            .string;
   } else if(hasContainer) {
      processContainer(container);
```

```
      if(evaluate) {
         if(!keywordDefined("scope") || (resolveKeyword("scope") == "global")
             )
           global.variables[variable] = out;
         else if(resolveKeyword("scope") == "local")
           local->variables[variable] = out;
         else if(resolveKeyword("scope") == "parent")
           local->parent->variables[variable] = out;
      } else {
         if(!keywordDefined("scope") || (resolveKeyword("scope") == "global")
             )
           global.variables[variable] = container;
         else if(resolveKeyword("scope") == "local")
           local->variables[variable] = container;
         else if(resolveKeyword("scope") == "parent")
           local->parent->variables[variable] = container;
      }
    } else {
      if(!keywordDefined("scope") || (resolveKeyword("scope") == "global"))
        global.variables[variable] = resolveKeyword("value");
      else if(resolveKeyword("scope") == "local")
        local->variables[variable] = resolveKeyword("value");
      else if(resolveKeyword("scope") == "parent")
        local->parent->variables[variable] = resolveKeyword("value");
    }
  }

  coreutils::MString Tag::processModifier(coreutils::MString &value, coreutils
      ::MString &modifier) {
    if(modifier == "")
      return value;
    if(modifier == "tobinary")
      return value.toBinary();
    else if(modifier == "frombinary")
      return value.fromBinary();
    else if(modifier == "tohex")
      return value.toHex();
    else if(modifier == "fromhex")
      return value.fromHex();
    else if(modifier == "tobase64")
      return value.toBase64();
    else if(modifier == "frombase64")
      return value.fromBase64();
    else if(modifier == "toupper")
      return value.toUpper();
    else if(modifier == "tolower")
      return value.toLower();
    else if(modifier == "tocgi")
      return value.toCGI();
    else if(modifier == "fromcgi")
      return value.fromCGI();
    throw coreutils::Exception("modifier not valid.");
  }

}
```

0.45 __tag.h

```
#ifndef ____tag_h__
#define ____tag_h__

#include "Tag.h"
#include "ZString.h"
#include "MString.h"
#include <map>

namespace jet {

   class __tag : public Tag {

    public:
      __tag(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

      std::map<coreutils::MString, coreutils::MString> tags;

   };

}

#endif
```

0.46 Tag.h

```
#ifndef __Tag_h__
#define __Tag_h__

#include "ZString.h"
#include "MString.h"
#include "Global.h"
#include <map>

namespace jet {

   class Tag : public coreutils::ZString {

    public:
      Tag(coreutils::ZString &in, coreutils::MString &parentOut, Global &global,
           Tag *parent = NULL, Tag *local = NULL, coreutils::ZString splitTagName
           = "");
      virtual ~Tag();

      coreutils::MString getVariable(coreutils::ZString &variable, bool
          inContainer = false);

      coreutils::MString resolveKeyword(coreutils::ZString keyword);
      std::map<coreutils::MString, coreutils::MString> variables;
      std::map<coreutils::MString, coreutils::MString> keywords;
      coreutils::ZString name;
      coreutils::ZString container;
      coreutils::ZString container2;
      Global &global;
      Tag *parent;
      Tag *local;

    protected:
      bool hasContainer = false;
      bool hasContainer2 = false;
      bool keywordDefined(coreutils::ZString variable);
      void parseContainer(coreutils::ZString &in, coreutils::MString &out,
          coreutils::ZString container2 = NULL, bool topLevel =  false);
      void processContainer(coreutils::ZString &container, coreutils::ZString
          container2 = NULL, bool topLevel = false);
      void copyContainer(coreutils::ZString &in, coreutils::MString &out);

      coreutils::MString &parentOut;
      coreutils::MString out;

      bool output = true;
      bool evaluate = true;
      bool filterBlankLines = false;
      bool trimLines = false;
      bool cleanWhitespace = false;

      void renderVariableName(coreutils::ZString &variable, coreutils::MString &
          name, coreutils::MString &modifier);
      void storeVariable(coreutils::ZString variable, coreutils::MString value,
          coreutils::ZString scope);
      void storeVariable(coreutils::ZString variable);
```

```
    private:
      bool containerOnly = false;
      coreutils::ZString splitTagName;

      int skipBlankLine(coreutils::ZString in);

      void scanContainer(coreutils::ZString &in);
      bool ifNested(coreutils::ZString &in);
      bool ifTagName(coreutils::ZString &in, const char *tag);
      bool ifTagName(coreutils::ZString &in);
      bool ifTagDefined(coreutils::ZString &in, coreutils::ZString &tag);
      bool ifEndTagName(coreutils::ZString &in);
      bool ifSplitTagName(coreutils::ZString &in);

      coreutils::MString processModifier(coreutils::MString &value, coreutils::
          MString &modifier);
  };

}

#endif
```

0.47 __until.cpp

```
#include "__until.h"
#include "Exception.h"
#include "Operand.h"
#include <iostream>

namespace jet {

   __until::__until(coreutils::ZString &in, coreutils::MString &parentOut,
       Global &global, Tag *parent, Tag *local) : Tag(in, parentOut, global,
       parent, this) {

      coreutils::MString result;
      bool booleanResult = false;
      bool exprMethod = false;
      coreutils::MString exprSaved;

      if(keywordDefined("value1")) {

         if(keywordDefined("expr"))
           throw coreutils::Exception("either␣value1␣or␣expr␣can␣be␣specified␣
               but␣not␣both.");
         if(!keywordDefined("value2"))
           throw coreutils::Exception("value2␣required␣if␣value1␣specified.");
         if(!keywordDefined("type"))
           throw coreutils::Exception("type␣expected␣if␣value1␣and␣value2␣
               specified.");

         int rc = keywords[resolveKeyword("value1")].compare(keywords[
             resolveKeyword("value2")]);
         if(((keywords[resolveKeyword("type")] == "eq") && (rc == 0)) ||
            ((keywords[resolveKeyword("type")] == "ne") && (rc != 0)) ||
            ((keywords[resolveKeyword("type")] == "lt") && (rc == -1)) ||
            ((keywords[resolveKeyword("type")] == "le") && (rc != 1)) ||
            ((keywords[resolveKeyword("type")] == "gt") && (rc == 1)) ||
            ((keywords[resolveKeyword("type")] == "ge") && (rc != -1)))
           booleanResult = true;
         else
           throw coreutils::Exception("type␣value␣must␣be␣'eq','ne','lt','le','
               gt','ge'.");
      }
      else if(keywordDefined("expr")) {
         if(keywordDefined("value2"))
           throw coreutils::Exception("value2␣should␣not␣be␣specified␣with␣expr.
               ");
         if(keywordDefined("type"))
           throw coreutils::Exception("type␣should␣not␣be␣specified␣with␣expr.")
               ;
         exprMethod = true;
         exprSaved = keywords["expr"];
      }
      do {
         processContainer(container);
         container.reset();
         if(exprMethod) {
            keywords["expr"].reset();
```

```
            keywords["expr"] = exprSaved;
            resolveKeyword("expr");
            booleanResult = Operand(keywords["expr"], *this).boolean;
         } else {
            booleanResult = false;
            int rc = keywords[resolveKeyword("value1")].compare(keywords[
                resolveKeyword("value2")]);
            if(((keywords[resolveKeyword("type")] == "eq") && (rc == 0)) ||
               ((keywords[resolveKeyword("type")] == "ne") && (rc != 0)) ||
               ((keywords[resolveKeyword("type")] == "lt") && (rc == -1)) ||
               ((keywords[resolveKeyword("type")] == "le") && (rc != 1)) ||
               ((keywords[resolveKeyword("type")] == "gt") && (rc == 1)) ||
               ((keywords[resolveKeyword("type")] == "ge") && (rc != -1)))
              booleanResult = true;
         }
      } while(booleanResult);

   }

}
```

0.48 __until.h

```
#ifndef ____until_h__
#define ____until_h__

#include "Tag.h"
#include <sstream>

namespace jet {

   class __until : public Tag {

    public:
      __until(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

   };

}

#endif
```

0.49 __while.cpp

```
#include "__while.h"
#include "Exception.h"
#include "Operand.h"
#include <iostream>

namespace jet {

   __while::__while(coreutils::ZString &in, coreutils::MString &parentOut,
       Global &global, Tag *parent, Tag *local) : Tag(in, parentOut, global,
       parent, this) {

      coreutils::MString result;
      bool booleanResult = false;
      bool exprMethod = false;
      coreutils::MString exprSaved;

      if(keywordDefined("value1")) {

         if(keywordDefined("expr"))
           throw coreutils::Exception("either␣value1␣or␣expr␣can␣be␣specified␣
               but␣not␣both.");
         if(!keywordDefined("value2"))
           throw coreutils::Exception("value2␣required␣if␣value1␣specified.");
         if(!keywordDefined("type"))
           throw coreutils::Exception("type␣expected␣if␣value1␣and␣value2␣
               specified.");

         int rc = keywords[resolveKeyword("value1")].compare(keywords[
             resolveKeyword("value2")]);
         if(((keywords[resolveKeyword("type")] == "eq") && (rc == 0)) ||
            ((keywords[resolveKeyword("type")] == "ne") && (rc != 0)) ||
            ((keywords[resolveKeyword("type")] == "lt") && (rc == -1)) ||
            ((keywords[resolveKeyword("type")] == "le") && (rc != 1)) ||
            ((keywords[resolveKeyword("type")] == "gt") && (rc == 1)) ||
            ((keywords[resolveKeyword("type")] == "ge") && (rc != -1)))
           booleanResult = true;
         else
           throw coreutils::Exception("type␣value␣must␣be␣'eq','ne','lt','le','
               gt','ge'.");
      }
      else if(keywordDefined("expr")) {
         if(keywordDefined("value2"))
           throw coreutils::Exception("value2␣should␣not␣be␣specified␣with␣expr.
               ");
         if(keywordDefined("type"))
           throw coreutils::Exception("type␣should␣not␣be␣specified␣with␣expr.")
               ;
         exprMethod = true;
         exprSaved = keywords["expr"];
         booleanResult = Operand(keywords["expr"], *this).boolean;
      }
      while(booleanResult) {
         processContainer(container);
         container.reset();
         if(exprMethod) {
```

```
            keywords["expr"].reset();
            keywords["expr"] = exprSaved;
            booleanResult = Operand(keywords["expr"], *this).boolean;
         } else {
            booleanResult = false;
            int rc = keywords[resolveKeyword("value1")].compare(keywords[
                resolveKeyword("value2")]);
            if(((keywords[resolveKeyword("type")] == "eq") && (rc == 0)) ||
               ((keywords[resolveKeyword("type")] == "ne") && (rc != 0)) ||
               ((keywords[resolveKeyword("type")] == "lt") && (rc == -1)) ||
               ((keywords[resolveKeyword("type")] == "le") && (rc != 1)) ||
               ((keywords[resolveKeyword("type")] == "gt") && (rc == 1)) ||
               ((keywords[resolveKeyword("type")] == "ge") && (rc != -1)))
              booleanResult = true;
         }
      }

   }

}
```

0.50 __whiledir.cpp

```
#include "__whiledir.h"
#include "Exception.h"
#include "__mysql.h"
#include <iostream>
#include <filesystem>
#include <vector>
#include <algorithm>

namespace jet {

   __whiledir::__whiledir(coreutils::ZString &in, coreutils::MString &parentOut,
        Global &global, Tag *parent, Tag *local) : Tag(in, parentOut, global,
       parent, this) {
      if(!keywordDefined("path"))
        throw coreutils::Exception("whiledir tag must specify a path.");
      if(keywordDefined("sort") && (resolveKeyword("sort") == "true")) {
         std::vector<std::filesystem::directory_entry> entries;
         for(auto const &entry : std::filesystem::directory_iterator(
             resolveKeyword("path").str()))
           entries.push_back(entry);
         std::sort(entries.begin(), entries.end(), [](const auto &a, const auto
             &b) { return a.path() < b.path(); });
         for(const auto &entry : entries) {
            if(keywordDefined("fullpath"))
              global.variables[resolveKeyword("fullpath")] = entry.path();
            if(keywordDefined("filename"))
              global.variables[resolveKeyword("filename")] = entry.path().
                  filename();
            if(keywordDefined("filenamenoextension"))
              global.variables[resolveKeyword("filenamenoextension")] = entry.
                  path().stem();
            processContainer(container);
            container.reset();
         }
      } else {
         for(auto const &entry : std::filesystem::directory_iterator(variables[
             resolveKeyword("path")].str())) {
            if(keywordDefined("fullpath"))
              global.variables[resolveKeyword("fullpath")] = entry.path();
            if(keywordDefined("filename"))
              global.variables[resolveKeyword("filename")] = entry.path().
                  filename();
            if(keywordDefined("filenamenoextension"))
              global.variables[resolveKeyword("filenamenoextension")] = entry.
                  path().stem();
            processContainer(container);
            container.reset();
         }
      }

   }

}
```

0.51 __whiledir.h

```
#ifndef ____whiledir_h__
#define ____whiledir_h__

#include "Tag.h"
#include "ZString.h"
#include "MString.h"

namespace jet {

   class __whiledir : public Tag {

    public:
      __whiledir(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

   };

}

#endif
```

0.52 __while.h

```
#ifndef ____while_h__
#define ____while_h__

#include "Tag.h"
#include <sstream>

namespace jet {

   class __while : public Tag {

    public:
      __while(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

   };

}

#endif
```

0.53 __whilerow.cpp

```
#include "__whilerow.h"
#include "Exception.h"
#include "__mysql.h"
#include <iostream>

namespace jet {

   __whilerow::__whilerow(coreutils::ZString &in, coreutils::MString &parentOut,
        Global &global, Tag *parent, Tag *local) : Tag(in, parentOut, global,
       parent, local) {

      int count = keywords["count"].asInteger();

      while  ((count != 0) && global.getSession(keywords["sessionid"])->hasRow()
          ) {
         processContainer(container);
         container.reset();
         global.getSession(keywords["sessionid"])->nextRow();
         --count;
      }

   }

}
```

0.54 __whilerow.h

```
#ifndef ____whilerow_h__
#define ____whilerow_h__

#include "Tag.h"
#include "ZString.h"
#include "MString.h"

namespace jet {

   class __whilerow : public Tag {

    public:
      __whilerow(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

   };

}

#endif
```

0.55 __write.cpp

```
#include "__write.h"
#include "Exception.h"
#include "Operand.h"
#include <iostream>
#include <fcntl.h>
#include <unistd.h>

namespace jet {

   __write::__write(coreutils::ZString &in, coreutils::MString &parentOut,
       Global &global, Tag *parent, Tag *local) : Tag(in, parentOut, global,
       parent, local) {
      output = false;
      int mode = 0;
      int len;
      processContainer(container);
      if(!keywordDefined("file"))
        throw coreutils::Exception("write␣tag␣must␣have␣file␣defined.");
      if(!keywordDefined("expr") && keywordDefined("value") && hasContainer)
        throw coreutils::Exception("write␣tag␣cannot␣have␣both␣value␣and␣a␣
            container.");
      if(keywordDefined("expr") && !keywordDefined("value") && hasContainer)
        throw coreutils::Exception("write␣tag␣cannot␣have␣both␣expr␣and␣a␣
            container.");
      if(keywordDefined("expr") && keywordDefined("value") && !hasContainer)
        throw coreutils::Exception("write␣tag␣cannot␣have␣both␣expr␣and␣value.")
            ;
      if(!keywordDefined("expr") && !keywordDefined("value") && !hasContainer)
        throw coreutils::Exception("write␣tag␣must␣have␣a␣value,␣expr␣or␣a␣
            container.");
      if(!keywordDefined("mode"))
        throw coreutils::Exception("write␣tag␣must␣have␣a␣mode␣keyword.");
      if(keywords[resolveKeyword("mode")] == "append")
        mode = O_APPEND;
      else if(keywords[resolveKeyword("mode")] == "overwrite")
        mode = O_TRUNC;
      else
        throw coreutils::Exception("mode␣keyword␣must␣be␣'overwrite'␣or␣'append
            '.");
      int fd = open(keywords[resolveKeyword("file")].c_str(), mode, 0644); //
          TODO: Need to add O_CREAT and AUTH flags.
      if(hasContainer && !evaluate)
        len = write(fd, container.getData(), container.getLength());
      else if(hasContainer && evaluate)
        len = write(fd, out.getData(), out.getLength());
      else if(!hasContainer && keywordDefined("value"))
        len = write(fd, variables[resolveKeyword("value")].getData(), keywords[
            resolveKeyword("value")].getLength());
      else if(!hasContainer && keywordDefined("expr"))
        len = write(fd, keywords["expr"].getData(), keywords["expr"].getLength()
            );
      close(fd);
   }

}
```

0.56 __write.h

```
#ifndef __write_h__
#define __write_h__

#include "Tag.h"
#include "ZString.h"
#include "MString.h"
#include <sstream>

namespace jet {

   class __write : public Tag {

    public:
      __write(coreutils::ZString &in, coreutils::MString &parentOut, Global &
          global, Tag *parent, Tag *local);

    protected:

   };

}

#endif
```