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(!variableDefined("pgm"))
        throw coreutils::Exception("pgm␣keyword␣must␣be␣specified.");
      resolveKeyword("pgm");
      argv[0] = variables["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(variableDefined(arg)) {
           resolveKeyword(arg);
           argv[ix] = variables[arg].c_str();
         } else
           break;
      }
      argv[ix] == NULL;
      pipe(fdo);
      pid = fork();
      if(pid == 0) {
         close(fdo[0]);
         dup2(fdo[1], 1);
         if(variableDefined("input")) {
            resolveKeyword("input");
            coreutils::ZString input(variables["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(variables["pgm"].c_str(), argv, global.envp);
         close(fdo[1]);
         exit(errno);
      }
      close(fdo[1]);
      if(variableDefined("name")) {
```

```
            resolveKeyword("name");
            if(!variableDefined("scope") || (variables["scope"] == "global"))
              global.variables[variables["name"]].read(fdo[0]);
            else if(variables["scope"] == "local")
              parent->variables[variables["name"]].read(fdo[0]);
            else if(variables["scope"] == "parent")
              parent->parent->variables[variables["name"]].read(fdo[0]);
            else
              throw coreutils::Exception("scope value is not valid.");

        } else
          out.read(fdo[0]);
        waitpid(pid, &status, 0);
        if(variableDefined("error")) {
            resolveKeyword("error");
            global.variables[variables["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, this) {
      if(hasContainer)
        parseContainer(container, containerOut);
      containerOut.reset();
      coreutils::ZString container3 = global.tags[name];
      hasContainer = true;
      processContainer(container3);
   }
}
```

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(!variableDefined("file"))
        throw coreutils::Exception("file must be sppecified for dump tag.");

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

      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.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 = variableDefined("name");
      if(variableDefined("start")) {
         resolveKeyword("start");
         counter = variables["start"].asDouble();
         variables["start"].reset();
      }
      if(variableDefined("end"))
        resolveKeyword("end");
      else
        throw coreutils::Exception("for␣tag␣requires␣end␣keyword.");
      if(variableDefined("step"))
        resolveKeyword("step");
      else
        throw coreutils::Exception("for␣tag␣requires␣step␣keyword.");
      for(double ix = counter; ix <= variables["end"].asDouble(); ix +=
          variables["step"].asDouble()) {
         variables["end"].reset();
         variables["step"].reset();
         if(nameDefined) {
            std::cout << ix << std::endl;
            if(!variableDefined("scope") || (variables["scope"] == "global"))
              global.variables[variables["name"]] = ix;
            else if(variables["scope"] == "local")
              local->variables[variables["name"]] = ix;
            else if(variables["scope"] == "parent")
              parent->local->variables[variables["name"]] = ix;
         }
         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 = variableDefined("name");
      if(variableDefined("start")) {
         resolveKeyword("start");
         counter = variables["start"].asDouble();
         variables["start"].reset();
      }
      if(variableDefined("end"))
        resolveKeyword("end");
      else
        throw coreutils::Exception("for␣tag␣requires␣end␣keyword.");
      if(variableDefined("step"))
        resolveKeyword("step");
      else
        throw coreutils::Exception("for␣tag␣requires␣step␣keyword.");
      for(double ix = counter; ix <= variables["end"].asDouble(); ix +=
          variables["step"].asDouble()) {
         variables["end"].reset();
         variables["step"].reset();
         if(nameDefined) {
            if(!variableDefined("scope") || (variables["scope"] == "global"))
              global.variables[variables["name"]] = ix;
            else if(variables["scope"] == "local")
              local->variables[variables["name"]] = ix;
            else if(variables["scope"] == "parent")
              parent->local->variables[variables["name"]] = ix;
         }
         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);
   }

   coreutils::MString& Global::processModifier(coreutils::MString &value,
       coreutils::MString &modifier) {
      if(modifier.getLength() == 0)
        return value;
      if(modifier == "tobinary")
        modifiers.processToBinaryModifier(value, lastConverted);
      if(modifier == "frombinary")
        modifiers.processFromBinaryModifier(value, lastConverted);
      if(modifier == "tohex")
        modifiers.processToHexModifier(value, lastConverted);
      if(modifier == "fromhex")
        modifiers.processFromHexModifier(value, lastConverted);
      if(modifier == "tobase64")
        modifiers.processToBase64Modifier(value, lastConverted);
      if(modifier == "frombase64")
        modifiers.processFromBase64Modifier(value, lastConverted);
      if(modifier == "toupper")
        modifiers.processToUpperModifier(value, lastConverted);
      if(modifier == "tolower")
        modifiers.processToLowerModifier(value, lastConverted);
```

```
   if(modifier == "tocgi")
     modifiers.processToCGIModifier(value, lastConverted);
   if(modifier == "fromcgi")
     modifiers.processFromCGIModifier(value, lastConverted);
   return lastConverted;
}

coreutils::ZString Global::getVariable(coreutils::ZString &variable, std::map
    <coreutils::MString, coreutils::MString> &lvariables) {
   if(variable.ifNext("$[")) {
      coreutils::MString name;
      coreutils::MString modifier;
      if(variable.ifNext("!")) {
         renderVariableName(variable, name, modifier, lvariables);
         return variables[name];
      } if(variable.ifNext(":")) {
         renderVariableName(variable, name, modifier, lvariables);
         if(name.find(":") == -1) {
            name << ":0";
         }
         return processModifier(cgiVariables[name], modifier);
      } if(variable.ifNext("@")) {
         // TODO: should only allow session variables. Allow substitution.
      } if(variable.ifNext("%")) {
         renderVariableName(variable, name, modifier, lvariables);
         return getenv(name.c_str());
      } else {
         renderVariableName(variable, name, modifier, lvariables);
         name.split(".");
         if(name.getList().size() == 1) {
            return processModifier(variables[name[0]], modifier);
         }
         return getSessionVariable(name);
      }
      throw coreutils::Exception("expected␣variable␣name␣or␣type␣designator."
          );
   } if(variable.ifNext("#[")) {
      std::cout << variable.unparsed() << std::endl;
      coreutils::MString name;
      coreutils::MString modifier;
      renderVariableName(variable, name, modifier, lvariables);
      return lvariables[name];
   }
   throw coreutils::Exception("Expecting␣a␣variable␣initializer␣('$['␣or␣
       '#[').");
}

void Global::renderVariableName(coreutils::ZString &variable, coreutils::
    MString &name, coreutils::MString &modifier, std::map<coreutils::MString,
    coreutils::MString> &lvariables) {
   while(!variable.ifNext("]")) {
      name << variable.getTokenInclude("#?
          ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789._-");
      if(variable.ifNext(";")) {
         renderVariableName(variable, modifier, modifier, lvariables);
         return;
      } else if(variable.ifNext(":")) {
```

```
         name << ":";
      } else if(variable.startsWith("$[") || variable.startsWith("#["))

        name << getVariable(variable, lvariables);
      else if(variable.ifNext("]"))
        return;
      else if(!variable.ifNextInclude("#?
          ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789._-"))
        throw coreutils::Exception("invalid␣variable␣name.");
   }
   return;
}

__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);
            modifiers.processFromCGIModifier(data, lastConverted);
            cgiVariables[namex] = lastConverted;
          } 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 "Modifiers.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);
      coreutils::MString& processModifier(coreutils::MString &value, coreutils::
          MString &modifier);
      coreutils::ZString getVariable(coreutils::ZString &variable, std::map<
          coreutils::MString, coreutils::MString> &lvariables);
      void renderVariableName(coreutils::ZString &variable, coreutils::MString &
          name, coreutils::MString &modifier, std::map<coreutils::MString,
          coreutils::MString> &lvariables);
      __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;

      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;
      coreutils::MString lastConverted;
      char **envp;
      Modifiers modifiers;

   };

}

#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(!variableDefined("name"))
        throw coreutils::Exception("header tag must have name defined.");
      if(!variableDefined("expr") && variableDefined("value") && hasContainer)
        throw coreutils::Exception("header tag cannot have both value and a
            container.");
      if(variableDefined("expr") && !variableDefined("value") && hasContainer)
        throw coreutils::Exception("header tag cannot have both expr and a
            container.");
      if(variableDefined("expr") && variableDefined("value") && !hasContainer)
        throw coreutils::Exception("header tag cannot have both expr and value."
            );
      if(!variableDefined("expr") && !variableDefined("value") && !hasContainer)
        throw coreutils::Exception("header tag must have a value, expr or a
            container.");
      resolveKeyword("name");
      if(variableDefined("expr")) {
         if(variableDefined("eval"))
           throw coreutils::Exception("Cannot use eval with expr.");
         global.headers[variables["name"]] = Operand(variables["expr"], global,
             parent->variables).string;
      } else if(hasContainer) {
         processContainer(container);
         if(evaluate) {
            global.headers[variables["name"]] = out;
         } else {
            global.headers[variables["name"]] = container;
         }
      } else {
         resolveKeyword("value");
         global.headers[variables["name"]] = variables["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(variableDefined("value1")) {
         resolveKeyword("value1");
         if(variableDefined("expr"))
           throw coreutils::Exception("Either␣value1␣or␣expr␣can␣be␣specified␣
               but␣not␣both.");
         if(variableDefined("value2")) {
            if(!variableDefined("type"))
              throw coreutils::Exception("type␣expected␣if␣value1␣and␣value2␣
                  specified.");
         } else
           throw coreutils::Exception("value2␣required␣if␣value1␣specified.");
         resolveKeyword("value2");
         resolveKeyword("type");
         int rc = variables["value1"].compare(variables["value2"]);
         if(((variables["type"] == "eq") && (rc == 0)) ||
            ((variables["type"] == "ne") && (rc != 0)) ||
            ((variables["type"] == "lt") && (rc == -1)) ||
            ((variables["type"] == "le") && (rc != 1)) ||
            ((variables["type"] == "gt") && (rc == 1)) ||
            ((variables["type"] == "ge") && (rc != -1)))
           booleanResult = true;
         else
           throw coreutils::Exception("type␣value␣must␣be␣'eq','ne','lt','le','
               gt','ge'.");
      } else if(variableDefined("expr")) {
         if(variableDefined("value2"))
           throw coreutils::Exception("value2␣should␣not␣be␣specified␣with␣expr.
               ");
         if(variableDefined("type"))
           throw coreutils::Exception("type␣should␣not␣be␣specified␣with␣expr.")
               ;
         booleanResult = Operand(variables["expr"], global, parent->variables).
             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(variables["sessionid"]))
        throw coreutils::Exception("sessionid does not exist.");
      resolveKeyword("sessionid");
      if(global.getSession(variables["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(!variableDefined("file"))
        throw coreutils::Exception("file␣keyword␣must␣be␣specified.");
      if(hasContainer)
        throw coreutils::Exception("include␣tag␣should␣not␣have␣a␣container.");
      hasContainer = true;
      resolveKeyword("file");
      coreutils::File file(variables["file"]);
      file.read();
      container = file.asZString();
      processContainer(container);
   }

}
```

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 <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
       ) {
      if(variableDefined("cgi"))
        resolveKeyword("cgi");
      if(variables["cgi"] == "true") {
         coreutils::ZString requestMethod(getenv("REQUEST_METHOD"));
         if(requestMethod == "POST") {
            coreutils::ZString contentLength(getenv("CONTENT_LENGTH"));
            coreutils::ZString contentType(getenv("CONTENT_TYPE"));

            std::ofstream outFile("/tmp/output.txt");

            coreutils::MString postdata;
            postdata.read(0);    // TODO: Need to limit the read characters to
                the CONTENT-LENGTH value;

            if(contentType == "multipart/form-data")
              global.setupFormData(postdata);
            else if(contentType == "application/x-www-form-urlencoded")
              global.setupFormURLEncoded(postdata);
         }
      }
      processContainer(container);
   }
}
```

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

```
#include "KeywordValue.h"
#include <iostream>

namespace jet {

   KeywordValue::KeywordValue(coreutils::ZString data, Global &global, std::map<
       coreutils::MString, coreutils::MString> &variables) : MString() {
      while(!data.eod()) {
         if(data.startsWith("$[") || data.startsWith("#[")) {
            write(global.getVariable(data, variables));
         } else {
            write(data.charAt(0));
            data.nextChar();
         }
      }
   }

   KeywordValue::~KeywordValue() {}

}
```

0.26 KeywordValue.h

```
#ifndef __KeywordValue_h__
#define __KeywordValue_h__

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

namespace  jet {

   ///
   /// KeywordValue will read the data ZString and convert any variable
       references.
   ///

   class KeywordValue : public coreutils::MString {

    public:
      KeywordValue(coreutils::ZString data, Global &global, std::map<coreutils::
          MString, coreutils::MString> &variables);
      virtual ~KeywordValue();
   };

}

#endif
```

0.27 Modifiers.cpp

```
#include "Modifiers.h"
#include "Exception.h"

namespace jet {

   void Modifiers::processToBinaryModifier(coreutils::MString &value, coreutils
       ::MString &lastConverted) {
      value.reset();
      lastConverted = "";
      char temp;
      while(!value.eod()) {
         temp = value.nextChar();
         if(strchr("\\'\".\0\r\n", temp))
           lastConverted.write('\\');
         lastConverted.write(temp);
      }
      value.reset();
   }

   void Modifiers::processFromBinaryModifier(coreutils::MString &value,
       coreutils::MString &lastConverted) {
      value.reset();
      lastConverted = "";
      while(!value.eod()) {
         if(value.ifNext("\\r"))
           lastConverted.write(13);
         else if(value.ifNext("\\n"))
           lastConverted.write(10);
         else if(value.ifNext("\\0"))
           lastConverted.write(0);
         else if(value.ifNext("\\\\"))
           lastConverted.write("\\");
         else if(value.ifNext("\\."))
           lastConverted.write(".");
         else if(value.ifNext("\\\""))
           lastConverted.write("\"");
         else if(value.ifNext("\\'"))
           lastConverted.write("'");
         else
           lastConverted.write(value.nextChar());
      }
      value.reset();
   }

   void Modifiers::processToHexModifier(coreutils::MString &value, coreutils::
       MString &lastConverted) {
      value.reset();
      lastConverted = "";
      char temp;
      while(!value.eod()) {
         temp = value.nextChar();
         char temp2 = temp;
         temp >>= 4;
         lastConverted.write(hexChar(temp));
         lastConverted.write(hexChar(temp2));
```

```
   }
   value.reset();
}

void Modifiers::processFromHexModifier(coreutils::MString &value, coreutils::
    MString &lastConverted) {
   value.reset();
   lastConverted = "";
   while(!value.eod()) {
      char ch1 = value.nextChar();
      ch1 -= 48;
      if(ch1 > 9)
        ch1 -= 7;
      ch1 <<= 4;
      ch1 &= 240;
      if(value.eod())
        coreutils::Exception("conversion␣from␣hex␣requires␣even␣number␣of␣
            characters.");
      char ch2 = value.nextChar();
      ch2 -= 48;
      if(ch2 > 9)
        ch2 -= 7;
      ch2 &= 15;
      ch1 |= ch2;
      lastConverted.write(ch1);
   }
   value.reset();
}

void Modifiers::processToBase64Modifier(coreutils::MString &value, coreutils
    ::MString &lastConverted) {
}

void Modifiers::processFromBase64Modifier(coreutils::MString &value,
    coreutils::MString &lastConverted) {
}

void Modifiers::processToUpperModifier(coreutils::MString &value, coreutils::
    MString &lastConverted) {
}

void Modifiers::processToLowerModifier(coreutils::MString &value, coreutils::
    MString &lastConverted) {
}

void Modifiers::processToCGIModifier(coreutils::MString &value, coreutils::
    MString &lastConverted) {
}

void Modifiers::processFromCGIModifier(coreutils::MString &value, coreutils::
    MString &lastConverted) {
   value.reset();
   lastConverted = "";
   while(!value.eod()) {
      char c = value.nextChar();
      if(c == '+')
        lastConverted.write('␣');
```

```
        else if(c == '%') {
           char ch1 = value.nextChar();
           ch1 -= 48;
           if(ch1 > 9)
             ch1 -= 7;
           ch1 <<= 4;
           ch1 &= 240;
           if(value.eod())
             coreutils::Exception("conversion from hex requires even number of
                 characters.");
           char ch2 = value.nextChar();
           ch2 -= 48;
           if(ch2 > 9)
             ch2 -= 7;
           ch2 &= 15;
           ch1 |= ch2;
           lastConverted.write(ch1);
        } else
          lastConverted.write(c);
      }
      value.reset();
   }

   char Modifiers::hexChar(char c) {
      c &= 15;
      c += 48;
      if(c > 57)
        c += 7;
      return c;
   }

}
```

0.28 Modifiers.h

```
#ifndef __MODIFIERS_H__
#define __MODIFIERS_H__

#include "MString.h"

namespace jet {

   class Modifiers {

    public:
      void processToBinaryModifier(coreutils::MString &value, coreutils::MString
           &lastConverted);
      void processFromBinaryModifier(coreutils::MString &value, coreutils::
          MString &lastConverted);
      void processToHexModifier(coreutils::MString &value, coreutils::MString &
          lastConverted);
      void processFromHexModifier(coreutils::MString &value, coreutils::MString
          &lastConverted);
      void processToBase64Modifier(coreutils::MString &value, coreutils::MString
           &lastConverted);
      void processFromBase64Modifier(coreutils::MString &value, coreutils::
          MString &lastConverted);
      void processToUpperModifier(coreutils::MString &value, coreutils::MString
          &lastConverted);
      void processToLowerModifier(coreutils::MString &value, coreutils::MString
          &lastConverted);
      void processToCGIModifier(coreutils::MString &value, coreutils::MString &
          lastConverted);
      void processFromCGIModifier(coreutils::MString &value, coreutils::MString
          &lastConverted);

    private:
      char hexChar(char c);

   };

}

#endif
```

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

      resolveKeyword("host");
      resolveKeyword("database");
      resolveKeyword("user");
      resolveKeyword("password");
      resolveKeyword("sessionid");

      sessionId = variables["sessionid"];

      global.addSession(sessionId, this);

      mysql = mysql_init(NULL);
      mysql = mysql_real_connect(mysql, variables["host"].c_str(), variables["
          user"].c_str(), variables["password"].c_str(), variables["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 {
                int index = column.asInteger();
                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, Global &global, std::map<coreutils::
       MString, coreutils::MString> &lvariables) {

      doubleValue = 0;

      in.skipWhitespace();

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

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

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

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

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

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

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

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

} 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, global, lvariables);
    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, global, lvariables);
    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, global, lvariables);
    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, global, lvariables);
      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, global, lvariables);
      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, global, lvariables);
      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, global, lvariables);
         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, global, lvariables);
         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, global, lvariables);
         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, global, lvariables);
          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 "Global.h"

namespace jet {

   class Operand {

    public:
      Operand(coreutils::ZString &in, Global &global, std::map<coreutils::
          MString, coreutils::MString> &lvariables);

      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(!variableDefined("file"))
        throw coreutils::Exception("file␣keyword␣must␣be␣specified.");
      if(!variableDefined("name"))
        throw coreutils::Exception("name␣keyword␣must␣be␣specified.");
      if(hasContainer)
        throw coreutils::Exception("read␣tag␣does␣not␣have␣a␣container.");
      resolveKeyword("file");
      resolveKeyword("name");
      fd = open(variables["file"].c_str(), O_RDONLY);
      if(fd < 0)
        throw coreutils::Exception("file␣name␣is␣not␣found.");
      global.variables[variables["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 "Operand.h"
#include "KeywordValue.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(!variableDefined("name"))
        throw coreutils::Exception("set␣tag␣must␣have␣name␣defined.");
      if(!variableDefined("expr") && variableDefined("value") && hasContainer)
        throw coreutils::Exception("set␣tag␣cannot␣have␣both␣value␣and␣a␣
            container.");
      if(variableDefined("expr") && !variableDefined("value") && hasContainer)
        throw coreutils::Exception("set␣tag␣cannot␣have␣both␣expr␣and␣a␣
            container.");
      if(variableDefined("expr") && variableDefined("value") && !hasContainer)
        throw coreutils::Exception("set␣tag␣cannot␣have␣both␣expr␣and␣value.");
      if(!variableDefined("expr") && !variableDefined("value") && !hasContainer)
        throw coreutils::Exception("set␣tag␣must␣have␣a␣value,␣expr␣or␣a␣
            container.");
      if(variableDefined("expr") && variableDefined("eval"))
        throw coreutils::Exception("Cannot␣use␣eval␣with␣expr.");

      resolveKeyword("name");

      if(variableDefined("expr")) {
         if(!variableDefined("scope") || (variables["scope"] == "global"))
           global.variables[variables["name"]] = Operand(variables["expr"],
               global, parent->variables).string;
         else if(variables["scope"] == "local")
           local->variables[variables["name"]] = Operand(variables["expr"],
               global, parent->variables).string;
         else if(variables["scope"] == "parent")
           local->parent->variables[variables["name"]] = Operand(variables["expr
               "], global, parent->variables).string;
      } else if(hasContainer) {
         processContainer(container);
         if(evaluate) {
            if(!variableDefined("scope") || (variables["scope"] == "global"))
              global.variables[variables["name"]] = out;
            else if(variables["scope"] == "local")
              local->variables[variables["name"]] = out;
            else if(variables["scope"] == "parent")
              local->parent->variables[variables["name"]] = out;
         } else {
            if(!variableDefined("scope") || (variables["scope"] == "global"))
              global.variables[variables["name"]] = container;
            else if(variables["scope"] == "local")
              local->variables[variables["name"]] = container;
            else if(variables["scope"] == "parent")
```

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

  }
}
```

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(variables["sessionid"]))
        throw coreutils::Exception("sessionid does not exist.");
      resolveKeyword("sessionid");
      processContainer(container);
      global.getSession(variables["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"

namespace jet {

   __stream::__stream(coreutils::ZString &in, coreutils::MString &parentOut,
       Global &global, Tag *parent, Tag *local) : Tag(in, parentOut, global,
       parent, this) {
      if(!variableDefined("name"))
        throw coreutils::Exception("stream␣tag␣must␣have␣a␣file␣name␣to␣stream."
            );

      // TODO: Output headers that have been written so far.
      // TODO: Open file with fairly small buffer size and write to end to cout
          and not the out buffers.
      // TODO: Force no further output from jet-2.0 at end of stream.

   }
}
```

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(!variableDefined(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(variableDefined("input")) {
            resolveKeyword("input");
            coreutils::ZString input(variables["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(variables["cmd"].c_str());
         close(fdo[1]);
         exit(errno);
      }
      close(fdo[1]);
      if(variableDefined("name"))
        global.variables[variables["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(!variableDefined("name"))
        throw coreutils::Exception("tag must have a name.");
      if(!hasContainer)
        throw coreutils::Exception("tag requires a container to process.");
      global.tags[variables["name"]] = container;  // TODO: process container
          for further tag definitions.
   }
}
```

0.44 Tag.cpp

```
#include "Tag.h"
#include "Exception.h"
#include "KeywordValue.h"
#include "Global.h"
#include "__mysql.h"
#include "__sql.h"
#include "__whilerow.h"
#include "__comment.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 "__whiledir.h"
#include "__tag.h"
#include "__dotag.h"
#include "__stream.h"
#include "__dump.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(variables.count(keywordName) == 0)
                    variables[keywordName] = in.getTokenExclude("\"");
                  else
                    throw coreutils::Exception("keyword␣name␣must␣be␣unique
                        ␣for␣tag.");
               }
               if(!in.ifNext("\"")) {}
            }
         }
         if(variableDefined("filterblanklines")) {
            filterBlankLines = variables["filterblanklines"] == "true" ?
                true: false;
         }
         if(variableDefined("trimlines")) {
            trimLines = variables["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(variableDefined("eval")) {
            if(variables["eval"] == "yes") {
               evaluate = true;
            } else if(variables["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);
}

void Tag::resolveKeyword(coreutils::ZString keyword) {
   variables[keyword] = KeywordValue(variables[keyword], global, parent->
       local->variables);
}

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

void Tag::parseContainer(coreutils::ZString &in, coreutils::MString &out) {
   coreutils::ZString tag;
   char *start = in.getCursor();
   while(!in.eod()) {
      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, "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, "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")) {
            while(!containerOut.eod())
              out.write(containerOut.nextChar());
            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(global.getVariable(in, local->variables));
      } 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::variableDefined(coreutils::ZString keyword) {
      return variables.find(keyword) != variables.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;
   }
}
```

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();

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

    protected:
      bool hasContainer;
      bool hasContainer2;
      bool variableDefined(coreutils::ZString variable);
      void parseContainer(coreutils::ZString &in, coreutils::MString &out);
      void processContainer(coreutils::ZString &container);
      void copyContainer(coreutils::ZString &in, coreutils::MString &out);

      Global &global;

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

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

    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);

    };

}

#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(variableDefined("value1")) {

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

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

```
            resolveKeyword("expr");
            booleanResult = Operand(variables["expr"], global, parent->variables
                ).boolean;
         } else {
            booleanResult = false;
            int rc = variables["value1"].compare(variables["value2"]);
            if(((variables["type"] == "eq") && (rc == 0)) ||
               ((variables["type"] == "ne") && (rc != 0)) ||
               ((variables["type"] == "lt") && (rc == -1)) ||
               ((variables["type"] == "le") && (rc != 1)) ||
               ((variables["type"] == "gt") && (rc == 1)) ||
               ((variables["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(variableDefined("value1")) {

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

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

```
            variables["expr"].reset();
            variables["expr"] = exprSaved;
            booleanResult = Operand(variables["expr"], global, parent->variables
                ).boolean;
         } else {
            booleanResult = false;
            int rc = variables["value1"].compare(variables["value2"]);
            if(((variables["type"] == "eq") && (rc == 0)) ||
               ((variables["type"] == "ne") && (rc != 0)) ||
               ((variables["type"] == "lt") && (rc == -1)) ||
               ((variables["type"] == "le") && (rc != 1)) ||
               ((variables["type"] == "gt") && (rc == 1)) ||
               ((variables["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(!variableDefined("path"))
        throw coreutils::Exception("whiledir tag must specify a path.");
      resolveKeyword("path");
      resolveKeyword("sort");
      if(variableDefined("sort") && (variables["sort"] == "true")) {
         std::vector<std::filesystem::directory_entry> entries;
         for(auto const &entry : std::filesystem::directory_iterator(variables["
             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(variableDefined("fullpath")) {
               resolveKeyword("fullpath");
               global.variables[variables["fullpath"]] = entry.path();
            }
            if(variableDefined("filename")) {
               resolveKeyword("filename");
               global.variables[variables["filename"]] = entry.path().filename()
                   ;
            }
            if(variableDefined("filenamenoextension")) {
               resolveKeyword("filenamenoextension");
               global.variables[variables["filenamenoextension"]] = entry.path()
                   .stem();
            }
            processContainer(container);
            container.reset();
         }
      } else {
         for(auto const &entry : std::filesystem::directory_iterator(variables["
             path"].str())) {
            if(variableDefined("fullpath")) {
               resolveKeyword("fullpath");
               global.variables[variables["fullpath"]] = entry.path();
            }
            if(variableDefined("filename")) {
               resolveKeyword("filename");
               global.variables[variables["filename"]] = entry.path().filename()
                   ;
            }
            if(variableDefined("filenamenoextension")) {
```

```
                resolveKeyword("filenamenoextension");
                global.variables[variables["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 = variables["count"].asInteger();

      while  ((count != 0) && global.getSession(variables["sessionid"])->hasRow
          ()) {
         processContainer(container);
         container.reset();
         global.getSession(variables["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(!variableDefined("file"))
        throw coreutils::Exception("write␣tag␣must␣have␣file␣defined.");
      resolveKeyword("file");
      if(!variableDefined("expr") && variableDefined("value") && hasContainer)
        throw coreutils::Exception("write␣tag␣cannot␣have␣both␣value␣and␣a␣
            container.");
      if(variableDefined("expr") && !variableDefined("value") && hasContainer)
        throw coreutils::Exception("write␣tag␣cannot␣have␣both␣expr␣and␣a␣
            container.");
      if(variableDefined("expr") && variableDefined("value") && !hasContainer)
        throw coreutils::Exception("write␣tag␣cannot␣have␣both␣expr␣and␣value.")
            ;
      if(!variableDefined("expr") && !variableDefined("value") && !hasContainer)
        throw coreutils::Exception("write␣tag␣must␣have␣a␣value,␣expr␣or␣a␣
            container.");
      if(!variableDefined("mode"))
        throw coreutils::Exception("write␣tag␣must␣have␣a␣mode␣keyword.");
      resolveKeyword("mode");
      if(variables["mode"] == "append")
        mode = O_APPEND;
      else if(variables["mode"] == "overwrite")
        mode = O_TRUNC;
      else
        throw coreutils::Exception("mode␣keyword␣must␣be␣'overwrite'␣or␣'append
            '.");
      int fd = open(variables["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 && variableDefined("value"))
        len = write(fd, variables["value"].getData(), variables["value"].
            getLength());
      else if(!hasContainer && variableDefined("expr"))
        len = write(fd, variables["expr"].getData(), variables["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
```