1 Attributes in Sage

It is possible to add information to the Sage nodes using **attributes**. An attribute has two values:

1. **a type**: the type is a positive integer, **negative integers** are reserved for the system.

2. **data**: the data field is a pointer to void that can be used to store any information.

Attributes can be attached to SgStatement, SgExpression, SgType or SgSymbol objects.

The attributes for all nodes in a file can be saved in a separate file and read back from a file to allow multiple passes. The following methods of **class SgFile** are provided:

- **void saveAttributes(char *file)**: saves the attributes data fields in **file**. The data is assumed to be contiguous in memory and of size given by **getAttributeSize()**.

- **void saveAttributes(char *file, void (*savefunction)(void *dat, FILE *f))**: saves the attributes using the provided data copy function. This is useful for instance when the attributes data have pointers to more data.

- **void readAttributes(char *file)**: Reads the attribute file generated by **void saveAttributes(char *file)**.

- **void readAttributes(char *file, void * (*readfunction)(FILE *f))**: Reads the attribute file generated by **void saveAttributes(char *file, void (*savefunction)(void *dat,FILE *f))**

- **int numberOfAttributes()**: gives the total number of attributes in a file.

- **SgAttribute *attribute(int i)**: accesses the ith attribute of the file ($i = 0, 1, ...$).

No special initialization routines are necessary to use the attribute system. However, only one .dep file can be processed at a time. The selection of the file is done by the SgProject method **project->file(i)** which sets the current file. As a consequence, when adding an attribute to a node (for instance an SgStatement object) the file that the node belongs to must be the current file. This limitation is consistent with the multi-file behavior of Sage.

Warning: if the .dep file is modified before being saved, the saving of the attributes must be done last so the node numbers are consistent. Saving the database in a file modifies the identifiers of Sage nodes.

1.1 Attribute Methods in the SgStatement, SgExpression, SgSymbol, SgType Classes

- **int numberOfAttributes()**: gives the number of attributes attached to the SgStatement (or SgExpression, SgSymbol, SgType) object.

- **int numberOfAttributes(int type)**: gives the number of attributes of type **type** attached to the object. The parameter **type** acts as a filter.

- **void *attributeValue(int i)**: returns the data field of the ith attribute ($i = 0, 1, ...$) attached to the node. Returns NULL if the node does not exist.

- **int attributeType(int i)**: returns the type of the ith attribute ($i = 0, 1, ...$) attached to the node. Returns **0** if the node does not exist.

- **void *attributeValue(int i,int type)**: same as **void *attributeValue(int i)** but only consider the attribute of type **type**.

- **void *deleteAttribute(int i)**: removes the ith attribute ($i = 0, 1, ...$) in the attribute list of the object. The data field is returned so it can be deleted if appropriate.

- **void addAttribute(int type, void *a, int size)**: adds an attribute to a node. There are no limits (except the available memory).

- **void addAttribute(int type)**: adds an attribute without data field. Data field is set to NULL.

- **void addAttribute(void *a, int size)**: adds an attribute with only a data field.

- **void addAttribute(SgAttribute *at)**: adds an attribute that has already been constructed.

- **SgAttribute *getAttribute(int i)**: returns the ith ($i = 0, 1, ...$) attribute of an object.

- **SgAttribute *getAttribute(int i,int type)**: returns the ith attribute of type **type** attached to the object.

1.2 The Attributes Class: class SgAttribute

- **SgAttribute(int t, void *pt, int size, SgStatement &st, int filenum)**: constructor for SgStatement attribute. The parameter **t** is the type of the attribute, **pt** a pointer to the data and **size** the size of the data (in bytes). This last parameter is useful if no function is given for reading and saving the attributes. The parameter **st** is the statement the attribute is attached to. When an attribute is created this way, it does not appear in the attributes list of the SgStatement node, but it can be added using the **addAttribute(SgAttribute *att)** method.

- **SgAttribute(int t, void *pt, int size, SgSymbol &st, int filenum)**: constructor for SgSymbol attribute.

- **SgAttribute(int t, void *pt, int size, SgExpression &st, int filenum)**: constructor for SgExpression attribute.

- **SgAttribute(int t, void *pt, int size, SgType &st, int filenum)**: constructor for SgType attribute.
- **int getAttributeType()**: get the type of an attribute.
- **void setAttributeType(int t)**: set the type of an attribute.
- **void *getAttributeData()**: get the data field of an attribute.
- **void *setAttributeData(void *d)**: set the data field of an attribute.
- **int getAttributeSize()**: get the size of the data field of an attribute.
- **void setAttributeSize(int s)**: set the size of the data field of an attribute.
- **typenode getTypeNode()**: get the type of the node the attribute is attached to. The type is an enumerate: **enum typenode {BIFNODE, LLNODE, SYMBNODE, TYPENODE,...}** corresponding respectively to SgStatement, SgExpression, SgSymbol and SgType.
- **void *getPtToSage()**: return a pointer to sage node the attribute is attached to. According to **getTypeNode()** this value can be casted to the right type.
- **void setPtToSage(void *sa)**: set the pointer to the Sage node.
- **void resetPtToSage()**: set the pointer to the Sage node to NULL.
- **void setPtToSage(SgStatement &st)**: set the pointer to the Sage node and also the data accessed by **getTypeNode()**.
- **void setPtToSage(SgSymbol &st)**: idem for Symbol.
- **void setPtToSage(SgExpression &st)**: idem for SgExpression.
- **void setPtToSage(SgType &st)**: idem for SgType.
- **SgStatement *getStatement()**: return the SgStatement pointer if it exists and if the attribute is attached to an SgStatement node. Otherwise this function returns NULL.
- **SgExpression *getExpression()**: idem for SgExpression.
- **SgSymbol *getSgSymbol()**: idem for Symbol.
- **SgType *getType()**: idem for SgType.
- **int getfileNumber()**: get the file number.

- **SgAttribute *copy()**: get a copy of an attribute. This function does not copy the data field.
- **SgAttribute *getNext()**: The attributes attached to the same node are chained through the next value.
- **void setNext(SgAttribute *s)**: set the list chaining of an attribute.

Note:

From the user's point of view, the preferred way to handle attributes is by using the attribute methods of SgStatement, SgExpression, SgSymbol, and SgType classes. The SgAttribute class is provided for those, who need non-standard functionality. In particular, the SgAttribute constructors should not be used directly. Instead, addAttribute and getAttribute methods should be used. For example, one might be tempted to write a code fragment like that:

```
SgExpression * CurrentExpression;
MyExprPtr = new MyExpr (  ...  );
...
SgAttribute *MyAttribute = new SgAttribute ( MY_TYPE,
                                             (void *) MyExprPtr,
                                             sizeof(MyExpr) ,
                                             *CurrentExpression , 0 );
MyExprPtr->SetAttribute ( MyAttribute );
CurrentExpression->addAttribute ( MyAttribute );
```

Instead, this code should be written as:

```
SgExpression * CurrentExpression;
MyExprPtr = new MyExpr (  ...  );
...
CurrentExpression->addAttribute ( MY_TYPE, (void *) MyExprPtr,
                                             sizeof(MyExpr) );

MyAttribute =  CurrentExpression->getAttribute( 0, MY_TYPE )
MyExprPtr->SetAttribute ( MyAttribute );
```

1.3 An Example

- The following piece of code adds an attribute to all the statements of all the files in a project.

```
        .....
#define MY_ATTRIBUTE_TYPE 2
```

```
      .....
nbfile = project->numberOfFiles();
for (i=0; i< nbfile; i++)
  {
    file = &(project->file(i));
    first =   file->firstStatement();
    temp = first;
    while (temp)
      {
        str = new char[256];
        sprintf(str,"this_is_an_attribute_added_statement_line_%d",
                temp->lineNumber());
        temp->addAttribute(MY_ATTRIBUTE_TYPE,(void *) str, strlen(str));
        temp = temp->lexNext();
      }
  }
```

- How to retrieve the attributes from the statement list.

```
for (i=0; i< nbfile; i++)
  {
    file = &(project->file(i));
    first =   file->firstStatement();
    temp = first;
    while (temp)
      {
        for (j=0; j< temp->numberOfAttributes(); j++)
          printf("Found attribute type %d ---> %s\n",
                 temp->attributeType(j),
                 (char *) temp->attributeValue(j));
        temp = temp->lexNext();
      }
  }
```

- How to retrieve the attributes directly from the file:

```
SgFile *file;
nba = file->numberOfAttributes();
for (i=0 ; i< nba; i++)
  {
    att = file->attribute(i);
  }
```

- How to save the attribute:

```
 for (i=0; i< nbfile; i++)
  {
    sprintf(nam,"attribute%d.dump",i);
    file = &(project->file(i));
    file->saveAttributes(nam);
  }
```

- the corresponding read:

```
for (i=0; i< nbfile; i++)
  {
    sprintf(nam,"attribute%d.dump",i);
    file = &(project->file(i));
    file->readAttributes(nam);
  }
```

- saving using a user function:

```
void  saveatt (void *dat,FILE *f)
{
  if (!dat || !f)
    return;
  fprintf(f,"%s\n",dat);
}
 ........

for (i=0; i< nbfile; i++)
    {
      sprintf(nam,"attribute%d.dump",i);
      file = &(project->file(i));
      file->saveAttributes(nam,saveatt);
    }
```

- The corresponding read:

```
void  *readatt (FILE *f)
{
  char *data;
  if (!f)
    return NULL;
  data = new char[256];
  fscanf(f,"%s",data);
  return (void *) data;
}
  ........
```

```
for (i=0; i< nbfile; i++)
  {
    sprintf(nam,"attribute%d.dump",i);
    file = &(project->file(i));
    file->readAttributes(nam,readatt);
  }
```

1.4 Predefined Attributes

- `#define DEPENDENCE_ATTRIBUTE   -1001`
- `#define INDUCTION_ATTRIBUTE    -1002`
- `#define ACCESS_ATTRIBUTE       -1003`
- `#define DEPGRAPH_ATTRIBUTE     -1004`
- `#define USEDLIST_ATTRIBUTE     -1005`
- `#define DEFINEDLIST_ATTRIBUTE  -1006`
- `#define NOGARBAGE_ATTRIBUTE    -1007`
- `#define GARBAGE_ATTRIBUTE      -1008`
- `#define ANNOTATION_EXPR_ATTRIBUTE   -1009`

See the information on data dependence analysis and garbage collection for details.

1.5 Limitations and BUGS

- When a copy of a node is done the attributes are not copied....

2 Garbage Collection

int SgFile::expressionGarbageCollection(int deleteExpressionNode, int verbose)
: remove all the not referenced SgExpression nodes from a file. The method returns the number of nodes deleted. The function also deletes the attributes attached to low level nodes. The parameters are the following :

int deleteExpressionNode : allows, if not zero, to remove the lower implementation of the node from data base.

int verbose : if non zero prints the message "garbage collection in process".

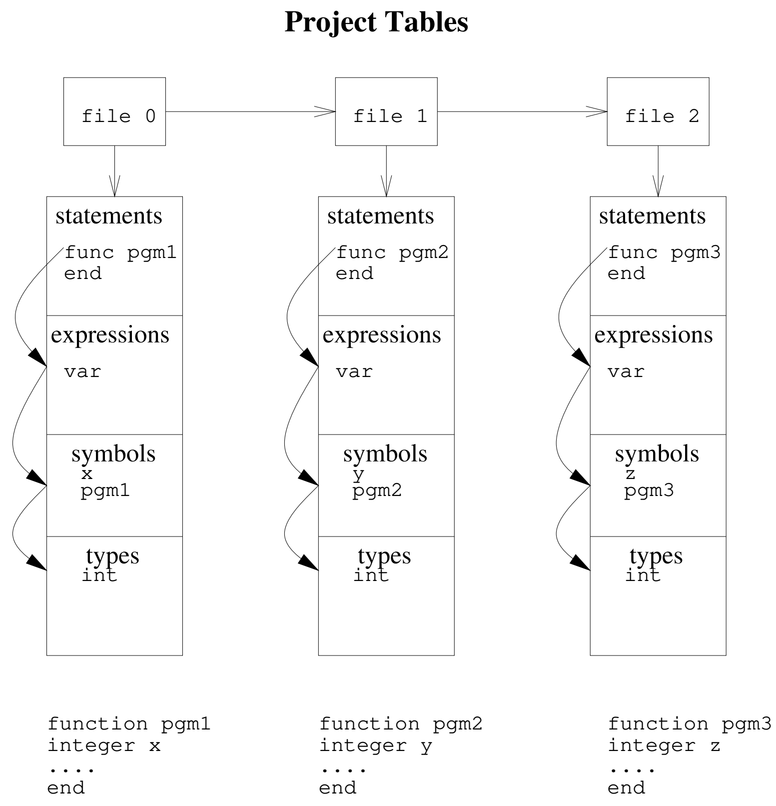

Figure 1: Database structure

Because the system cannot be aware of all the SgExpression nodes in the system (for instance node created but not yet referenced in the data base) it is possible to avoid the deletion of those nodes by adding an attribute with type **NOGARBAGE_ATTRIBUTE**.

3 Multiple Files Project

Multiple files capabilities exist in sage but there exists a set of limitations: only one file can be selected at a time. A file is selected by the methods:

```
SgProject::file(int i)
SgFile::SgFile(char * dep_file_name)
```

When a file is selected then all the operation in the data base will use the current file data structure (see Figure 1). For instance, if a statement is created, it is added in the current file structure. When the project is opened then the first file is automatically selected.

Here is an example of the usage of the multiple file capability:

```
  project =  new SgProject("test.proj");
  nbfile = project->numberOfFiles();
```

```
for (i=0; i< nbfile; i++)
  {
    file = &(project->file(i));
    printf("|||||||||||||||||||||||||||||||||||||||||||||||||||||||||\n");
    // any operations here will operate on file number i
    // file i has been selected by the method project->file(i)

    // do work here.......

    file->unparsestdout();
  }
```

3.1 Example: How to Retrieve a Function in Multiple Files

Here is an example of code to retrieve a function code in a multiple file project :

```
nbfile = project->numberOfFiles();
printf("Give the name of the function: ");
scanf("%s",str);
for (i=0; i< nbfile; i++)
  {
    file = &(project->file(i));
    if (!file)
      {
        Message("file not found",0);
        exit(1);
      }
    nbfunc = file->numberOfFunctions();
    for (j=0; j<nbfunc; j++)
      {
        func = file->functions(j);
        if (!strcmp(func->symbol()->identifier(),str))
          {
            printf("Function %s found in file %s\n",
                   func->symbol()->identifier(),
                   project->fileName(i));
            break;
          }
      }
  }
```

3.2 From one File to Another

Copy across files must be done by the member function **inline SgSymbol &SgSymbol::copyAcrossFiles(SgStatement &where)** where the parameter **where** indicates where the statements will be inserted. This method can only copy a full function, subroutine, class or structure. It duplicates all the expressions, symbols and types referenced in the body of a function, so that after the copy there are no more references to the original file.

To illustrate the problem of copying across files, let's consider this incorrect example. The result is illustrated in Figure 2:

```
project =  new SgProject("test.proj");
nbfile = project->numberOfFiles();
file = &(project->file(0));
func = file->functions(0);  // take the first function of file 0 to
                            //be copied and inserted in the other file
func->extractStmt(); // extract to allow copy
for (i=1; i< nbfile; i++)
  {
    file = &(project->file(i));
    if (!file)
      {
        Message("file not found",0);
        exit(1);
      }
    first =   file->firstStatement();
    copyoffunc = &(func->copy()); // THIS IS REAL WRONG
    first->insertStmtAfter(*copyoffunc);
    printf("|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||\n");
    file->unparsestdout();
    // unparse correctly but database is corrupted.
    sprintf(str,"debug%d.dep",i);
    file->saveDepFile(str); // incorrect database output.
  }
```

Remark: Even though the multiple file operation is very limited and fragile it is still possible to use it for creating complex data structures easily. In one file an object to be copied can be created by the parser. It can then be copied into another file to be used as a template.

```
.....
project =  new SgProject("test.proj");
nbfile = project->numberOfFiles();
// set to the first file
file = &(project->file(0));
// get the first function
func = file->functions(0);
// get the symbol
fc = func->symbol();
```

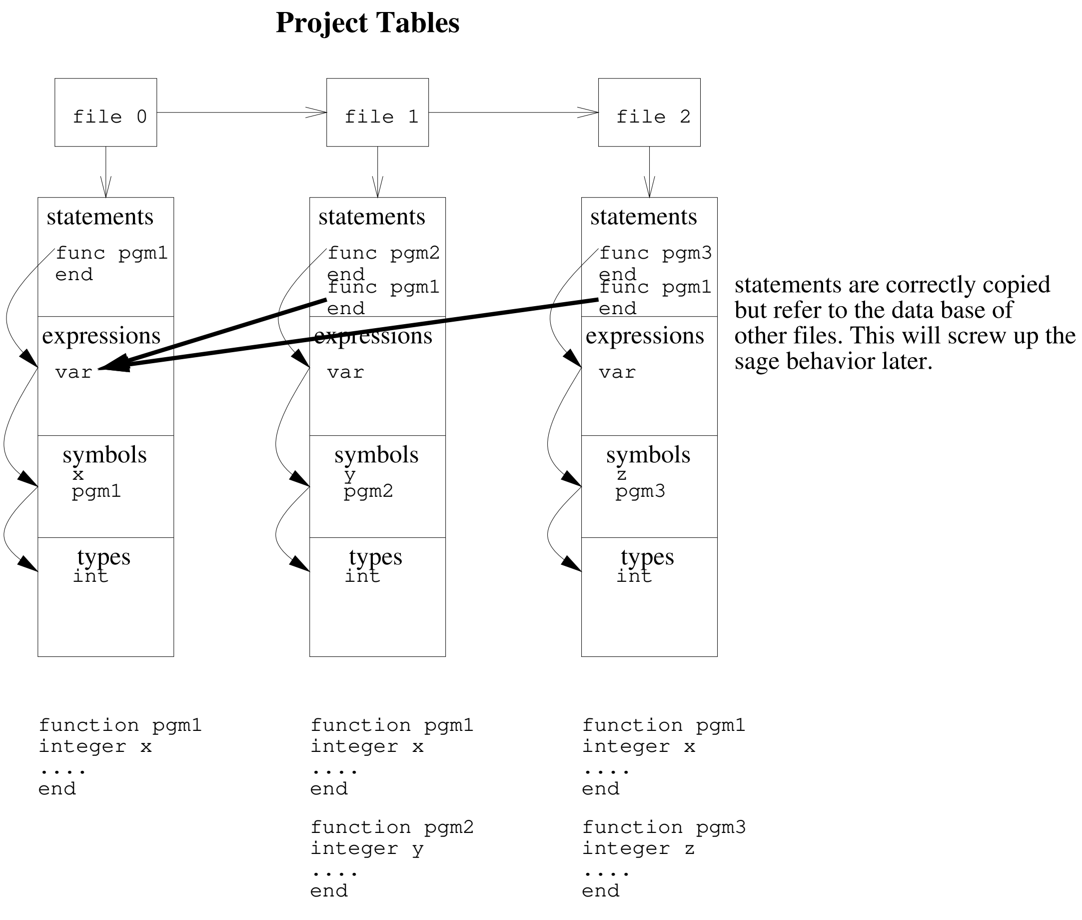

Figure 2: Database structure after the invalid copy

```
  // the following loop add the first function
  // of file number 0 to all the other files
  for (i=1; i< nbfile; i++)
    {
      file = &(project->file(i));
      first =   file->firstStatement();
      // call the copy across file function
      fccop =  &(fc->copyAcrossFiles(*first));
      // how to get the corresponding statement
      copyoffunc = fccop->body();
      file->unparsestdout();
    }
   .....
}
```

4 Data Dependences For Fortran In Sage

Sage provides data dependence computation for Fortran using the Omega Test. The information is provided using the attribute mechanism in Sage.

4.1 Initializing the Data Dependences

To initialize the data dependences in Sage two function are available:

1. `void doDependenceAnalysisOnTheFullFile(SgFile *file, int printdep,`
   `int printannotation, int verbose)`

2. `void doDependenceAnalysisForAFunction(SgFile *file,  SgStatement *func, int printdep,`
   `int printannotation, int verbose)`

The first function computes the data dependence for a full file and the second applies only on a function basis. These two functions also initialize the annotation system. The parameters are the following :

SgFile *file : The current file

SgStatement *func : the function

int printdep : display the data dependences if not zero

int printannotation : display the annotation found if not zero

int verbose : display the current function name.

In order for dependence analysis to work, the following file needs to be included:

```
#include "depInterfaceExt.h"
```

and the program must be linked with the library:

```
sage/Sage++/extentions/SageAnalysisTool/libDep.a
```

Here is, for example, the structure of a program (The full example can be found in /sage/Sage++/extentions/DemoDataDep/analyze.c)

```
.......
#include "sage++user.h"
#include "depInterfaceExt.h"
.......
main(){
.........
   project =  new SgProject(ifi);
   file = &(project->file(0));
   ........
   if (file->languageType() != ForSrc)
    {
      Message("This is not a Fortran file; exiting",0);
      exit(1);
    }
   doDependenceAnalysisOnTheFullFile(file,0,0,0);
   .......
}
```

4.2 How to Retrieve the Data Dependence Information

Attributes are used to store the information in the database:

DEPENDENCE_ATTRIBUTE : there is an attribute for each data dependence. These attributes are attached to outermost loops statements only.

```
SgStatement *stmt; // a statement
depNode *dep;
int i, nbatt;

nbatt = stmt->numberOfAttributes(DEPENDENCE_ATTRIBUTE);
for (i=0; i< nbatt; i++)
 {
    dep = (depNode *) stmt->attributeValue(i,DEPENDENCE_ATTRIBUTE);
    ...........
 }

// the dep node class
```

```
class depNode {
  .....
public:
  SgStatement *stmtin;  // source statement of the dependence
  SgStatement *stmtout; // sink statement of the dependence
  SgExpression *varin;  // source expression of the dependence
  SgExpression *varout; // sink expression of the dependence
  int          typedep; // WRONGDEP:     ????
                        // ARRAYDEP:     dependence on an array ref
                        // PRIVATEDEP:   The scalar variable (varin->symbol())
                        //               referenced can be privatized
                        // REDUCTIONDEP: The scalar variable (varin->symbol())
                        //               is subject to a reduction computation.
                        // SCALARDEP:    The scalar reference was not found to
                        //               be a reduction or private.
  int          kinddep; // The type of the dependence: ddflow, ddanti, ddoutput
  int lenghtvect;       // lenght of the vector data dep
  int distance[MAXNESTEDLOOP]; // distance if known (0 is the outermost loop)
  int knowndist[MAXNESTEDLOOP];// indicate if the distance is known
// here is an example how to use these two last vectors:
// for (i=1;i<= lenghtvect; i++)
//  {
//    if (knowndist[i])
//       printf("%d", distance[i]);
//    else
//       {
//         if (distance[i] & DEPZERO)
//          {
//            printf("0");
//          }
//         if (distance[i] & DEPGREATER)
//          {
//            printf("+");
//          }
//         if (distance[i] & DEPLESS)
//          {
//            printf("-");
//          }
//        }
//     if (i< lenghtvect)
//        printf(", ");
//   }

  depNode(SgStatement *sin,SgStatement *sout,
                SgExpression *vin,SgExpression *vout,
```

```
                int tdep, int kdep,
                int *dist, int *kdist, int le);
  ~depNode();
  void setNext(depNode *n); // set the next dependence node in the list
  depNode *nextNode();      // next dependence node in the list
  void displayDep();        // display the dependence
}
```

INDUCTION_ATTRIBUTE : Lists of the induction variables

```
  int j;
  SgStatement *loop;     // a loop statement
  Set         *inducset; // this is a set structure
                         //  (see /sage/Sage++/extentions/SageAnalysisTool/Set.h)
  PT_INDUCVAR *ind;
  inducset = (Set *) loop->attributeValue(0,INDUCTION_ATTRIBUTE);
  for (j=0; j < inducset->size(); j++)
    {
      ind = (PT_INDUCVAR) inducset->getElement(j);
      .....
    }

struct inducvar {
  int constante;         // indicate if constant or induction
  SgStatement *stmt;     // the loop it is coming from;
  SgExpression *var;     // this is a SgVarRef node;
  SgExpression *stride;  // the stride;
  SgExpression *lbound;  // the lower bound
  SgExpression *ubound;  // the upper bound
  Set *reachdef;         // not used currently;
  int level;             // internal info;
  int loopnum;           // internal info;
  int include;           // internal info;
}
typedef struct inducvar *PT_INDUCVAR;
```

ACCESS_ATTRIBUTE : The attribute contains a list of the access to variable in linear form for array access.

```
  int i;
  SgStatement     *loop;     // a loop statement
  Set             *accesset; // this is a set structure
                             //(see /sage/Sage++/extentions/SageAnalysisTool/Set.h)
  PT_ACCESSARRAY  el;
```

```
  accesset = (Set *) loop->attributeValue(0,ACCESS_ATTRIBUTE);
  for (i=0 ; i< accesset->size(); i++)
    {
      el =  (PT_ACCESSARRAY) accesset->getElement(i);
      .......
    }


struct arrayAccess {
  SgStatement *stmt;                                    // The statement the access
                                                        //  is coming from
  SgExpression *var;                                    // The access expression
  int          linear[MAXDIMARRAY][MAXNESTEDLOOP];      // matrix of access
  int          isLinear[MAXDIMARRAY];                   // flag for each line to indicate
                                                        // if linear or not
  int          cst[MAXNESTEDLOOP];                      // constante value
  // here is an example of storage
  //  do i = 1,n
  //    do j = 1,n
  //      a(2*i+3, 4*i + 5*j + 6) = ...
  //    enddo
  //  enddo
  // gives the following values of sets;
  //
  // isLinear            linear              cst
  // [0] = 1      [0][0] = 2 [0][1] = 0     [0] = 3
  // [1] = 1      [1][0] = 4 [1][1] = 5     [1] = 6
  //
  int          nbdim;                                   // the number of dimension of the
                                                        // array access (or an upper bound)
  int          size;                                    // the number of induction var that
                                                        // enclosed the statement
                                                        // (2 for previous example)
  int          type;                                    // internal info
  int          rw;                                      // if different from zero this is an
                                                        //  access in read otherwise in write
  int          scalar;                                  // if different from zero then this
                                                        // is a scalar variable access
  int          level;                                   // internal info
}
typedef struct arrayAccess *PT_ACCESSARRAY;
```

DEPGRAPH_ATTRIBUTE : The attribute contains a pointer to the class depGraph which gathers all the information concerning the data dependences. The attribute is attached to outermost loops statements only.

```
if (doloop = isSgForStmt(temp))
  {
     depg = (depGraph *) doloop->attributeValue(0,DEPGRAPH_ATTRIBUTE);
     if (depg)
        {
          // some computation here using the dep graph;
        }
  }

class depGraph {
  depNode     *current;                       // list of dependence (also stored in
                                              // the DEPENDENCE_ATTRIBUTE)
  depNode     *first;                         // first depNode
  depNode     *last;                          // last depNode
 public:
  SgFile *file;                               // the file
  Set *arrayRef;                              // same as ACCESS_ATTRIBUTE
  Set *induc;                                 // same as INDUCTION_ATTRIBUTE
  SgSymbol     *tsymb[MAXNESTEDLOOP];         // array of the loop nest symbol
                                              // (internal use)
  int          loopnum;                       // internal use
  SgStatement *func;                          // the function the loop belongs to
  SgStatement *loop;                          // the outermost loop
  int perfectNestedLevel;                     // number of perfectly nested level
  int nbstmt;                                 // internal use
  SgStatement **tabstat;                      // internal use
  int *tabtag;                                //internal use

  depGraph(SgFile *fi, SgStatement *f,SgStatement *l);
  ~depGraph();
  void addAnEdge(SgStatement *sin,SgStatement *sout,
                 SgExpression *vin,SgExpression *vout,
                 int tdep, int kdep,
                 int *dist, int *kdist,int le);
  depNode *firstNode();                         // set current on the first depNode
  nextEdge();                                   // set current on next depNode;
  void display();                               // display the data dependence;
  void scalarRefAnalysis(SgStatement *loop);    // do the scalar analysis
  void redoScalarRefAnalysis(SgStatement *loop); // redo the scalar analysis for a
                                                // an enclosed loop
                                                // useful to parallelize innermost
                                                // loops for instance
  // more there for internal purpose;
};
```

USEDLIST_ATTRIBUTE : for each statement, contains the list of used variables. These attributes are attached to executable statements only. Here is an example of how to access the attribute :

```
SgStatement *stmt;             // a statement
SgExpression *use, *varaccess; // this is an expression list node;
for (use = (SgExpression *) stmt->attributeValue(0,USEDLIST_ATTRIBUTE);
     use; use = use->rhs())
 {
    varaccess =   use->lhs();
    ......
    if (isSgArrayRefExp(varaccess)) // for instance
      {
       ......
      }
 }
```

DEFINEDLIST_ATTRIBUTE : for each statement, contains the list of defined variables. These attributes are attached to executable statements only.

```
SgStatement *stmt; // a statement
SgExpression *def; // this is an expression list node;
def = (SgExpression *) stmt->attributeValue(0,DEFINEDLIST_ATTRIBUTE);
```

4.3 Limitations and Bugs

1. The data dependence graph attributes cannot yet be saved to a file.