Sage++

A Class library for Building
Fortran 90 and C++ Restructuring Tools
November 1993

Please report any bugs you encounter in Sage++ to sage-bugs@cica.indiana.edu

1 Introduction

1.1 Aims and Objectives

Sage++ is an attempt to provide an object oriented toolkit for building program transformation systems for Fortran 77, Fortran 90, C and C++ languages. Sage++ is intended to be used by researchers interested in building parallelizing compilers, performance analysis tools, and source code optimizers. It is designed as an open C++ class library that provides the user with a set of parsers, a structured parse tree, a symbol and type table and access to programmer annotations embedded in the source text. The heart of the system is a set of functions that allow the tool builder complete freedom in restructuring the parse tree and a mechanism (called unparsing) to generating new source code from the restructured internal form.

The library is organized as a class hierarchy that provides access to the parse tree, symbol table and type table for each file in an application project. There are five basic families of classes in the library Project and files, Statements, Expressions, Symbols and Types.

1.2 History

Sage++ is based on an older system called Sigma (faust) which was, in turn, based on the Blaze compiler designed by Piyush Mehrotra in 1984. In its original form, Sigma was a tool kit for program restructuring that was accessed through the EMACS text editor (sigmacs). The primary advantage of EMACS is that it has powerful, built-in programming system, ELISP, that allows it to be highly customized. Tool builders were able to call Sigma operations from ELISP code and interactively restructure programs.

Because many potential users were interested in building "stand alone" tools (not linked to EMACS), we designed an C function interface to the Sigma system, called Sigma II (sigma), which provided a high level view of the "data base" consisting of the parse trees of the source code and the associated data dependence information. Because the underlying data structures generated by the parsers are very complex and awkward to use, Sigma II provided a level of abstraction that allowed the tool builder to work in terms of source program units like statements and expressions rather than the "low level" bit fields and linked lists of the internal structures. About the same time a group at the University of Rennes and IRISA in France, designed a Sigma "ToolBox" that provided access to more powerful transformations and users annotations in the source code. Though the ToolBox was more powerful, it also required more knowledge of the underling parser data structures.

The design of Sage++ is based on the IRISA ToolBox, but it provides an additional level of abstraction similar to, but more flexible than the Sigma II interface. One important difference between Sage and Sigma is the treatment of data dependances and control flow information.

The Sigma System, has a built-in control flow and data dependence analysis package. While this analysis system has many advanced features, such as full symbolic analysis and rudimentary interprocedural capabilities, it was also limited in scope and hard to use. What was more important, it was imbedded at the lowest level of software and written in terms of the parser data structures. Consequently it was nearly impossible to modify by users wishing to experiment with more recent advances in data dependence analysis theory.

In Sage++ it has been decided to add the control flow structures and data dependence analysis primitives on top of the user level class library. In this way, they can be easily modified or extended by the tool user. This aspect of Sage++ is not yet complete.

1.3 Overview

In this section we provide an overview of the Sage++ library. There are five basic families of classes in the library: *Projects and Files* which correspond to source files in a multi source application project, *Statements* which correspond to the basic source statements in Fortran90, C and C++, *Expressions* which are contained within statements, *Symbols* which are the basic user defined identifiers, and *Types* which are associated with each identifier and expression.

In Sage++, program parsing and program analysis and restructuring are divided into two phases. Application projects in Fortran77, Fortran90, C and C++ are first parsed, one file at a time to produce a machine independent binary internal format called a *.dep* file. For example, given a application with source files `Main.f`, `Subs.f`, `c++funs.C`, `cfuns.c` one invokes the Fortran parser `cfp` or the C parser `pc++` to generate the corresponding .dep files. Finally the user builds a *project* file, `MyProject.proj` which lists each of the .dep files, one per line. In this example, the .proj file is

```
Main.dep
Subs.dep
c++funs.dep
cfuns.dep
```

The source language type is encoded within the .dep file. It should be noted that the .dep file is a complete translation of the source including comments and the original source, up to line and column positions of tokens can be regenerated.

The purpose of the project file is so that it is possible to exploit interprocedural analysis.

1.4 Limitations

Sage++ has proven to be a powerful tool for our compiler prototyping experiments, but it still has a number of important limitations. The most important of these is that it is not easy for users to add language extensions to Fortran or C to the system. In principle this is not difficult. To add a new statement to the language one must extend the parser which is based on a the GNU Bison version of YACC. A new node type must be added to the internal form and a corresponding subclass added to the Sage++ hierarchy. the unparser module, which is table driven must be extended to recognize this new node. While we have done this several times (we have added some of the PCF extensions to Fortran and extended C++ to define our pC++ language (pC++)), it is not an easy task because it requires a complete understanding of the internal parser structures.

2 Projects and Files

A *project* is a group of parsed source files (*.dep files*) making up a program that is to be manipulated by sage. Sage can only deal with one project at a time (?). On disk, a project is represented as a file with name ending in .proj, containing a list of names of the component .dep files, one file name per line. A *file* is one of the parsed source files in a project. Each .dep file contains a parse tree. The root of the tree for a file is called the *global node* and its immediate children are the top level definitions and functions in the files. The file also contains a symbol table and type table. Refer to the overview for more details about .dep files.

2.1 SgProject

Represents the files in the current project, for all languages.

```
class  SgProject {
  public:
           SgProject(char * proj_file_name);
  int      numberOfFiles();
  SgFile & file(int i);
  char *   fileName(int i);
  int      Fortranlanguage();
  int      Clanguage();
  void     addFile(char * dep_file_name);
  void     deleteFile(SgFile * file);
};
```

SgProject is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135
- see [Restructure - main], page 139
- see ⟨undefined⟩ [Instrument], page ⟨undefined⟩
- see [Instrument - InitSymbols], page 140
- see [Instrument - InitFunctionTable], page 141
- see [Instrument - FInitialize], page 143
- see [Instrument - ProjectUnparse], page 144

- see [Instrument - FAnalyze], page 145
- see [Instrument - CAnalyze], page 154
- see [Instrument - CInitSymbols], page 155
- see [Instrument - CInitialize], page 155
- see [Instrument - OpenProject], page 157
- see [Expand Syntax - CAnalyze], page 167
- see [Expand Syntax - ProjectUnparse], page 168
- see [Expand Syntax - main], page 168
- see [Dump Info - ProjectUnparse], page 169
- see [Dump Info - main], page 185

Member Functions:

•**SgProject**(char * proj_file_name)

Sage is initialized by instantiating an SgProject with this constructor. '`proj_file_name`' is a string containing the name of the .proj file that corresponds to the program to be manipulated.

•int **numberOfFiles**()

Returns the number of files in the project.

•SgFile & **file**(int i)

Returns a reference to the '`i`'th file in the project.

•char * **fileName**(int i)

Returns the name of the '`i`'th file in the project.

•int **Fortranlanguage**()

Returns nonzero if every file in the project is a Fortran file.

•int **Clanguage**()

Returns nonzero if at least one file in the project is not a Fortran file.

•void **addFile**(char * dep_file_name)

Currently unimplemented.

•void **deleteFile**(SgFile * file)

Currently unimplemented.

SgProject Usage

Typical usage of SgProject is as in the following example:

```
#include "sage++user.h"
main()
{
  SgProject P("myproject.proj"); // opens and initializes project
  for(int i = 0; i < P.numberOfFiles(); i++){
    printf(" file is %s\n", P.fileName(i)");
    DoSomethingTo(P.file(i));
  };
}
```

Here's another one:

```
#include <stdio.h>
#include <GetOpt.h>
#include "sage++user.h"

int
main (int argc, char **argv)
{
  GetOpt getopt (argc, argv, "i:o:");
  SgProject *project;
  int option_char;
  char *ifile, *ofile;

  while ((option_char = getopt ()) != EOF)
    switch (option_char)
      {
         case 'i': ifile = getopt.optarg; break;
         case 'o': ofile = getopt.optarg; break;
         case '?': fprintf (stderr,
                     "usage: %s [i<input-proj> o<unparsed-file>]\n", argv[0]);
      }

  project = new SgProject (ifile);

  /* do something with the project */
}
```

2.2 SgFile

This class provides access to the local symbol and type tables, and the top level definitions, in the .dep file it corresponds to.

Represents files, for all languages.

```
class  SgFile {
  public:
  PTR_FILE filept;
                 SgFile(char* file_name);
  int            languageType();
  void           saveDepFile(char *dep_file);
  void           unparse(FILE *filedisc);
  void           unparsestdout();

  SgStatement *  mainProgram();
  SgStatement *  functions(int i);
  int            numberOfFunctions();
  SgStatement *  getStruct(int i);
  int            numberOfStructs();

  SgStatement *  firstStatement();
  SgSymbol *     firstSymbol();
  SgType *       firstType();
  SgExpression * firstExpression();

  SgExpression * SgExpressionWithId(int i);
  SgStatement *  SgStatementWithId( int id);
  SgStatement *  SgStatementAtLine(int lineno);
  SgSymbol *     SgSymbolWithId( int id);
  SgType *       SgTypeWithId( int id);

};
```

SgFile is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135
- see [Restructure - main], page 139
- see [Instrument - InitSymbols], page 140
- see [Instrument - InitFunctionTable], page 141
- see [Instrument - ProjectUnparse], page 144
- see [Instrument - FAnalyze], page 145
- see [Instrument - ListCollections], page 153
- see [Instrument - ListCollectionInstances], page 153
- see [Instrument - ListCollectionInvocations], page 153
- see [Instrument - CAnalyze], page 154
- see [Instrument - CInitSymbols], page 155

- see [Instrument - IsFortran], page 158
- see [Instrument - IsC], page 158
- see [Expand Syntax - ExpandSyntax], page 162
- see [Expand Syntax - Init], page 166
- see [Expand Syntax - CAnalyze], page 167
- see [Expand Syntax - ProjectUnparse], page 168
- see [Expand Syntax - main], page 168
- see [Dump Info - ProjectUnparse], page 169
- see [Dump Info - classifyStatements], page 170
- see [Dump Info - classifyExpressions], page 181
- see [Dump Info - classifySymbols], page 185
- see [Dump Info - main], page 185

Member Functions:

•**SgFile**(char* file_name)

Provides a way to generate a new SgFile which can be added to the project. (? How?)

•int **languageType**()

Returns CSrc (for C) or ForSrc (for Fortran).

•void **saveDepFile**(char *dep_file)

Saves the (internal representation of the) file as a .dep file with filename `dep_file`.

•void **unparse**(FILE *filedisc)

Unparses the (internal representation of the) file and saves the result in the file with file descriptor `filedisc`.

•void **unparsestdout**()

Unparses the (internal representation of the) file to standard output.

•SgStatement * **mainProgram**()

Returns the statement in the file at which the main program is located, or NULL if the main program is not located in this file.

•SgStatement * **functions**(int i)

Returns the ith function in the file.

•int **numberOfFunctions**()

Returns the number of functions in the file.

•SgStatement * **getStruct**(int i)

Returns the `ith` structure (? or class or union?) in the file.

•int **numberOfStructs**()

Returns the number of structures in the file.

•SgStatement * **firstStatement**()

Returns the first statement in the file.

•SgSymbol * **firstSymbol**()

Returns the first symbol in the file's symbol table.

•SgType * **firstType**()

Returns the first type in the file's type table.

•SgExpression * **firstExpression**()

Returns the first expression in the file.

•SgExpression * **SgExpressionWithId**(int i)

Returns the expression in the file with id `i`.

•SgStatement * **SgStatementWithId**(int id)

Returns the statement in the file with id `id`.

•SgStatement * **SgStatementAtLine**(int lineno)

Returns the statement at line `i` in the file.

•SgSymbol * **SgSymbolWithId**(int id)

Returns the symbol in the file with id `id`.

•SgType * **SgTypeWithId**(int id)

Returns the type in the file with id `id`.

SgFile Usage

Normally, you won't instantiate an SgFile yourself; you'll instantiate a project, then access the files in the project through the file() member of SgProject. There are two ways to traverse a file. One way is to access each subroutine and function by means of the '`numberOfFunctions()`' and '`functions(i)`' methods. For example, to print the names of all the subroutines in the project:

```
SgProject *project;

...

/* from Sage++ demo 'instrument', in function InitFunctionTable()) */

for (int i = 0; i < project->numberOfFiles(); i++) {
  SgFile *f;

  f = &(project->file(i));

  int num_routines;

  num_routines = f->numberOfFunctions();
  for (int j = 0; j < num_routines; j++){
      SgStatement *sub;
      SgSymbol *subsym;

      sub = f->functions(j);
      subsym = sub->symbol();

      printf("Function %d's name is %s\n", i*Files+j, subsym->identifier());
  }
}
```

The other way to traverse a file is to access it in lexical order starting with '`firstStatement()`':

```
void
traverseFile(SgFile *f)
{
  SgStatement *s;

  for (s = f->firstStatement(); s; s = s->lexNext()) {
    /* Do something to the statement */
  }
}
```

3 Statements

Files are broken down into *statements*. '`SgStatement`' is the base class for the representation of statements. SgStatement has many derived classes, corresponding to the many different kinds of statements in the languages Sage++ can deal with.

Each statement has a unique integer identifier *id*, and an integer *variant* tag which can be used to identify its type. These are available through SgStatement's '`id()`' and '`variant()`' member functions. SgStatement's '`thebif`' member provides access to the low level representation of statements, *bif nodes*.

Each statement has a context: it may have a lexical predecessor and a lexical successor, and it may be nested within a control block of another statement, or in a block structured definition like a C struct. This enclosing statement is called the *control parent*; it defines the basic structure of the parse tree.

3.1 SgStatement

```
class  SgStatement {
// Discuss about control parent, BIF structure etc
  public:
  PTR_BFND thebif;
                 SgStatement(int variant);
                 SgStatement(PTR_BFND bif);

  // info about statement
  int            lineNumber();          // source text line number
  int            id();                  // unique id;
  int            variant();             // the type of the statement
  SgExpression * expr(int i); // i = 0,1,2 returns the i-th expression.

  int            hasSymbol();  // returns TRUE if tmt has symbol, FALSE otherwise
  // returns the symbol field. Used by loop headers to point to the
  // loop variable symbol; Used by function and subroutine headers to
  // point to the function or subroutine name.
  SgSymbol *     symbol();        // returns the symbol field.
  char *         fileName();

  int            hasLabel();     // returns 1 if there is a label on the stmt.
  SgLabel *      label();          // the label
```

```
    // modifying the info.
    void           setlineNumber(int n); // change the line number info
    void           setId(int n);         // cannot change the id info
    void           setVariant(int n);    // change the type of the statement
    void           setExpression(int i, SgExpression &e); // change the i-th expression
    void           setLabel(SgLabel &l); // change the label
    void           setSymbol(SgSymbol &s); // change the symbol

    // Control structure
    SgStatement *  lexNext();   // the next statement in lexical order.
    SgStatement *  lexPrev();   // the previous stmt in lexical order.
    SgStatement *  controlParent(); // the enclosing control statement

    void           setLexNext(SgStatement &s); // change the lexical ordering
    void           setControlParent(SgStatement &s); // change the control parent.

  // Access statement using the tree structure
  // Describe BLOB lists here?

    int            numberOfChildrenList1();
    int            numberOfChildrenList2();
    SgStatement *  childList1(int i);
    SgStatement *  childList2(int i);
    SgStatement *  nextInChildList();
```

```
    SgStatement *  lastDeclaration();
    SgStatement *  lastExecutable();
    SgStatement *  lastNodeOfStmt();
    SgStatement *  nodeBefore();
    void           insertStmtBefore(SgStatement &s);
    void           insertStmtBefore(SgStatement &s, SgStatement &cp);
    void           insertStmtAfter(SgStatement &s);
    void           insertStmtAfter(SgStatement &s, SgStatement &cp);
    SgStatement *  extractStmt();
    SgStatement *  extractStmtBody();
    void           replaceWithStmt(SgStatement &s);
    void           deleteStmt();
    SgStatement  & copy();
    int            isIncludedInStmt(SgStatement &s);
    void           replaceSymbByExp(SgSymbol &symb, SgExpression &exp);
    void           replaceSymbBySymb(SgSymbol &symb, SgSymbol &newsymb);
    void           replaceSymbBySymbSameName(SgSymbol &symb, SgSymbol &newsymb);
    void           replaceTypeInStmt(SgType &old, SgType &newtype);
    char*          unparse();
    void           unparsestdout();
    void           sunparse(char *buffer); //unparsing functions.
    char *         comments();      //preceding comment lines.
    void           addComment(char *com);
    int            numberOfComments(); //number of preceeding comments.
    int            hasAnnotations();   //1 if there are annotations; 0 otherwise
    // These function must be removed. Doesn't make sense here.
    int            IsSymbolInScope(SgSymbol &symb); // TRUE if symbol is in scope
    int            IsSymbolReferenced(SgSymbol &symb);
    SgStatement *  getScopeForDeclare(); // return where a variable can be declared;
  };
```

SgStatement is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135
- see ⟨undefined⟩ [Instrument], page ⟨undefined⟩
- see [Instrument - InitFunctionTable], page 141
- see [Instrument - InsertFCallNode], page 142
- see [Instrument - FixLoops], page 143
- see [Instrument - FInstrumentSub], page 143
- see [Instrument - FInstrument], page 144
- see [Instrument - UnparseSub], page 145
- see [Instrument - VistaInstrument], page 145
- see [Instrument - FAnalyze], page 145
- see [Instrument - inMethodOfTheElement], page 149

- see [Instrument - whichFunctionAmI], page 150
- see [Instrument - isReferenceToCollection], page 150
- see [Instrument - ListCollections], page 153
- see [Instrument - ListCollectionInstances], page 153
- see [Instrument - ListCollectionInvocations], page 153
- see [Instrument - InsertCCallNode], page 155
- see [Instrument - CTimingInstrumentSub], page 156
- see ⟨undefined⟩ [Expand Syntax], page ⟨undefined⟩
- see [Expand Syntax - ExpandSyntax], page 162
- see [Expand Syntax - SearchInExpForCollectionArrayRef], page 162
- see [Expand Syntax - ReplaceWithGetElemPart], page 163
- see [Expand Syntax - Init], page 166
- see [Dump Info - classifyStatements], page 170
- see [Dump Info - doRoutineHeader], page 176
- see [Dump Info - doIfStmts], page 176
- see [Dump Info - doLoopStmts], page 177
- see [Dump Info - doIOStmts], page 178
- see [Dump Info - doGoToStmt], page 178
- see [Dump Info - doVarDecl], page 178

Member Functions:

•**SgStatement**(int variant)

•**SgStatement**(PTR_BFND bif)

•int **lineNumber**()

•int **id**()

•int **variant**()

•SgExpression * **expr**(int i)

There can be up to three expressions associated with any statement. For example, the C '`for`' loop is of the form '`for(expr_1; expr_2; expr_3)`'.

•int **hasSymbol**()

•SgSymbol * **symbol**()

Used for the name of a subroutine in a subroutine header statement and the induction variable of a DO loop.

•char * **fileName**()

•int **hasLabel**()

•SgLabel * **label**()

•void **setlineNumber**(int n)

•void **setId**(int n)

•void **setVariant**(int n)

•void **setExpression**(int i, SgExpression &e)

•void **setLabel**(SgLabel &l)

•void **setSymbol**(SgSymbol &s)

•SgStatement * **lexNext**()

•SgStatement * **lexPrev**()

•SgStatement * **controlParent**()

•void **setLexNext**(SgStatement &s)

•void **setControlParent**(SgStatement &s)

•int **numberOfChildrenList1**()

•int **numberOfChildrenList2**()

•SgStatement * **childList1**(int i)

•SgStatement * **childList2**(int i)

•SgStatement * **nextInChildList**()

•SgStatement * **lastDeclaration**()

If the statement is not a leaf of the parse tree, this function returns a pointer to the last declaration in the subtree the statement represents.

•SgStatement * **lastExecutable**()

•SgStatement * **lastNodeOfStmt**()

If the statement is not a leaf of the parse tree, this function returns a pointer to the last statement in the subtree it defines.

•SgStatement * **nodeBefore**()

•void **insertStmtBefore**(SgStatement &s)

•void **insertStmtBefore**(SgStatement &s, SgStatement &cp)

•void **insertStmtAfter**(SgStatement &s)

•void **insertStmtAfter**(SgStatement &s, SgStatement &cp)

•SgStatement * **extractStmt**()

•SgStatement * **extractStmtBody**()

•void **replaceWithStmt**(SgStatement &s)

•void **deleteStmt**()

•SgStatement & **copy**()

•int **isIncludedInStmt**(SgStatement &s)

•void **replaceSymbByExp**(SgSymbol &symb, SgExpression &exp)

•void **replaceSymbBySymb**(SgSymbol &symb, SgSymbol &newsymb)

•void **replaceSymbBySymbSameName**(SgSymbol &symb, SgSymbol &newsymb)

•void **replaceTypeInStmt**(SgType &old, SgType &newtype)

•char* **unparse**()

•void **unparsestdout**()

•void **sunparse**(char *buffer)

•char * **comments**()

Comments preceding or appended on the right of the statement.

•void **addComment**(char *com)

•int **numberOfComments**()

•int **hasAnnotations**()

•int **IsSymbolInScope**(SgSymbol &symb)

•int **IsSymbolReferenced**(SgSymbol &symb)

•SgStatement * **getScopeForDeclare**()

SgStatement Usage

Once a statement has been extracted from the parse tree one may make a copy of it with 'copy()' and insert the copy or the original in another position: S1.insertStmtAfter(S2) will insert S2 in the lexical position following S1.

One may also search a statement and its subexpressions for a symbol and replace all occurrences with another symbol or expression. Similarly, one may replace any reference to a given type with another type.

To illustrate the use of the statement classes consider the following simple example. Suppose we wish to traverse a file and apply the unrolling transformation to all the inner most loops whose body consists of only assignment statements. The function that accomplishes this is shown below.

```
void UnrollLoops(SgFile *file, int unroll_factor){
   SgStatement *s = file->firstStatement();
   SgForStmt *loop;
   while(s){
      if(loop = isSgForStmt(s)){
          if(loop->isAssignLoop()) loop->unrollLoop(unroll_factor);
          }
      s = s->lexNext();
   }
}
```

This function illustrates an important aspect of the programming style used in sage++. The main body of the function is a loop which scans the statements in the file in lexical order. The variable 's' is a pointer to a generic statement object. There are two ways to tell if a statement is a loop. One is to check to see if the variant is FOR_NODE. The other way is to use a special casting function, in this case, 'isSgForStmt()' which will check the variant and return a pointer cast to the correct subclass or NULL if it is not a match. Because C++ is a strongly typed language, if we wish to apply the methods of the subclass, such a cast is necessary. A function of the type

```
SgSUBCLASS * isSgSUBCLASS( SgBASECLASS *)
```

is provided for each subclass of SgStatement, SgExpression and SgType. We feel that this mechanism is an effective way to use the strong typing of C++ to reduce errors.

3.2 Header Statements

SgFuncHedrStmt

Represents functions, for all languages.

For base class, see [SgProcHedrStmt], page 27.

```
class  SgFuncHedrStmt: public SgProcHedrStmt {
  // Fortran and C function.
  // variant == FUNC_HEDR
public:
             SgFuncHedrStmt(SgSymbol &name, SgStatement Body);
             SgFuncHedrStmt(SgSymbol &name, SgType &type, SgStatement Body);
             SgFuncHedrStmt(SgSymbol &name, SgSymbol &resultName, SgType &type,
                                 SgStatement Body);
             SgFuncHedrStmt(SgSymbol &name);
             SgFuncHedrStmt(char *name);

  SgSymbol * resultName();  // name of result variable.;
  int        setResultName(SgSymbol &symbol); // set name of result variable.;

  SgType *   returnedType();  // type of returned value
  void       setReturnedType(SgType &type);  // set type of returned value
};
```

SgFuncHedrStmt is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135
- see [Instrument - inMethodOfTheElement], page 149
- see [Instrument - whichFunctionAmI], page 150
- see [Dump Info - doRoutineHeader], page 176

Member Functions:

•**SgFuncHedrStmt**(SgSymbol &name, SgStatement Body)

•**SgFuncHedrStmt**(SgSymbol &name, SgType &type, SgStatement Body)

•**SgFuncHedrStmt**(SgSymbol &name, SgSymbol &resultName, SgType &type, SgStatement Body)

•**SgFuncHedrStmt**(SgSymbol &name)

•**SgFuncHedrStmt**(char *name)

•SgSymbol * **resultName**()

•int **setResultName**(SgSymbol &symbol)

•SgType * **returnedType**()

•void **setReturnedType**(SgType &type)

SgBasicBlockStmt

Represents C basic blocks.

For base class, see Section 3.1 [SgStatement], page 11.

```
class SgBasicBlockStmt: public SgStatement {
  // in C we have: {  body; }
  // variant == BASIC_BLOCK
public:
   SgBasicBlockStmt();
};
```

SgBasicBlockStmt is used in the following places in the example programs:

- see [Expand Syntax - ReplaceWithGetElemPart], page 163

Member Functions:

•**SgBasicBlockStmt**()

SgClassStmt

Represents C++ classes.

A class declaration consists of a name, a list of super classes, a body consisting of public, private and protected functions and fields (called *Vars* here), and a list of variables that name instances, arrays of instances, pointers to instances and references to instances of the class.

For base class, see Section 3.1 [SgStatement], page 11.

```
class SgClassStmt: public SgStatement {
  // C++ class statement
  //    class name : superclass_list ElementTypeOf collection_name {
  //            body
  //     } variables_list;
  // variant == CLASS_DECL
```

```
public:
                SgClassStmt(int variant);
                SgClassStmt(SgSymbol &name);
  int           numberOfSuperClasses();
  SgSymbol *    name();
  SgSymbol *    superClass(int i);
  void          setSuperClass(int i, SgSymbol &symb);
#if 0
  int           numberOfVars();            // variables in variables_list
  SgExpression  variable(int i);  // i-th variable in variable_list
  SgExpression  collectionName(); // if an ElementType class.

  // body manipulation functions.
  int           numberOfPublicVars();
  int           numberOfPrivateVars();
  int           numberOfProtectedVars();
  SgSymbol *    publicVar(int i);
  SgSymbol *    protectedVar(int i);
  SgSymbol *    privateVar(int i);
  void          addPublicVar(SgSymbol &s);
  void          addPrivateVar(SgSymbol &s);
  void          addProtectedVar(SgSymbol &s);
  int           numberOfPublicFuns();
  int           numberOfPrivateFuns();
  int           numberOfProtectedFuns();
  SgStatement * publicFun(int i);
  SgStatement * protectedFun(int i);
  SgStatement * privateFun(int i);
  void          addPublicFun(SgStatement &s);
  void          addPrivateFun(SgStatement &s);
  void          addProtectedFun(SgStatement &s);
#endif
};
```

SgClassStmt is used in the following places in the example programs:

- see [Expand Syntax - Init], page 166

Member Functions:

•**SgClassStmt**(int variant)

•**SgClassStmt**(SgSymbol &name)

•int **numberOfSuperClasses**()

•SgSymbol * **name**()

•SgSymbol * **superClass**(int i)

•void **setSuperClass**(int i, SgSymbol &symb)

•int **numberOfVars**()

•SgExpression **variable**(int i)

•SgExpression **collectionName**()

•int **numberOfPublicVars**()

•int **numberOfPrivateVars**()

•int **numberOfProtectedVars**()

•SgSymbol * **publicVar**(int i)

•SgSymbol * **protectedVar**(int i)

•SgSymbol * **privateVar**(int i)

•void **addPublicVar**(SgSymbol &s)

•void **addPrivateVar**(SgSymbol &s)

•void **addProtectedVar**(SgSymbol &s)

•int **numberOfPublicFuns**()

•int **numberOfPrivateFuns**()

•int **numberOfProtectedFuns**()

•SgStatement * **publicFun**(int i)

•SgStatement * **protectedFun**(int i)

•SgStatement * **privateFun**(int i)

•void **addPublicFun**(SgStatement &s)

•void **addPrivateFun**(SgStatement &s)

•void **addProtectedFun**(SgStatement &s)

SgStructStmt

Represents C++ structures.

For base class, see [SgClassStmt], page 19.

```
class SgStructStmt: public SgClassStmt {
  // basic C++ structure
  // struct name ;
  //                  body
  //              } variables_list;
  // variant == STRUCT_DECL
public:
  // consider like a class.
   SgStructStmt();
   SgStructStmt(SgSymbol &name);

};
```

Member Functions:

•**SgStructStmt**()

•**SgStructStmt**(SgSymbol &name)

SgUnionStmt

Represents C++ unions.

For base class, see [SgClassStmt], page 19.

```
class SgUnionStmt: public SgClassStmt {
  // basic C++ structure
  // union name  {
  //     body
  //    } variables_list;
  // variant == UNION_DECL
public:
  // consider like a class.
   SgUnionStmt();
   SgUnionStmt(SgSymbol &name);
};
```

Member Functions:

• **SgUnionStmt**()

• **SgUnionStmt**(SgSymbol &name)

SgEnumStmt

Represents C++ enumerations.

For base class, see [SgClassStmt], page 19.

```
class SgEnumStmt: public SgClassStmt {
  // basic C++ structure
  // enum name  {
  //     body
  //    } variables_list;
  // variant == ENUM_DECL
public:
  // consider like a class.
   SgEnumStmt();
   SgEnumStmt(SgSymbol &name);
};
```

Member Functions:

• **SgEnumStmt**()

• **SgEnumStmt**(SgSymbol &name)

SgCollectionStmt

Represents pC++ collections.

For base class, see [SgClassStmt], page 19.

```
class SgCollectionStmt: public SgClassStmt {
  // basic C++ structure
  // collection name ;
  //                   body
  //               } variables_list;
  // variant == COLLECTION_DECL
```

```
public:
                SgCollectionStmt();
                SgCollectionStmt(SgSymbol &name);
#if 0
  int           numberOfElemMethods();
  SgStatement * elementMethod(int i);
  void          addElementMethod(SgStatement &s);
#endif
  SgStatement * firstElementMethod();
};
```

SgCollectionStmt is used in the following places in the example programs:

- see [Instrument - ListCollections], page 153

Member Functions:

•**SgCollectionStmt**()

•**SgCollectionStmt**(SgSymbol &name)

•int **numberOfElemMethods**()

•SgStatement * **elementMethod**(int i)

•void **addElementMethod**(SgStatement &s)

•SgStatement * **firstElementMethod**()

SgProgHedrStmt

Represents Fortran program blocks.

For base class, see Section 3.1 [SgStatement], page 11.

```
class  SgProgHedrStmt: public SgStatement {
  // fortran Program block
  // variant == PROG_HEDR
```

```
public:
                SgProgHedrStmt(PTR_BFND bif);
                SgProgHedrStmt(int variant);
                SgProgHedrStmt(SgSymbol &name, SgStatement Body);
                SgProgHedrStmt(SgSymbol &name);
                SgProgHedrStmt(char *name);
  SgSymbol &    name();
  void          setName(SgSymbol &symbol); // set program name

  int           numberOfFunctionsCalled();  // the number of functions called
  SgSymbol *    calledFunction(int i);// the i-th called function
  int           numberOfStmtFunctions();   // the number of statement funcions;
  SgStatement * statementFunc(int i); // the i-th statement function;
  int           numberOfEntryPoints();     // the number of entry points;
  SgStatement * entryPoint(int i); // the i-th entry point;
  int           numberOfParameters();       // the number of parameters;
  SgSymbol *    parameter(int i);     // the i-th parameter
  int           numberOfSpecificationStmts();
  SgStatement * specificationStmt(int i);
  int           numberOfExecutionStmts();
  SgStatement * executionStmt(int i);
  int           numberOfInternalFunctionsDefined();
  SgStatement * internalFunction(int i);
  int           numberOfInternalSubroutinesDefined();
  SgStatement * internalSubroutine(int i);
  int           numberOfInternalSubProgramsDefined();
  SgStatement * internalSubProgram(int i);

#if 0
  SgSymbol &    addVariable(SgType &T, char *name);
                                      //add a declaration for new variable

  SgStatement & addCommonBlock(char *blockname, int noOfVars, SgSymbol *Vars); // add a new common block
#endif
  int           isSymbolInScope(SgSymbol &symbol);
  int           isSymbolDeclaredHere(SgSymbol &symbol);

  // global analysis data

  int           numberOfVarsUsed();  // list of used variable access sections
  SgExpression * varsUsed(int i); // i-th var used section descriptor
  int           numberofVarsMod();   // list of modifed variable access sections
  SgExpression * varsMod(int i);   // i-th var mod section descriptor
};
```

SgProgHedrStmt is used in the following places in the example programs:

- see [Dump Info - doRoutineHeader], page 176

Member Functions:

•**SgProgHedrStmt**(PTR_BFND bif)

•**SgProgHedrStmt**(int variant)

•**SgProgHedrStmt**(SgSymbol &name, SgStatement Body)

•**SgProgHedrStmt**(SgSymbol &name)

•**SgProgHedrStmt**(char *name)

•SgSymbol & **name**()

•void **setName**(SgSymbol &symbol)

•int **numberOfFunctionsCalled**()

•SgSymbol * **calledFunction**(int i)

•int **numberOfStmtFunctions**()

•SgStatement * **statementFunc**(int i)

•int **numberOfEntryPoints**()

•SgStatement * **entryPoint**(int i)

•int **numberOfParameters**()

•SgSymbol * **parameter**(int i)

•int **numberOfSpecificationStmts**()

•SgStatement * **specificationStmt**(int i)

•int **numberOfExecutionStmts**()

•SgStatement * **executionStmt**(int i)

•int **numberOfInternalFunctionsDefined**()

•SgStatement * **internalFunction**(int i)

•int **numberOfInternalSubroutinesDefined**()

•SgStatement * **internalSubroutine**(int i)

•int **numberOfInternalSubProgramsDefined**()

•SgStatement * **internalSubProgram**(int i)

•SgSymbol & **addVariable**(SgType &T, char *name)

•SgStatement & **addCommonBlock**(char *blockname, int noOfVars, SgSymbol *Vars)

•int **isSymbolInScope**(SgSymbol &symbol)

•int **isSymbolDeclaredHere**(SgSymbol &symbol)

•int **numberOfVarsUsed**()

•SgExpression * **varsUsed**(int i)

•int **numberofVarsMod**()

•SgExpression * **varsMod**(int i)

SgProcHedrStmt

Represents Fortran subroutines.

For base class, see [SgProgHedrStmt], page 24.

```
class  SgProcHedrStmt: public SgProgHedrStmt {
  // Fortran subroutine
  // variant == PROC_HEDR
public:
                SgProcHedrStmt(int variant);
                SgProcHedrStmt(SgSymbol &name, SgStatement Body);
                SgProcHedrStmt(SgSymbol &name);
                SgProcHedrStmt(char *name);
  void          AddArg(SgExpression &arg);
  int           isRecursive();  // 1 if recursive.;
  int           numberOfEntryPoints();      // the number of entry points
                                         // other than the main, 0 for C funcs.
  SgStatement * entryPoint(int i);  // the i-th entry point
  // this is incorrect. Takes only subroutines calls into account.
  // Should be modified to take function calls into account too.
  int           numberOfCalls();            // number of calls to this proc.
  SgStatement * call(int i);       // position of the i-th call.
};
```

SgProcHedrStmt is used in the following places in the example programs:

- see [Dump Info - doRoutineHeader], page 176

Member Functions:

•**SgProcHedrStmt**(int variant)

•**SgProcHedrStmt**(SgSymbol &name, SgStatement Body)

•**SgProcHedrStmt**(SgSymbol &name)

•**SgProcHedrStmt**(char *name)

•void **AddArg**(SgExpression &arg)

•int **isRecursive**()

•int **numberOfEntryPoints**()

•SgStatement * **entryPoint**(int i)

•int **numberOfCalls**()

•SgStatement * **call**(int i)

SgBlockDataStmt

Represents Fortran block data statements.

For base class, see Section 3.1 [SgStatement], page 11.

```
class  SgBlockDataStmt: public SgStatement {
#if 0
  // Fortran Block Data statement
  // variant == BLOCK_DATA
public:
             SgBlockDataStmt(SgSymbol &name, SgStatement &body);

  SgSymbol * name();  // block data name if given;
  int        setName(SgSymbol &symbol);           // set block data name

  int        isSymbolInScope(SgSymbol &symbol);
  int        isSymbolDeclaredHere(SgSymbol &symbol);
};
```

Member Functions:

•**SgBlockDataStmt**(SgSymbol &name, SgStatement &body)

•SgSymbol * **name**()

•int **setName**(SgSymbol &symbol)

•int **isSymbolInScope**(SgSymbol &symbol)

•int **isSymbolDeclaredHere**(SgSymbol &symbol)

SgModuleStmt

Represents Fortran 90 module statements.

For base class, see Section 3.1 [SgStatement], page 11.

```
class  SgModuleStmt: public SgStatement {
#if 0
  // Fortran 90 Module statement
  // variant ==  MODULE_STMT
public:
                SgModuleStmt(SgSymbol &moduleName, SgStatement &body);
                SgModuleStmt(SgSymbol &moduleName);

  SgSymbol *    moduleName();               // module name
  void          setName(SgSymbol &symbol);        // set module name

  int           numberOfSpecificationStmts();
  int           numberOfRoutinesDefined();
  int           numberOfFunctionsDefined();
  int           numberOfSubroutinesDefined();

  SgStatement * specificationStmt(int i);
  SgStatement * routine(int i);
  SgStatement * function(int i);
  SgStatement * subroutine(int i);

  int           isSymbolInScope(SgSymbol &symbol);
  int           isSymbolDeclaredHere(SgSymbol &symbol);

  SgSymbol &    addVariable(SgType &T, char *name);
                                        //add a declaration for new variable

  SgStatement * addCommonBlock(char *blockname, int noOfVars, SgSymbol *Vars); // add a new common block
};
```

Member Functions:

•**SgModuleStmt**(SgSymbol &moduleName, SgStatement &body)

•**SgModuleStmt**(SgSymbol &moduleName)

•SgSymbol * **moduleName**()

•void **setName**(SgSymbol &symbol)

•int **numberOfSpecificationStmts**()

•int **numberOfRoutinesDefined**()

•int **numberOfFunctionsDefined**()

•int **numberOfSubroutinesDefined**()

•SgStatement * **specificationStmt**(int i)

•SgStatement * **routine**(int i)

•SgStatement * **function**(int i)

•SgStatement * **subroutine**(int i)

•int **isSymbolInScope**(SgSymbol &symbol)

•int **isSymbolDeclaredHere**(SgSymbol &symbol)

•SgSymbol & **addVariable**(SgType &T, char *name)

•SgStatement * **addCommonBlock**(char *blockname, int noOfVars, SgSymbol *Vars)

SgInterfaceStmt

Represents Fortran 90 operator interface statements.

For base class, see Section 3.1 [SgStatement], page 11.

```
class  SgInterfaceStmt: public SgStatement {
#if 0
  // Fortran 90 Operator Interface Statement
  // variant == INTERFACE_STMT
public:
                SgInterfaceStmt(SgSymbol &name, SgStatement &body,
                                  SgStatement &scope);
```

```
  SgSymbol *    interfaceName();               // interface name if given
  int           setName(SgSymbol &symbol);           // set interface name

  int           numberOfSpecificationStmts();

  SgStatement * specificationStmt(int i);

  int           isSymbolInScope(SgSymbol &symbol);
  int           isSymbolDeclaredHere(SgSymbol &symbol);
};
```

Member Functions:

•**SgInterfaceStmt**(SgSymbol &name, SgStatement &body, SgStatement &scope)

•SgSymbol * **interfaceName**()

•int **setName**(SgSymbol &symbol)

•int **numberOfSpecificationStmts**()

•SgStatement * **specificationStmt**(int i)

•int **isSymbolInScope**(SgSymbol &symbol)

•int **isSymbolDeclaredHere**(SgSymbol &symbol)

3.3 Declaration Statements

SgDeclarationStatement

For base class, see Section 3.1 [SgStatement], page 11.

```
class  SgDeclarationStatement: public SgStatement {
  // Declaration class
  // abstract class
public:
                 SgDeclarationStatement(int variant);
```

```
    SgExpression * varList();
    int            numberOfVars();
    SgExpression * var(int i);
    void           deleteVar(int i);
    void           deleteTheVar(SgExpression &var);
    void           addVar(SgExpression &exp);
  };
```

Member Functions:

•**SgDeclarationStatement**(int variant)

•SgExpression * **varList**()

•int **numberOfVars**()

•SgExpression * **var**(int i)

•void **deleteVar**(int i)

•void **deleteTheVar**(SgExpression &var)

•void **addVar**(SgExpression &exp)

SgVarDeclStmt

Represents declaration statements, for all languages.

For usage, see [SgSymbol Usage], page 108.

For base class, see [SgDeclarationStatement], page 31.

```
  class  SgVarDeclStmt: public SgDeclarationStatement {
    // Declaration Statement
    // variant == VAR_DECL
```

```
public:
  // varRefValList is a list of low-level nodes of
  // variants VAR_REFs or ARRAY_REFs or ASSIGN_OPs
                 SgVarDeclStmt(SgExpression &varRefValList,
                               SgExpression &attributeList, SgType &type);
                 SgVarDeclStmt(SgExpression &varRefValList, SgType &type);
  SgType *       type();  // the type;
  int            numberOfAttributes(); // the number of F90 attributes;
  // the attributes are: PARAMETER_OP | PUBLIC_OP |
  //    PRIVATE_OP | ALLOCATABLE_OP | EXTERNAL_OP |
  //    OPTIONAL_OP | POINTER_OP | SAVE_OP TARGET_OP
#if 0
  SgExpression & attribute(int i);
#endif
  int            numberOfSymbols();  // the number of variables declared;
  SgSymbol *     symbol(int i);

  void           deleteSymbol(int i);
  void           deleteTheSymbol(SgSymbol &symbol);
  SgExpression * initialValue(int i);  // the initial value ofthe i-th variable
  void           setInitialValue(int i, SgExpression &initVal); // sets the initial value ofthe i-th variable
  // an alternative way to initialize variables. The low-level node (VAR_REF or ARRAY_REF) is
  // replaced by a ASSIGN_OP low-level node.
};
```

SgVarDeclStmt is used in the following places in the example programs:

- see [Dump Info - doVarDecl], page 178

Member Functions:

•**SgVarDeclStmt**(SgExpression &varRefValList, SgExpression &attributeList, SgType &type)

•**SgVarDeclStmt**(SgExpression &varRefValList, SgType &type)

•SgType * **type**()

•int **numberOfAttributes**()

•SgExpression & **attribute**(int i)

•int **numberOfSymbols**()

•SgSymbol * **symbol**(int i)

•void **deleteSymbol**(int i)

•void **deleteTheSymbol**(SgSymbol &symbol)

•SgExpression * **initialValue**(int i)

•void **setInitialValue**(int i, SgExpression &initVal)

SgVarListDeclStmt

Represents lists of variable declarations, for all languages.

For base class, see [SgDeclarationStatement], page 31.

```
class  SgVarListDeclStmt: public SgDeclarationStatement {
  // Declaration Statement
  // variant == INTENT_STMT, OPTIONAL_STMT, SAVE_STMT, PUBLIC_STMT,
  // PRIVATE_STMT, EXTERNAL_STAT, INTRINSIC_STAT, DIM_STAT,
  // ALLOCATABLE_STAT, POINTER_STAT, TARGET_STAT, MODULE_PROC_STMT
public:
             SgVarListDeclStmt(int variant, SgExpression &symbolRefList);
             SgVarListDeclStmt(int variant, SgSymbol &symbolList,
                                SgStatement &scope);


  int        numberOfSymbols();
  SgSymbol * symbol(int i);
  void       appendSymbol(SgSymbol &symbol);
  void       deleteSymbol(int i);
  void       deleteTheSymbol(SgSymbol &symbol);
};
```

SgVarListDeclStmt is used in the following places in the example programs:

- see [Dump Info - doVarDecl], page 178

Member Functions:

•**SgVarListDeclStmt**(int variant, SgExpression &symbolRefList)

•**SgVarListDeclStmt**(int variant, SgSymbol &symbolList, SgStatement &scope)

•int **numberOfSymbols**()

•SgSymbol * **symbol**(int i)

•void **appendSymbol**(SgSymbol &symbol)

•void **deleteSymbol**(int i)

•void **deleteTheSymbol**(SgSymbol &symbol)

SgNestedVarListDeclStmt

Represents Fortran lists of lists of variable declarations.

For base class, see [SgDeclarationStatement], page 31.

```
class SgNestedVarListDeclStmt: public SgDeclarationStatement {
  // Declaration statement
  // variant == NAMELIST_STAT, EQUI_STAT, COMM_STAT
  // These statements have the format of a list of variable lists. For example,
  // EQUIVALENCE (A, C, D), (B, G, F), ....
public:
                 SgNestedVarListDeclStmt(int variant,
                                          SgExpression &listOfVarList);
  // varList must be of low-level variant appropriate to variant. For example,
  // if the variant is COMM_STAT, listOfVarList must be of variant COMM_LIST.

  SgExpression * lists();
  int            numberOfLists();
  SgExpression * list(int i);
#if 0
  SgExpression * leadingVar(int i);
#endif
  void           addList(SgExpression &list);
  void           addVarToList(SgExpression &varRef);
  void           deleteList(int i);
  void           deleteTheList(SgExpression &list);
  void           deleteVarInList(int i, SgExpression &varRef);
  void           deleteVarInTheList(SgExpression &list, SgExpression &varRef);
};
```

SgNestedVarListDeclStmt is used in the following places in the example programs:

- see [Dump Info - doVarDecl], page 178

Member Functions:

•**SgNestedVarListDeclStmt**(int variant, SgExpression &listOfVarList)

•SgExpression * **lists**()

•int **numberOfLists**()

•SgExpression * **list**(int i)

•SgExpression * **leadingVar**(int i)

•void **addList**(SgExpression &list)

•void **addVarToList**(SgExpression &varRef)

•void **deleteList**(int i)

•void **deleteTheList**(SgExpression &list)

•void **deleteVarInList**(int i, SgExpression &varRef)

•void **deleteVarInTheList**(SgExpression &list, SgExpression &varRef)

SgParameterStmt

Represents Fortran constants.

For base class, see [SgDeclarationStatement], page 31.

```
class SgParameterStmt: public SgDeclarationStatement {
  // Fortran constants declaration statement
  // variant = PARAM_DECL
public:
             SgParameterStmt(SgExpression &constants, SgExpression &values);
             SgParameterStmt(SgExpression &constantsWithValues);

  int        numberOfConstants();    // the number of constants declared

  SgSymbol * constant(int i);  // the i-th variable

  void       addConstant(SgSymbol &constant, SgExpression &value);
  void       deleteConstant(int i);
  void       deleteTheConstant(SgSymbol &constant);
};
```

SgParameterStmt is used in the following places in the example programs:

- see [Dump Info - doVarDecl], page 178

Member Functions:

•**SgParameterStmt**(SgExpression &constants, SgExpression &values)

•**SgParameterStmt**(SgExpression &constantsWithValues)

•int **numberOfConstants**()

•SgSymbol * **constant**(int i)

•void **addConstant**(SgSymbol &constant, SgExpression &value)

•void **deleteConstant**(int i)

•void **deleteTheConstant**(SgSymbol &constant)

SgImplicitStmt

Represents Fortran implicit type declarations.

For base class, see [SgDeclarationStatement], page 31.

```
class SgImplicitStmt: public SgDeclarationStatement {
  // Fortran implicit type declaration statement
  // variant = IMPL_DECL
public:
                 SgImplicitStmt(SgExpression &implicitLists);

  int            numberOfImplicitTypes();  // the number of implicit types declared;
  SgType *       implicitType(int i); // the i-th implicit type
  SgExpression * implicitRangeList(int i) ;
  void           appendImplicitNode(SgExpression &impNode);
#if 0
  void           addImplicitType(SgType Type, char alphabet[]);
  int            deleteImplicitItem(int i);
  int            deleteTheImplicitItem(SgExpression &implicitItem);
#endif
};
```

SgImplicitStmt is used in the following places in the example programs:

- see [Dump Info - doVarDecl], page 178

Member Functions:

•**SgImplicitStmt**(SgExpression &implicitLists)

•int **numberOfImplicitTypes**()

•SgType * **implicitType**(int i)

•SgExpression * **implicitRangeList**(int i)

•void **appendImplicitNode**(SgExpression &impNode)

•void **addImplicitType**(SgType Type, char alphabet[])

•int **deleteImplicitItem**(int i)

•int **deleteTheImplicitItem**(SgExpression &implicitItem)

SgStmtFunctionStmt

Represents Fortran statement function declarations.

For base class, see [SgDeclarationStatement], page 31.

```
class  SgStmtFunctionStmt: public SgDeclarationStatement {
#if 0
  // Fortran statement function declaration
  // variant == STMTFN_DECL
public:
             SgStmtFunctionStmt(SgSymbol &name, SgExpression &args,
                                SgStatement Body);
  SgSymbol * name();
  void       setName(SgSymbol &name);
  SgType *   type();
  int        numberOfParameters();       // the number of parameters
  SgSymbol * parameter(int i);     // the i-th parameter
};
```

Member Functions:

•**SgStmtFunctionStmt**(SgSymbol &name, SgExpression &args, SgStatement Body)

•SgSymbol * **name**()

•void **setName**(SgSymbol &name)

•SgType * **type**()

•int **numberOfParameters**()

•SgSymbol * **parameter**(int i)

SgStructureDeclStmt

Represents Fortran 90 structure declarations.

For base class, see [SgDeclarationStatement], page 31.

```
class SgStructureDeclStmt: public SgDeclarationStatement {
  // Fortran 90 structure declaration statement
  // variant == STRUCT_DECL
public:
       SgStructureDeclStmt(SgSymbol &name, SgExpression &attributes,
                           SgStatement &body);

#if 0
  int  isPrivate();
  int  isPublic();
  int  isSequence();
#endif
};
```

SgStructureDeclStmt is used in the following places in the example programs:

- see [Dump Info - doVarDecl], page 178

Member Functions:

•**SgStructureDeclStmt**(SgSymbol &name, SgExpression &attributes, SgStatement &body)

•int **isPrivate**()

•int **isPublic**()

•int **isSequence**()

SgUseStmt

Represents Fortran 90 module usage statements.

For base class, see [SgDeclarationStatement], page 31.

```
class SgUseStmt: public SgDeclarationStatement {
#if 0
  // Fortran 90 module usuage statement
  // variant = USE_STMT
public:
                 SgUseStmt(SgSymbol &moduleName, SgExpression &renameList,
                             SgStatement &scope);
  // renameList must be a list of low-level nodes of variant RENAME_NODE

  int            isOnly();
  SgSymbol *     moduleName();
  void           setModuleName(SgSymbol &moduleName);
  int            numberOfRenames();
  SgExpression * renameNode(int i);
  void           addRename(SgSymbol &localName, SgSymbol &useName);
  void           addRenameNode(SgExpression &renameNode);
  void           deleteRenameNode(int i);
  void           deleteTheRenameNode(SgExpression &renameNode);
};
```

Member Functions:

•**SgUseStmt**(SgSymbol &moduleName, SgExpression &renameList, SgStatement &scope)

•int **isOnly**()

•SgSymbol * **moduleName**()

•void **setModuleName**(SgSymbol &moduleName)

•int **numberOfRenames**()

•SgExpression * **renameNode**(int i)

•void **addRename**(SgSymbol &localName, SgSymbol &useName)

•void **addRenameNode**(SgExpression &renameNode)

•void **deleteRenameNode**(int i)

•void **deleteTheRenameNode**(SgExpression &renameNode)

SgMiscellStmt

Represents Fortran 90 contains statements, private statements, sequence statements.

For base class, see [SgDeclarationStatement], page 31.

```
class  SgMiscellStmt: public SgDeclarationStatement {
#if 0
  // Fortran 90 simple miscellaneous statements
  // variant == CONTAINS_STMT, PRIVATE_STMT, SEQUENCE_STMT
public:
   SgMiscellStmt(int variant);
};
```

Member Functions:

•**SgMiscellStmt**(int variant)

3.4 Control Statements

SgForStmt

Represents C for. Represents Fortran do.

This class has a set of high level functions for semantic analysis and transformations. This includes the analysis of implicit induction variables defined in the loop body as well as information about the interactions with other loops that may in the same loop nest. In addition there are very high level transformation such as normalization, interchange, unrolling, fusion. (In the current version of the system many of these functions are still un-implemented because they depend on data dependence information.)

For base class, see Section 3.1 [SgStatement], page 11.

```
class  SgForStmt: public SgStatement {
  // for Fortran Do and C for
  // variant = FOR_NODE
public:
                 SgForStmt(SgSymbol &do_var, SgExpression &start,
                             SgExpression &end, SgExpression &step,
                             SgStatement &body);
                 SgForStmt(SgSymbol *do_var, SgExpression *start,
                             SgExpression *end, SgExpression *step,
                             SgStatement *body);
                 SgForStmt(SgSymbol &do_var, SgExpression &start,
                             SgExpression &end, SgStatement &body);
                 SgForStmt(SgExpression &start, SgExpression &end,
                             SgExpression &step, SgStatement &body);

  SgSymbol       doName();                // the name of the loop (for F90.)
  void           setDoName(SgSymbol &doName);// sets the name of the loop(for F90)

  SgExpression * start();
  void           setStart(SgExpression &lbound);

  SgExpression * end();
  void           setEnd(SgExpression &ubound);

  SgExpression * step();
  void           setStep(SgExpression &step);

  SgLabel *      endOfLoop();

//SgExpression & bounds();   // bounds are returned as a triplet lb:ub;
//void           setBounds(SgTripletOp &bounds); // bounds are passed as a triplet lb:ub;

  // body is returned with control end statement
  //   still attached.
  SgStatement *  body();
  // s is assumed to terminate with a
  //   control end statement.
  void           set_body(SgStatement &s);
#if 0
  int            replaceBody(SgStatement &s); // new body = s and lex successors.


  int            numberOfInductVars();        // 1 if an induction variable can be found.
  SgSymbol *     inductionVar(int i);   // i-th induction variable
  SgExpression * indVarRange(int i); // range of i-th ind. var.
#endif
  int            isPerfectLoopNest();
  SgStatement *  getNextLoop();
  SgStatement *  getPreviousLoop();   // returns outer nested loop
  SgStatement *  getInnermostLoop();  // returns innermost nested loop
```

```
#if 0
  int            isLinearLoopNest();          // TRUE if the bound and step of the loops
                                   // in the loop nest are linear expressions
                                 // and use the index variables of the previous
                                // loops of the nest.
#endif
  int            isEnddoLoop();            // TRUE if the loop ends with an Enddo
  int            convertLoop();            // Convert the loop into a Good loop.
#if 0
  int            isAssignLoop();          // TRUE if the body consists only of assignments
  int            isAssignIfLoop();        // TRUE if the body consists only of assigments
  //             and conditional statements.
  //high level program transformations.
  // Most are from SIGMA Toolbox by F.Bodin et al.
  // Semantics can be found in the above reference.
  int            tiling_p(int i);
  int            tiling(int i, int tab[]);
  int            stripMining(int i);
  SgStatement    distributeLoop(int i);
  SgStatement    distributeLoopSCC();
  SgStatement    loopFusion(SgForStmt &loop);
  SgStatement    unrollLoop(int i);
  int            interchangeLoops(SgForStmt &loop);
  int            interchangeWithLoop(int i);
  int            normalized();
  int            NormalizeLoop();
  int            vectorize();
  int            vectorizeNest();
  int            ExpandScalar(SgSymbol &symbol, int i);
  int            ScalarForwardSubstitution(SgSymbol &symbol);
  int            pullStatementToFront(SgStatement &s);
  int            pullStatementToEnd(SgStatement &s);
#endif
};
```

SgForStmt is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135
- see [Instrument - FixLoops], page 143
- see [Dump Info - doLoopStmts], page 177

Member Functions:

•**SgForStmt**(SgSymbol &do_var, SgExpression &start, SgExpression &end, SgExpression &step, SgStatement &body)

•**SgForStmt**(SgSymbol *do_var, SgExpression *start, SgExpression *end, SgExpression *step, SgStatement *body)

•**SgForStmt**(SgSymbol &do_var, SgExpression &start, SgExpression &end, SgStatement &body)

•**SgForStmt**(SgExpression &start, SgExpression &end, SgExpression &step, SgStatement &body)

•SgSymbol **doName**()

•void **setDoName**(SgSymbol &doName)

•SgExpression * **start**()

•void **setStart**(SgExpression &lbound)

•SgExpression * **end**()

•void **setEnd**(SgExpression &ubound)

•SgExpression * **step**()

•void **setStep**(SgExpression &step)

•SgLabel * **endOfLoop**()

•//SgExpression & **bounds**()

•//void **setBounds**(SgTripletOp &bounds)

•SgStatement * **body**()

•void **set_body**(SgStatement &s)

•int **replaceBody**(SgStatement &s)

•int **numberOfInductVars**()

•SgSymbol * **inductionVar**(int i)

•SgExpression * **indVarRange**(int i)

•int **isPerfectLoopNest**()

•SgStatement * **getNextLoop**()

•SgStatement * **getPreviousLoop**()

•SgStatement * **getInnermostLoop**()

•int **isLinearLoopNest**()

•int **isEnddoLoop**()

•int **convertLoop**()

•int **isAssignLoop**()

•int **isAssignIfLoop**()

•int **tiling_p**(int i)

•int **tiling**(int i, int tab[])

•int **stripMining**(int i)

•SgStatement **distributeLoop**(int i)

•SgStatement **distributeLoopSCC**()

•SgStatement **loopFusion**(SgForStmt &loop)

•SgStatement **unrollLoop**(int i)

•int **interchangeLoops**(SgForStmt &loop)

•int **interchangeWithLoop**(int i)

•int **normalized**()

•int **NormalizeLoop**()

•int **vectorize**()

•int **vectorizeNest**()

•int **ExpandScalar**(SgSymbol &symbol, int i)

•int **ScalarForwardSubstitution**(SgSymbol &symbol)

•int **pullStatementToFront**(SgStatement &s)

•int **pullStatementToEnd**(SgStatement &s)

SgDoWhileStmt

Represents C do...while. Represents Fortran dowhile.

For base class, see [SgWhileStmt], page 46.

```
class  SgDoWhileStmt: public SgWhileStmt {
  // For Fortran  dowhile().. and C do {....) while();
  // variant = DO_WHILE_NODE
public:
                  SgDoWhileStmt(SgExpression &cond, SgStatement &body);
};
```

Member Functions:

•// For Fortran **dowhile**()

•**SgDoWhileStmt**(SgExpression &cond, SgStatement &body)

SgWhileStmt

Represents C while() .

For base class, see Section 3.1 [SgStatement], page 11.

```
class  SgWhileStmt: public SgStatement {
  // for C       while()
  // variant = WHILE_NODE
public:
                 SgWhileStmt(int variant);
                 SgWhileStmt(SgExpression &cond, SgStatement &body);
  SgExpression * conditional();      // the while test
#if 0
  int            numberOfInductVars(); // 1 if an induction variable can be found.
  SgSymbol *     inductionVar(int i);    // i-th induction variable
  SgExpression * indVarRange(int i); // range of i-th ind. var.
#endif
  void           replaceBody(SgStatement &s); // new body = s and lex successors.
};
```

SgWhileStmt is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135
- see [Dump Info - doLoopStmts], page 177

Member Functions:

- // for C **while**()
- **SgWhileStmt**(int variant)
- **SgWhileStmt**(SgExpression &cond, SgStatement &body)
- SgExpression * **conditional**()
- int **numberOfInductVars**()
- SgSymbol * **inductionVar**(int i)
- SgExpression * **indVarRange**(int i)
- void **replaceBody**(SgStatement &s)

SgIfStmt

Represents C if. Represents Fortran if...then...else.

For base class, see Section 3.1 [SgStatement], page 11.

```
class  SgIfStmt: public SgStatement {
  // For Fortran if then else and C  if()
  // variant == IF_NODE
public:
                                     SgIfStmt(int variant);
                                     SgIfStmt(SgExpression &cond,
                                                SgStatement &trueBody,
                                                SgStatement &falseBody,
                                                SgSymbol &construct_name);
                                     SgIfStmt(SgExpression &cond,
                                                SgStatement &trueBody,
                                                SgStatement &falseBody);
                                     SgIfStmt(SgExpression &cond,
                                                SgStatement &trueBody);
  SgStatement *                      trueBody();      // the first stmt in the True clause
  // SgBlock is needed?
  SgStatement *                      trueBody(int i); // i-th stmt in True clause
  SgStatement *                      falseBody();     // the first stmt in the False
  SgStatement *                      falseBody(int i);// i-th statement of the body.
  SgExpression *                     conditional();  // the while test
  SgSymbol *                         construct_name();
  void                               replaceTrueBody(SgStatement &s);// new body=s and lex successors.
  void                               replaceFalseBody(SgStatement &s);//new body=s and lex successors.
```

```
};
```

SgIfStmt is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135
- see [Dump Info - doIfStmts], page 176

Member Functions:

•// For Fortran if then else and C **if**()

•**SgIfStmt**(int variant)

•**SgIfStmt**(SgExpression &cond, SgStatement &trueBody, SgStatement &falseBody, SgSymbol &construct_name)

•**SgIfStmt**(SgExpression &cond, SgStatement &trueBody, SgStatement &falseBody)

•**SgIfStmt**(SgExpression &cond, SgStatement &trueBody)

•SgStatement * **trueBody**()

•SgStatement * **trueBody**(int i)

•SgStatement * **falseBody**()

•SgStatement * **falseBody**(int i)

•SgExpression * **conditional**()

•SgSymbol * **construct_name**()

•void **replaceTrueBody**(SgStatement &s)

•void **replaceFalseBody**(SgStatement &s)

SgSwitchStmt

Represents C switch. Represents Fortran case.

For base class, see Section 3.1 [SgStatement], page 11.

```
class  SgSwitchStmt: public SgStatement {
  // Fortran Case and C  switch();
  // variant == SWITCH_NODE
public:
                         SgSwitchStmt(SgExpression &selector,
                                        SgStatement &caseOptionList,
                                        SgSymbol &constructName);
  SgExpression *         selector();  // the switch selector
  void                   setSelector(SgExpression &cond);
  int                    numberOfCaseOptions();       // the number of cases
  SgStatement *          caseOption(int i);  // i-th case block
  void                   addCaseOption(SgStatement &caseOption);
#if 0
  void                   deleteCaseOption(int i);
#endif
};
```

Member Functions:

•// Fortran Case and C **switch**()

•**SgSwitchStmt**(SgExpression &selector, SgStatement &caseOptionList, SgSymbol &construct-Name)

•SgExpression * **selector**()

•void **setSelector**(SgExpression &cond)

•int **numberOfCaseOptions**()

•SgStatement * **caseOption**(int i)

•void **addCaseOption**(SgStatement &caseOption)

•void **deleteCaseOption**(int i)

SgLogIfStmt

Represents Fortran logical if.

For base class, see Section 3.1 [SgStatement], page 11.

```
class  SgLogIfStmt: public SgStatement {
  // For Fortran logical if  - only one body statement allowed
  // variant == LOGIF_NODE
```

```
public:
                 SgLogIfStmt(int variant);
                 SgLogIfStmt(SgExpression &cond, SgStatement &s);
  SgStatement *  body();  // returns reference to first stmt in the body
  SgExpression * conditional();  // the while test
  // check if the statement s is a single statement.
  void           setBody(SgStatement &s); // new body = s
  // this code won't work, since after the addition false
  //   clause, it should become SgIfThenElse statement.
  void           addFalseClause(SgStatement &s); // make it into if-then-else
  SgIfStmt *     convertLogicIf();
};
```

SgLogIfStmt is used in the following places in the example programs:

- see [Dump Info - doIfStmts], page 176

Member Functions:

•**SgLogIfStmt**(int variant)

•**SgLogIfStmt**(SgExpression &cond, SgStatement &s)

•SgStatement * **body**()

•SgExpression * **conditional**()

•void **setBody**(SgStatement &s)

•void **addFalseClause**(SgStatement &s)

•SgIfStmt * **convertLogicIf**()

SgIfElseIfStmt

Represents Fortran if...then...elseif...elseif.

For base class, see [SgIfStmt], page 47.

```
class  SgIfElseIfStmt: public SgIfStmt {
#if 0
  // For Fortran if then elseif .. elseif ... case
  // variant == ELSEIF_NODE
```

```
public:
                 SgIfElseIfStmt(SgExpression &condList, SgStatement &blockList,
                                  SgSymbol &constructName);
  int            numberOfConditionals();       // the number of conditionals
  SgStatement *  body(int b);          // block b
  void           setBody(int b);              // sets block
  SgExpression * conditional(int i); // the i-th conditional
  void           setConditional(int i);       // sets the i-th conditional
  void           addClause(SgExpression &cond, SgStatement &block);
  void           removeClause(int b);          // removes block b and it's conditional
};
```

Member Functions:

•**SgIfElseIfStmt**(SgExpression &condList, SgStatement &blockList, SgSymbol &constructName)

•int **numberOfConditionals**()

•SgStatement * **body**(int b)

•void **setBody**(int b)

•SgExpression * **conditional**(int i)

•void **setConditional**(int i)

•void **addClause**(SgExpression &cond, SgStatement &block)

•void **removeClause**(int b)

SgArithIfStmt

Represents Fortran arithmentic if.

For base class, see Section 3.1 [SgStatement], page 11.

```
class  SgArithIfStmt: public SgStatement {
  // For Fortran Arithementic if
  // variant == ARITHIF_NODE
```

```
public:
                 SgArithIfStmt(int variant);
                 SgArithIfStmt(SgExpression &cond, SgLabel &llabel,
                                 SgLabel &elabel, SgLabel &glabel);
  SgExpression * conditional();
  void           set_conditional(SgExpression &cond);
  SgExpression * label(int i);  // the <, ==, and > goto labels. in order 0->2.
  void           setLabel(SgLabel &label);
};
```

SgArithIfStmt is used in the following places in the example programs:

- see [Dump Info - doIfStmts], page 176

Member Functions:

•**SgArithIfStmt**(int variant)

•**SgArithIfStmt**(SgExpression &cond, SgLabel &llabel, SgLabel &elabel, SgLabel &glabel)

•SgExpression * **conditional**()

•void **set_conditional**(SgExpression &cond)

•SgExpression * **label**(int i)

•void **setLabel**(SgLabel &label)

SgWhereStmt

Represents Fortran where.

For base class, see [SgLogIfStmt], page 49.

```
class  SgWhereStmt: public SgLogIfStmt {
  // fortran Where stmt
  // variant == WHERE_NODE
public:
   SgWhereStmt(SgExpression &cond, SgStatement &body);
};
```

Member Functions:

•**SgWhereStmt**(SgExpression &cond, SgStatement &body)

SgWhereBlockStmt

Represents Fortran where...elsewhere.

For base class, see [SgIfStmt], page 47.

```
class  SgWhereBlockStmt: public SgIfStmt {
  // fortran Where - Elsewhere stmt
  // variant == WHERE_BLOCK_STMT
public:
   SgWhereBlockStmt(SgExpression &cond, SgStatement &trueBody,
                      SgStatement &falseBody);
};
```

Member Functions:

•**SgWhereBlockStmt**(SgExpression &cond, SgStatement &trueBody, SgStatement &falseBody)

SgCaseOptionStmt

Represents case option statement.

For base class, see Section 3.1 [SgStatement], page 11.

```
class SgCaseOptionStmt: public SgStatement {
  // Fortran case option statement
  // variant == CASE_NODE
public:
                SgCaseOptionStmt(SgExpression &caseRangeList,
                                   SgStatement &body, SgSymbol &constructName);

  SgExpression * caseRangeList();
  void           setCaseRangeList(SgExpression &caseRangeList);
  SgExpression * caseRange(int i);
  void           setCaseRange(int i, SgExpression &caseRange);
  SgStatement *  body();
  void           setBody(SgStatement &body);
```

```
};
```

Member Functions:

•**SgCaseOptionStmt**(SgExpression &caseRangeList, SgStatement &body, SgSymbol &constructName)

•SgExpression * **caseRangeList**()

•void **setCaseRangeList**(SgExpression &caseRangeList)

•SgExpression * **caseRange**(int i)

•void **setCaseRange**(int i, SgExpression &caseRange)

•SgStatement * **body**()

•void **setBody**(SgStatement &body)

3.5 Executable and Other Statements

SgExecutableStatement

For base class, see Section 3.1 [SgStatement], page 11.

```
class  SgExecutableStatement: public  SgStatement {
  // this is really a non-control, non-declaration stmt.
  // no special functions here.
public:
   SgExecutableStatement(int variant);
};
```

Member Functions:

•**SgExecutableStatement**(int variant)

SgContinueStmt

Represents continue, for all languages.

For base class, see [SgExecutableStatement], page 54.

```
class  SgContinueStmt: public SgExecutableStatement {
  // variant == CONT_STAT in Fortran and
  // variant == CONTINUE_NODE in C
public:
   SgContinueStmt();
};
```

Member Functions:

•**SgContinueStmt**()

SgControlEndStmt

Represents ends of basic blocks, for all languages.

For base class, see [SgExecutableStatement], page 54.

```
class  SgControlEndStmt: public SgExecutableStatement {
  // the end of a basic block
  // variant == CONTROL_END
public:
   SgControlEndStmt(int variant);
   SgControlEndStmt();
};
```

SgControlEndStmt is used in the following places in the example programs:

- see [Expand Syntax - ReplaceWithGetElemPart], page 163

Member Functions:

•**SgControlEndStmt**(int variant)

•**SgControlEndStmt**()

SgReturnStmt

Represents return statements, for all languages.

For base class, see [SgExecutableStatement], page 54.

```
class  SgReturnStmt: public SgExecutableStatement {
  // the return (expr) node
  // variant == RETURN_NODE//RETURN_STAT
public:
                 SgReturnStmt(SgExpression &returnValue);
                 SgReturnStmt();
  SgExpression * returnValue();
  void           setReturnValue(SgExpression &retVal);
};
```

SgReturnStmt is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135

Member Functions:

•**SgReturnStmt**(SgExpression &returnValue)

•**SgReturnStmt**()

•SgExpression * **returnValue**()

•void **setReturnValue**(SgExpression &retVal)

SgGotoStmt

Represents goto, for all languages.

For base class, see [SgExecutableStatement], page 54.

```
class  SgGotoStmt: public SgExecutableStatement {
  // the fortran or C goto
  // variant == GOTO_NODE
public:
                SgGotoStmt(SgLabel &label);
  SgLabel *     branchLabel();
#if 0
  SgStatement * target(); //the statement we go to
#endif
};
```

SgGotoStmt is used in the following places in the example programs:

- see [Dump Info - doGoToStmt], page 178

Member Functions:

•**SgGotoStmt**(SgLabel &label)

•SgLabel * **branchLabel**()

•SgStatement * **target**()

SgCExpStmt

Represents C expression statements.

For usage, see [SgExpression Usage], page 74.

For base class, see [SgExecutableStatement], page 54.

```
class  SgCExpStmt: public SgExecutableStatement {
  // C non-control expression Statment
  // variant == EXPR_STMT_NODE
public:
                 SgCExpStmt(SgExpression &exp);
                 SgCExpStmt(SgExpression &lhs, SgExpression &rhs);
  SgExpression * expr();  // the expression
  void           replaceExpression(SgExpression &e); // replace exp with e
};
```

SgCExpStmt is used in the following places in the example programs:

- see [Instrument - InsertCCallNode], page 155
- see [Expand Syntax - ReplaceWithGetElemPart], page 163

Member Functions:

•**SgCExpStmt**(SgExpression &exp)

•**SgCExpStmt**(SgExpression &lhs, SgExpression &rhs)

•SgExpression * **expr**()

•void **replaceExpression**(SgExpression &e)

SgBreakStmt

Represents C break.

For base class, see [SgExecutableStatement], page 54.

```
class  SgBreakStmt: public SgExecutableStatement {
  // C break stmt -jkg
  // variant == BREAK_NODE
public:
   SgBreakStmt();
};
```

Member Functions:

•**SgBreakStmt**()

SgAssignStmt

Represents Fortran assignment statements.

For base class, see [SgExecutableStatement], page 54.

```
class SgAssignStmt: public SgExecutableStatement {
  // Fortran assignment Statment
  // variant == ASSIGN_STAT
public:
                SgAssignStmt(int variant);
                SgAssignStmt(SgExpression &lhs, SgExpression &rhs);
  SgExpression * lhs();  // the left hand side
  SgExpression * rhs();  // the right hand side
  void          replaceLhs(SgExpression &e); // replace lhs with e
  void          replaceRhs(SgExpression &e); // replace rhs with e
#if 0
  SgExpression * varReferenced();
  SgExpression * varUsed();
  SgExpression * varDefined();
#endif
};
```

SgAssignStmt is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135
- see [Dump Info - classifyStatements], page 170

Member Functions:

•**SgAssignStmt**(int variant)

•**SgAssignStmt**(SgExpression &lhs, SgExpression &rhs)

•SgExpression * **lhs**()

•SgExpression * **rhs**()

•void **replaceLhs**(SgExpression &e)

•void **replaceRhs**(SgExpression &e)

•SgExpression * **varReferenced**()

•SgExpression * **varUsed**()

•SgExpression * **varDefined**()

SgPointerAssignStmt

Represents Fortran pointer assignment statements.

For base class, see [SgAssignStmt], page 58.

```
class  SgPointerAssignStmt: public SgAssignStmt {
  // Fortran  pointer assignment statement
  // variant == POINTER_ASSIGN_STAT
public:
   SgPointerAssignStmt(SgExpression lhs, SgExpression rhs);
};
```

Member Functions:

•**SgPointerAssignStmt**(SgExpression lhs, SgExpression rhs)

SgHeapStmt

Represents Fortran allocate and deallocate.

For base class, see [SgExecutableStatement], page 54.

```
class  SgHeapStmt: public SgExecutableStatement {
  // Fortran heap space allocation and deallocation statements
  // variant == ALLOCATE_STMT or DEALLOCATE_STMT
public:
                 SgHeapStmt(int variant, SgExpression &allocationList,
                              SgExpression &statVariable);
  SgExpression * allocationList();
  void           setAllocationList(SgExpression &allocationList);
  SgExpression * statVariable();
  void           setStatVariable(SgExpression &statVar);
};
```

Member Functions:

•**SgHeapStmt**(int variant, SgExpression &allocationList, SgExpression &statVariable)

•SgExpression * **allocationList**()

•void **setAllocationList**(SgExpression &allocationList)

•SgExpression * **statVariable**()

•void **setStatVariable**(SgExpression &statVar)

SgNullifyStmt

Represents Fortran pointer initialization statement.

For base class, see [SgExecutableStatement], page 54.

```
class  SgNullifyStmt: public SgExecutableStatement {
  // Fortran pointer initialization statement
  // variant == NULLIFY_STMT
public:
                 SgNullifyStmt(SgExpression &objectList);
  SgExpression * nullifyList();
  void           setNullifyList(SgExpression &nullifyList);
};
```

Member Functions:

•**SgNullifyStmt**(SgExpression &objectList)

•SgExpression * **nullifyList**()

•void **setNullifyList**(SgExpression &nullifyList)

SgLabelListStmt

Represents Fortran statements containing lists of labels.

For base class, see [SgExecutableStatement], page 54.

```
class SgLabelListStmt: public SgExecutableStatement {
  // the fortran
  // statements containg a list of labels
```

```
public:
                 SgLabelListStmt(int variant);
  int            numberOfTargets();
  SgExpression * labelList();
  void           setLabelList(SgExpression &labelList);
#if 0
  SgStatement *  target(int i); //the statement we go to
#endif
};
```

Member Functions:

•**SgLabelListStmt**(int variant)

•int **numberOfTargets**()

•SgExpression * **labelList**()

•void **setLabelList**(SgExpression &labelList)

•SgStatement * **target**(int i)

SgAssignedGotoStmt

Represents Fortran assigned goto statement.

For base class, see [SgLabelListStmt], page 61.

```
class  SgAssignedGotoStmt: public SgLabelListStmt {
  // the fortran
  // variant == ASSGOTO_NODE
public:
             SgAssignedGotoStmt(SgSymbol &symbol, SgExpression &labelList);
  SgSymbol * symbol();
  void       setSymbol(SgSymbol &symb);
};
```

SgAssignedGotoStmt is used in the following places in the example programs:

- see [Dump Info - doGoToStmt], page 178

Member Functions:

•**SgAssignedGotoStmt**(SgSymbol &symbol, SgExpression &labelList)

•SgSymbol * **symbol**()

•void **setSymbol**(SgSymbol &symb)

SgComputedGotoStmt

Represents Fortran computed goto statement.

For base class, see [SgLabelListStmt], page 61.

```
class  SgComputedGotoStmt: public SgLabelListStmt {
  // the fortran goto
  // variant == COMGOTO_NODE
public:
                 SgComputedGotoStmt(SgExpression &expr, SgLabel &label);
  void           addLabel(SgLabel &label);
  SgExpression * exp();
  void           setExp(SgExpression &exp);
};
```

SgComputedGotoStmt is used in the following places in the example programs:

- see [Dump Info - doGoToStmt], page 178

Member Functions:

•**SgComputedGotoStmt**(SgExpression &expr, SgLabel &label)

•void **addLabel**(SgLabel &label)

•SgExpression * **exp**()

•void **setExp**(SgExpression &exp)

SgStopOrPauseStmt

Represents Fortran stop statement.

For base class, see [SgExecutableStatement], page 54.

```
class  SgStopOrPauseStmt: public SgExecutableStatement {
  // the fortran stop
  // variant == STOP_STAT
public:
                 SgStopOrPauseStmt(int variant, SgExpression &expr);
  SgExpression * exp();
  void           setExp(SgExpression &exp);
};
```

Member Functions:

•**SgStopOrPauseStmt**(int variant, SgExpression &expr)

•SgExpression * **exp**()

•void **setExp**(SgExpression &exp)

SgCallStmt

Represents Fortran call statement.

For base class, see [SgExecutableStatement], page 54.

```
class  SgCallStmt: public SgExecutableStatement {
  // the fortran call
  // variant == PROC_STAT
public:
                 SgCallStmt(SgSymbol &name, SgExpression &args);
                 SgCallStmt(SgSymbol &name);
  SgSymbol *     name();    // name of subroutine being called
  int            numberOfArgs();  // the number of arguement expressions
  void           addArg(SgExpression &arg);
  SgExpression * arg(int i); // the i-th argument expression
```

```
#if 0
  // global analysis functions
  int            numberOfVarsUsed();
  SgExpression * varsUsed(int i);  // i-th region description
  int            numberOfVarsMod();
  SgExpression * varsMod(int i);  // i-th region description
#endif
};
```

SgCallStmt is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135
- see [Instrument - InsertFCallNode], page 142
- see [Dump Info - classifyStatements], page 170

Member Functions:

•**SgCallStmt**(SgSymbol &name, SgExpression &args)

•**SgCallStmt**(SgSymbol &name)

•SgSymbol * **name**()

•int **numberOfArgs**()

•void **addArg**(SgExpression &arg)

•SgExpression * **arg**(int i)

•int **numberOfVarsUsed**()

•SgExpression * **varsUsed**(int i)

•int **numberOfVarsMod**()

•SgExpression * **varsMod**(int i)

SgIOStmt

For base class, see [SgExecutableStatement], page 54.

```
class SgIOStmt: public SgExecutableStatement {
  // fortran input/output and their control statements
  // abstract class
public:
   SgIOStmt(int variant);
};
```

SgIOStmt is used in the following places in the example programs:

- see [Dump Info - doIOStmts], page 178

Member Functions:

•**SgIOStmt**(int variant)

SgInputOutputStmt

Represents Fortran read, write, and print.

For base class, see [SgIOStmt], page 65.

```
class SgInputOutputStmt: public SgIOStmt {
  // fortran input and output statements
  // variant = READ_STAT, WRITE_STATE, PRINT_STAT
public:
                 SgInputOutputStmt(int variant, SgExpression &specList,
                                    SgExpression &itemList);
  SgExpression * specList();
  void           setSpecList(SgExpression &specList);
  SgExpression * itemList();
  void           setItemList(SgExpression &itemList);
};
```

SgInputOutputStmt is used in the following places in the example programs:

- see [Dump Info - doIOStmts], page 178

Member Functions:

•**SgInputOutputStmt**(int variant, SgExpression &specList, SgExpression &itemList)

•SgExpression * **specList**()

•void **setSpecList**(SgExpression &specList)

•SgExpression * **itemList**()

•void **setItemList**(SgExpression &itemList)

SgIOControlStmt

Represents Fortran open, close, inquire, backspace, rewind, endfile, and format.

For base class, see [SgExecutableStatement], page 54.

```
class SgIOControlStmt: public SgExecutableStatement {
  // fortran input/output control and editing statements
  // variant = OPEN_STAT, CLOSE_STAT, INQUIRE_STAT, BACKSPACE_STAT,
  // REWIND_STAT, ENDFILE_STAT, FORMAT_STAT
public:
                 SgIOControlStmt(int variant,
                                  SgExpression &controlSpecifierList);
  SgExpression * controlSpecList();
  void           setControlSpecList(SgExpression &controlSpecList);
};
```

SgIOControlStmt is used in the following places in the example programs:

- see [Dump Info - doIOStmts], page 178

Member Functions:

•**SgIOControlStmt**(int variant, SgExpression &controlSpecifierList)

•SgExpression * **controlSpecList**()

•void **setControlSpecList**(SgExpression &controlSpecList)

SgCycleStmt

Represents Fortran 90 cycle statements.

For base class, see [SgExecutableStatement], page 54.

```
class  SgCycleStmt: public SgExecutableStatement {
  // the fortran 90 cycle statement
  // variant == CYCLE_STMT
public:
             SgCycleStmt(SgSymbol &symbol);
  SgSymbol * constructName();  // the name of the loop to cycle
  void       setConstructName(SgSymbol &constructName);
};
```

Member Functions:

•**SgCycleStmt**(SgSymbol &symbol)

•SgSymbol * **constructName**()

•void **setConstructName**(SgSymbol &constructName)

SgExitStmt

Represents Fortran 90 exit statements.

For base class, see [SgControlEndStmt], page 55.

```
class  SgExitStmt: public SgControlEndStmt {
  // the fortran 90 exit statement
  // variant == EXIT_STMT
public:
             SgExitStmt(SgSymbol &construct_name);
  SgSymbol * constructName();  // the name of the loop to cycle
  void       setConstructName(SgSymbol &constructName);
};
```

Member Functions:

•**SgExitStmt**(SgSymbol &construct_name)

•SgSymbol * **constructName**()

•void **setConstructName**(SgSymbol &constructName)

4 Expressions

Many statements contain expressions. '`SgExpression`' is the base class for all expressions. Unlike the SgStatement class hierarchy, there is not a subclass for every expression operator type. Basic binary operators such as the standard arithmetic operators are identified only the integer variant returned by the SgExpression member function '`variant()`'. The expression nodes that have their own subclass do so because they have a special type of constructor or special fields.

4.1 SgExpression

SgExpression is the base class for all the other expression classes. Every expression may have up to two operands (or subexpressions), the left hand side '`lhs()`' and the right hand side '`rhs()`'. Each expression has a type '`type()`', may have a symbol '`symbol()`', and has a unique identifier '`id()`'.

SgExpression is used to represent the basic binary operators, for C (? and Fortran?), which are listed below along with their variants:

+ ADD_OP
- SUBT_OP
* MULT_OP
/ DIV_OP
% MOD_OP

< LT_OP
> GT_OP
<= LE_OP
>= GE_OP

& BITAND_OP
| BITOR_OP
&& AND_OP
|| OR_OP
<< LSHIFT_OP
>> RSHIFT_OP

= ASSGN_OP
+= PLUS_ASSGN_OP
&= AND_ASSGN_OP
*= MULT_ASSGN_OP
/= DIV_ASSGN_OP
%= MOD_ASSGN_OP
^= XOR_ASSGN_OP
<<= LSHIFT_ASSGN_OP
>>= RSHIFT_ASSGN_OP

== EQ_OP
!= NEQ_OP

-> POINTST_OP
:: SCOPE_OP
. BIT_NUMBER

The SgExpression class:

```
class  SgExpression {
public:
  PTR_LLND thellnd;
  // generic expression class.
                SgExpression(int variant, SgExpression &lhs, SgExpression &rhs,
                               SgSymbol &s, SgType &type);
                SgExpression(int variant);
                SgExpression(PTR_LLND ll);

  SgExpression * lhs();
  SgExpression * rhs();
  SgExpression * operand(int i);
  int            variant();
  SgType *       type();
  SgSymbol *     symbol();
  int            id();
  SgExpression * nextInExprTable();

  void           setLhs(SgExpression &e);
  void           setRhs(SgExpression &e);
  void           setSymbol(SgSymbol &s);
  void           setType(SgType &t);
  void           setVariant(int v);

  SgExpression & copy();
  char *         unparse();
  void           sunparse(char *buffer);
  void           unparsestdout();
  SgExpression * IsSymbolInExpression(SgSymbol &symbol);
  void           replaceSymbolByExpression(SgSymbol &symbol, SgExpression &expr);
  SgExpression * symbRefs();
  SgExpression * arrayRefs();
  int            linearRepresentation(int *coeff, SgSymbol **symb, int *cst,
                                       int size);
  SgExpression * normalForm(int n, SgSymbol *s);
  SgExpression * coefficient(SgSymbol &s);
  int            isInteger();
  int            valueInteger();
};
```

SgExpression is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135
- see [Instrument - VistaInstrument], page 145
- see [Instrument - isReferenceToMethodOfElement], page 147
- see [Instrument - isReferenceToClassOfElement], page 148
- see [Instrument - isReferenceToCollection], page 150
- see [Instrument - ListCollectionInvocations], page 153
- see [Instrument - CTimingInstrumentSub], page 156
- see ⟨undefined⟩ [Expand Syntax], page ⟨undefined⟩
- see [Expand Syntax - isReferenceToElementField], page 160
- see [Expand Syntax - isArrayRefOfCollection], page 161
- see [Expand Syntax - ExpandSyntax], page 162
- see [Expand Syntax - SearchInExpForCollectionArrayRef], page 162
- see [Expand Syntax - ReplaceWithGetElemPart], page 163
- see [Dump Info - classifyStatements], page 170
- see [Dump Info - doLoopStmts], page 177
- see [Dump Info - classifyExpressions], page 181

Member Functions:

•**SgExpression**(int variant, SgExpression &lhs, SgExpression &rhs, SgSymbol &s, SgType &type)

•**SgExpression**(int variant)

•**SgExpression**(PTR_LLND ll)

•SgExpression * **lhs**()

•SgExpression * **rhs**()

•SgExpression * **operand**(int i)

•int **variant**()

•SgType * **type**()

•SgSymbol * **symbol**()

•int **id**()

•SgExpression * **nextInExprTable**()

•void **setLhs**(SgExpression &e)

•void **setRhs**(SgExpression &e)

•void **setSymbol**(SgSymbol &s)

•void **setType**(SgType &t)

•void **setVariant**(int v)

•SgExpression & **copy**()

Returns a deep copy.

•char * **unparse**()

•void **sunparse**(char *buffer)

•void **unparsestdout**()

•SgExpression * **IsSymbolInExpression**(SgSymbol &symbol)

Returns the most nested expression containing a reference to symbol.

•void **replaceSymbolByExpression**(SgSymbol &symbol, SgExpression &expr)

•SgExpression * **symbRefs**()

•SgExpression * **arrayRefs**()

Returns a list of array references in the expression.

•int **linearRepresentation**(int *coeff, SgSymbol **symb,int *cst, int size)

•SgExpression * **normalForm**(int n, SgSymbol *s)

Does a normal form simplify as a linear combination of the symbols in s.

•SgExpression * **coefficient**(SgSymbol &s)

Finds the multiplicative coefficient of s in the expression.

•int **isInteger**()

TRUE if the expression can be reduced to an integer value.

•int **valueInteger**()

If isInteger() is true, returns the integer value the expression can be reduced to.

SgExpression Usage

The standard binary operators do not have explicit subclasses (or constructors). However each of these operators has been overloaded as "friends" of the SgExpression class, so building expressions with these operators is a very simple task. For example, to build an expression of the form

```
(X+3)*Y+6.4
```

the program only needs two symbols and two value expressions:

```
SgVariableSymb xsymb("X"), ysymb("Y");
SgVarRefExp x(xsymb), y(ysymb);
SgValueExp  three(3), fltvalue(6.4);
SgExpression &e = (x + three)*y + fltvalue;
```

The variables X and Y are first created as symbols, then expressions that are references to the symbols are created as SgVarRefExp objects. Notice that we have not given types to the variables X and Y and declarations for X and Y have not yet been generated.

In this code, e is now a reference to the root (+) of the parse tree for the desired expression. Note that the constructors for the SgValueExp class allow any type of literal to be created.

A few standard operators have not been overloaded because this would create confusion with standard C++ operations. These include =, ==, !=, ->, and ::. They are constructed by the functions

```
SgExpression   SgAssignOp( SgExpression &lhs, SgExpression &rhs);
SgExpression       SgEqOp( SgExpression &lhs, SgExpression &rhs);
SgExpression      SgNeqOp( SgExpression &lhs, SgExpression &rhs);
SgExpression  SgPointStOp( SgExpression &lhs, SgExpression &rhs);
SgExpression    SgScopeOp( SgExpression &lhs, SgExpression &rhs);
```

Therefore, to build a C assignment statement of the form

```
X = (X+3)*Y+6.4
```

using the definitions for variables X, Y, three and fltvalue given above, we write

```
SgCExpStmt c_stmt(SgAssignOp(x.copy(), (x + three)*y + fltvalue));
```

In the case of Fortran note that the assignment is a statement and not an expression. To build the same statement in Fortran we write

```
SgAssignStmt fortran_stmt(x.copy(), (x + three)*y + fltvalue);
```

The subclasses are all designed to provide handy constructors so that it is easy to build expressions and extract values.

A function of the type

```
SgSUBCLASS * isSgSUBCLASS( SgBASECLASS *)
```

is provided for each subclass of SgStatement, SgExpression and SgType. We feel that this mechanism is an effective way to use the strong typing of C++ to reduce errors.

4.2 SgValueExp

Represents literals, for all languages.

Variants:

integers: INT_VAL
characters: CHAR_VAL
floating point numbers: FLOAT_VAL
double precision floating point numbers: DOUBLE_VAL
strings: STRING_VAL
complex numbers: COMPLEX_VAL

For usage:
see [SgExpression Usage], page 74;
see [SgSubscriptExp Usage], page 88.

For base class, see Section 4.1 [SgExpression], page 70.

```
class  SgValueExp: public SgExpression {
public:
                 SgValueExp(int value);
                 SgValueExp(char char_val);
                 SgValueExp(float float_val);
                 SgValueExp(double double_val);
                 SgValueExp(char *string_val);
                 SgValueExp(double real, double imaginary);
                 SgValueExp(SgValueExp &real, SgValueExp &imaginary);
  void           setValue(int int_val);
  void           setValue(char char_val);
  void           setValue(float float_val);
  void           setValue(double double_val);
  void           setValue(char *string_val);
  void           setValue(double real, double im);
  void           setValue(SgValueExp &real, SgValueExp & im);
  int            intValue();
  char*          floatValue();
  char           charValue();
  char*          doubleValue();
  char *         stringValue();
  SgExpression * realValue();
  SgExpression * imaginaryValue();
```

```
};
```

SgValueExp is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135
- see [Instrument - InsertFCallNode], page 142
- see [Dump Info - classifyExpressions], page 181

Member Functions:

•**SgValueExp**(int value)

•**SgValueExp**(char char_val)

•**SgValueExp**(float float_val)

•**SgValueExp**(double double_val)

•**SgValueExp**(char *string_val)

•**SgValueExp**(double real, double imaginary)

•**SgValueExp**(SgValueExp &real, SgValueExp &imaginary)

•void **setValue**(int int_val)

•void **setValue**(char char_val)

•void **setValue**(float float_val)

•void **setValue**(double double_val)

•void **setValue**(char *string_val)

•void **setValue**(double real, double im)

•void **setValue**(SgValueExp &real, SgValueExp & im)

•int **intValue**()

•char* **floatValue**()

•char **charValue**()

•char* **doubleValue**()

•char * **stringValue**()

•SgExpression * **realValue**()

•SgExpression * **imaginaryValue**()

4.3 SgFunctionCallExp

Represents function calls, for all languages.

Variant: FUNC_CALL

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgFunctionCallExp: public SgExpression {
public:
                 SgFunctionCallExp(PTR_LLND ll);
                 SgFunctionCallExp(SgSymbol &fun, SgExpression &paramList);
                 SgFunctionCallExp(SgSymbol &fun);
  SgSymbol *     funName();
  SgExpression * args();
  int            numberOfArgs();
  SgExpression * arg(int i);
  void           addArg(SgExpression &arg);
};
```

SgFunctionCallExp is used in the following places in the example programs:

- see [Instrument - isReferenceToMethodOfElement], page 147
- see [Instrument - isReferenceToClassOfElement], page 148
- see [Instrument - isReferenceToCollection], page 150
- see [Instrument - InsertCCallNode], page 155
- see [Expand Syntax - ReplaceWithGetElemPart], page 163

Member Functions:

•// **function_name**(expr1, expr2, ...)

•**SgFunctionCallExp**(PTR_LLND ll)

•**SgFunctionCallExp**(SgSymbol &fun, SgExpression &paramList)

•**SgFunctionCallExp**(SgSymbol &fun)

•SgSymbol * **funName**()

•SgExpression * **args**()

•int **numberOfArgs**()

•SgExpression * **arg**(int i)

•void **addArg**(SgExpression &arg)

4.4 SgExprListExp

Represents lists of expressions, for all languages.

Variant: EXPR_LIST

For usage, see [SgSymbol Usage], page 108.

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgExprListExp: public SgExpression {
public:
                  SgExprListExp(PTR_LLND ll);
                  SgExprListExp();
                  SgExprListExp(SgExpression &ptr);
  int             length();
  SgExpression *  elem(int i);
  SgExprListExp * next();
  SgExpression *  value();
  void            setValue(SgExpression &ptr);
  void            append(SgExpression &arg);
  void            linkToEnd(SgExpression &arg);
};
```

SgExprListExp is used in the following places in the example programs:

- see [Instrument - VistaInstrument], page 145
- see [Instrument - ListCollectionInvocations], page 153
- see [Instrument - InsertCCallNode], page 155
- see [Instrument - CTimingInstrumentSub], page 156
- see [Expand Syntax - ExpandSyntax], page 162
- see [Expand Syntax - ReplaceWithGetElemPart], page 163

Member Functions:

•**SgExprListExp**(PTR_LLND ll)

•**SgExprListExp**()

•**SgExprListExp**(SgExpression &ptr)

•int **length**()

•SgExpression * **elem**(int i)

•SgExprListExp * **next**()

•SgExpression * **value**()

•void **setValue**(SgExpression &ptr)

•void **append**(SgExpression &arg)

•void **linkToEnd**(SgExpression &arg)

4.5 SgVarRefExp

Represents scalar variable references and non-indexed array references, for all languages.

Variant: VAR_REF

For usage:
see [SgExpression Usage], page 74;
see [SgSubscriptExp Usage], page 88.

For base class, see Section 4.1 [SgExpression], page 70.

```
class  SgVarRefExp: public SgExpression {
public:
                 SgVarRefExp(PTR_LLND ll);
                 SgVarRefExp(SgSymbol &s);
  SgExpression * progatedValue(); // if scalar propogation worked
};
```

SgVarRefExp is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135
- see [Instrument - isReferenceToMethodOfElement], page 147
- see [Instrument - isReferenceToClassOfElement], page 148
- see [Instrument - isReferenceToCollection], page 150
- see [Expand Syntax - isReferenceToElementField], page 160
- see [Expand Syntax - isArrayRefOfCollection], page 161
- see [Expand Syntax - ReplaceWithGetElemPart], page 163
- see [Expand Syntax - Init], page 166

Member Functions:

•**SgVarRefExp**(PTR_LLND ll)

•**SgVarRefExp**(SgSymbol &s)

•SgExpression * **progatedValue**()

4.6 SgArrayRefExp

Represents array references, for all languages.

Variant: ARRAY_REF

For usage, see [SgSubscriptExp Usage], page 88.

For base class, see Section 4.1 [SgExpression], page 70.

```
class  SgArrayRefExp: public SgExpression {
public:
                 SgArrayRefExp(PTR_LLND ll);
                 SgArrayRefExp(SgSymbol &s);
                 SgArrayRefExp(SgSymbol &s, SgExpression &subscripts);
                 SgArrayRefExp(SgSymbol &s, SgExpression &sub1,
                               SgExpression &sub2);

                 SgArrayRefExp(SgSymbol &s, SgExpression &sub1,
                               SgExpression &sub2, SgExpression &sub3);
```

```
                 SgArrayRefExp(SgSymbol &s, SgExpression &sub1,
                                SgExpression &sub2, SgExpression &sub3,
                                SgExpression &sub4);
  int            numberOfSubscripts();  // the number of subscripts in reference
  SgExpression * subscripts();
  SgExpression * subscript(int i);
  void           addSubscript(SgExpression &e);
  void           setSymbol(SgSymbol &s);
};
```

SgArrayRefExp is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135
- see [Instrument - isReferenceToCollection], page 150
- see [Expand Syntax - isReferenceToElementField], page 160

Member Functions:

•**SgArrayRefExp**(PTR_LLND ll)

•**SgArrayRefExp**(SgSymbol &s)

•**SgArrayRefExp**(SgSymbol &s, SgExpression &subscripts)

•**SgArrayRefExp**(SgSymbol &s, SgExpression &sub1,SgExpression &sub2)

•**SgArrayRefExp**(SgSymbol &s, SgExpression &sub1,SgExpression &sub2,SgExpression &sub3)

•**SgArrayRefExp**(SgSymbol &s, SgExpression &sub1,SgExpression &sub2,SgExpression &sub3,SgExpression &sub4)

•int **numberOfSubscripts**()

•SgExpression * **subscripts**()

•SgExpression * **subscript**(int i)

•void **addSubscript**(SgExpression &e)

•void **setSymbol**(SgSymbol &s)

4.7 SgInitListExp

Represents initializations of the form {expr1,expr2,expr3}, for all languages.

Variant: INIT_LIST

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgInitListExp: public SgExpression {
public:
                 SgInitListExp(PTR_LLND ll);
                 SgInitListExp(SgExpression &expr_list);
                 SgInitListExp(int n, SgExpression *components);

  SgExpression * arg(int i); // the i-th term
  int            numberOfArgs();
  void           setArg(int i, SgExpression &e);
};
```

Member Functions:

•**SgInitListExp**(PTR_LLND ll)

•**SgInitListExp**(SgExpression &expr_list)

•**SgInitListExp**(int n, SgExpression *components)

•SgExpression * **arg**(int i)

•int **numberOfArgs**()

•void **setArg**(int i, SgExpression &e)

4.8 SgRecordRefExp

Represents C references of fields in structures.

Variant: RECORD_REF

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgRecordRefExp: public SgExpression {
```

```
public:
                 SgRecordRefExp(PTR_LLND ll);
                 SgRecordRefExp(SgSymbol &recordName, char *fieldName);
                 SgRecordRefExp(SgExpression &recordExp, char *fieldName);
  SgSymbol *     fieldName();
  SgSymbol *     recordName();
  SgExpression * record();
};
```

Member Functions:

•**SgRecordRefExp**(PTR_LLND ll)

•**SgRecordRefExp**(SgSymbol &recordName, char *fieldName)

•**SgRecordRefExp**(SgExpression &recordExp, char *fieldName)

•SgSymbol * **fieldName**()

•SgSymbol * **recordName**()

•SgExpression * **record**()

4.9 SgPntrArrRefExp

Represents C pointers used as arrays.

Variant: ARRAY_OP

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgPntrArrRefExp: public SgExpression {
```

```
public:
                 SgPntrArrRefExp(PTR_LLND ll);
                 SgPntrArrRefExp(SgExpression &p);
                 SgPntrArrRefExp(SgExpression &p, SgExpression &subscripts);
                 SgPntrArrRefExp(SgExpression &p, int n, SgExpression &sub1,
                                   SgExpression &sub2);
                 SgPntrArrRefExp(SgExpression &p, int n, SgExpression &sub1,
                                   SgExpression &sub2, SgExpression &sub3);
                 SgPntrArrRefExp(SgExpression &p, int n, SgExpression &sub1,
                                   SgExpression &sub2, SgExpression &sub3,
                                   SgExpression &sub4);
  int            dimension();  // the number of subscripts in reference
  SgExpression * subscript(int i);
  void           addSubscript(SgExpression &e);
  void           setPointer(SgExpression &p);
};
```

Member Functions:

•**SgPntrArrRefExp**(PTR_LLND ll)

•**SgPntrArrRefExp**(SgExpression &p)

•**SgPntrArrRefExp**(SgExpression &p, SgExpression &subscripts)

•**SgPntrArrRefExp**(SgExpression &p, int n, SgExpression &sub1, SgExpression &sub2)

•**SgPntrArrRefExp**(SgExpression &p, int n, SgExpression &sub1, SgExpression &sub2, SgExpression &sub3)

•**SgPntrArrRefExp**(SgExpression &p, int n, SgExpression &sub1, SgExpression &sub2, SgExpression &sub3, SgExpression &sub4)

•int **dimension**()

•SgExpression * **subscript**(int i)

•void **addSubscript**(SgExpression &e)

•void **setPointer**(SgExpression &p)

4.10 SgPointerDerefExp

Represents C pointer dereferences.

Variant: DEREF_OP

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgPointerDerefExp: public SgExpression {
public:
                 SgPointerDerefExp(PTR_LLND ll);
                 SgPointerDerefExp(SgExpression &pointerExp);
  SgExpression * pointerExp();
};
```

Member Functions:

•**SgPointerDerefExp**(PTR_LLND ll)

•**SgPointerDerefExp**(SgExpression &pointerExp)

•SgExpression * **pointerExp**()

4.11 SgFuncPntrExp

Represents C uses of pointers to functions.

Variant: FUNCTION_OP

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgFuncPntrExp: public SgExpression {
public:
                 SgFuncPntrExp(PTR_LLND ll);
                 SgFuncPntrExp(SgExpression &ptr);
  SgExpression * funExp();
  void           setFunExp(SgExpression &s);
  int            numberOfArgs();
  SgExpression * arg(int i);
  void           addArg(SgExpression &arg);
};
```

Member Functions:

•**SgFuncPntrExp**(PTR_LLND ll)

•**SgFuncPntrExp**(SgExpression &ptr)

•SgExpression * **funExp**()

•void **setFunExp**(SgExpression &s)

•int **numberOfArgs**()

•SgExpression * **arg**(int i)

•void **addArg**(SgExpression &arg)

4.12 SgSubscriptExp

Represents C array references low:up:stride for extended C.
Represents Fortran 90 array references low:up:stride.

Variant: DDOT

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgSubscriptExp: public SgExpression {
public:
                 SgSubscriptExp(PTR_LLND ll);
                 SgSubscriptExp(SgExpression &lbound, SgExpression &ubound,
                                 SgExpression &step);
                 SgSubscriptExp(SgExpression &lbound, SgExpression &ubound);
  // perhaps this function can use LlndMapping
  SgExpression * lbound();
  SgExpression * ubound();
  SgExpression * step();
};
```

Member Functions:

•**SgSubscriptExp**(PTR_LLND ll)

•**SgSubscriptExp**(SgExpression &lbound, SgExpression &ubound, SgExpression &step)

•**SgSubscriptExp**(SgExpression &lbound, SgExpression &ubound)

•SgExpression * **lbound**()

•SgExpression * **ubound**()

•SgExpression * **step**()

4.13 SgSubscriptExp

Represents C array references low:up:stride for extended C.
Represents Fortran 90 array references low:up:stride.

Variant: DDOT

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgSubscriptExp: public SgExpression {
public:
                 SgSubscriptExp(PTR_LLND ll);
                 SgSubscriptExp(SgExpression &lbound, SgExpression &ubound,
                                 SgExpression &step);
                 SgSubscriptExp(SgExpression &lbound, SgExpression &ubound);
  // perhaps this function can use LlndMapping
  SgExpression * lbound();
  SgExpression * ubound();
  SgExpression * step();
};
```

Member Functions:

•**SgSubscriptExp**(PTR_LLND ll)

•**SgSubscriptExp**(SgExpression &lbound, SgExpression &ubound, SgExpression &step)

•**SgSubscriptExp**(SgExpression &lbound, SgExpression &ubound)

•SgExpression * **lbound**()

•SgExpression * **ubound**()

•SgExpression * **step**()

SgSubscriptExp Usage

To build a fortran 90 array reference X(1:100:3, i+3) or the "extended" C array reference X[1:100:3][i+3] we can write

```
SgVariableSymb xsymb("X"), isymb("i");
SgVarRefExp  i(isymb);
SgValueExp  three(3), one(1), hunderd(100);
SgSubscriptExp range(one, hundred, three);
SgArrayRefExp(xsymb, range, i+three.copy());
```

4.14 SgUnaryExp

Represents unary expressions, for all languages.

The expressions represented, and their variants, are:

*expr DEREF_OP
&expr ADDRESS_OP
sizeof(expr) SIZE_OP
-expr MINUS_OP
+expr UNARY_ADD_OP
++ (on left) PLUSPLUS_OP
++ (on right) PLUSPLUS_OP
– (on left) MINUSMINUS_OP
– (on right) MINUSMINUS_OP
~expr BIT_COMPLEMENT_OP
!expr NOT_OP

(? Do any of these apply to Fortran?)

For base class, see Section 4.1 [SgExpression], page 70.

```
class  SgUnaryExp: public  SgExpression {
public:
                 SgUnaryExp(PTR_LLND ll);
                 SgUnaryExp(int variant, SgExpression & e);
                 SgUnaryExp(int variant, int post, SgExpression & e);
  int            post();
  SgExpression & operand();
};
```

Member Functions:

•**SgUnaryExp**(PTR_LLND ll)

•**SgUnaryExp**(int variant, SgExpression & e)

•**SgUnaryExp**(int variant, int post, SgExpression & e)

•int **post**()

•SgExpression & **operand**()

4.15 SgCastExp

Represents C casts.

Variant: CAST_OP

For base class, see Section 4.1 [SgExpression], page 70.

```
class  SgCastExp: public SgExpression {
public:
   SgCastExp(PTR_LLND ll);
   SgCastExp(SgType &t, SgExpression &e);
   SgCastExp(SgType &t);
};
```

Member Functions:

•**SgCastExp**(PTR_LLND ll)

•**SgCastExp**(SgType &t, SgExpression &e)

•**SgCastExp**(SgType &t)

4.16 SgExprIfExp

Represents C conditional expression operator ?:.

Variant: EXPR_IF

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgExprIfExp: public SgExpression {
public:
                 SgExprIfExp(PTR_LLND ll);
                 SgExprIfExp(SgExpression &exp1, SgExpression &exp2,
                             SgExpression &exp3);
  SgExpression & conditional();
  SgExpression & trueExp();
  SgExpression & falseExp();
  void           setConditional(SgExpression &c);
  void           setTrueExp(SgExpression &t);
  void           setFalseExp(SgExpression &f);
};
```

Member Functions:

•**SgExprIfExp**(PTR_LLND ll)

•**SgExprIfExp**(SgExpression &exp1,SgExpression &exp2, SgExpression &exp3)

•SgExpression & **conditional**()

•SgExpression & **trueExp**()

•SgExpression & **falseExp**()

•void **setConditional**(SgExpression &c)

•void **setTrueExp**(SgExpression &t)

•void **setFalseExp**(SgExpression &f)

4.17 SgDeleteExp

Represents C++ delete.

Variant: DELETE_OP

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgDeleteExp: public SgExpression {
```

```
public:
   SgDeleteExp(PTR_LLND ll);
   SgDeleteExp(SgExpression &size, SgExpression &expr);
   SgDeleteExp(SgExpression &expr);
};
```

Member Functions:

•**SgDeleteExp**(PTR_LLND ll)

•**SgDeleteExp**(SgExpression &size, SgExpression &expr)

•**SgDeleteExp**(SgExpression &expr)

4.18 SgNewExp

Represents **C++** new.

Variant: NEW_OP.

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgNewExp: public SgExpression {
public:
                 SgNewExp(PTR_LLND ll);
                 SgNewExp(SgType &t);
                 SgNewExp(SgType &t, SgExpression &e);
#if 0
  SgExpression & numberOfArgs();
  SgExpression & argument(int i);
#endif
};
```

Member Functions:

•**SgNewExp**(PTR_LLND ll)

•**SgNewExp**(SgType &t)

•**SgNewExp**(SgType &t, SgExpression &e)

•SgExpression & **numberOfArgs**()

•SgExpression & **argument**(int i)

4.19 SgThisExp

Represents C++ this.

Variant: THIS_NODE

For base class, see Section 4.1 [SgExpression], page 70.

```
class  SgThisExp: public SgExpression {
public:
   SgThisExp(PTR_LLND ll);
   SgThisExp(SgType &t);
};
```

Member Functions:

•**SgThisExp**(PTR_LLND ll)

•**SgThisExp**(SgType &t)

4.20 SgKeywordValExp

Represents Fortran keyword values in I/O statements etc..

Variant is KEYWORD_VAL.

For base class, see Section 4.1 [SgExpression], page 70.

```
class  SgKeywordValExp: public SgExpression {
public:
         SgKeywordValExp(char *name);
  char * value();
};
```

Member Functions:

•**SgKeywordValExp**(char *name)

•char * **value**()

4.21 SgRefExp

Represents Fortran const references, type references, and interface references.

Variants: CONST_REF, TYPE_REF, INTERFACE_REF

For base class, see Section 4.1 [SgExpression], page 70.

```
class  SgRefExp: public SgExpression {
public:
   SgRefExp(PTR_LLND ll);
   SgRefExp(int variant, SgSymbol &s);
};
```

Member Functions:

•**SgRefExp**(PTR_LLND ll)

•**SgRefExp**(int variant, SgSymbol &s)

4.22 SgVecConstExp

Represents Fortran vector constants [expr1,expr2,expr3].

Variant: VECTOR_CONST

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgVecConstExp: public SgExpression {
```

```
public:
                 SgVecConstExp(PTR_LLND ll);
                 SgVecConstExp(SgExpression &expr_list);
                 SgVecConstExp(int n, SgExpression *components);

  SgExpression * arg(int i); // the i-th term
  int            numberOfArgs();
  void           setArg(int i, SgExpression &e);
};
```

Member Functions:

•**SgVecConstExp**(PTR_LLND ll)

•**SgVecConstExp**(SgExpression &expr_list)

•**SgVecConstExp**(int n, SgExpression *components)

•SgExpression * **arg**(int i)

•int **numberOfArgs**()

•void **setArg**(int i, SgExpression &e)

4.23 SgObjectListExp

Represents Fortran equivalence, namelist, and common.

Variants: EQUI_LIST, NAMELIST_LIST, COMM_LIST

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgObjectListExp: public SgExpression {
public:
                 SgObjectListExp(PTR_LLND ll);
                 SgObjectListExp(int variant, SgSymbol &object,
                                  SgExpression &list);
                 SgObjectListExp(int variant, SgExpression &objectRef,
                                  SgExpression &list);
  SgSymbol *     object();
  SgExpression * objectRef();
  int            listLength();
  SgSymbol *     symbol(int i);        //i'th symbol
  SgExpression * objectRef(int i);  //i'th symbol
```

```
};
```

Member Functions:

•**SgObjectListExp**(PTR_LLND ll)

•**SgObjectListExp**(int variant, SgSymbol &object, SgExpression &list)

•**SgObjectListExp**(int variant,SgExpression &objectRef, SgExpression &list)

•SgSymbol * **object**()

•SgExpression * **objectRef**()

•int **listLength**()

•SgSymbol * **symbol**(int i)

•SgExpression * **objectRef**(int i)

4.24 SgSpecPairExp

Represents Fortran default control arguments to i/o statements.

Variant: SPEC_PAIR

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgSpecPairExp: public SgExpression {
public:
                 SgSpecPairExp(PTR_LLND ll);
                 SgSpecPairExp(SgExpression &arg, SgExpression &value);
                 SgSpecPairExp(SgExpression &arg);
                 SgSpecPairExp(char *arg, char *value);
  SgExpression * arg();
  SgExpression * value();
};
```

Member Functions:

•**SgSpecPairExp**(PTR_LLND ll)

•**SgSpecPairExp**(SgExpression &arg, SgExpression &value)

•**SgSpecPairExp**(SgExpression &arg)

•**SgSpecPairExp**(char *arg, char *value)

•SgExpression * **arg**()

•SgExpression * **value**()

4.25 SgIOAccessExp

Represents Fortran index variable bound instantiations and do loop range representation.

Variant: IOACCESS

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgIOAccessExp: public SgExpression {
public:
   SgIOAccessExp(PTR_LLND ll);
   SgIOAccessExp(SgSymbol &s, SgExpression lbound, SgExpression ubound,
                 SgExpression step);
   SgIOAccessExp(SgSymbol &s, SgExpression lbound, SgExpression ubound);
};
```

Member Functions:

•**SgIOAccessExp**(PTR_LLND ll)

•**SgIOAccessExp**(SgSymbol &s, SgExpression lbound, SgExpression ubound, SgExpression step)

•**SgIOAccessExp**(SgSymbol &s, SgExpression lbound, SgExpression ubound)

4.26 SgImplicitTypeExp

Represents Fortran index variable bound instantiations.

Variant: IMPL_TYPE

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgImplicitTypeExp: public SgExpression {
public:
                SgImplicitTypeExp(PTR_LLND ll);
                SgImplicitTypeExp(SgType &type, SgExpression &rangeList);
  SgType *      type();
  SgExpression * rangeList();
  char *        alphabeticRange();
};
```

Member Functions:

•**SgImplicitTypeExp**(PTR_LLND ll)

•**SgImplicitTypeExp**(SgType &type, SgExpression &rangeList)

•SgType * **type**()

•SgExpression * **rangeList**()

•char * **alphabeticRange**()

4.27 SgTypeExp

Represents Fortran type expressions.

Variant: TYPE_OP

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgTypeExp: public SgExpression {
public:
           SgTypeExp(PTR_LLND ll);
           SgTypeExp(SgType &type);
  SgType * type();
```

```
};
```

Member Functions:

•**SgTypeExp**(PTR_LLND ll)

•**SgTypeExp**(SgType &type)

•SgType * **type**()

4.28 SgSeqExp

Represents Fortran seq expressions.

Variant: SEQ

(? What is this?)

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgSeqExp: public SgExpression {
public:
                 SgSeqExp(PTR_LLND ll);
                 SgSeqExp(SgExpression &exp1, SgExpression &exp2);
  SgExpression * front();
  SgExpression * rear();
};
```

Member Functions:

•**SgSeqExp**(PTR_LLND ll)

•**SgSeqExp**(SgExpression &exp1, SgExpression &exp2)

•SgExpression * **front**()

•SgExpression * **rear**()

4.29 SgStringLengthExp

Represents Fortran string length expressions.

Variant: LEN_OP

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgStringLengthExp: public SgExpression {
public:
                 SgStringLengthExp(PTR_LLND ll);
                 SgStringLengthExp(SgExpression &length);
  SgExpression * length();
};
```

Member Functions:

•**SgStringLengthExp**(PTR_LLND ll)

•**SgStringLengthExp**(SgExpression &length)

•SgExpression * **length**()

4.30 SgDefaultExp

Represents Fortran default.

Variant: DEFAULT

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgDefaultExp: public SgExpression {
public:
   SgDefaultExp(PTR_LLND ll);
   SgDefaultExp();
};
```

Member Functions:

•**SgDefaultExp**(PTR_LLND ll)

•**SgDefaultExp**()

4.31 SgLabelRefExp

Represents Fortran label references.

Variant: LABEL_REF

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgLabelRefExp: public SgExpression {
public:
            SgLabelRefExp(PTR_LLND ll);
            SgLabelRefExp(SgLabel &label);
  SgLabel * label();
};
```

Member Functions:

•**SgLabelRefExp**(PTR_LLND ll)

•**SgLabelRefExp**(SgLabel &label)

•SgLabel * **label**()

4.32 SgConstExp

Represents Fortran 90 array constructors.

Variant: CONSTRUCTOR_REF

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgConstExp:  public SgExpression {
public:
                 SgConstExp(PTR_LLND ll);
                 SgConstExp(SgExpression &values);
  int            numberOfArgs();
  SgExpression * arg(int i);
```

```
};
```

Member Functions:

•**SgConstExp**(PTR_LLND ll)

•**SgConstExp**(SgExpression &values)

•int **numberOfArgs**()

•SgExpression * **arg**(int i)

4.33 SgStructConstExp

Represents Fortran 90 structure constructors.

Variant: STRUCTURE_CONSTRUCTOR

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgStructConstExp:  public SgExpression {
public:
                SgStructConstExp(PTR_LLND ll);
  // further checks on values need to be done.
                SgStructConstExp(SgSymbol &structName, SgExpression &values);
                SgStructConstExp(SgExpression  &typeRef, SgExpression &values);
  int           numberOfArgs();
  SgExpression * arg(int i);
};
```

Member Functions:

•**SgStructConstExp**(PTR_LLND ll)

•**SgStructConstExp**(SgSymbol &structName, SgExpression &values)

•**SgStructConstExp**(SgExpression &typeRef, SgExpression &values)

•int **numberOfArgs**()

•SgExpression * **arg**(int i)

4.34 SgAttributeExp

Represents Fortran 90 attributes.

Variants: PARAMETER_OP, PUBLIC_OP, PRIVATE_OP, ALLOCATABLE_OP, DIMENSION_OP, EXTERNAL_OP, IN_OP, OUT_OP, INOUT_OP, INTRINSIC_OP, POINTER_OP, OPTIONAL_OP, SAVE_OP, TARGET_OP

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgAttributeExp: public SgExpression {
public:
   SgAttributeExp(PTR_LLND ll);
   SgAttributeExp(int variant);
};
```

Member Functions:

•**SgAttributeExp**(PTR_LLND ll)

•**SgAttributeExp**(int variant)

4.35 SgKeywordArgExp

Represents Fortran 90 keyword arguments.

Variant: KEYWORD_ARG

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgKeywordArgExp: public SgExpression {
public:
                 SgKeywordArgExp(PTR_LLND ll);
                 SgKeywordArgExp(char *argName, SgExpression &exp);
  SgSymbol *     arg();
  SgExpression * value();
};
```

Member Functions:

•**SgKeywordArgExp**(PTR_LLND ll)

•**SgKeywordArgExp**(char *argName, SgExpression &exp)

•SgSymbol * **arg**()

•SgExpression * **value**()

4.36 SgUseOnlyExp

Represents Fortran 90 ONLY attribute of USE statements.

Variant: ONLY_NODE

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgUseOnlyExp: public SgExpression {
public:
                 SgUseOnlyExp(PTR_LLND ll);
                 SgUseOnlyExp(SgExpression &onlyList);
  SgExpression * onlyList();
};
```

Member Functions:

•**SgUseOnlyExp**(PTR_LLND ll)

•**SgUseOnlyExp**(SgExpression &onlyList)

•SgExpression * **onlyList**()

4.37 SgUseRenameExp

Represents Fortran 90 USE statement renamings.

Variant: RENAME_NODE

For base class, see Section 4.1 [SgExpression], page 70.

```
class SgUseRenameExp: public SgExpression {
public:
                 SgUseRenameExp(PTR_LLND ll);
                 SgUseRenameExp(SgSymbol &newName, SgSymbol &oldName);
  SgSymbol *     newName();
  SgSymbol *     oldName();
  SgExpression * newNameExp();
  SgExpression * oldNameExp();
};
```

Member Functions:

•**SgUseRenameExp**(PTR_LLND ll)

•**SgUseRenameExp**(SgSymbol &newName, SgSymbol &oldName)

•SgSymbol * **newName**()

•SgSymbol * **oldName**()

•SgExpression * **newNameExp**()

•SgExpression * **oldNameExp**()

5 Symbols

5.1 SgSymbol

SgSymbol is the base class for the classes representing symbols. As with the other classes there is a 'variant()' which identifies the subclass and an integer identifier 'id()' that is unique for each symbol. The symbols in each file are organized as a list which is accessible from the file object as 'SgFile::firstSymbol()' and the following symbol to the current symbol is located with 'next()'. Every symbol has a character string 'identifier()', a 'type()', which is described in detail below, and a statement called 'scope()' that is the statement in which declaration is scoped. (It is the control parent of the declaration statement). The statement where the variable is declared is given by 'declaredInStmt()'.

In addition, there are functions that can be used to generate a copy of the symbol. There are three forms of the copy. The basic copy make a simple copy of the symbol table entry. The level one copy also generates new type information (?correct?). The level two copy also copies the declaration body.

```
class SgSymbol {
```

```
public:
  // basic class contains
  PTR_SYMB thesymb;
                SgSymbol(int variant, char *identifier, SgType &t,
                           SgStatement &scope);
                SgSymbol(int variant, char *identifier, SgStatement &scope);
                SgSymbol(int variant, char *name);
                SgSymbol(int variant);
                SgSymbol(PTR_SYMB symb);
  int           variant();
  int           id();             // unique identifier
  char *        identifier();   // the text name for the symbol.
  SgType *      type();       // the type of the symbol
  void          setType(SgType &t);       // the type of the symbol
  SgStatement * scope(); // the SgControlStatement where defined.
  SgSymbol *    next();     // next symbol reference.
  SgStatement * declaredInStmt();  // the declaration statement
  SgSymbol &    copy();
  SgSymbol &    copyLevel1(); // copy also parameters
  SgSymbol &    copyLevel2(); // copy parameters, body also
  int           attributes();    // the Fortran 90 attributes
  void          setAttribute(int attribute);
  void          removeAttribute(int attribute);
  void          declareTheSymbol(SgStatement &st);
  SgStatement * body(); // the body of the symbol if has one (like, function call, class,...)
};
```

SgSymbol is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135
- see [Instrument - Fortran Program Transformations], page 140
- see [Instrument - InitFunctionTable], page 141
- see [Instrument - InsertFCallNode], page 142
- see [Instrument - pCxx Program Transformations], page 147
- see [Instrument - isReferenceToMethodOfElement], page 147
- see [Instrument - isReferenceToClassOfElement], page 148
- see [Instrument - isReferenceToCollection], page 150
- see [Instrument - ListCollectionInvocations], page 153
- see [Instrument - InsertCCallNode], page 155
- see [Instrument - CTimingInstrumentSub], page 156
- see ⟨undefined⟩ [Expand Syntax], page ⟨undefined⟩
- see [Expand Syntax - isReferenceToElementField], page 160
- see [Expand Syntax - SearchInExpForCollectionArrayRef], page 162

- see [Expand Syntax - ReplaceWithGetElemPart], page 163
- see [Expand Syntax - Init], page 166
- see [Dump Info - classifyStatements], page 170
- see [Dump Info - doRoutineHeader], page 176
- see [Dump Info - doSymb], page 179
- see [Dump Info - doSymbAttribs], page 179
- see [Dump Info - classifySymbols], page 185

Member Functions:

•**SgSymbol**(int variant, char *identifier, SgType &t, SgStatement &scope)

•**SgSymbol**(int variant, char *identifier, SgStatement &scope)

•**SgSymbol**(int variant, char *name)

•**SgSymbol**(int variant)

•**SgSymbol**(PTR_SYMB symb)

•int **variant**()

•int **id**()

•char * **identifier**()

•SgType * **type**()

•void **setType**(SgType &t)

•SgStatement * **scope**()

•SgSymbol * **next**()

•SgStatement * **declaredInStmt**()

•SgSymbol & **copy**()

•SgSymbol & **copyLevel1**()

•SgSymbol & **copyLevel2**()

•int **attributes**()

•void **setAttribute**(int attribute)

•void **removeAttribute**(int attribute)

•void **declareTheSymbol**(SgStatement &st)

•SgStatement * **body**()

If the symbol refers to a function, struct, subroutine or class, this returns a pointer to the SgStatement object that is the declaration of the symbol.

SgSymbol Usage

As an example of how to traverse the symbol table, consider the problem of looking for a symbol table entry for a given member function in a given class. There are several ways to do this. The scheme shown below will search the table for the class by name. Then it searches the field list for the member function and returns the pointer to the symbol object if it is found.

```
SgSymbol *memberFunction(char *className, char *functionName){
  SgSymbol *s, *fld;
  SgClassSymb *cl;
  SgMemberFuncSymb *f;
  int i;

  for (s=file.firstSymbol(); s ; s = s->next())
    if (!strcmp(s->identifier(), className))
       break;

  if (s == NULL)
     return NULL;

  if (cl = isSgClassSymb(s)){
    i = 1;
    while(fld = cl->field(i++)){
      if ((f = isSgMemberFuncSymb(fld)) &&
          !strcmp(f->identifier(), functionName))
         return f;
    }
  }
  return NULL;
}
```

As another example, consider the task of generating temporary variables that match the type and initial constructor parameters of a given variable symbol. More specifically, consider the declarations

```
int *x[100], y[34];
myClass object(parm1, parm2);
```

Each of these statements is of class SgVarDeclStmt which has a type and an expression list. In the first case the expression list has two items (called *values* in SgExprListExp): '`x[100], y[34]`', and the second declaration has one item '`object(parm1, parm2)`'. The following function creates a new temporary symbol with name _Ti where i is an integer counter and given a symbol table entry, finds the declaration of the symbol, and creates an identical declaration for the temporary.

```
char name_counter = '0';
SgSymbol * DeclareTemporaryLike(SgSymbol * x){
   SgStatement  *old_decl_stmt;
   SgSymbol *tmp;
   SgVarDeclStmt *decl_stmt;
   SgExpression *exp;
   char *name;
   SgExprListExp *elist;


   // now find the declaration statement for x and get
   // the correct initializing expression.
   old_decl_stmt = x.declaredInStmt()
      // get the expression list of variables delcared
   exp = old_decl_stmt->expr(0);

   // look for the reference to our symbol x.
   while(elist = isSgExprListExp(exp)){
if((elist->value())->isSymbolInExpression(*x)
            break;
        exp = elist->next();
}
   if(exp == NULL) return NULL;

   // now create a new expression list and
   // and a new declaration statement
   elist = new SgExprListExp(*(exp->copy());
   decl_stmt = new SgVarDeclStmt(*elist,old_decl_stmt->type());

   // create a new variable of the same class
   // generate the name as "_Ti" where i is counter.
   name = new char(10);
   sprintf(name, "_T%c", name_counter++);
   tmp = new SgVariableSymb(name, (x->type())->copy(), *scope);

   // replace x with the new variable and insert
   // the new decl statement after the old one.
   decl_stmt->replaceSymbBySymb(*x, *tmp);
   old_decl_stmt->insertStmtAfter(*decl_stmt);
   return tmp;
}
```

5.2 SgVariableSymb

Represents variable symbols, for all languages.

Variant: VARIABLE_NAME

For usage,
see [SgExpression Usage], page 74;
see [SgSubscriptExp Usage], page 88.

For base class, see Section 5.1 [SgSymbol], page 105.

```
class SgVariableSymb: public SgSymbol {
public:
                 SgVariableSymb(char *identifier, SgType &t, SgStatement &scope);
                 SgVariableSymb(char *identifier, SgType &t);
                 SgVariableSymb(char *identifier, SgStatement &scope);
                 SgVariableSymb(char *identifier);
#if 0
  int            isAttributeSet(int attribute);
  void           setAttribute(int attribute);

  int            numberOfUses();            // number of uses.
  SgStatement  * useStmt(int i);  // statement where i-th use occurs
  SgExpression * useExpr(int i);  // expression where i-th use occurs
  int            numberOfDefs();
#endif
};
```

SgVariableSymb is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135
- see ⟨undefined⟩ [Expand Syntax], page ⟨undefined⟩
- see [Expand Syntax - ReplaceWithGetElemPart], page 163
- see [Expand Syntax - Init], page 166
- see [Dump Info - classifySymbols], page 185

Member Functions:

•**SgVariableSymb**(char *identifier, SgType &t, SgStatement &scope)

•**SgVariableSymb**(char *identifier, SgType &t)

•**SgVariableSymb**(char *identifier, SgStatement &scope)

•**SgVariableSymb**(char *identifier)

•int **isAttributeSet**(int attribute)

•void **setAttribute**(int attribute)

•int **numberOfUses**()

Returns the number of uses.

•SgStatement * **useStmt**(int i)

Returns the ith use of the variable.

•SgExpression * **useExpr**(int i)

Returns the ith expression in which the variable is used.

•int **numberOfDefs**()

The number of definitions of the variable.

5.3 SgConstantSymb

Represents symbols for constants, for all languages.

Variant: CONST_NAME

For base class, see Section 5.1 [SgSymbol], page 105.

```
class SgConstantSymb: public SgSymbol {
public:
                 SgConstantSymb(char *identifier, SgStatement &scope,
                                SgExpression &value);
  SgExpression * constantValue();
};
```

SgConstantSymb is used in the following places in the example programs:

- see [Dump Info - classifySymbols], page 185

Member Functions:

•**SgConstantSymb**(char *identifier, SgStatement &scope, SgExpression &value)

•SgExpression * **constantValue**()

5.4 SgFunctionSymb

Represents symbols for subroutines, symbols for functions, and symbols for main programs, for all languages.

Variants: PROGRAM_NAME, PROCEDURE_NAME, MEMBER_FUNC and FUNCTION_NAME.

For base class, see Section 5.1 [SgSymbol], page 105.

```
class SgFunctionSymb: public SgSymbol {
  // a subroutine, function or main program
  // variant == PROGRAM_NAME, PROCEDURE_NAME, or FUNCTION_NAME
public:
             SgFunctionSymb(int variant);
             SgFunctionSymb(int variant, char *identifier, SgType &t,
                                 SgStatement &scope);
  void       addParameter(int n, SgSymbol &parameters);
  void       insertParameter(int position, SgSymbol &symb);
  int        numberOfParameters();
  SgSymbol * parameter(int i);
  SgSymbol * result();
  void       setResult(SgSymbol &symbol);
#if 0
  int        isRecursive();
  int        setRecursive();
#endif
};
```

SgFunctionSymb is used in the following places in the example programs:

- see [Instrument - InitSymbols], page 140
- see [Instrument - CInitSymbols], page 155
- see ⟨undefined⟩ [Expand Syntax], page ⟨undefined⟩
- see [Expand Syntax - ReplaceWithGetElemPart], page 163
- see [Expand Syntax - Init], page 166
- see [Dump Info - classifySymbols], page 185

Member Functions:

•**SgFunctionSymb**(int variant)

•**SgFunctionSymb**(int variant, char *identifier, SgType &t, SgStatement &scope)

•void **addParameter**(int n, SgSymbol &parameters)

•void **insertParameter**(int position, SgSymbol &symb)

•int **numberOfParameters**()

Returns number of parameters in the formal parameter list (of a function).

•SgSymbol * **parameter**(int i)

Returns the ith parameter in the formal parameter list (of a function).

•SgSymbol * **result**()

For Fortran, returns a pointer to the result symbol.

•void **setResult**(SgSymbol &symbol)

For Fortran.

•int **isRecursive**()

For Fortran, returns TRUE if the function is recursive.

•int **setRecursive**()

5.5 SgLabelSymb

Represents C label symbols.

Variant: LABEL_NAME

For base class, see Section 5.1 [SgSymbol], page 105.

```
class SgLabelSymb: public SgSymbol {
public:
   SgLabelSymb(char *name);
};
```

Member Functions:

•**SgLabelSymb**(char *name)

5.6 SgTypeSymb

Represents C typedef'ed symbols.

Variant: TYPE_NAME

For base class, see Section 5.1 [SgSymbol], page 105.

```
class SgTypeSymb: public SgSymbol {
#if 0
public:
           SgTypeSymb(char *name, SgType &baseType);
  SgType & baseType();
};
```

Member Functions:

•**SgTypeSymb**(char *name, SgType &baseType)

•SgType & **baseType**()

The type the symbol represents.

5.7 SgClassSymb

Represents C symbols for classes, symbols for unions, symbols for structs, and symbols for collections.

Variants: CLASS_NAME, UNION_NAME, STRUCT_NAME and COLLECTION_NAME.

For usage, see [SgSymbol Usage], page 108.

For base class, see Section 5.1 [SgSymbol], page 105.

```
class SgClassSymb: public SgSymbol {
public:
            SgClassSymb(int variant, char *name, SgStatement &scope);
  int       numberOfFields();
  SgSymbol * field(int i);
```

```
};
```

Member Functions:

•**SgClassSymb**(int variant, char *name, SgStatement &scope)

•int **numberOfFields**()

Returns the number of members.

•SgSymbol * **field**(int i)

Returns the ith member.

5.8 SgFieldSymb

Represents C symbols for fields in enums, symbols for fields in structs, and symbols for fields in classes.

Variants: ENUM_NAME and FIELD_NAME.

For base class, see Section 5.1 [SgSymbol], page 105.

```
class SgFieldSymb: public SgSymbol {
public:
  // no check is made to see if the field "identifier"
  //   already exists in the structure.
             SgFieldSymb(char *identifier, SgType &t, SgSymbol &structureName);
  int        offset();        // position in the structure
  SgSymbol * structureName();  // parent structure
  SgSymbol * nextField();
  int        isMethodOfElement();
#if 0
  int        isPrivate();
  int        isSequence();
  void       setPrivate();
  void       setSequence();
#endif
};
```

SgFieldSymb is used in the following places in the example programs:

- see [Instrument - isReferenceToMethodOfElement], page 147
- see [Instrument - isReferenceToClassOfElement], page 148
- see [Expand Syntax - isReferenceToElementField], page 160

Member Functions:

•**SgFieldSymb**(char *identifier, SgType &t, SgSymbol &structureName)

•int **offset**()

•SgSymbol * **structureName**()

•SgSymbol * **nextField**()

•int **isMethodOfElement**()

•int **isPrivate**()

•int **isSequence**()

•void **setPrivate**()

•void **setSequence**()

5.9 SgMemberFuncSymb

Represents C symbols for members of structs and classes.

Variants: MEMBER_FUNC, MEMB_PRIVATE, MEMB_PUBLIC, MEMB_PROTECTED and MEMB_METHOELEM.

For usage, see [SgSymbol Usage], page 108.

For base class, see Section 5.4 [SgFunctionSymb], page 112.

```
class SgMemberFuncSymb: public SgFunctionSymb {
public:
             SgMemberFuncSymb(char *identifier, SgType &t, SgStatement &cla,
                              int status);
#if 0
  int        status();
  int        isVirtual();       // 1 if virtual.
```

```
#endif
  int        isMethodOfElement();
  SgSymbol * className();
  void       setClassName(SgSymbol &symb);
};
```

SgMemberFuncSymb is used in the following places in the example programs:

- see [Instrument - isReferenceToMethodOfElement], page 147
- see [Instrument - isReferenceToClassOfElement], page 148
- see [Instrument - inMethodOfTheElement], page 149
- see [Instrument - whichFunctionAmI], page 150

Member Functions:

•**SgMemberFuncSymb**(char *identifier, SgType &t, SgStatement &cla, int status)

•int **status**()

Returns the protection status of the symbol.

•int **isVirtual**()

•int **isMethodOfElement**()

•SgSymbol * **className**()

Returns the symbol table entry for the enclosing class or struct.

•void **setClassName**(SgSymbol &symb)

5.10 SgLabelVarSymb

Represents Fortran symbols for label variables for assigned goto statements.

Variant: LABEL_NAME

For base class, see Section 5.1 [SgSymbol], page 105.

```
class SgLabelVarSymb: public SgSymbol {
public:
   SgLabelVarSymb(char *name, SgStatement &scope);
```

```
};
```

SgLabelVarSymb is used in the following places in the example programs:

- see [Dump Info - classifySymbols], page 185

Member Functions:

•**SgLabelVarSymb**(char *name, SgStatement &scope)

5.11 SgExternalSymb

Represents Fortran symbols for external functions.

Variant: ROUTINE_NAME

For base class, see Section 5.1 [SgSymbol], page 105.

```
class SgExternalSymb: public SgSymbol {
public:
   SgExternalSymb(char *name, SgStatement &scope);
};
```

SgExternalSymb is used in the following places in the example programs:

- see [Dump Info - classifySymbols], page 185

Member Functions:

•**SgExternalSymb**(char *name, SgStatement &scope)

5.12 SgConstructSymb

Represents Fortran symbols for construct names.

Variant: CONSTRUCT_NAME

For base class, see Section 5.1 [SgSymbol], page 105.

```
class SgConstructSymb: public SgSymbol {
  // for fortran statement with construct names
public:
   SgConstructSymb(char *name, SgStatement &scope);
};
```

SgConstructSymb is used in the following places in the example programs:

- see [Dump Info - classifySymbols], page 185

Member Functions:

•**SgConstructSymb**(char *name, SgStatement &scope)

5.13 SgModuleSymb

Represents Fortran symbols for module statements.

For base class, see Section 5.1 [SgSymbol], page 105.

```
class SgModuleSymb: public SgSymbol {
// A lot of work needs to be done on this class.

  // for fortran module statement
  // variant == MODULE_NAME
public:
   SgModuleSymb(char *name);
};
```

SgModuleSymb is used in the following places in the example programs:

- see [Dump Info - classifySymbols], page 185

Member Functions:

•**SgModuleSymb**(char *name)

5.14 SgInterfaceSymb

Represents Fortran 90 symbols for module interface statements.

Variant: (? MODULE_NAME or INTERFACE_NAME?)

For base class, see Section 5.1 [SgSymbol], page 105.

```
class SgInterfaceSymb: public SgSymbol {
public:
   SgInterfaceSymb(char *name, SgStatement &scope);
};
```

SgInterfaceSymb is used in the following places in the example programs:

- see [Dump Info - classifySymbols], page 185

Member Functions:

•**SgInterfaceSymb**(char *name, SgStatement &scope)

6 Types

Sage's type classes correspond to the type structures of the languages Sage can deal with. Types characterize the properties of symbols. Because C++ and Fortran 90 have fairly rich type structures, there are many Sage type classes.

6.1 SgType

SgType is the base class for all classes that represent types. As with symbols, types in a file form a list that can be accessed with the function '`SgFile::firstType()`'. The '`next()`' function returns the type node following '`this`'. The function '`copy()`' returns a copy of the type node. Many types are defined in terms of other types. For example, an array type has a base type that may be a pointer type which has a base type that may be an integer or a class, etc. The '`baseType()`' function always access the next type in the chain. For the language pC++ there is a dummy type called '`ElementType`' which is used as a "place holder" or parameter for a type that is supplied later. Another function '`equivalentToType()`' can be used to test to types to see if they are equivalent.

```
class SgType {
public:
  PTR_TYPE thetype;
            SgType(int variant);
            SgType(PTR_TYPE type);
  int       variant();
  int       id();
  SgSymbol * symbol();
  SgType &   copy();
  SgType *   next();
  int       isTheElementType();
  int       equivalentToType(SgType &type);
  SgType *   internalBaseType();
  int       hasBaseType();
  SgType *   baseType();
};
```

SgType is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135
- see [Instrument - Fortran Program Transformations], page 140

- see [Instrument - isReferenceToClassOfElement], page 148
- see [Instrument - isReferenceToCollection], page 150
- see [Instrument - ListCollectionInstances], page 153
- see ⟨undefined⟩ [Expand Syntax], page ⟨undefined⟩
- see [Expand Syntax - isArrayRefOfCollection], page 161
- see [Expand Syntax - ReplaceWithGetElemPart], page 163
- see [Expand Syntax - MatchingDummyVar], page 165
- see [Expand Syntax - InDummyVarTable], page 166
- see [Expand Syntax - Init], page 166
- see [Dump Info - classifyStatements], page 170
- see [Dump Info - doFullType], page 180

Member Functions:

•**SgType**(int variant)

•**SgType**(PTR_TYPE type)

•int **variant**()

•int **id**()

•SgSymbol * **symbol**()

•SgType & **copy**()

•SgType * **next**()

•int **isTheElementType**()

•int **equivalentToType**(SgType &type)

•SgType * **internalBaseType**()

•int **hasBaseType**()

•SgType * **baseType**()

SgType Usage

To illustrate the use of the type table, consider the simple problem of identifying when a variable in a variable reference expression is of a given user defined class. More specifically, given

```
class myClass;
myClass x, y;
y = x + 2;
```

then if e is the expression representing the variable reference to x, we would like a function isVarRefOfClass(e, "myClass") that will return 1 because the class type matches. To do this we first check to see if e is indeed a variable reference. Then, extracting the type of the symbol we should find a derived type. From the derived type we can find the name of the class.

```
int  isVarRefOfClass(SgExpression *e, char *classname){
   SgSymbol *s;
   SgDerivedType *d;
   SgClassSymb *cl;
   SgVarRefExp *exp;
   if((exp = isSgVarRefExp(e)) == NULL) return 0;
   s = exp->symbol();
   if((d = isSgDerivedType(s->type())) == NULL) return 0;
   if(cl = isSgClassSymb(d->typeName()))
         if(!strcmp(cl->identifier(), classname))
                return 1;
   return 0;
}
```

A function of the type

```
SgSUBCLASS * isSgSUBCLASS( SgBASECLASS *)
```

is provided for each subclass of SgStatement, SgExpression and SgType. We feel that this mechanism is an effective way to use the strong typing of C++ to reduce errors.

6.2 SgArrayType

Represents types for arrays, for all languages.

The parser gives each array variable its own Array type descriptor object.

Variant: T_ARRAY

For base class, see Section 6.1 [SgType], page 121.

```
class SgArrayType: public SgType {
public:
                 SgArrayType(SgType &base_type);
  int            dimension();
  SgExpression * sizeInDim(int i);
  SgType *       baseType();
  void           setBaseType(SgType &bt);
  void           addRange(SgExpression &e);
```

```
};
```

SgArrayType is used in the following places in the example programs:

- see [Restructure - addStuffToProgram], page 135
- see [Dump Info - doFullType], page 180

Member Functions:

•**SgArrayType**(SgType &base_type)

Each array must have a base type which may be any type object.

•int **dimension**()

•SgExpression * **sizeInDim**(int i)

Returns the expression that describes the the shape of the array. If this expression is a SgSubscriptExpr then it is a Fortran array which has a non-unit starting index and a upper bound index.

•SgType * **baseType**()

•void **setBaseType**(SgType &bt)

Modifies the base type.

•void **addRange**(SgExpression &e)

Allows a new dimension to be added.

6.3 SgClassType

Represents C types for structs, types for unions, and types for enums. Represents C++ types for base classes. Represents pC++ types for collections. Represents Fortran types for records.

This type is very simple. It contains the standard type information from SgType, a pointer to the class declaration statement, and a pointer to the symbol that is the first field in the structure. Note: derived classes have a different type.

Variants: T_STRUCT, T_ENUM, T_CLASS, T_ENUM, and T_COLLECTION.

For base class, see Section 6.1 [SgType], page 121.

```
class SgClassType: public SgType {
#if 0
  // a C struct or Fortran Record, a C++ class, a C Union and a C Enum
  // and a pC++ collection.  note: derived classes are another type.
  // this type is very simple.  it only contains the standard type
  // info from SgType and a pointer to the class declaration stmt
  // and a pointer to the symbol that is the first field in the struct.
  // variant == T_STRUCT, T_ENUM, T_CLASS, T_ENUM, T_COLLECTION
public:
  // why is struct_decl needed. No appropriate field found.
  // assumes that first_field has been declared as
  // FIELD_NAME and the remaining fields have been stringed to it.
              SgClassType(int variant, char *name, SgStatement &struct_decl,
                             int num_fields, SgSymbol &first_field);
  SgStatement & structureDecl();
  SgSymbol *    firstFieldSymb();
  SgSymbol *    fieldSymb(int i);
  int           numberOfFields();
};
```

Member Functions:

•**SgClassType**(int variant, char *name, SgStatement &struct_decl, int num_fields, SgSymbol &first_field)

•SgStatement & **structureDecl**()

Returns a reference to the source code that defines this structure.

•SgSymbol * **firstFieldSymb**()

Returns a pointer to the symbol entry for the first field. Symbols for All other fields are chained to that symbol.

•SgSymbol * **fieldSymb**(int i)

Returns a pointer to the 'i'th field.

•int **numberOfFields**()

6.4 SgFunctionType

Represents types for functions, for all languages.

The important information here is the type of the returned value.

Variant: T_FUNCTION

For base class, see Section 6.1 [SgType], page 121.

```
class SgFunctionType: public SgType {
#if 0
public:
             SgFunctionType(SgType &return_val_type);
  SgType & returnedValue();
  void     changeReturnedValue(SgType &rv);
};
```

Member Functions:

•**SgFunctionType**(SgType &return_val_type)

•SgType & **returnedValue**()

•void **changeReturnedValue**(SgType &rv)

Allows modification of the returned value.

6.5 SgDescriptType

Represents C unsigned, signed, long, short, extern, static, and register.

Represents C++ private, virtual, inline, volatile, const, auto, and friend.

Represents Fortran 90 syn, shared, future, and global.

This class represents descriptor objects that modify other type objects. For example, in the C the expression

```
long volatile int x;
```

long and volatile are modifiers and there is a descriptor type whose base type is the real type of x. the modifier is an integer with bits set if the modifier holds.

The modifier bits are: BIT_SYN, BIT_SHARED, BIT_PRIVATE, BIT_FUTURE, BIT_VIRTUAL, BIT_INLINE, BIT_UNSIGNED, BIT_SIGNED, BIT_LONG, BIT_SHORT, BIT_VOLATILE, BIT_CONST, BIT_TYPEDEF, BIT_EXTERN, BIT_AUTO, BIT_STATIC, BIT_REGISTER, and BIT_FRIEND

Variant: T_DESCRIPT

For base class, see Section 6.1 [SgType], page 121.

```
class SgDescriptType: public SgType {
public:
        SgDescriptType(SgType &base_type, int bit_flag);
  int   modifierFlag();
  void  setModifierFlag(int flag);
};
```

SgDescriptType is used in the following places in the example programs:

- see [Dump Info - doFullType], page 180

Member Functions:

•**SgDescriptType**(SgType &base_type, int bit_flag)

•int **modifierFlag**()

•void **setModifierFlag**(int flag)

6.6 SgPointerType

Represents C pointer type.

Variant: T_POINTER

The important information is encoded in the base type, i.e. the type of thing this pointer can point to.

For base class, see Section 6.1 [SgType], page 121.

```
class SgPointerType: public SgType {
  // A pointer type contains only one hany bit of information:
  // the base type.
  // variant == T_POINTER
public:
           SgPointerType(SgType &base_type);
  SgType * baseType();
  int      indirection();
  void     setIndirection(int);
  void     setBaseType(SgType &baseType);
};
```

SgPointerType is used in the following places in the example programs:

- see [Instrument - isReferenceToCollection], page 150
- see [Expand Syntax - ReplaceWithGetElemPart], page 163
- see [Expand Syntax - Init], page 166
- see [Dump Info - doFullType], page 180

Member Functions:

•**SgPointerType**(SgType &base_type)

•SgType * **baseType**()

•int **indirection**()

•void **setIndirection**(int)

•void **setBaseType**(SgType &baseType)
Can modify the base type.

6.7 SgDerivedType

Represents C typedefed types. Represents C++ instance variables.

Variants: T_DERIVED_TYPE, T_DERIVED_CLASS

For base class, see Section 6.1 [SgType], page 121.

```
class SgDerivedType: public SgType {
public:
              SgDerivedType(SgSymbol &type_name);
  SgSymbol *  typeName();
};
```

SgDerivedType is used in the following places in the example programs:

- see [Instrument - isReferenceToClassOfElement], page 148
- see [Dump Info - doFullType], page 180

Member Functions:

•**SgDerivedType**(SgSymbol &type_name)

•SgSymbol * **typeName**()

Returns the object of class SgTypeSymb or SgClassSymb to which this type refers.

6.8 SgReferenceType

Represents C++ reference types.

Variant: T_REFERENCE

For base class, see Section 6.1 [SgType], page 121.

```
class SgReferenceType: public SgType {
  // A reference (&xx in c+=) type contains only one hany bit of information:
  // the base type.
  // variant == T_REFERENCE
public:
           SgReferenceType(SgType &base_type);
  SgType * baseType();
  void     setBaseType(SgType &baseType);
};
```

SgReferenceType is used in the following places in the example programs:

- see [Instrument - isReferenceToCollection], page 150
- see [Dump Info - doFullType], page 180

Member Functions:

•**SgReferenceType**(SgType &base_type)

•SgType * **baseType**()

•void **setBaseType**(SgType &baseType)

6.9 SgDerivedClassType

Represents C++ types of derived classes.

For base class, see Section 6.1 [SgType], page 121.

```
class SgDerivedClassType: public SgType {
  // variant == T_DERIVED_CLASS
public:
             SgDerivedClassType(SgSymbol &type_name);
  SgSymbol * typeName();
};
```

Member Functions:

•**SgDerivedClassType**(SgSymbol &type_name)

•SgSymbol * **typeName**()

6.10 SgDerivedCollectionType

Represents pC++ collection classes.

For example, if we have

```
Collection DistributedArray {body1} ;
class object {body2} ;
DistributedArray<object>  X;
```

then the type of X is SgDerivedCollectionType.

Variant: T_DERIVED_COLLECTION

For base class, see Section 6.1 [SgType], page 121.

```
class SgDerivedCollectionType: public SgType {
  // for example:
  // Collection DistributedArray {body1} ;
  // class object {body2} ;
  // DistributedArray<object>  X;
  // X is of type with variant = T_DERIVED_COLLECTION
```

```
public:
                  SgDerivedCollectionType(SgSymbol &s, SgType &t);
  SgType *        elementClass();
  void            setElementClass(SgType &ty);
  SgSymbol *      collectionName();
  SgStatement *   createCollectionWithElemType();
};
```

SgDerivedCollectionType is used in the following places in the example programs:

- see [Instrument - isReferenceToClassOfElement], page 148
- see [Instrument - isReferenceToCollection], page 150
- see [Instrument - ListCollectionInstances], page 153
- see [Instrument - ListCollectionInvocations], page 153
- see [Instrument - CTimingInstrumentSub], page 156
- see ⟨undefined⟩ [Expand Syntax], page ⟨undefined⟩
- see [Expand Syntax - isArrayRefOfCollection], page 161
- see [Expand Syntax - SearchInExpForCollectionArrayRef], page 162
- see [Expand Syntax - ReplaceWithGetElemPart], page 163
- see [Expand Syntax - Init], page 166
- see [Dump Info - doFullType], page 180

Member Functions:

•**SgDerivedCollectionType**(SgSymbol &s, SgType &t)

t is the collection type.

•SgType * **elementClass**()

Returns the type of the collection element.

•void **setElementClass**(SgType &ty)

Sets the type of the collection element.

•SgSymbol * **collectionName**()

Returns the object of class SgClassSymb that is the name of the collection.

•SgStatement * **createCollectionWithElemType**()

7 Labels

7.1 SgLabel

Represents labels, for all languages.

```
class SgLabel {
public:
  PTR_LABEL thelabel;
       SgLabel(PTR_LABEL lab);
       SgLabel(int i);
  int  id();
  int  getLastLabelVal();
};
```

Member Functions:

•**SgLabel**(PTR_LABEL lab)

•**SgLabel**(int i)

•int **id**()

•int **getLastLabelVal**()

8 Non-Member Functions

SgType *SgTypeInt(); SgType *SgTypeChar(); SgType *SgTypeFloat(); SgType *SgTypeDouble(); SgType *SgTypeVoid(); SgType *SgTypeBool(); SgType *SgTypeDefault();

SgUnaryExp & SgDerefOp(SgExpression &e); SgUnaryExp & SgAddrOp(SgExpression &e); SgUnaryExp & SgUMinusOp(SgExpression &e); SgUnaryExp & SgUPlusOp(SgExpression &e); SgUnaryExp & SgPrePlusPlusOp(SgExpression &e); SgUnaryExp & SgPreMinusMinusOp(SgExpression &e); SgUnaryExp & SgPostPlusPlusOp(SgExpression &e); SgUnaryExp & SgPostMinusMinusOp(SgExpression &e); SgUnaryExp & SgBitCompfOp(SgExpression &e); SgUnaryExp & SgNotOp(SgExpression &e); SgUnaryExp & SgSizeOfOp(SgExpression &e); SgUnaryExp & makeAnUnaryExpression(int code,PTR_LLND ll1);

SgValueExp * isSgValueExp(SgExpression *pt); SgKeywordValExp * isSgKeywordValExp(SgExpression *pt); SgCastExp * isSgCastExp(SgExpression *pt); SgDeleteExp * isSgDeleteExp(SgExpression *pt); SgNewExp * isSgNewExp(SgExpression *pt); SgExprIfExp * isSgExprIfExp(SgExpression *pt); SgFunctionCallExp * isSgFunctionCallExp(SgExpression *pt); SgFuncPntrExp * isSgFuncPntrExp(SgExpression *pt); SgExprListExp * isSgExprListExp(SgExpression *pt); SgRefExp * isSgRefExp (SgExpression *pt); SgVarRefExp * isSgVarRefExp (SgExpression *pt); SgThisExp * isSgThisExp (SgExpression *pt); SgArrayRefExp * isSgArrayRefExp (SgExpression *pt); SgPntrArrRefExp * isSgPntrArrRefExp(SgExpression *pt); SgPointerDerefExp * isSgPointerDerefExp (SgExpression *pt); SgRecordRefExp * isSgRecordRefExp (SgExpression *pt); SgStructConstExp* isSgStructConstExp (SgExpression *pt); SgConstExp* isSgConstExp (SgExpression *pt); SgVecConstExp * isSgVecConstExp (SgExpression *pt); SgInitListExp * isSgInitListExp (SgExpression *pt); SgObjectListExp * isSgObjectListExp (SgExpression *pt); SgAttributeExp * isSgAttributeExp (SgExpression *pt); SgKeywordArgExp * isSgKeywordArgExp (SgExpression *pt); SgSubscriptExp* isSgSubscriptExp (SgExpression *pt); SgUseOnlyExp * isSgUseOnlyExp (SgExpression *pt); SgUseRenameExp * isSgUseRenameExp (SgExpression *pt); SgSpecPairExp * isSgSpecPairExp (SgExpression *pt); SgIOAccessExp * isSgIOAccessExp (SgExpression *pt); SgImplicitTypeExp * isSgImplicitTypeExp (SgExpression *pt); SgTypeExp * isSgTypeExp (SgExpression *pt); SgSeqExp * isSgSeqExp (SgExpression *pt); SgStringLengthExp * isSgStringLengthExp (SgExpression *pt); SgDefaultExp * isSgDefaultExp (SgExpression *pt); SgLabelRefExp * isSgLabelRefExp (SgExpression *pt); SgProgHedrStmt * isSgProgHedrStmt (SgStatement *pt); SgProcHedrStmt * isSgProcHedrStmt (SgStatement *pt); SgFuncHedrStmt * isSgFuncHedrStmt (SgStatement *pt); SgClassStmt * isSgClassStmt (SgStatement *pt); SgStructStmt * isSgStructStmt (SgStatement *pt); SgUnionStmt * isSgUnionStmt (SgStatement *pt); SgEnumStmt * isSgEnumStmt (SgStatement *pt); SgCollectionStmt * isSgCollectionStmt (SgStatement *pt); SgBasicBlockStmt * isSgBasicBlockStmt (SgStatement *pt); SgForStmt * isSgForStmt (SgStatement *pt); SgWhileStmt * isSgWhileStmt (SgStatement *pt); SgDoWhileStmt * isSgDoWhileStmt (SgStatement *pt); SgLogIfStmt * isSgLogIfStmt (SgStatement *pt); SgIfStmt * isSgIfStmt (SgStatement *pt); SgArithIfStmt * isSgArithIfStmt (SgStatement *pt); SgWhereStmt * isSgWhereStmt (SgStatement *pt); SgWhereBlockStmt * isSgWhereBlockStmt (SgStatement *pt); SgSwitchStmt * isSgSwitchStmt (SgStatement *pt); SgCaseOptionStmt * isSgCaseOptionStmt (SgStatement *pt); SgExecutableStatement * isSgExecutableStatement (SgStatement *pt); SgAssignStmt * isSgAssignStmt (SgStatement *pt); SgCExpStmt * isSgCExpStmt (SgStatement *pt); SgPointerAssignStmt * isSgPointerAssignStmt (SgStatement *pt); SgHeapStmt * isSgHeapStmt (SgStatement *pt); SgNullifyStmt * isSgNullifyStmt (SgStatement *pt); SgContinueStmt * isSgContinueStmt (SgStatement *pt); SgControlEndStmt * isSgControlEndStmt (SgStatement *pt); SgBreakStmt * isSgBreakStmt (SgStatement *pt); SgCycleStmt * isSgCycleStmt (SgStatement *pt); SgReturnStmt * isSgReturnStmt (SgStatement *pt); SgExitStmt * isSgExitStmt (SgStatement *pt); SgGotoStmt * isSgGotoStmt (SgStatement *pt); SgLabelListStmt * isSgLabelListStmt (SgStatement *pt); SgAssignedGotoStmt * isSgAssignedGotoStmt (SgStatement *pt); SgComputedGotoStmt * isSgComputedGotoStmt (SgStatement *pt); SgStopOrPauseStmt * isSgStopOrPauseStmt (SgStatement *pt); SgCallStmt* isSgCallStmt (SgStatement *pt); SgIOStmt * isSgIOStmt (SgStatement *pt); SgInputOutputStmt * isSgInputOutputStmt (SgStatement *pt); SgIOControlStmt * isSgIOControlStmt (SgStatement *pt); SgDeclarationStatement * isSgDeclarationStatement (SgStatement *pt); SgVarDeclStmt * isSgVarDeclStmt (SgStatement *pt); SgVarListDeclStmt * isSgVarListDeclStmt (SgStatement *pt); SgStructureDeclStmt * isSgStructureDeclStmt (SgStatement *pt); SgNestedVarListDeclStmt* isSgNestedVarListDeclStmt (SgStatement *pt); SgParameterStmt * isSgParameterStmt (SgStatement *pt); SgImplicitStmt * isSgImplicitStmt (SgStatement *pt); SgVariableSymb * isSgVariableSymb (SgSymbol *pt); SgConstantSymb * isSgConstantSymb (SgSymbol *pt); SgFunctionSymb * isSgFunctionSymb (SgSymbol *pt); SgMemberFuncSymb * isSgMemberFuncSymb (SgSymbol *pt); SgFieldSymb * isSgFieldSymb (SgSymbol *pt); SgClassSymb * isSgClassSymb (SgSymbol *pt); SgLabelSymb * isSgLabelSymb (SgSymbol *pt); SgLabelVarSymb * isSgLabelVarSymb (SgSymbol *pt); SgExternalSymb * isSgExternalSymb (SgSymbol *pt); SgConstructSymb * isSgConstructSymb (SgSymbol *pt); SgInterfaceSymb * isSgInterfaceSymb (SgSymbol *pt); SgModuleSymb * isSgModuleSymb (SgSymbol *pt); SgArrayType * isSgArrayType (SgType *pt); SgPointerType * isSgPointerType (SgType *pt); SgReferenceType * isSgReferenceType (SgType *pt); SgDerivedType * isSgDerivedType (SgType *pt); SgDerivedClassType * isSgDerivedClassType (SgType *pt); SgDescriptType * isSgDescriptType (SgType *pt); SgDerivedCollectionType* isSgDerivedCollectionType (SgType *pt);

9 Example Programs

9.1 Restructure

```
/*********************************************************************/
/*                    pC++/Sage++  Copyright (C) 1993                */
/*  Indiana University  University of Oregon  University of Rennes   */
/*********************************************************************/


#include <stdio.h>
#include <malloc.h>
#include "sage++user.h"

// this is an example of how to use the sage library
// to restructure fortran programs.  note: this does
// not use any dependence analysis, so it is not
// very ambitious.  it is just an example of
// how to create variable, declare them, creates
// expressions , create statements and functions
// and install them in a program.
```

Restructure - addStuffToProgram

```
void addStuffToProgram( SgProject *project, SgFile *file){
  SgValueExp c1(1), c2(2), c3(3), c100(100);
  SgExpression *pt;
  SgVarRefExp  *e1, *e2, *e3, *e4;
  SgStatement *themain, *first, *firstex, *last;
  SgFuncHedrStmt *ptfunc;
  SgSymbol *ptsymb;
  SgSymbol *i1;
  SgSymbol *i2;
  SgSymbol *i3;
  SgSymbol *i4;
  SgSymbol *anarray;
  SgAssignStmt *stmt, *stmt1;
  SgIfStmt *anif;
  SgStatement *anotherif;
  SgWhileStmt *awhile;
  SgForStmt *afor;
  SgReturnStmt *areturn;
  SgCallStmt *afuncall;
  SgArrayType *typearray;
```

```
SgType basetype(T_FLOAT);


printf("There is %d files in that project\n",project->numberOfFiles());

// get the first statement in the file this is the GLOBAL_BFND, i.e.
// the header for the file and not a real statement in the program.

first = (file->firstStatement());
// find the main program.
themain = (file->mainProgram());
if(!themain){
         printf("no main program here!\n");
         return;
 }

// first create a new function

ptfunc = new SgFuncHedrStmt("funct1");

// next add some parameters.

ptsymb = new SgVariableSymb("var1");
pt = new SgVarRefExp(*ptsymb);
ptfunc->AddArg(*pt);

ptsymb = new SgVariableSymb("var2");
pt = new SgVarRefExp(*ptsymb);
ptfunc->AddArg(*pt);

// Now insert our function after the file
// header. this is a bit hard to read.
// x->instertStmtAfter(y) means put y after x.

first->insertStmtAfter(*ptfunc);

// lets add a statement to that function
// first declare some integer variables.

i1 = new SgVariableSymb("i1");
i1->declareTheSymbol(*ptfunc);
e1 = new SgVarRefExp(*i1);

i2 = new SgVariableSymb("i2");
i2->declareTheSymbol(*ptfunc);
e2 = new SgVarRefExp(*i2);

i3 = new SgVariableSymb("i3");
i3->declareTheSymbol(*ptfunc);
e3 = new SgVarRefExp(*i3);
```

```
i4 = new SgVariableSymb("i4");
i4->declareTheSymbol(*ptfunc);
e4 = new SgVarRefExp(*i4);

// next find the last declaration in the function.
// so that we can add statements after it.

firstex = (ptfunc->lastDeclaration());

// make the assignment i1 = i2 + (i3 + 1)*i4

stmt = new SgAssignStmt((*e1), (*e2) + ((*e3) + c1) * (*e4));

// mane another assignment i2 = i3

stmt1 = new SgAssignStmt(*e2,*e3);

// make an if statement: if( 1.gt.2) then
//                          stmt1
//                       else
//                          a copy of stmt
//                       endif

anif = new SgIfStmt(c1 > c2 , *stmt1, stmt->copy());

// make another if statement that is a copy of the last.

anotherif = &(anif->copy());

// make a while loop: while (i4 < 2)
//                         a copy of the if statement
//                    endwhile

awhile = new SgWhileStmt( (*e4)< c2 , anif->copy());

// make a do loop:  do i1 = 1, 2, 3
//                      a copy of the while statement
//                  enddo

afor = new SgForStmt(* i1, c1, c2, c3, awhile->copy());

// now a return statement

areturn = new SgReturnStmt();

// now a subroutine call:  call funct1(1,2,3)
```

```
    afuncall = new SgCallStmt(*ptfunc->symbol());
    afuncall->addArg(c1.copy());
    afuncall->addArg(c2.copy());
    afuncall->addArg(c3.copy());

  // let insert what we have created into the subroutine.
  // note these will come out in reverse order of insertion
  // becuase we keep inserting them after the last declaration.

    firstex->insertStmtAfter(*anif);
    firstex->insertStmtAfter(stmt->copy());
    firstex->insertStmtAfter(*awhile);
    firstex->insertStmtAfter(*afor);

  // to do it the other way, find the last statement.
  // and insert the return at the end.

    last = (ptfunc->lastExecutable());
    last->insertStmtAfter(*areturn);

  // now put the if and the function call in the main
  // program.

    themain->insertStmtAfter(*anotherif);
    themain->insertStmtAfter(*afuncall);

  // Let's create an array.  basetype was declared as float above.
  // first we create a type.  in sage an array type is its base
  // type plus its dimensions (ranges).

     typearray = new SgArrayType(basetype);
     typearray->addRange(c1);
     typearray->addRange(c2);
     typearray->addRange(c3);

  // now create the array symbol of this type.

     anarray = new SgVariableSymb("Array1",*typearray);
     anarray->declareTheSymbol(*ptfunc);

  // make an array expression Array1(i1, i2, i3)

     pt =  new SgArrayRefExp(*anarray,*e1,*e2,*e3);

  // make an assignment statement
  //   Array1(i1, i2, i3) = i2 + (Array1(i1,i2,i3) + 1)*Array1(i1,i2,i3)

     stmt = new SgAssignStmt((*pt), (*e2) + ((*pt) + c1) * (*pt));

  // insert it into our subroutine.
```

```
     firstex->insertStmtAfter(*stmt);

  // unparse the file
    file->unparsestdout();

  // and save the modified dep file.

    file->saveDepFile("debug.dep");

  }
```

Restructure - main

```
//This main program assumes there is a fortran file called.
// x.f which has been parsed with cfp to generate a x.dep file.
// also there must be a project file x.proj  which
// is just one line containing the name x.dep
// multiple source files may be added to the project by adding
// the corresponding dep file name (one to a line.)
// however, sage has not been tested on multiple source
// files lately and it probably will not work.

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

// better to grab the names from argv.
// but lets keep it simple.
  SgProject project("x.proj");
  SgFile file("x.f");
  int c;

// first check to see what language the source file is.
// each dep file has the language type encoded.

  switch (file.languageType()) {
  case CSrc: // it is C or C++
         printf("this is a c program.  this example is for fortran \n");
    return -1;
  case ForSrc: // it is a fortran 77 or fortran 90 program.
      addStuffToProgram(&project, &file);
      // ProjectUnparse(&project);
  }
  return 0;
}
```

9.2 Instrument

```
/*******************************************************************/
/*                 pC++/Sage++  Copyright (C) 1993                 */
/*  Indiana University  University of Oregon  University of Rennes   */
/*******************************************************************/


#include <stdio.h>
#include <malloc.h>
#define DISTRIBUTION
#include "sage++user.h"

#define MAXSUBS 100

struct ftable {
  char *function_name;
  SgStatement *function;
} ftable;

struct ftable function_table[MAXSUBS];
int totalfunctions;
static int event = 0;
SgProject *project;
```

Instrument - Fortran Program Transformations

```
// Fortran program transformations

SgType *SgTypeInt();

SgSymbol *entry, *eexit, *loop_entry, *loop_exit;
char *fnames[] = {"entry_event",
  "exit_event",
  "loop_entry_event",
  "loop_exit_event"};

#define ENTRY_EVENT 0
#define EXIT_EVENT  1
#define LOOP_ENTRY  2
#define LOOP_EXIT   3

#define AFTER  0
#define BEFORE 1
```

Instrument - InitSymbols

```
// Initialize the function symbols (used in the instrumentation)
void
InitSymbols(SgProject *p)
{
  SgFile *f;

  f = &(p->file(0));
  entry = new SgFunctionSymb(PROCEDURE_NAME,fnames[0],
     *SgTypeInt(),*(f->firstStatement()));
  eexit  = new SgFunctionSymb(PROCEDURE_NAME,fnames[1],
     *SgTypeInt(), *(f->firstStatement()));
  loop_entry= new SgFunctionSymb(PROCEDURE_NAME,fnames[2],
      *SgTypeInt(), *(f->firstStatement()));
  loop_exit = new SgFunctionSymb(PROCEDURE_NAME,fnames[3],
     *SgTypeInt(), *(f->firstStatement()));
}
```

Instrument - InitFunctionTable

```
// Initialize the function table (used for searching for functions).
void
InitFunctionTable(SgProject *project)
{

  int Files = project->numberOfFiles();
  int k = 0;

  for (int i = 0; i < project->numberOfFiles(); i++) {
    SgFile *f;

    f = &(project->file(i));

    int num_routines;

    num_routines = f->numberOfFunctions();
    for (int j = 0; j < num_routines; j++){
      SgStatement *sub;
      SgSymbol *subsym;

      sub = function_table[k].function = f->functions(j);
      subsym = sub->symbol();
      function_table[k++].function_name = subsym->identifier();
#if 0
      printf("Function %d's name is %s\n", i*Files+j, subsym->identifier());
```

```
#endif
    }
  }

  totalfunctions = k;
}
```

Instrument - InsertFCallNode

```
// Inserting a call to the instrumentation library
void InsertFCallNode(int event_type, SgStatement *s, int how)
{
  SgSymbol *symb;

  switch (event_type) {
    case ENTRY_EVENT: symb = entry;
                    break;
    case EXIT_EVENT:  symb = eexit;
                    break;
    case LOOP_ENTRY:  symb = loop_entry;
                    break;
    case LOOP_EXIT :  symb = loop_exit;
                    break;
  }


  SgCallStmt *c;
  SgValueExp *ev;

  c = new SgCallStmt(*symb);
  ev = new SgValueExp(event);
  c->addArg(*ev);
  delete(ev);

  event++;

  switch (how) {
    case BEFORE:
         s->insertStmtBefore(*c);
         break;
    case AFTER:
         s->insertStmtAfter(*c);
         break;
  }
}
```

Instrument - FixLoops

```
// Fixing the loops in the subroutine (massaging the program)
void
FixLoops(SgStatement* s)
{
  SgStatement *br;
  SgForStmt *f;

  for (br = s->lexNext(); ; br = br->lexNext()) {
    if (f = isSgForStmt(br))
      if (! f->isEnddoLoop())
f->convertLoop();
   if (br == s->lastNodeOfStmt())
     break;

  }
}
```

Instrument - FInstrumentSub

```
// Instrumentation running within a subroutine.
void
FInstrumentSub(SgStatement* s)
{
  SgStatement *br;

  for (br = s->lexNext(); ; br = br->lexNext()) {
    switch (br->variant()){
    case FOR_NODE:
      InsertFCallNode(LOOP_ENTRY, br, BEFORE);
      InsertFCallNode(LOOP_EXIT, br->lastNodeOfStmt(), AFTER);
      break;
    default:
         break;
    }
    if (br == s->lastNodeOfStmt())
      break;
  }
}
```

Instrument - FInitialize

```
// Basic initialization
```

```
extern "C"
void
FInitialize(SgProject *project)
{
  InitSymbols(project);
  InitFunctionTable(project);
}
```

Instrument - FInstrument

```
// Instrumentation call (search for function symbol based on function name)
extern "C"
void
FInstrument(char *name)
{
  SgStatement *sub;

  for(int i=0; i < totalfunctions; i++) {
    if (strcmp(name, function_table[i].function_name) == 0)
      break;
  }

  sub = function_table[i].function;
  FixLoops(sub);
  InsertFCallNode(ENTRY_EVENT, sub->lastDeclaration(), AFTER);
  InsertFCallNode(EXIT_EVENT, sub->lastExecutable(), AFTER);
  FInstrumentSub(function_table[i].function);
}
```

Instrument - ProjectUnparse

```
extern "C"
void
ProjectUnparse(SgProject *project)
{
  for (int i = 0; i < project->numberOfFiles(); i++) {
    SgFile *f;

    f = &(project->file(i));
    f->unparsestdout();
  }
}
```

Instrument - UnparseSub

```
extern "C"
void
UnparseSub(char *name){
  SgStatement *sub;

  for(int i=0; i < totalfunctions; i++) {
    if (strcmp(name, function_table[i].function_name) == 0)
      break;
  }

  sub = function_table[i].function;
  sub->unparsestdout();
}
```

Instrument - VistaInstrument

```
// needs to be filled in (used for instrumentation for animation)
void
VistaInstrument(SgStatement *s)
{
  SgStatement *br;
  SgExprListExp *l;
  SgExpression *e;
  SgExpression *a;

  for (br = s->lexNext(); ; br = br->lexNext()) {
    for (l = isSgExprListExp(br->expr(0)); l ; l = l->next()){
      e = l->value();
    }

    if (br == s->lastNodeOfStmt())
      break;
  }
}
```

Instrument - FAnalyze

```
// Instrumenting the entire program.

void
FAnalyze(SgProject *project)
```

```
{
  char s[32];


  printf ("Project has %d number of files\n", project->numberOfFiles());
  for (int i = 0; i < project->numberOfFiles(); i++) {
    SgFile *f;

    f = &(project->file(i));
    printf("Source file %s has", project->fileName(i));

    int num_routines;

    num_routines = f->numberOfFunctions();
    printf ("%d routines\n", num_routines);
    for (int j = 0; j < num_routines; j++) {
      SgStatement *sub;
      SgStatement *exe;

      sub = f->functions(j);
      exe = sub->lastDeclaration();
      InsertFCallNode(ENTRY_EVENT, exe, AFTER);
      FixLoops(sub);
//    FInstrumentSub(sub);
    }
    f->unparsestdout();
    printf ("\n Done unparsing\n");
#if 0
    f->saveDepFile(sprintf(s,"debug%d.dep",i));
#endif
  }

  for (i = 0; i < project->numberOfFiles(); i++) {
    SgFile *f;

    f = &(project->file(i));
    printf("Source file %s has", project->fileName(i));

    int num_routines;

    num_routines = f->numberOfFunctions();
    printf ("%d routines\n", num_routines);
    for (int j = 0; j < num_routines; j++) {
      SgStatement *sub;

      sub = f->functions(j);
      VistaInstrument(sub);
    }
    f->unparsestdout();
    printf ("\n Done unparsing\n");
```

```
#if 0
    f->saveDepFile(sprintf(s,"debug%d.dep",i));
#endif
  }

}
```

Instrument - pCxx Program Transformations

```
// pCxx Program transformations.

SgSymbol *timer_clear, *timer_start, *timer_stop, *timer_elapsed;

char *cnames[] = {"Test_CMMD_node_timer_clear",
  "Test_CMMD_node_timer_start",
  "Test_CMMD_node_timer_stop",
  "Test_CMMD_node_timer_elapsed"};

#define TIMER_CLEAR 0
#define TIMER_START 1
#define TIMER_STOP  2
#define TIMER_ELAPSED   3
```

Instrument - isReferenceToMethodOfElement

```
// Basic help routines (all obtained from the compiler)
// Should be made part of the library.
SgSymbol *
isReferenceToMethodOfElement(SgExpression *expr)
{
   SgVarRefExp *varef;
   SgFunctionCallExp *fcall;
   SgSymbol *symb;
   SgMemberFuncSymb *symbm;
```

```
   SgFieldSymb *symbf;
   if (!expr)
     return NULL;
   if (varef = isSgVarRefExp(expr))
     {
       symb = varef->symbol();
     }
   if (fcall = isSgFunctionCallExp(expr))
     {
       symb = fcall->symbol();
     }
   if (symbm = isSgMemberFuncSymb(symb))
     {
       if (symbm->isMethodOfElement())
         return symbm;
       else
         return NULL;
     }
   if (symbf = isSgFieldSymb(symb))
     {
       if (symbf->isMethodOfElement())
         return symbf;
       else
         return NULL;
     }
   return 0;
}
```

Instrument - isReferenceToClassOfElement

```
SgSymbol *
isReferenceToClassOfElement(SgDerivedCollectionType *type,SgExpression *expr)
{
  SgVarRefExp *varef;
  SgFunctionCallExp *fcall;
  SgSymbol *symb,*elsymb;
  SgMemberFuncSymb *symbm;
  SgFieldSymb *symbf;
  SgType *elmetype;
  SgDerivedType *eltype;

   if (!expr || !type)
     return NULL;
  elmetype = type->elementClass();
  if (!elmetype)
    return NULL;
  if (elmetype->isTheElementType())
    return NULL;
```

```
  if (eltype = isSgDerivedType(elmetype))
    {
      if (!(elsymb = eltype->typeName()))
        {
          return NULL;
        }
    } else
      return NULL;

   if (varef = isSgVarRefExp(expr))
     {
       symb = varef->symbol();
     }
   if (fcall = isSgFunctionCallExp(expr))
     {
       symb = fcall->symbol();
     }

   if (symbm = isSgMemberFuncSymb(symb))
     {
       if (symbm->className() == elsymb)
         return symbm;
       else
         return NULL;
     }
   if (symbf = isSgFieldSymb(symb))
     {
       if (symbf->structureName() == elsymb)
         return symbf;
       else
         return NULL;
     }
   return NULL;
}
```

Instrument - inMethodOfTheElement

```
int
inMethodOfTheElement(SgStatement *where)
{
  SgFuncHedrStmt *fh;
  SgMemberFuncSymb *symb;

  if (!where)
    return 0;
```

```
  if (where->variant() == GLOBAL)
    return 0;

  if (fh = isSgFuncHedrStmt(where))
    {
      if (symb = isSgMemberFuncSymb(fh->symbol()))
        {
          return symb->isMethodOfElement();
        }
    }
  return inMethodOfTheElement(where->controlParent());
}
```

Instrument - whichFunctionAmI

```
SgFuncHedrStmt *
whichFunctionAmI(SgStatement *where)
{
  SgFuncHedrStmt *fh;
  SgMemberFuncSymb *symb;

  if (!where)
    return NULL;

  if (where->variant() == GLOBAL)
    return NULL;

  if (fh = isSgFuncHedrStmt(where))
    {
      return fh;
    }
  return whichFunctionAmI(where->controlParent());
}
```

Instrument - isReferenceToCollection

```
//
// where is used to disambiguate this
//
SgDerivedCollectionType *
isReferenceToCollection(SgExpression *expr, SgStatement *where)
 {
```

```
    SgType *type = NULL;
    SgExpression *lhs;
    SgVarRefExp *varef;
    SgPointerType *tpt;
    SgReferenceType *tref;
    SgDerivedCollectionType *dstype;
    SgArrayRefExp *array;
    SgSymbol *symb;
    SgFunctionCallExp *func;

    if (!expr)
      return NULL;
    switch (expr->variant())
      {
      case RECORD_REF:
      case POINTST_OP:
      case DEREF_OP:
        lhs = expr->lhs();
        break;
      case VAR_REF:
      case ARRAY_REF:
      case FUNC_CALL:
        lhs = expr;
        break;
        default :
// default should be to try to generate a copy for the expression
```

```
// and see if expression becomes call the function again.
         return NULL;
     }
   if (!lhs)
     return NULL;
   if (varef = isSgVarRefExp(lhs))
     {
       if (varef->symbol())
         type = varef->symbol()->type();
     } else
       if (isSgThisExp(lhs))
         {
           if (inMethodOfTheElement(where))
             return 0; // this refer to the element
           type = lhs->type();
         } else
           if (array = isSgArrayRefExp(lhs))
             {
               symb = array->symbol();
               if (symb)
                 type = symb->type();
               if (type)
                 type = type->internalBaseType();
             } else
               if (func = isSgFunctionCallExp(lhs))
                 {
                   symb = func->symbol();
                   if (symb)
                     type = symb->type();
                 } else
                   if  (isSgPointerDerefExp(lhs))
                     {
                       type = isReferenceToCollection(lhs,where);
                     } else
                       if  (isSgExprListExp(lhs))
                         {
                           type = isReferenceToCollection(lhs->lhs(),where);
                         } else
                           return NULL;
   if (!type)
     return NULL;
   switch (expr->variant())
     {
     case FUNC_CALL:
     case ARRAY_REF:
     case VAR_REF:
     case RECORD_REF:
       if (dstype = isSgDerivedCollectionType(type))
         return dstype;
       if (tref = isSgReferenceType(type))
         {
           type = tref->baseType();
           if (dstype = isSgDerivedCollectionType(type))
             return dstype;
         }
       break;
     case DEREF_OP:
     case POINTST_OP:
       if (tpt = isSgPointerType(type))
         {
           type = tpt->baseType();
           if (dstype = isSgDerivedCollectionType(type))
             return dstype;
           if (tref = isSgReferenceType(type))
             {
               type = tref->baseType();
               if (dstype = isSgDerivedCollectionType(type))
                 return dstype;
             }
         }
       break;
     }
   return NULL;
 }
```

Instrument - ListCollections

```
// Simple routines to look into a pCxx Program.
void
ListCollections(SgFile *file)
{
  SgStatement *s;
  SgCollectionStmt *c;

  for (s = file->firstStatement(); s ; s = s->lexNext()){
    if (c = isSgCollectionStmt(s)){
      c->unparsestdout();
    }
  }
}
```

Instrument - ListCollectionInstances

```
void
ListCollectionInstances(SgFile *file)
{
  SgType *type;
  SgDerivedCollectionType *c;

  for (type = file->firstType(); type ; type = type->next()){
    if (c = isSgDerivedCollectionType(type)){
      SgStatement *s;

      printf("%s<%s>\n",c->collectionName()->identifier(),
              c->elementClass()->symbol()->identifier());
    }
  }
}
```

Instrument - ListCollectionInvocations

```
void
ListCollectionInvocations(SgFile *f)
```

```
{
  SgStatement *br;
  SgExprListExp *l;
  SgExpression *e;
  SgStatement *s;
  SgDerivedCollectionType *a;
  SgSymbol *b;

  s = f->firstStatement();
  for (br = s; br; br = br->lexNext())
    for (l = isSgExprListExp(br->expr(0)); l ; l = l->next()){
      e = l->value();
      switch (e->variant()) {
        case DEREF_OP:
        case RECORD_REF:
        case POINTST_OP:
  a = isReferenceToCollection(e, br);
  b = isReferenceToMethodOfElement(e->rhs());
  if (a && !b)
    b = isReferenceToClassOfElement(a, e->rhs());
  if (a && b)
    br->unparsestdout();
      }
    }
}
```

Instrument - CAnalyze

```
void
CAnalyze(SgProject *project){

  for (int i = 0; i < project->numberOfFiles(); i++) {
    SgFile *f;

    f = &(project->file(i));
    ListCollections(f);
    ListCollectionInstances(f);
    printf("Source file %s has ", project->fileName(i));

    int num_routines;

    num_routines = f->numberOfFunctions();
    printf ("%d routines\n", num_routines);
    ListCollectionInvocations(f);
  }
}
```

Instrument - CInitSymbols

```
// Initializing the timer function symbols.
void CInitSymbols(SgProject *p)
{
  SgFile *f;

  f = &(p->file(0));
  timer_clear = new SgFunctionSymb(FUNCTION_NAME,cnames[0],
   *SgTypeInt(),*(f->firstStatement()));
  timer_start  = new SgFunctionSymb(FUNCTION_NAME,cnames[1],
    *SgTypeInt(), *(f->firstStatement()));
  timer_stop = new SgFunctionSymb(FUNCTION_NAME,cnames[2],
  *SgTypeInt(), *(f->firstStatement()));
  timer_elapsed = new SgFunctionSymb(FUNCTION_NAME,cnames[3],
     *SgTypeInt(), *(f->firstStatement()));
}
```

Instrument - CInitialize

```
extern "C"
void
CInitialize(SgProject *project)
{
  CInitSymbols(project);
  InitFunctionTable(project);
}
```

Instrument - InsertCCallNode

```
// Insert a C function call statement
void
InsertCCallNode(int event_type, SgStatement *s, int how)
{
  SgSymbol *symb;
```

```
  switch (event_type) {
    case TIMER_CLEAR: symb = timer_clear;
                      break;
    case TIMER_START: symb = timer_start;
                      break;
    case TIMER_STOP:  symb = timer_stop;
                      break;
    case TIMER_ELAPSED: symb = timer_elapsed;
                       break;
  }

  SgFunctionCallExp *func;
  SgExprListExp *exp;
  SgCExpStmt *c;

  func = new SgFunctionCallExp(*symb);
  exp = new SgExprListExp();
  exp->setValue(*func);
  c = new SgCExpStmt (*exp);


  switch (how) {
    case BEFORE:
         s->insertStmtBefore(*c);
         break;
    case AFTER:
         s->insertStmtAfter(*c);
         break;
  }
}
```

Instrument - CTimingInstrumentSub

```
// Instrument a pC++ function for timing collection invocations.

void
CTimingInstrumentSub(SgStatement *s){

  SgStatement *br;
  SgExprListExp *l;
  SgExpression *e;
  SgDerivedCollectionType *a;
  SgSymbol *b;

  for (br = s->lastDeclaration(); ; br = br->lexNext()) {
    for (l = isSgExprListExp(br->expr(0)); l ; l = l->next()){
      e = l->value();
```

```
      switch(e->variant()){
      case DEREF_OP:
      case POINTST_OP:
      case RECORD_REF:
a = isReferenceToCollection(e, br);
b = isReferenceToMethodOfElement(e->rhs());
if (a && !b)
  b = isReferenceToClassOfElement(a, e->rhs());

if (a && isSgCExpStmt(br)) {
  InsertCCallNode(TIMER_START, br, BEFORE);
  InsertCCallNode(TIMER_STOP, br, AFTER);
        InsertCCallNode(TIMER_ELAPSED, br->lexNext(), AFTER);
}
      }
    }
    if (br == s->lastNodeOfStmt())
      break;
  }
}
```

Instrument - CTimingInstrument

```
// Instrumentation based on function name.

extern "C"
void
CTimingInstrument(char *name){

  for(int i=0; i < totalfunctions; i++) {
    if (strcmp(name, function_table[i].function_name) == 0)
      break;
  }

  CTimingInstrumentSub(function_table[i].function);
}
```

Instrument - OpenProject

```
extern "C"
void
OpenProject(char *proj) {
```

```
  project = new SgProject (proj);
}
```

Instrument - IsFortran

```
#define TRUE  1
#define FALSE 0

extern "C"
int
IsFortran() {
  SgFile *file;

  file = &(project->file(0));
  if (file->languageType() == ForSrc)
    return TRUE;
  else
    return FALSE;
}
```

Instrument - IsC

```
extern "C"
int
IsC() {
  SgFile *file;

  file = &(project->file(0));
  if (file->languageType() == CSrc)
    return TRUE;
  else
    return FALSE;
}
```

9.3 Expand Syntax

```
/*********************************************************************/
/*                  pC++/Sage++  Copyright (C) 1993                  */
/*  Indiana University  University of Oregon  University of Rennes   */
/*********************************************************************/
```

```
/*****************************************************************
**  A Simple application of Sage++ that analyzes pCxx Programs and does
**  one simple source transformation to eliminate some extra communication
**  and expand some syntax.
**
**  pC++ has distributed data-structures called Collections that are
**  homogenous and they contain classes as elements. In a pCxx Program to
**  refer to any particular element an array like syntax is used. To refer
**  to a field within an element that resides on someother processor is
**  expensive at present. The reason being when a reference to a nonlocal
**  element occurs the whole element is brought in and then the particular
**  field that is required is extracted from it.
**
**  This application analyzes pCxx Program for "Collection Array
**  References" and transforms them so that the communication for getting
**  fields of non local elements is not more than the "length" of the
**  field.
**
**  As an example consider the following pCxx Program and in
**  Processor_Main are some potential non local references.
**
**  class Simple{
**   public:
**    int foo;
**    float bar[10];
**    void hello() {printf("hello world\n");};
**  };
**
**  #include "distarray.h"
**
**  Collection DistBlkVector: SuperKernel
**  {
**   public:
**    DistBlkVector(Distribution *T, Align *A);
**   MethodOfElement:
**    virtual void hello();
**  };
**
**
**  DistBlkVector::DistBlkVector(Template *T, Align *A):SuperKernel(T, A)
**  {}
**
**
**  void Processor_Main()
**  {
**    int i,j;
**    Processors P;
**    Distribution T(GRIDSIZE,&P,BLOCK);
**    Align  A(GRIDSIZE,"[ALIGN(V[i],T[i])]");
```

```
**    Distribution T1(GRIDSIZE,GRIDSIZE&P,BLOCK, BLOCK);
**    Align A1(GRIDSIZE,"[ALIGN(V[i][j],T[i][j])]");
**    DistBlkVector<Simple>   x(&T,&A);
**    DistBlkVector<Simple>   y(&T,&A);
**
**    x(i)->foo; // xformed to x.Get_ElementPart_Int(i, <foo-offset>, <sizeof (int)>)
**    x(i)->hello();
**
**    y(i,j)->bar // xformed to y.GetElementPart(i, <bar-offset>, 10 * <sizeof (float)>)
**    y(i,j)->hello();
**
**  }
**
*****************************************************************/


#include <stdio.h>
#include <malloc.h>
#include <sage++user.h>


// Make the dummy var table into a class and clean it up.
#define MAXDUMMYVAR 20

SgSymbol     *DummyVar[MAXDUMMYVAR];
SgType       *DummyVarType[MAXDUMMYVAR];
SgExpression *DummyVarExpr[MAXDUMMYVAR];

int TotalDummyVars = 0;

SgFunctionSymb *GetElementPart, *GetElemInt, *GetElemFloat, *GetElemDouble;
SgVariableSymb *addr1, *addr2;

void SearchInExpForCollectionArrayRef(SgStatement *br, SgExpression *e,
      int & level);
void ReplaceWithGetElemPart(SgStatement *br, SgExpression *e,
    SgDerivedCollectionType *a, SgSymbol *b, int level);
int MatchingDummyVar(SgType *x);
```

Expand Syntax - isReferenceToElementField

```
SgSymbol *
isReferenceToElementField(SgExpression *expr)
{
  SgVarRefExp *varef;
  SgArrayRefExp *aref;
  SgSymbol *symb;
  SgFieldSymb *symbf;
```

```
  if (!expr)
    return NULL;
  if (varef = isSgVarRefExp(expr))
    {
      symb = varef->symbol();
      if (symbf = isSgFieldSymb(symb))
return symbf;
    }
  else if (aref = isSgArrayRefExp(expr))
    {
      symb = aref->symbol();
      if (symbf = isSgFieldSymb(symb))
return symbf;
    }
  return NULL;
}
```

Expand Syntax - isArrayRefOfCollection

```
// Looks into the expression for collection Array References
// Returns the type of the collection if it finds one.

SgDerivedCollectionType *
isArrayRefOfCollection(SgExpression *expr)
{
  SgType *type = NULL;
  SgExpression *lhs;
  SgVarRefExp *varef;
  SgDerivedCollectionType *dstype;

  if (!expr)
    return NULL;

  switch (expr->variant())
    {
    case FUNCTION_OP:
      lhs = expr->lhs();

      break;
    default :
      return NULL;
    }

  if (!lhs)
    return NULL;
```

```
  if (varef = isSgVarRefExp(lhs))
    {
      if (varef->symbol())
type = varef->symbol()->type();
    }

  if (!type)
    return NULL;

  if (dstype = isSgDerivedCollectionType(type))
    return dstype;

  return NULL;
}
```

Expand Syntax - ExpandSyntax

```
void
ExpandSyntax(SgFile *f)
{
  SgStatement *br;
  SgExprListExp *l;
  SgExpression *e;
  SgStatement *s;
  int lvl = 0;

  s = f->firstStatement();
  for (br = s; br; br = br->lexNext())
    if (isSgCExpStmt(br)) {
      for (l = isSgExprListExp(br->expr(0)); l ; l = l->next()) {
e = l->value();
SearchInExpForCollectionArrayRef(br,e, lvl);
lvl = 0;
      }
    }
}
```

Expand Syntax - SearchInExpForCollectionArrayRef

```
// Need the level stuff to handle multiple references collection field
// references in one statement.
```

```
void SearchInExpForCollectionArrayRef(SgStatement *br, SgExpression *e,
      int & level)
{

  if (e == NULL)
    return;

  if (e->variant() == POINTST_OP) {

    SgDerivedCollectionType *a;
    SgSymbol *b;

    a = isArrayRefOfCollection(e->lhs());
    b = isReferenceToElementField(e->rhs());

    if (a && b) {
      ReplaceWithGetElemPart(br, e, a, b, level);
      level = level + 1;
    }
  }
  else {
    SearchInExpForCollectionArrayRef(br, e->lhs(), level);
    SearchInExpForCollectionArrayRef(br, e->rhs(), level);
  }
}
```

Expand Syntax - ReplaceWithGetElemPart

```
void ReplaceWithGetElemPart(SgStatement *br, SgExpression *e,
    SgDerivedCollectionType *a, SgSymbol *b, int level)
{

  SgBasicBlockStmt bgn;
  SgControlEndStmt nd;
  SgStatement *ext, *brnext;
  int matchingdummy;
  SgType *btype;

  matchingdummy = MatchingDummyVar(a->elementClass());
  if (!matchingdummy)
    Message("Dummy variable has not been created!\n", 0);

  br->insertStmtBefore(bgn, *(br->controlParent()));
  brnext = br->lexNext();
  ext = br->extractStmt();
```

```
  SgExpression dummy(RECORD_REF);
  dummy.setLhs(*DummyVarExpr[matchingdummy]);
  dummy.setRhs(*(e->rhs()));


  char addr1str[16], addr2str[16];
  if (level == 0) {
    strcpy(addr1str, "addr1");
    strcpy(addr2str, "addr2");
  }
  else {
    sprintf(addr1str, "addr1_%d", level);
    sprintf(addr2str, "addr2_%d", level);
  }

  SgPointerType p(*SgTypeInt());
  SgVariableSymb addr1(addr1str,p);
  SgVariableSymb addr2(addr2str,p);
  SgVarRefExp addr1ref(addr1), addr2ref(addr2);



  SgTypeArray *arrtype;
  if ((arrtype = isSgTypeArray(b->type())) && isSgArrayRefExp(e->rhs()))
    btype = arrtype->baseType();
  else
    btype = b->type();

  SgFunctionSymb *fsymb;

  if (btype->equivalentToType(*SgTypeInt()))
    fsymb = GetElemInt;
  else if (btype->equivalentToType(*SgTypeFloat()))
    fsymb = GetElemFloat;
  else if (btype->equivalentToType(*SgTypeDouble()))
    fsymb = GetElemDouble;
  else
    fsymb = GetElementPart;

  SgFunctionCallExp func(*fsymb);
  func.addArg(*(e->lhs()->rhs()));
  func.addArg(addr2ref - addr1ref);


// sizeof will be wrong for array's like x(i)->foo where foo
// is an array so do more work. Handle only one, two and three dimension
// cases.

// Need to work harder here. What about stuff like x(i)->foo[10] where the
// field foo is a 2d array.
```

```
  if ((arrtype = isSgTypeArray(b->type())) && isSgVarRefExp(e->rhs())) {
    if (arrtype->dimension() == 1)
      func.addArg(SgSizeOfOp(dummy) * arrtype->sizeInDim(0)->copy());
    else if (arrtype->dimension() == 2)
      func.addArg(SgSizeOfOp(dummy) * arrtype->sizeInDim(0)->copy()
  * arrtype->sizeInDim(1)->copy());
    else if (arrtype->dimension() == 3)
      func.addArg(SgSizeOfOp(dummy) * arrtype->sizeInDim(0)->copy()
  * arrtype->sizeInDim(1)->copy()
  * arrtype->sizeInDim(2)->copy());
  }
  else
    func.addArg(SgSizeOfOp(dummy));

  SgVarRefExp *collref;
  collref = isSgVarRefExp(e->lhs()->lhs());

  e->setVariant(RECORD_REF);
  e->setLhs(*collref);
  e->setRhs(func);


  SgExpression x1(ASSGN_OP);
  x1.setLhs(addr1ref);
  x1.setRhs(SgAddrOp(*DummyVarExpr[matchingdummy]));
  SgExprListExp exp1(x1);
  SgCExpStmt addr1assgn(exp1);

  SgExpression x2(ASSGN_OP);
  x2.setLhs(addr2ref);
  x2.setRhs(SgAddrOp(dummy));
  SgExprListExp exp2(x2);
  SgCExpStmt addr2assgn(exp2);

  bgn.insertStmtAfter(addr1assgn);
  addr1assgn.insertStmtAfter(addr2assgn);
  addr2assgn.insertStmtAfter(*ext);
  ext->insertStmtAfter(nd);

  addr1.declareTheSymbol(bgn);
  addr2.declareTheSymbol(bgn);
}
```

Expand Syntax - MatchingDummyVar

```
int MatchingDummyVar(SgType *x)
```

```
{
  for (int i=0; i < TotalDummyVars; i++)
    if (DummyVarType[i]->equivalentToType(*x))
      return i;
  return 0;
}
```

Expand Syntax - InDummyVarTable

```
int InDummyVarTable(SgType *x)
{

  for (int i=0; i < MAXDUMMYVAR; i++)
    if (DummyVarType[i] != NULL)
      if (DummyVarType[i]->equivalentToType(*x))
return TRUE;

  return FALSE;
}
```

Expand Syntax - Init

```
void Init( SgFile *f) {
  char strin[256];
  SgStatement *fr;
  SgPointerType *p;
  SgSymbol *symb;

  fr = f->firstStatement();

  p = new SgPointerType(*SgTypeInt());
  GetElementPart = new SgFunctionSymb(FUNCTION_NAME, "Get_ElementPart", *p, *fr);
  GetElemInt = new SgFunctionSymb(FUNCTION_NAME, "Get_ElementPart_Int", *p, *fr);
  GetElemFloat = new SgFunctionSymb(FUNCTION_NAME, "Get_ElementPart_Float", *p, *fr);
  GetElemDouble = new SgFunctionSymb(FUNCTION_NAME, "Get_ElementPart_Double", *p, *fr);

  for (int i=0; i < MAXDUMMYVAR; i++)
    {
      DummyVar[i] = NULL;
      DummyVarType[i] = NULL;
      DummyVarExpr[i] = NULL;
    }

  SgType *type;
```

```
  SgDerivedCollectionType *ct;

  for (i=0,type = f->firstType(); type; type = type->next())
    {
      if (ct = isSgDerivedCollectionType(type))
{
  SgType *elementype;
  elementype = ct->elementClass();
  if (!InDummyVarTable(elementype))
    {
      SgClassStmt *fordef;

      sprintf(strin, "Dummy_%d", i);
      DummyVar[i] = new SgVariableSymb(strin, *elementype, *fr);
      DummyVar[i]->declareTheSymbol(*fr);
      DummyVarType[i] = elementype;
      DummyVarExpr[i] = new SgVarRefExp(*DummyVar[i]);

      fordef = new SgClassStmt(*elementype->symbol());
      fr->insertStmtAfter(*fordef);

      i++;
    }
      }
    }

  TotalDummyVars = i;
}
```

Expand Syntax - CAnalyze

```
void
CAnalyze(SgProject * project){

  for (int i = 0; i < project->numberOfFiles(); i++) {
    SgFile *f;


    f = &(project->file(i));
    Init(f);
    ExpandSyntax(f);
  }
}
```

Expand Syntax - ProjectUnparse

```
void
ProjectUnparse(SgProject *project)
{
  for (int i = 0; i < project->numberOfFiles(); i++) {
    SgFile *f;

    f = &(project->file(i));
    f->unparsestdout();
  }
}
```

Expand Syntax - main

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

  SgProject project("simple.proj");
  SgFile file("simple.pc");
  int c;

  switch (file.languageType()) {
  case CSrc:

#if 1
    CAnalyze(&project);
    ProjectUnparse(&project);
#endif
    break;

  case ForSrc:
    printf("Wrong language!\n");
    exit(-1);
  }
}
```

9.4 Dump Info

```
/*********************************************************************/
/*                  pC++/Sage++  Copyright (C) 1993                  */
/*  Indiana University  University of Oregon  University of Rennes   */
/*********************************************************************/
```

```
#include <stdio.h>
#include <malloc.h>
#include "sage++user.h"

// This program demonstrates the functions that access the
// contents of the parsed program stored in the .dep files.
// It is very simple.  It traverses the program file in lexical
// order and examines each statement and prints out the
// information stored in the file about the statement.

// THIS PROGRAM WORKS, BUT IT IS NOT FINISHED!!!
```

Dump Info - variants

```
// each object in the .dep file has an integer "variant" to
// describe its type.  There is a standard defined name for
// each and that may be retrieved from the array "tag" below.
#define MAXTAGS 1000
static char *tag[MAXTAGS];
void initVariantNames(){
   for(int i = 0; i < MAXTAGS; i++) tag[i] = NULL;
#include "../h/tag.h"
}
void printVariantName(int i){
     if((i >= 0 && i < MAXTAGS) && tag[i]) printf("%s", tag[i]);
     else printf("not a known node variant");
}
```

Dump Info - ProjectUnparse

```
// the unparse operation regenerates source code.
// The function below prints the entire project to
// stdout.  note:  the current unparser will only print
// to stdout.  in the near future we will have a version
// that will print to a string buffer.

void
ProjectUnparse(SgProject *project)
{
  for (int i = 0; i < project->numberOfFiles(); i++) {
    SgFile *f;
```

```
      f = &(project->file(i));
      f->unparsestdout();
    }
  }
```

Dump Info - classifyStatements

```
// some forward declarations.

void doExpr(SgExpression *e);
void doSymb(SgSymbol *s);
void doType(SgType *t);
void doRoutineHeader(SgStatement *s);
void doIfStmts(SgStatement *s);
void doLoopStmts(SgStatement *s);
void doIOStmts(SgStatement *s);
void doGoToStmt(SgStatement *s);
void doVarDecl(SgStatement *s);
void doFullType(SgType *t);
void doSymbAttribs(SgSymbol *s);

// this function traverses the list of statements in a file.
// for each statement it prints out all the information available
// in the .dep data base.

void classifyStatements(SgFile *f){
  SgStatement *line;
  SgStatement *s;
  int i,j;

  // grab the first statement in the file.
  s = f->firstStatement();

  // follow the statements in lexical order.
  // an alternative is to follow them in scope order
  // using a recursive traversal, but this is simpler
  // for what we want to do here.

  for (line = s; line; line = line->lexNext()){
   printf("-------------------------------------------------------------------\n");
        // print the line to the standard out.
        // note: if the statement controls other blocks
        // nested within, it will print the entire block
        // of "control children".
        line->unparsestdout();
```

```
        // print out the variant tag, i.e. the type of statement.
        printf("STMT[%d] at line %d has variant:",
                                      line->id(), line->lineNumber());
        printVariantName(line->variant());
        printf("\n");

        // is this a labeled statement? if so print the label.
        if(line->hasLabel()){
SgLabel *lab = line->label();
               printf("has label[%d]\n", lab->id());
}

        // are there comments that precede this statement?
        // if so print them out.
 if(line->comments())
printf("comments befor this line:\n%s", line->comments());
```

```
          // for each type of statement call the appropriate special case
          // function.
          switch(line->variant()){
                case GLOBAL:
                      printf("The head-of-fole global node. not really a statement\n");
                      break;
                case PROG_HEDR:
                case PROC_HEDR:
                case FUNC_HEDR:
                     printf(" a new routine.\n");
                     doRoutineHeader(line);
                     break;
                case BASIC_BLOCK :
                     printf(" a beginning of a control block for C programs \n");
                     break;
                case CONTROL_END:
                     printf(" end of a control block for stmt %d\n",
                              (line->controlParent())->id());
                     break;
                case IF_NODE:
                case ELSEIF_NODE:
                case ARITHIF_NODE:
                case LOGIF_NODE:
                      doIfStmts(line);
                      break;
                case LOOP_NODE:
                case FOR_NODE:
                case FORALL_NODE:
                case WHILE_NODE:
                      doLoopStmts(line);
                      break;
                case EXIT_NODE:
                     // note the use of the special cast function
                     // we are casting to a subtype but this is o.k.
                     // with this function.
    {SgExitStmt *   exst =  isSgExitStmt(line);
     printf("an exit statement. exit from:\n");
     doSymb(exst->constructName());
                      }
     break;
                case ASSIGN_STAT:
     { SgAssignStmt *ass =  isSgAssignStmt(line);
     printf("an assignment\n");
                      // print out the two expressions on each side of =
     printf("left hand side:\n");
doExpr(ass->lhs());
     printf("\nright hand side:\n");
```

```
doExpr(ass->rhs());
                     printf("\n");
     }
     break;
               case PROC_STAT:
    { int i;
                      SgCallStmt *call = isSgCallStmt(line);
                     // a "call f(...) statement.  extract the passed
                     // arguement expressions and print them.
                     printf("a subroutine call. passed args are\n");
     for(i = 0; i < call->numberOfArgs(); i++){
printf("argument %d:\n", i);
doExpr(call->arg(i));
        printf("\n");
}
                      }
                     break;
               case WHERE_NODE:
               case ALLDO_NODE:
               case IDENTIFY:
               case FORMAT_STAT:
               case STOP_STAT:      break;
               case RETURN_STAT:    break;
               case GOTO_NODE:
               case ASSGOTO_NODE:
               case COMGOTO_NODE:
doGoToStmt(line);
                        break;
               case VAR_DECL:
               case INTENT_STMT:
               case OPTIONAL_STMT:
               case PUBLIC_STMT:
               case PRIVATE_STMT:
               case DIM_STAT:
               case ALLOCATABLE_STMT:
               case POINTER_STMT:
               case TARGET_STMT:
               case MODULE_PROC_STMT:
               case EXTERN_STAT:
               case COMM_STAT:
               case NAMELIST_STAT:
               case EQUI_STAT:
               case PARAM_DECL:
               case IMPL_DECL:
               case SAVE_DECL:
               case DATA_DECL:
```

```
doVarDecl(line);
                        break;
               case PAUSE_NODE:     break;
               case STOP_NODE:      break;
               case ASSLAB_STAT:    break;
               case COMMENT_STAT:    break;
               case CONT_STAT:
                    printf(" continue statement: end of cntl for stmt %d\n",
                             (line->controlParent())->id());
    break;
               case ENTRY_STAT:
               case STMTFN_STAT:
               case BLOCK_DATA:
               case INTRIN_STAT:
               case PROC_COM: /* process common */
               case ATTR_DECL: /* attribute declaration */
                     printf("another fortran statement\n");
     break;
                // the statements below are all valid, and will be
                // treated in example when I have more time.
               case READ_STAT:
               case WRITE_STAT:
               case PRINT_STAT:
               case BACKSPACE_STAT:
               case REWIND_STAT :
               case ENDFILE_STAT:
               case INQUIRE_STAT:
               case OPEN_STAT:
               case CLOSE_STAT:
               case OTHERIO_STAT:
```

```
doIOStmts(line);
                        break;
               case INCLUDE_STAT:
               case ALLOCATE_STMT:
               case NULLIFY_STMT:
               case DEALLOCATE_STMT:
               case SEQUENCE_STMT:
               case CYCLE_STMT:
               case EXIT_STMT:
               case CONTAINS_STMT:
               case WHERE_BLOCK_STMT:
               case MODULE_STMT:
               case USE_STMT:
               case INTERFACE_STMT:
               case OVERLOADED_ASSIGN_STAT:
               case POINTER_ASSIGN_STAT:
               case OVERLOADED_PROC_STAT:
                        printf("yet another fortran 90 statement.\n");
                        break;
               case DECOMPOSITION_STMT:  /* Old Fortran D stuff */
               case ALIGN_STMT:
               case DISTRIBUTE_STMT:
               case REDUCE_STMT:
                         printf("Old Fortran D. not treated here..needs update\n");
                         break;
               case CDOALL_NODE:  /* added for Cedar Fortran */
               case SDOALL_NODE:
               case DOACROSS_NODE:
               case CDOACROSS_NODE:
                        printf("Cedar Fortran.  who cares anymore. \n");
                        break;
               case PARDO_NODE:     /* Following added for PCF Fortran */
               case PARSECTIONS_NODE:
               case SECTION_NODE:
               case GUARDS_NODE:
               case LOCK_NODE:
               case UNLOCK_NODE:
               case CRITSECTION_NODE:
               case POST_NODE:
               case WAIT_NODE:
               case CLEAR_NODE:
               case POSTSEQ_NODE:
               case WAITSEQ_NODE:
               case SETSEQ_NODE:
               case ASSIGN_NODE:
               case RELEASE_NODE:
               case PRIVATE_NODE:
               case SCOMMON_NODE:
               case PARREGION_NODE:
               case PDO_NODE:
               case PSECTIONS_NODE:
               case SINGLEPROCESS_NODE:
               case SKIPPASTEOF_NODE:
                   printf("Old pcf fortran. needs update.  who cares anymore. \n");
   break;
               default:
                   printf("unknown statement. not treated here..I am tired.\n");
                   break;
         }
   printf("---------------------------------------------------------------------\n");
    }
```

```
}
```

Dump Info - doRoutineHeader

```
// print out the name and formal parameters for this subroutine,
// function or main program header.

void doRoutineHeader(SgStatement *s){
SgProgHedrStmt *    prog;
SgProcHedrStmt *    subr;
SgFuncHedrStmt *    func;
int i;
SgSymbol *sy;

   if(subr = isSgProcHedrStmt (s)){
printf("symbol:%s", (subr->name()).identifier());
        printf(" which is a suroutine.\n");
        printf("parameter list:\n");
for(i = 0; i < subr->numberOfParameters(); i++){
sy = subr->parameter(i);
printf("parameter(%d):", i); doSymb(sy);
                printf(" of type: "); doFullType(sy->type());
doSymbAttribs(sy);
printf("\n");
}
}
   else if(func = isSgFuncHedrStmt (s)){
printf("symbol:%s", (func->name()).identifier());
        printf("which is a function of type ");
doFullType((func->name()).type());
        printf("\n");
        printf("parameter list:\n");
for(i = 0; i < func->numberOfParameters(); i++){
sy = func->parameter(i);
printf("parameter(%d):", i); doSymb(sy);
                printf(" of type: "); doFullType(sy->type());
doSymbAttribs(sy);
printf("\n");
}
}
}
```

Dump Info - doIfStmts

```
// this routine handles the different types of "if" statements.
// not finished.
void doIfStmts(SgStatement *s){
SgLogIfStmt *    logif;
SgIfStmt *       ifst;
SgArithIfStmt *  arithif;

   if (logif = isSgLogIfStmt (s)){
printf("logical if: condition EXPR[%d] ", logif->conditional()->id());
(logif->conditional())->unparsestdout();
printf(" :if true do STMT[%d]\n", (logif->body())->id());
}
   else if(ifst = isSgIfStmt (s)){}
   else if(arithif = isSgArithIfStmt (s)){}
}
```

Dump Info - doLoopStmts

```
// this routine handles the different types of loop statements.
// only does do loops now.  more later.

void doLoopStmts(SgStatement *s){
SgForStmt *  doloop;
SgWhileStmt *  whileloop;
SgExpression *e;
   if (doloop = isSgForStmt(s)){
printf("a DO LOOP: control var:");
doSymb(doloop->symbol());
printf("\n lower bound = ");
if(e = doloop->start()) e->unparsestdout();
printf("  , upper bound = ");
if(e = doloop->end()) e->unparsestdout();
if(e = doloop->step()){
printf(" , step = ");
  e->unparsestdout();
}
printf(" \n");
if(doloop->endOfLoop()){
printf("end of loop label[%d]\n",(doloop->endOfLoop())->id());
}
}
   else if(whileloop = isSgWhileStmt (s)){}
   else printf("not finished\n");
}
```

Dump Info - doIOStmts

```
// print out info about I/O statements.
// not done.

void doIOStmts(SgStatement *s){
SgIOStmt *   iost;
SgInputOutputStmt * inoust;
SgIOControlStmt *  iocntl;
   if (iost = isSgIOStmt(s)){}
   else if(inoust = isSgInputOutputStmt (s)){}
   else if(iocntl = isSgIOControlStmt(s)){};
}

// print out the information about the different types of
// goto statements.
// not done.  only does standard goto's.
```

Dump Info - doGoToStmt

```
void doGoToStmt(SgStatement *s){
SgGotoStmt *   gotostmt;
SgAssignedGotoStmt * assgoto;
SgComputedGotoStmt * compgoto;
  if(gotostmt = isSgGotoStmt(s)){
printf("garden variety goto: branch to stmt with label [ %d ]\n",
gotostmt->branchLabel()->id());

}
  else if(assgoto = isSgAssignedGotoStmt(s)){}
  else if(compgoto = isSgComputedGotoStmt(s)){}
}
```

Dump Info - doVarDecl

```
// this does variable declarations.
// there are lots of different forms that a
// declaration can take.  this just does the standard
// types now.  not done yet.

void doVarDecl(SgStatement *s){
SgVarDeclStmt * var;
SgVarListDeclStmt *  varlist;
SgStructureDeclStmt * strucdecl;
```

```
SgNestedVarListDeclStmt*  declst;
SgParameterStmt *    parm;
SgImplicitStmt *     implic;
   if (var = isSgVarDeclStmt(s)){
 int i;
         printf("Variables declared here are:\n");
 for(i = 0; i < var->numberOfSymbols(); i++){
SgSymbol *s;
                s =  var->symbol(i);
                doSymb(s); printf(" of type: ");
if(s->type()) doFullType(s->type());
printf("\n");
}
         }
   else if (varlist = isSgVarListDeclStmt (s)){}
   /* else if (strucdecl = isSgStructureDeclStmt (s)){} */
   else if ( declst = isSgNestedVarListDeclStmt (s)){}
   else if ( parm = isSgParameterStmt (s)){}
   else if ( implic =  isSgImplicitStmt (s)){}
   else printf("not here!\n");
}
```

Dump Info - doSymb

```
void doSymb(SgSymbol *s){
    printf("SYMB[%d]:%s",s->id(),s->identifier());
}
```

Dump Info - doSymbAttribs

```
void doSymbAttribs(SgSymbol *s){
int i;
```

```
if(s->attributes()){
    i = s->attributes();
    printf(" attributes =( ");
              if(i & ALLOCATABLE_BIT ){ printf(" allocatable,"); }
              if(i & DIMENSION_BIT  ){ printf(" dim,"); }
              if(i & EXTERNAL_BIT ){ printf(" extern,"); }
              if(i & IN_BIT  ){ printf(" in,"); }
              if(i & INOUT_BIT  ){ printf(" inout,"); }
              if(i & INTRINSIC_BIT  ){ printf(" intrinsic,"); }
              if(i & OPTIONAL_BIT  ){ printf(" optional,"); }
              if(i & OUT_BIT  ){ printf(" out,"); }
              if(i & PARAMETER_BIT  ){ printf(" parameter,"); }
              if(i & POINTER_BIT ){ printf(" pointer,"); }
              if(i & PRIVATE_BIT ){ printf(" private,"); }
              if(i & PUBLIC_BIT  ){ printf(" public,"); }
              if(i & SAVE_BIT  ){ printf(" save,"); }
              if(i & SEQUENCE_BIT  ){ printf(" seq,"); }
              if(i & RECURSIVE_BIT  ){ printf(" recur,"); }
              if(i & TARGET_BIT  ){ printf(" target,"); }
    printf(")");
}
}
```

Dump Info - doFullType

```
// given a type, traverse it and generate full infomration
void doFullType(SgType *t){
   SgArrayType *arrayt;
   SgPointerType *ptr;
   SgReferenceType *ref; // only for C++
   SgDerivedType *deriv;
   SgDescriptType *descr; // only for C and C++
   SgDerivedCollectionType *col; // only for pC++
   int i,n;

   if(arrayt = isSgArrayType(t)){
        printf("dimension(");
n = arrayt->dimension();
for(i = 0; i < n; i++){
(arrayt->sizeInDim(i))->unparsestdout();
if(i < n-1) printf(", ");
}
printf(") ");
}
   else{
```

```
switch(t->variant()){
           case T_INT:     printf("integer "); break;
           case T_FLOAT:   printf("real "); break;
           case T_DOUBLE:  printf("double precision "); break;
           case T_CHAR:    printf("character "); break;
           case T_BOOL:    printf("boolean "); break;
           case T_STRING:  printf("string "); break;
           case T_RECORD:  printf("record.. more later "); break;
           case T_UNKNOWN: printf("unknown "); break;
           case T_COMPLEX: printf("complex "); break;
           case LOCAL:
           case INPUT:
           case OUTPUT:
           case IO:
               printf("do these later\n");
         break;
   }
   }
   if(t->hasBaseType()){
printf("of ");
doFullType(t->baseType());
}
}
```

Dump Info - classifyExpressions

```
// help functions for classifyExpression

void doExpr(SgExpression *e){
    printf("EXPR[%d]:",e->id());
    e->unparsestdout();
}

int lhsId(SgExpression *e){
    if(e->lhs()) return (e->lhs())->id();
    else return -1;
}
int rhsId(SgExpression *e){
    if(e->rhs()) return (e->rhs())->id();
    else return -1;
}
int symbId(SgExpression *e){
    if(e->symbol()) return (e->symbol())->id();
    else return -1;
}
char * symbName(SgExpression *e){
    if(e->symbol()) return (e->symbol())->identifier();
    else return "";
```

```
}
int typeId(SgExpression *e){
    if(e->type()) return (e->type())->id();
    else return -1;
}


void classifyExpressions(SgFile *f){
    SgExpression *e;
    for(e = f->firstExpression(); e; e = e->nextInExprTable()){
       SgValueExp *valexp = isSgValueExp(e);
       printf("EXPR[%3d] ", e->id());
       switch(e->variant()){
               case INT_VAL:
                    printf("integer value:%d ", valexp->intValue());
    break;
               case FLOAT_VAL:
                    printf("real value:%f ", valexp->floatValue());
    break;
               case DOUBLE_VAL:
                    printf("double value:%f ", valexp->doubleValue());
    break;
               case BOOL_VAL:
                    printf("bool value:%i ", valexp->intValue());
    break;
               case CHAR_VAL:
                    printf("char value:%c ", valexp->charValue());
    break;
               case STRING_VAL:
                    printf("string value:%s ", valexp->stringValue());
    break;
               case COMPLEX_VAL:
                     printf("complex: real part id=%d im part id=%d ",
                                      lhsId(e),rhsId(e));
    break;
              case KEYWORD_VAL:   /* Strings to be printed with quotes */
```

```
break;
               case CONST_REF:
     printf("constant ref: symb[%d]:%s type[%d] ",
                                  symbId(e), symbName(e), typeId(e));
                     break;
               case VAR_REF:
     printf("variable ref: symb[%d]:%s type[%d] ",
          symbId(e), symbName(e), typeId(e));
     break;
               case ARRAY_REF:
     printf("array ref: symb[%d]:%s type[%d] subscript list id (%d) ",
          symbId(e), symbName(e), typeId(e), lhsId(e));
     break;
               case RECORD_REF:
               case LABEL_REF:
break;
               case VAR_LIST:
               case EXPR_LIST:
               case RANGE_LIST:
                       printf("this item [%d]  next list [%d] ", lhsId(e), rhsId(e));
break;
               case CASE_CHOICE:
               case DEF_CHOICE:
               case VARIANT_CHOICE:
```

```
break;
               case DDOT:
               case RANGE_OP:
               case UPPER_OP:
               case LOWER_OP:
               case EQ_OP:
               case LT_OP:
               case GT_OP:
               case NOTEQL_OP:
               case LTEQL_OP:
               case GTEQL_OP:
               case ADD_OP:
               case SUBT_OP:
               case OR_OP:
               case MULT_OP:
               case DIV_OP:
               case MOD_OP:
               case AND_OP:
               case EXP_OP:
               case ARRAY_MULT:
               case CONCAT_OP: /* cancatenation of strings */
               case XOR_OP: /* .XOR. in fortran */
               case EQV_OP: /* .EQV. in fortran */
               case NEQV_OP: /* .NEQV. in fortran */
                       printf("left oprnd[%d] right oprnd[%d] ", lhsId(e), rhsId(e));
                      break;
               case MINUS_OP: /* unary operations */
               case NOT_OP:
                       printf("operand   [%d]                 ", lhsId(e));
      break;
               case STAR_RANGE: /* operations with no operands 360.. */
      break;
               case PROC_CALL:
               case FUNC_CALL:
               case CONSTRUCTOR_REF:
               case ACCESS_REF:
               case CONS:
               case ACCESS:
               case IOACCESS:
               case CONTROL_LIST:
               case SEQ:
               case SPEC_PAIR:
               case COMM_LIST:
               case STMT_STR:
               case EQUI_LIST:
               case IMPL_TYPE:
               case STMTFN_DECL:
               case BIT_COMPLEMENT_OP:
               case EXPR_IF:
               case EXPR_IF_BODY:
               case FUNCTION_REF:
               case UNARY_ADD_OP:
               case SIZE_OP:
               case INTEGER_DIV_OP:
               case SUB_OP:
               case NAMELIST_LIST:
               case ORDERED_OP:     /* Following added for PCF FORTRAN */
               case EXTEND_OP:
               case MAXPARALLEL_OP:
               case SAMETYPE_OP:
               case TYPE_REF:     /* Added for FORTRAN 90 */
               case STRUCTURE_CONSTRUCTOR:
               case ARRAY_CONSTRUCTOR:
               case SECTION_REF:
               case VECTOR_SUBSCRIPT:
               case SECTION_OPERANDS:
               case KEYWORD_ARG:
               case OVERLOADED_CALL:
               case INTERFACE_REF:
               case RENAME_NODE:
               case TYPE_NODE:
               case PAREN_OP:
               case PARAMETER_OP:
               case PUBLIC_OP:
               case PRIVATE_OP:
               case ALLOCATABLE_OP:
               case DIMENSION_OP:
               case EXTERNAL_OP:
               case IN_OP:
               case OUT_OP:
               case INOUT_OP:
               case INTRINSIC_OP:
               case POINTER_OP:
               case OPTIONAL_OP:
               case SAVE_OP:
               case TARGET_OP:
               case ONLY_NODE:
               case LEN_OP:
               case TYPE_OP:
               default:
                   break;
        }
    printf(" variant: ");
    printVariantName(e->variant());
    printf(" expr is: ");
    e->unparsestdout();
    printf("\n");
    }
```

```
}
```

Dump Info - classifySymbols

```
void classifySymbols(SgFile *f){
    SgSymbol *s;
    SgVariableSymb *vs;
    SgConstantSymb *cs;
    SgFunctionSymb *fs;
    SgLabelVarSymb *lvs;
    SgExternalSymb *exts;
    SgConstructSymb *cnsts;
    SgInterfaceSymb *ifs;
    SgModuleSymb *mods;
    for(s = f->firstSymbol(); s; s = s->next()){
printf("SYMB[%d]: %s = %s  scope=STMT[%d] type:",
s->id(),tag[s->variant()], s->identifier(),
                (s->scope())?(s->scope())->id(): -1);
if(s->type()) doFullType(s->type());
doSymbAttribs(s);
        printf("\n");
switch(s->variant()){
            case CONST_NAME:
            case VARIABLE_NAME:
            case PROGRAM_NAME:
            case PROCEDURE_NAME:
            case LABEL_VAR: /* dest of assigned goto stmt */
            case FUNCTION_NAME:
            case LABEL_NAME: /* new added for VPC */
            case ROUTINE_NAME: /*added for external statement*/
            case CONSTRUCT_NAME:
            case INTERFACE_NAME:
            case MODULE_NAME:
                       printf(" need work\n");
       break;
            default:
       printf("forgot one\n");
       break;
    }
       }

    }
```

Dump Info - main

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

  SgProject project("x.proj");
  SgFile file("x.f");
  int c;

  initVariantNames();

  switch (file.languageType()) {
  case CSrc:
        printf("this is a c program.  this example is for fortran \n");
        return -1;
  case ForSrc:
        classifyStatements(&file);
        printf("**************************************************\n");
        printf("**** Expression Table ****************************\n");
        printf("**************************************************\n");
        classifyExpressions(&file);
        printf("**************************************************\n");
        printf("**** Symbol  Table *******************************\n");
        printf("**************************************************\n");
        classifySymbols(&file);
  }
  return 0;
}
```

10 Indices

10.1 Index - Sage Classes

S

10.2 Index - Sage Functions

A

B

E

F

G

H

I

L

M

N

O

P

T

10.3 Index - C

10.4 Index - Fortran

>

^

<

A

B

C

D

E

F

G

I

K

L

M

N

O

T

U

V

W

10.5 Index - Examples

B

G

I

L

S

10.6 Index - Concepts

Table of Contents