Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 8512

A programmer's guide for Nestor *Version 1*

Georges-André Silber

6th October 1998

Technical Report N° 1998-04

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

A programmer's guide for Nestor
Version 1

Georges-André Silber

6th October 1998

Abstract

This is a programmer's guide for Nestor, a library to easily manipulate Fortran programs through a high level internal representation based on C++ classes. Nestor is a research tool that can be used to quickly implement source to source transformations. The input of the library is Fortran 77, Fortran 90, and HPF 2.0. Its current output supports the same languages plus some dialects such as Petit, OpenMP, CrayMP. Nestor is light, ready to use (`http://www.ens-lyon.fr/~gsilber/nestor`), fully documented and is well suited for Fortran to Fortran transformations.

Keywords: Library, program transformations, HPF, parallelization, object oriented.

Résumé

Ce rapport est un guide du programmeur pour Nestor, une bibliothèque pour manipuler facilement des programme Fortran à l'aide d'une représentation interne de haut niveau qui se fonde sur des classes C++. Nestor est un outil de recherche qui peut être utilisé pour implanter rapidement des transformation source à source. Les langages reconnus par la librairie sont Fortran 77, Fortran 90 et HPF 2.0. Les langages disponibles en sortie sont les précédents plus des dialectes de Fortran comme Petit, OpenMP, CrayMP, etc. Nestor est léger, prêt à être utilisé (`http://www.ens-lyon.fr/~gsilber/nestor`) et complètement documenté. De plus, Nestor est bien adapté aux transformations source à source de Fortran.

Mots-clés : Bibliothèque, transformation de programmes, HPF, parallélisation, orienté objet.

A programmer's guide for Nestor

Georges-André Silber

6th October 1998

Contents

1 Introduction

Automatic parallelization consists in transforming automatically a sequential program to a parallel program, for instance Fortran 77 or Fortran 90 to High Performance Fortran (HPF). The goal of Nestor is to provide a programming environment to build systems that transform programs. Furthermore, Nestor already integrates a set of algorithms that detect parallelism into loop nests and automatically transform sequential Fortran into HPF. Nestor is our development tool for the implementation of our theoretical results and is integrated as a parallelizing tool into TransTool [1].

Nestor can be used by researchers interested by parallelizing compilers, performance analysis tools or more generally by source code optimizers. There are a lot of libraries transforming source code into an intermediate representation (SUIF [4], PIPS [3]), but this intermediate representation is sometimes at a too low level (difficult to handle) or represents a subset of language. For example, SUIF does not analyze Fortran, but uses *f2c*, a public domain program, to transform Fortran source code into C source code and then analyzes it. Our library suppresses these two defaults. By using the front-end of an excellent HPF compiler from the public domain, Adaptor (written by T. Brandes [2]), we have built a system that gives an intermediate representation of an HPF 2.0 source code under the form of a hierarchical collection of C++ objects. Under this form, the code can be analyzed, transformed and unparsed in HPF 2.0 very easily.

The global structure of Nestor is presented by the Fig.1. Nestor is splitted into three different libraries (node, base, and graph) that are dependent (base needs node and graph needs base) and that provide different functionalities.

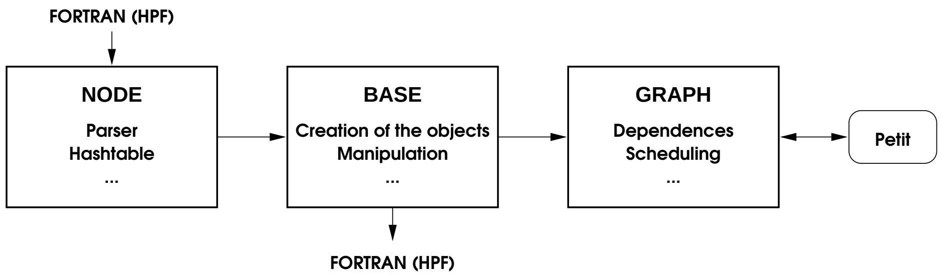

Figure 1: Global organization of Nestor.

This organization is flexible and allows to use only the needed parts of Nestor. For instance, it is possible to build a syntax checker for HPF codes only by using the node library. If a user is not concerned by program dependences, it does not have to link the graph library. This design approach permits to add functionalities to Nestor in an incremental way without rewriting all the underlying bricks. In the following, we present in details the different libraries of Nestor.

2 The node library

This library is a modified version of the front-end of Adaptor that recognizes HPF 2.0. This allows Nestor to handle real codes, instead of only considering a subset of language or an *ad-hoc* language. Furthermore, this approach has the advantage of keeping the semantic of a high level language, instead of translating Fortran to low level C.

The front-end has been extended to handle new directives which allow the user to control the parallelization of its code. For instance, there is a new directive `SINGLE` that can be placed in front of a loop nest that signifies to Nestor that it must treat this loop nest. Other directives allow to define area of code that have to be analyzed. These directives drive the transformations that are done on a source code: completely or semi-automatically.

The parsing phase builds an abstract syntax tree in memory. The node library offers C functions (not documented here) to access and travel around the tree. These functions are used by the base library to build a hierarchical collection of C++ objects representing the source code. They are written in a way such that replacing the parser is an easy task and offer an easy interface for other languages.

For supporting semi-automatic parallelization, it is useful to define parts of code that have to be parallelized and others that are not to be taken into account. The base library offers a mechanism, by the use of new directives, that permits to ignore parts of code that are known to be sequential and to emphase parts that must be parallelized. These directives have the same structure than the HPF directives except that they begin with the keyword `cNESTOR$`.

The first directive (SINGLE) has to be placed in front of a `DO` or a `FORALL` statement:

```
!NESTOR$ SINGLE
    do i = 1,N
      a(i) = c(i-1) + b(i+1)
    enddo
```

This directive set the flag nestor_flag to 1 in the object NstStatementDo or NstStatementForall during the parsing.

The second directive (BEGIN-END) defines a region used to enclose a part of code:

```
!NESTOR$ BEGIN
    if (a(1) .eq. 0) then
      do i = 1,N
        a(i) = c(i-1) + b(i+1)
      enddo
    endif
!NESTOR$ END
```

This creates a new object NstStatementNestor enclosing the statements inside.

The third directive (ALL) has to be placed in a subroutine, a function or a program between the declarations and the statements:

```
% cat exe.f
      PROGRAM tot
      integer NBLOCK, IBLOCK, ILOC
      integer  PPTR(1:100),IBLEN(1:100)
      real XPLUS(1:100), XDPLUS(1:100)
      real X(1:100), XD(1:100), XDD(1:100)
      real  DELTAT
!NESTOR$ ALL
      do IBLOCK = 1, NBLOCK
         ILOC = PPTR(IBLOCK)
         do I = 1, IBLEN(IBLOCK)
            XDPLUS(ILOC+I-1) = XD(ILOC+I-1) + .1*DELTAT * XDD(ILOC+I-1)
            XPLUS(ILOC+I-1) = X(ILOC+I-1) + DELTAT * XDPLUS(ILOC+I-1)
         enddo
      enddo
      end
% nstprs exe.f
      program TOT
      integer*4 NBLOCK
      integer*4 IBLOCK
      integer*4 ILOC
      integer*4 PPTR (1:100)
      integer*4 IBLEN (1:100)
      real*4 XPLUS (1:100)
      real*4 XDPLUS (1:100)
      real*4 X (1:100)
```

```
      real*4 XD (1:100)
      real*4 XDD (1:100)
      real*4 DELTAT
!NESTOR$ BEGIN
      do IBLOCK = 1,NBLOCK
         ILOC = PPTR(IBLOCK)
         do I = 1,IBLEN(IBLOCK)
            XDPLUS(ILOC+I-1) = XD(ILOC+I-1)+.1*DELTAT*XDD(ILOC+I-1)
            XPLUS(ILOC+I-1) = X(ILOC+I-1)+DELTAT*XDPLUS(ILOC+I-1)
         end do
      end do
!NESTOR$ END
      end program TOT
```

The mask `GlobalModeMask` passed to the constructor of the class NstComputationUnit has the same effect.

3 The base library

The base library of Nestor contains the definition of all the classes representing the differents syntactic elements of HPF 2.0, Fortran 90 and Fortran 77. The main purpose of this library is to provide the user an easy framework and programming tool to write source-to-source transformations of Fortran codes. Classes defined in the base library permit to create new object, i.e. new syntactic elements and then, to create a new source code from scratch. Secondly, a special class (NstComputationUnit) is used as an interface with the node library, creating a set of objects representing the parsed source code.

A source code transformed by the base library into C++ objects can be seen as a syntactic tree where each node is an object with methods and attributes representing the syntactic element. These objects can be easily modified, replaced, moved at another place of the hierarchical structure. For instance, a list of statements is represented by a class derived from the standard C++ class list where it is easy to reorganize the order of the statements. Each object representing a syntactic element has its own constructor that allows the programmer to add new objects and then new statements, variables, procedures in the source code. Each object has its own output function, a redefinition of the C++ operator «, permiting an easy unparse of every object or of the whole code. This output function can be configured, allowing to unparse in different languages (now, only HPF and Petit language are supported).

Suppose we want to create a program from scratch, i.e. creating a hierarchical structure of objects representing a source code. For instance, suppose we want to create the structure corresponding to the following code:

```
      PROGRAM simple
      integer n
      parameter (n = 100)
      integer a(1:n)
      integer b(1:n)
      integer c(1:n)
      integer i
      do i = 2,n
         a(i) = b(i)
         c(i) = a(i-1)
      enddo
      end
```

There are two ways of creating such a structure. First, if this source code is in a file named '`simple_code.f`', it is easy to write a program using the base library and the node library to parse the file and create the structure. Here is the C++ code:

```
# include <libnstbase.H>
int main(int ac, char **av)
{
  NstComputationUnit program("simple_code.f");
  cout << program;
}
```

The constructor of NstComputationUnit parses the file with the node library and then creates a hierarchical structure of objects representing the code in the object program. The command:

```
cout << program;
```

unparses the program with the methods of the base library on the standard output. Suppose now that we want to create this code from scratch, i.e we want to build the structure ourselves. Here is the C++ code that creates an object program, an instance of the class NstUnitProgram that contains all the syntactical elements created:

```
# include <libnstbase.H>
int main(int ac, char **av)
{
  // Create identifiers
  NstIdentifier prog_id = "simple";
  NstIdentifier a_id = "a";
  NstIdentifier b_id = "b";
  NstIdentifier c_id = "c";
  NstIdentifier i_id = "i";
  NstIdentifier n_id = "n";

  // Create an empty program
  NstUnitProgram program (prog_id);

  // Declare the parameter
  NstInteger hundred = 100;
  NstDeclarationParameter n_param (program, n_id, nst_integer_type, hundred);
  // Create an access to the parameter n
  NstVariableUsed n_used (*n_param.object());

  // Creates an integer array type (a(1:n))
  NstTypeArray array_type (nst_integer_type);
  NstShape shape (n_used);
  array_type.shapes.push_back(&shape);
  // Declare the variable a,b,c
  NstDeclarationVariable a_var (program, a_id, array_type);
  NstDeclarationVariable b_var (program, b_id, array_type);
  NstDeclarationVariable c_var (program, c_id, array_type);
  // Create an access to the variable a,b,c
  NstVariableUsed a_used (*a_var.object());
  NstVariableUsed b_used (*b_var.object());
  NstVariableUsed c_used (*c_var.object());

  // Declare the variable i
```

```
  NstDeclarationVariable i_var (program, i_id, nst_integer_type);
  // Create accesses to the loop variable i
  NstVariableLoop i_used (*i_var.object());

  // Creates the loop nest
  NstInteger two = 2;
  NstStatementDo i_loop (i_used, two, n_used);
  // Put the loop nest into the program
  program.statements.push_back(&i_loop);

  // First statement (a(i) = b(i))
  // the left part of the assignment
  NstVariableIndexed lvalue1 (a_used);
  lvalue1.indexes.push_back(&i_used);
  // The right part (b(i))
  NstVariableIndexed rvalue1 (b_used);
  rvalue1.indexes.push_back(&i_used);
  // Creates the assignment
  NstSimpleAssign assign1(lvalue1, rvalue1);
  NstStatementBasic bassign1(assign1);
  i_loop.body.push_back(&bassign1);

  // second statement (c(i) = a(i-1))
  // the left part of the assignment
  NstVariableIndexed lvalue2 (c_used);
  lvalue2.indexes.push_back(&i_used);
  // The right part (a(i-1))
  NstVariableIndexed rvalue2 (a_used);
  NstInteger one = 1;
  NstExpressionBinary iminus1 (NST_OPERAND_MINUS, i_used, one);
  rvalue2.indexes.push_back(&iminus1);
  // Creates the assignment
  NstSimpleAssign assign2(lvalue2, rvalue2);
  NstStatementBasic bassign2(assign2);
  i_loop.body.push_back(&bassign2);

  // Unparse the code
  cout << program;
}
```

This code can be found in the nestor directory under the directory `src/doc` and is called `gene_code.cc`. Try compiling this example by typing

```
make bin/gene_code
```

and try to execute it. Now, we have seen how to get a structure representing a source code. Now, we will see how to modify this structure to transform source code. Suppose we want to transform the previous code by distributing the do loop, i.e. we want to obtain:

```
      PROGRAM simple
      integer n
      parameter (n = 100)
      integer a(1:n)
      integer b(1:n)
      integer c(1:n)
```

```
      integer i
!HPF$ INDEPENDENT
      do i = 2,n
         a(i) = b(i)
      enddo
!HPF$ INDEPENDENT
      do i = 2,n
         c(i) = a(i-1)
      enddo
      end
```

to run the two loops in parallel. This is easily done by the following procedure:

```
void distribute_single_loop_with_2_statements(NstStatementList& sl)
{
  NstStatementDo* do_loop;
  NstStatementDo* new_loop;
  NstStatementList::iterator sli;
  for (sli = sl.begin(); sli != sl.end(); sli++)
    {
      do_loop = (*sli)->isNstStatementDo();
      if (do_loop)
        {
          if (do_loop->body.size() == 2)
            {
              new_loop = (NstStatementDo*)do_loop->clone();
              do_loop->independent = TRUE;
              new_loop->independent = TRUE;
              do_loop->body.pop_back();
              new_loop->body.pop_front();
              sli++;
              sl.insert(sli, new_loop);
              sli--;
            }
        }
    }
}
```

This procedure examines all the list passed as parameter and for each do loop with two statements, splits the loop in two loops with one statement in each loop. Furthermore, it declares the loop as being INDEPENDENT. Note that we do not test if the loop is actually independent.

3.1 The main class

---

class **NstTree**

---

virtual int **type**() const
virtual const char* **class_name**() const
virtual const char* **class_type**() const

virtual void **error**(const char* st) const
virtual void **warning**(const char* st) const
class **Warning** {}

unsigned long **tag**() const
unsigned long **tag**(unsigned long t) const

virtual NstTree* **init**()
virtual NstTree* **next**()
void **traversal**(int (*action)(NstTree*, void*), void* extra = NULL)

friend ostream& **operator**«(ostream& s, const NstTree& t)

virtual NstStatement* **isNstStatement**()
virtual NstDeclaration* **isNstDeclaration**()
virtual NstUnit* **isNstUnit**()
virtual NstSimple* **isNstSimple**()
virtual NstIdentifier* **isNstIdentifier**()
virtual NstParameter* **isNstParameter**()
virtual NstExpression* **isNstExpression**()
virtual NstProcessor* **isNstProcessor**()
virtual NstDistSpec* **isNstDistSpec**()
virtual NstDistFormat* **isNstDistFormat**()
virtual NstAlignSpec* **isNstAlignSpec**()
virtual NstVarDesc* **isNstVarDesc**()
virtual NstType* **isNstType**()
virtual NstShape* **isNstShape**()
virtual NstObject* **isNstObject**()
virtual NstComputationUnit* **isNstComputationUnit**()
virtual NstTree* **isNstList**()

This class is the main class of the `base` library. All the other classes of the library inherit (directly or undirectly) this class. It provides virtual methods to get the type of a derived object and some error methods. See appendix A for all derived types. Each terminal child class redefines the virtual method type(). It permits to know the actual type of an object, even if we only know its NstTree part (or another ancestor of the class). Here is an example, with the NstStatementContinue class that is defined further:

```
NstStatementContinue test_object;
NstTree* tree = (NstTree*)&test_object;
int rt = tree->type();
```

The variable `rt` will contain NST_STATEMENT_CONTINUE, because the object `tree` is in fact an instance of the class NstStatementContinue.

- virtual const char* **class_name**() const
  Returns the string corresponding to the class of the object.
- virtual const char* **class_type**() const
  Returns the string corresponding to the type of the object.
- virtual int **type**() const
  Returns the type NST_NOTHING.
- virtual void **error**(const char* st) const
  Prints on the standard error stream a message of the form:

  ```
  (Nestor error: NST_STATEMENT_CONTINUE: st)
  ```

  and exits the processus with the return code EXIT_FAILURE.
- virtual void **warning**(const char* st) const
  Prints on the standard error stream a message of the form:

  ```
  (Nestor warning: NST_STATEMENT_CONTINUE: st)
  ```

  and throws the exception Warning.
- unsigned long **tag**() const
  Returns the tag (an integer) stored in the object.
- unsigned long **tag**(unsigned long t) const
  Replaces the current tag with t and returns the old one.
- virtual NstTree* **init**()
  Init the list of components of the object. Returns the first component of the list. Returns NULL if the list has no components.
- virtual NstTree* **next**()
  Returns the next components in the list of components of the object. A call to init() has to be made before any call to this method. Returns NULL when the list is empty.
- void **traversal**(int (*action)(NstTree*, void*), void* extra = NULL)
  Recursive traversal of the AST. This method traverses the tree and calls the function action with the current node as parameter and the pointer extra. If the function action returns a value different from 0, the recursion stops.
- friend ostream& **operator«**(ostream& s, const NstTree& t)
  This operator unparses an object derived from the NstTree class on a stream. It calls private virtual methods that are redefined for each child object for each defined unparse language. This operator can be globally configured for a language by using the function:

  NstUnparseMode **nst_unparse_mode**(NstUnparseMode m)

  that returns the old unparse mode and where m is the new unparse mode. The unparse mode can be choosen between these ones:

UnparseFortran	Fortran/HPF 2.0 (default)
UnparsePetit	Petit mode

  Note that there is another function:

  NstUnparseMode **nst_unparse_mode**()

that gives the current unparse mode. When the unparse mode if UnparseFortran, the user can select the Fortran dialect that is unparsed with the function:

NstFortranType **nst_fortran_type**(NstFortranType t)

where t can be choosed between the following types:

HPF	HPF2.0 (default)
SunMP	SunMP (F77)
CrayMP	CrayMP (F90)
OpenMP	OpenMP

Note that there is another function:

NstFortranType **nst_fortran_type**()

that gives the current Fortran dialect.

- virtual NstStatement* **isNstStatement**()
  Returns NULL.
- virtual NstDeclaration* **isNstDeclaration**()
  Returns NULL.
- virtual NstUnit* **isNstUnit**()
  Returns NULL.
- virtual NstSimple* **isNstSimple**()
  Returns NULL.
- virtual NstIdentifier* **isNstIdentifier**()
  Returns NULL.
- virtual NstParameter* **isNstParameter**()
  Returns NULL.
- virtual NstExpression* **isNstExpression**()
  Returns NULL.
- virtual NstProcessor* **isNstProcessor**()
  Returns NULL.
- virtual NstDistSpec* **isNstDistSpec**()
  Returns NULL.
- virtual NstDistFormat* **isNstDistFormat**()
  Returns NULL.
- virtual NstAlignSpec* **isNstAlignSpec**()
  Returns NULL.
- virtual NstVarDesc* **isNstVarDesc**()
  Returns NULL.
- virtual NstType* **isNstType**()
  Returns NULL.
- virtual NstShape* **isNstShape**()
  Returns NULL.

- virtual NstObject* **isNstObject()**
  Returns NULL.

- virtual NstComputationUnit* **isNstComputationUnit()**
  Returns NULL.

- virtual NstTree* **isNstList()**
  Returns NULL.

Here are some examples to illustrate the use of the NstTree class. First, an example that plays with tags:

```
% cat test_tags.cc
# include <libnstbase.H>
void test_tags(NstTree& tree)
{
  unsigned long ot;
  cout << "current tag: " << tree.tag() << endl;
  ot = tree.tag(12UL);
  cout << "new tag    : " << tree.tag() << endl;
  cout << "old tag    : " << ot << endl;
}
% test_tags
current tag: 0
new tag    : 12
old tag    : 0
```

Second, an example to illustrate error messages:

```
% cat test_error.cc
# include <libnstbase.H>
void test_error(NstTree& tree)
{
  try {
    tree.warning("throwing exception..");
  } catch (NstTree::Warning) {
    cerr << "catched." << endl;
  }
  tree.error("finished");
}
void main()
{
  NstIdentifier toto;
  test_error(toto);
}
% test_error
(Nestor warning: NST_IDENTIFIER: throwing exception...)
catched.
(Nestor error: NST_IDENTIFIER: finished.)
```

Finally, an example to illustrate the unparse:

```
% cat test_unparse.cc
# include <libnstbase.H>
void main()
{
```

```
  NstSTatementContinue cont;
  cont.label = 100;
  NstInteger entier = 45;
  cout << "The integer has the value: " << entier << endl;
  cout << cont << endl;
}
% test_unparse
The integer has the value: 45
  100 CONTINUE
```

The next example is recursive traversal of an AST that prints the type of each object encountered. The method class_type() is defined for each class and returns a string containing the type of the class. This example illustrates the use of the init() and next() methods. These methods are defined for each class, even for classes representing lists. Consequently, they can be used to traverse a list instead of using an object `iterator`.

```
void print_type(NstTree* t)
{
  cout << t->class_type() << endl;
  NstTree* current = t->init();
  while (current) {
    print_type(current);
    current = t->next();
  }
}
```

3.2 Identifiers

class **NstIdentifier** : public **NstTree**

int **type**() const
NstIdentifier* **isNstIdentifier**()

NstIdentifier()
NstIdentifier(const string& s)
NstIdentifier(const char* s)

const string& **id_string**() const
string **id_string**(const string& s)
string **id_string**(const char* s)
int **is_empty**()const

friend int **operator==**(const NstIdentifier& a, const NstIdentifier& b)
friend int **operator==**(const NstIdentifier& a, const char* b)
friend int **operator==**(const char* a, const NstIdentifier& b)
friend int **operator==**(const NstIdentifier& a, const string& b)
friend int **operator==**(const string& a, const NstIdentifier& b)

This class permits to manipulate identifiers used to represent variables and various other names that resides in a piece of code. This class offers a reliable mechanism to create unique identifiers.

- int **type**() const
  Returns the type NST_IDENTIFIER.

- NstIdentifier* **isNstIdentifier**()
  Returns this.

- **NstIdentifier**()
  Creates a new instance of this class containing an unique identifier of the form `nst_`d where d is choosen to ensure that the identifier is unique (no other identifier with the same name has been created by a constructor of the class NstIdentifier before this call).

- **NstIdentifier**(const string& s)
  Creates a new instance of this class containing s.

- **NstIdentifier**(const char* s)
  Creates a new instance of this class containing s.

- const string& **id_string**() const
  Returns the string contained in the object.

- string **id_string**(const string& s)
  Replaces the current string by s and returns the old one.

- string **id_string**(const char* s)
  Replaces the current string by s and returns the old one.

- int **is_empty**()const
  Returns 1 if the identifier is empty, 0 elsewhere.

- friend int **operator==()**
  Checks the equality between two identifiers or strings (case insensitive).

  Here is an example of the use of this class:

```
% cat test_identifier.cc
# include <libnstbase.H>
void main()
{
  NstIdentifier i1;
  NstIdentifier i2("hello");
  string st("hello2");
  string st2("Hello2");
  NstIdentifier i3(st);
  NstIdentifier i6(st2);
  NstIdentifier i4("NST_1");
  NstIdentifier i5;

  cout << i1 << endl;
  cout << i2 << endl;
  cout << i3 << " " << i3.number() << endl;
  cout << i6 << " " << i6.number() << endl;
  cout << i4 << endl;
  cout << i5 << endl;
}
% test_identifier
nst_0
hello
hello2 4
Hello2 5
NST_1
nst_1
```

  Another example to illustrate the method id_string():

```
% cat test_identifier2.cc
# include <libnstbase.H>
void main()
{
  NstIdentifier i1("hello");
  NstIdentifier i2 = i1;
  string str;

  str = i1.id_string("world.");
  cout << str << ", " << i1.id_string() << endl;
  (void)i2.id_string("");
  cout << "i2 is " << (i2.is_empty()?"empty":"not empty") << endl;
}
% test_identifier2
hello, world.
i2 is empty
```

  Finally, an example to illustrate the use of the operator =:

```
% cat test_identifier3.cc
# include <libnstbase.H>
```

```
void main()
{
  NstIdentifier i1("hello");
  NstIdentifier i2("hillo");
  string str("hullo");

  cout << "i1 and i2 are " << (i1 == i2?"equal":"not equal") << endl;
  cout << "i1 and hello are " << (i1 == "hello"?"equal":"not equal") << endl;
  cout << "i2 and str are " << (i2 == str?"equal":"not equal") << endl;
}
% test_identifier3
i1 and i2 are not equal
i1 and hello are equal
i2 and str are not equal
```

3.3 Unit of computation: parsing a source code

---

class **NstComputationUnit** : public **NstTree**

---

int **type**() const
NstComputationUnit* **isNstComputationUnit**()

NstComputationUnit()
NstComputationUnit(const char* file_name, unsigned long options = NoOptionsMask)

NstUnitList **units**
NstSymbolTable **intrinsics**
NstSymbolTable **externals**

This class is used to parse a file containing source code and to transform it into a hierarchical bunch of objects. An empty instance of the NstComputationUnit can also be created. The language supported is HPF 2.0. Here is an example of a simple program that parses a file named `toto.f`, and after the creation of the objects corresponding to the syntactical elements of the code, unparses it to the standard output:

```
NstComputationUnit program("toto.f");
cout << program;
```

Note that any source code transformation can be inserted between these two lines of code. These two lines of code are a simple syntactical and semantical analyzer.

There is a global variable unparse_stab which, when set to 1, affects the unparsing of the units: the symbol table is unparsed as a comment before the declarations in Fortran mode.

- int **type**() const
  Returns the type NST_COMPUTATION_UNIT.
- NstComputationUnit* **isNstComputationUnit**()
  Returns this.
- **NstComputationUnit**()
  Creates a new empty instance (object) of this class.
- **NstComputationUnit**(const char* file_name, unsigned long options = NoOptionsMask)
  Creates a new instance of this class. Parses the file file_name according to options, an optional argument that can be set with the flags:

NoOptionsMask	No options
DebugUnparse	Unparse by the node library
FreeCodeMask	Parse free format code (Fortran 90)
GlobalModeMask	Tag all the `DO` and `FORALL` loops (see 2)
VerboseMask	Verbose mode

  that can be combined with the operator |. By default, the file is parsed with the Fortran 77 format. This constructor uses the library `node` to check the correctness of the code and to build a syntactic tree. From this syntactic tree, all the objects corresponding to the syntactical elements of the code are created, starting from the list of the units (functions, subroutines) contained in the source code.
- NstUnitList **units**
  The list of the units of the source code.

- **NstSymbolTable intrinsics**
  The list of the intrinsics defined by the `node` library. This list is empty if the object has been created from scratch.

- **NstSymbolTable externals**
  The list of the externals found in the source code.

Here is an example that gives the number of units in the source code and their names:

```
% cat test_compunit.cc
# include <libnstbase.H>
int main(int ac, char **av)
{
  if (ac < 2) return -1;
  NstComputationUnit program(av[1], FreeCodeMask);
  cout << "This source code has ";
  cout << program.units.size() << " units:" << endl;
  NstUnitList::iterator i;
  for (i = program.units.begin(); i != program.units.end(); i++)
    cout << *(*i)->name() << endl;
}
% cat units.f
      subroutine toto ()
      real a
      a = 1
      end subroutine toto

      program main
      call toto()
      end program main
% test_compunit toto.f
This source code has 2 units:
toto
main
```

3.4 Units: functions, subroutines, etc.

---

class **NstUnit** : public **NstTree**

NstUnit* **isNstUnit**()
virtual NstUnitProgram* **isNstUnitProgram**()
virtual NstUnitSubroutine* **isNstUnitSubroutine**()
virtual NstUnitModule* **isNstUnitModule**()
virtual NstUnitFunction* **isNstUnitFunction**()
virtual NstUnitBlockData ***isNstUnitBlockData**()

int **line**
NstIdentifier* **name**() const
const NstDeclarationList& **declarations**() const
const NstSymbolTable& **table_of_symbols**() const
const NstUnitList& **internals**() const
NstObjectBaseProcedure* **object**() const

NstUnitList* **in_list**() const
void **attach_to_list**(NstUnitList& ul)
void **remove_from_list**()

An unit is a program, a function, a subroutine, a module or a block data. An unit contains declarations, internals (Fortran 90) and statements (depending on the type of unit!). The class NstUnit is the base class for unit classes. It contains informations common to all types of unit.

The two methods attach_to_list() and remove_from_list() are important in this class. They permit to attach or remove an object from a list (the list returned by the method in_list()). An unit can be in different lists, but can be attached to only one list.

- NstUnit* **isNstUnit**()
  Returns this.

- virtual NstUnitProgram* **isNstUnitProgram**()
  Returns NULL.

- virtual NstUnitSubroutine* **isNstUnitSubroutine**()
  Returns NULL.

- virtual NstUnitModule* **isNstUnitModule**()
  Returns NULL.

- virtual NstUnitFunction* **isNstUnitFunction**()
  Returns NULL.

- virtual NstUnitBlockData ***isNstUnitBlockData**()
  Returns NULL.

- int **line**
  Stores the line number in the source code where the unit begin (assigned at parse time).

- NstIdentifier* **name**() const
  Returns the name of the unit. This name is not stored in this object but in the NstObjectBaseProcedure object associated with it.

- const NstDeclarationList& **declarations()** const
  Returns the list of the declarations of the unit. For declaring a new variable into an unit, see the class NstDeclaration.

- const NstSymbolTable& **table_of_symbols()** const
  Returns the table of symbols of the unit.

- const NstUnitList& **internals()** const
  Returns the list of the internals (functions or subroutines) of the unit. For declaring a new internal, see the classes NstUnitSubroutine and NstUnitFunction.

- NstObjectBaseProcedure* **object()** const
  Returns the object associated with the unit. This object stores the symbol table and the name of the unit.

- NstUnitList* **in_list()** const
  This method returns a pointer to the list the unit is attached to. This method returns NULL if the unit is not attached to a list. See the method below for attaching an unit to a list. All the NstUnit objects created by the NstComputationUnit class are attached to their list, but a newly created object is not attached to a list.

- void **attach_to_list**(NstUnitList& ul)
  Attaches the unit to the list ul. This method will return an error if the unit is already in a list. Note that this method does not put the unit into the list, it just tells to the unit that it is in the list, so be careful! Example:

```
NstComputationUnit cunit;
NstUnitProgram prog("toto");
NstUnitList dummy;
cunit.units.push_back(&prog);
dummy.push_back(&prog);
cout << prog.in_list(); // Return NULL
prog.attach_to_list(cunit.units);
cout << prog.in_list(); // Return &cunit.units
```

- void **remove_from_list()**
  Removes an unit from a list. Note that this method does not remove actually the unit from the list it is in, it just tells to the unit that it is not in the list anymore, so be careful! Example:

```
cunit.units.pop_back();
cout << prog.in_list(); // Return &cunit.units
prog.remove_from_list();
cout << prog.in_list(); // Return NULL
// prog is still in the list dummy
```

---

class **NstUnitModule** : public **NstUnit**

---

int **type()** const
NstUnitModule* **isNstUnitModule()**

NstUnitModule(NstIdentifier& id)

A module (Fortran 90) is an unit of compilation that contains variable declarations, new types, subroutines, functions.

- int **type**() const
  Returns the type NST_UNIT_MODULE.
- NstUnitModule* **isNstUnitModule**()
  Return this.
- **NstUnitModule**(NstIdentifier& id)
  Creates a new instance of this class, a module named id. A new NstObjectBaseProcedure object is created and associated to it.

---

class **NstUnitBlockData** : public **NstUnit**

---

int **type**() const
NstUnitBlockData* **isNstUnitBlockData**()

NstUnitBlockData(NstIdentifier& id)

Corresponds to a block data program unit. The use of modules should makes the use of block data program units unnecessary.

- int **type**() const
  Returns the type NST_UNIT_BLOCK_DATA.
- NstUnitBlockData* **isNstUnitBlockData***()
  Return this.
- **NstUnitBlockData**(NstIdentifier& id)
  Creates a new instance of this class, a block data named id. A new NstObjectBaseProcedure object is created and associated to it.

---

class **NstUnitProgram** : public **NstUnit**

---

int **type**() const
NstUnitProgram* **isNstUnitProgram**()

NstUnitProgram (NstIdentifier& id)
NstStatementList **statements**

This class permits to represent fortran programs.

- int **type**() const
  Returns the type NST_UNIT_PROGRAM.
- NstUnitProgram* **isNstUnitProgram**()
  Return this.
- **NstUnitProgram**(NstIdentifier& id)
  Creates a new instance of this class, a program named id. A new NstObjectBaseProcedure object is created and associated to it.
- NstStatementList **statements**
  This attribute contains the list of statements of the program. The list representation allows to easily add, remove, or reorder statements.

---

class **NstUnitSubroutine** : public **NstUnit**

---

int **type**() const
NstUnitSubroutine* **isNstUnitSubroutine***()

NstUnitSubroutine(NstIdentifier& id)
NstUnitSubroutine(NstIdentifier& id, NstUnit& obp)

const NstDeclarationList& **formals**()const
NstStatementList **statements**
bool **is_recursive**
bool **is_pure**

This class permits to represent fortran subroutines.

- int **type**() const
  Returns the type NST_UNIT_SUBROUTINE.

- NstUnitSubroutine* **isNstUnitSubroutine**()
  Return this.

- **NstUnitSubroutine**(NstIdentifier& id)
  Creates a new instance of this class, a subroutine named id. A new NstObjectBaseProcedure object is created and associated to it.

- **NstUnitSubroutine**(NstIdentifier& id, NstUnit& obp)
  Creates a new instance of this class, a subroutine named id. A new NstObjectBaseProcedure object is created and associated to it. The subroutine is added to the internals of the unit obp and declared in its symbol table.

- const NstDeclarationList& **formals**()const
  Returns the formal parameters of the subroutine. See the class NstDeclarationVarParam for declaring a new parameter.

- NstStatementList **statements**
  This attribute contains the list of statements of the subroutine. The list representation allows to easily add, remove, or reorder statements.

- bool **is_recursive**
  Tells if the subroutine is recursive or not.

- bool **is_pure**
  Tells if the subroutine is pure or not.

---

class **NstUnitFunction** : public **NstUnit**

---

int **type**() const
NstUnitFunction* **isNstUnitFunction***()

NstUnitFunction(NstIdentifier& id, NstType& tp)
NstUnitFunction(NstIdentifier& id, NstType& tp, NstUnit& obp)
const NstDeclarationList& **formals**()const
NstStatementList **statements**
int **is_recursive**
int **is_pure**
NstVariableUsed* **result_id**() const
NstType* **function_type**() const

This class represents fortran functions.

- int **type**() const
  Returns the type NST_UNIT_FUNCTION.

- NstUnitFunction* **isNstUnitFunction**()
  Return this.

- **NstUnitFunction**(NstIdentifier& id, NstType& tp)
  Creates a new instance of this class, a function of type tp named id. A new NstObjectBaseProcedure object is created and associated to it. A new variable is created in the function that is the return result of the function. This new variable is only included in the symbol table of the function and has the type of the function. The name of this variable is id_result_d, where d is an unique number.

- **NstUnitFunction**(NstIdentifier& id, NstType& tp, NstUnit& obp)
  Creates a new instance of this class, a function of type tp named id. A new NstObjectBaseProcedure object is created and associated to it. The function is added to the internals of the unit obp and declared in its symbol table. A new variable is created in the function that is the return result of the function. This new variable is only included in the symbol table of the function and has the type of the function. The name of this variable is id_result_d, where d is an unique number.

- const NstDeclarationList& **formals**()const
  Returns the formal parameters of the function. See the class NstDeclarationVarParam for declaring a new parameter.

- NstStatementList **statements**
  This attribute contains the list of statements of the function. The list representation allows to easily add, remove, or reorder statements.

- bool **is_recursive**
  Tells if the function is recursive or not.

- bool **is_pure**
  Tells if the function is pure or not.

- NstVariableUsed* **result_id**() const
  Returns the variable used for returning the result of the function.

- NstType* **function_type**() const
  Returns the type of the function.

This example creates a new real function that returns 3:

```
% cat test_fct.cc
# include <libnstbase.H>
int main(int ac, char **av)
{
  NstIdentifier tutu ("toto");
  NstInteger three = 3;
  // New function
  NstUnitFunction stoto(tutu, nst_real_type);
  // New statement
  NstSimpleAssign sa(*stoto.result_id(), three);
  NstStatementBasic sb(sa);
  stoto.statements.push_back(&sb);
  cout << stoto;
}
% test_fct
      real*4 function toto() result (toto_result_0)
      toto_result_0 = 3
      end function TOTO
```

This example replaces the name of the first internal of a program by `new_id`:

```
% cat test_nameunit.cc
# include <libnstbase.H>
int main(int ac, char **av)
{
  if (ac < 2) return -1;
  NstComputationUnit program(av[1]);
  NstUnit* first_unit = program.units.front();
  NstUnitProgram* prg = first_unit->isNstUnitProgram();
  if (!prg) return -1;
  NstUnit* unit = prg->internals().front();
  NstIdentifier* unit_name = unit->name();
  unit_name->id_string("new_id");
  cout << program;
}
% cat units.f
      program main
      call toto()
      contains
      subroutine toto ()
      a = 3
      end subroutine toto
      end program main
% test_nameunit units.f
      program main
      call new_id()
      contains
      subroutine new_id ()
      a = 3
      end subroutine new_id
      end program main
```

This exemple prints the declarations of a program:

```
% cat test_declarations.cc
```

```
# include <libnstbase.H>
int main(int ac, char **av)
{
  if (ac < 2) return -1;
  NstComputationUnit program(av[1]);
  NstUnit* first_unit = program.units.front();
  cout << first_unit->declarations();
}
% cat exe.f
      PROGRAM tot
      integer NBLOCK, IBLOCK, ILOC
      integer  PPTR(1:100),IBLEN(1:100)
      real XPLUS(1:100), XDPLUS(1:100)
      real X(1:100), XD(1:100), XDD(1:100)
      real  DELTAT
      integer big_text (1:11,1:23,1:34)
      do IBLOCK = 1, NBLOCK
         ILOC = PPTR(IBLOCK)
         do I = 1, IBLEN(IBLOCK)
            XDPLUS(ILOC+I-1) = XD(ILOC+I-1) + .1*DELTAT * XDD(ILOC+I-1)
            XPLUS(ILOC+I-1) = X(ILOC+I-1) + DELTAT * XDPLUS(ILOC+I-1)
         enddo
      enddo
      end
% test_declarations exe.f
      integer*4 NBLOCK
      integer*4 IBLOCK
      integer*4 ILOC
      integer*4 PPTR (1:100)
      integer*4 IBLEN (1:100)
      real*4 XPLUS (1:100)
      real*4 XDPLUS (1:100)
      real*4 X (1:100)
      real*4 XD (1:100)
      real*4 XDD (1:100)
      real*4 DELTAT
      integer*4 BIG_TEXT (1:11,1:23,1:34)
```

This example creates a new subroutine and declares three arrays into it:

```
% cat test_sub.cc
# include <libnstbase.H>
int main(int ac, char **av)
{
  // Identifiers
  NstIdentifier s_id ("dummy");
  NstIdentifier a_id = "a";
  NstIdentifier b_id = "b";
  NstIdentifier c_id = "c";
  // New Shape
  NstShape shape (100);
  // New array type
  NstTypeArray ar_type (nst_real_type);
  ar_type.shapes.push_back(&shape);
```

```
  ar_type.shapes.push_back(&shape);
  // New subroutine
  NstUnitSubroutine subr(s_id);
  // Array declaration
  NstDeclarationVariable a_decl(subr, a_id, ar_type);
  NstDeclarationVariable b_decl(subr, b_id, ar_type);
  NstDeclarationVariable c_decl(subr, c_id, ar_type);
  // Unparsing
  cout << subr;
}
% test_sub
      subroutine dummy()
      real*4 a (1:100,1:100)
      real*4 b (1:100,1:100)
      real*4 c (1:100,1:100)
      end subroutine dummy
```

class **NstUnitList** : public **list<NstUnit*>**, public **NstTree**

NstTree* **isNstList**()
int **type**() const

NstUnitList()

NstUnit* **search**(NstIdentifier& id) const
NstUnit* **rsearch**(NstIdentifier& id) const
NstUnit* **search**(const char* id) const
NstComputationUnit* **in_computation_unit**() const
NstUnit* **in_unit**() const

void **copy**(const NstUnitList& s)

This class represents a list of units (an unit is a function, a program, a subroutine, a module, etc...). The class list is the doubly linked list of the Standard Template Library (STL).

- NstTree* **isNstList**()
  Returns this.
- int **type**() const
  Returns the type NST_UNIT_LIST.
- **NstUnitList**()
  Creates a new instance of this class, an empty list.
- NstUnit* **search**(NstIdentifier& id) const
  Searchs in the list the unit with the name id. Returns NULL if the unit is not found.
- NstUnit* **rsearch**(NstIdentifier& id) const
  Searchs recursively in the list the unit with the name id. Returns NULL if the unit is not found.
- NstUnit* **search**(const char* id) const
  Searchs in the list the unit with the name id. Returns NULL if the unit is not found.
- NstComputationUnit* **in_computation_unit**() const
  Returns the NstComputationUnit object if the list is an attribute of it. Returns NULL elsewhere.

- NstUnit* **in_unit**() const
  Returns the NstUnit object the list is in. Returns NULL if the list is not in an unit.

- void **copy**(const NstUnitList& s)
  Copies all units of the list s into this list. Each unit of the list s is attached to the current list: the units of s must not be attached to any list.

3.5 Declarations

---

class **NstDeclaration** : public **NstTree**

NstDeclaration* **isNstDeclaration**()
virtual NstDeclarationVariable* **isNstDeclarationVariable**()
virtual NstDeclarationTemplate* **isNstDeclarationTemplate**()
virtual NstDeclarationProcessors* **isNstDeclarationProcessors**()
virtual NstDeclarationDistribute* **isNstDeclarationDistribute**()
virtual NstDeclarationAlign* **isNstDeclarationAlign**()
virtual NstDeclarationImplicit* **isNstDeclarationImplicit**()
virtual NstDeclarationFormat* **isNstDeclarationFormat**()
virtual NstDeclarationDynamic* **isNstDeclarationDynamic**()
virtual NstDeclarationVarParam* **isNstDeclarationVarParam**()
virtual NstDeclarationExternal* **isNstDeclarationExternal**()
virtual NstDeclarationIntrinsic* **isNstDeclarationIntrinsic**()
NstIdentifier* **name**() const
NstObject* **object**() const
int **line**

This class is the base class for declarations and contains common information for all declarations. Each declaration is associated with a **Nstobject** object that contains information about the entity declared.

- NstDeclaration* **isNstDeclaration**()
  Returns this.
- virtual NstDeclarationVariable* **isNstDeclarationVariable**()
  Return NULL.
- virtual NstDeclarationTemplate* **isNstDeclarationTemplate**()
  Return NULL.
- virtual NstDeclarationProcessors* **isNstDeclarationProcessors**()
  Return NULL.
- virtual NstDeclarationDistribute* **isNstDeclarationDistribute**()
  Return NULL.
- virtual NstDeclarationAlign* **isNstDeclarationAlign**()
  Return NULL.
- virtual NstDeclarationImplicit* **isNstDeclarationImplicit**()
  Return NULL.
- virtual NstDeclarationFormat* **isNstDeclarationFormat**()
  Return NULL.
- virtual NstDeclarationDynamic* **isNstDeclarationDynamic**()
  Return NULL.
- virtual NstDeclarationVarParam* **isNstDeclarationVarParam**()
  Return NULL.

- virtual NstDeclarationExternal* **isNstDeclarationExternal**()
  Return NULL.

- virtual NstDeclarationIntrinsic* **isNstDeclarationIntrinsic**()
  Return NULL.

- NstIdentifier* **name**() const
  Returns the name of the entity declared. The identifier is not stored in the object but in the NstObject associated with the declaration. This identifier can be null if the declaration has no name.

- NstObject* **object**() const
  Return the NstObject object associated with the declaration. This method can return NULL if there is no object created with this kind of declaration. For instance, this field will return NULL in the case of a NstDeclarationImplicit.

- int **line**
  Store the line number of a parsed declaration.

---

class **NstDeclarationVariable** : public **NstDeclaration**

---

int **type**() const
NstDeclarationVariable* **isNstDeclarationVariable**()

NstDeclarationVariable(NstUnit& bm, NstIdentifier& i, NstType& t)

NstType* **declaration_type**() const
NstType* **declaration_type**(NstType& new_type)
NstObjectVariable* **object**() const

This class permits to declare a new variable inside an unit. It creates a new object in the table of symbols of the unit and insert the declaration into the declaration list.

- int **type**() const
  Returns the type NST_DECLARATION_VARIABLE.

- NstDeclarationVariable* **isNstDeclarationVariable**()
  Returns this.

- **NstDeclarationVariable**(NstUnit& bm, NstIdentifier& i, NstType& t)
  Creates a new instance of this class. Declares a new variable of type t with identifier i into the unit bm. This object is added to the list of declarations of the unit bm. A new NstObjectVariable object is created and stored in the symbol table of the unit bm. If there is already an entity declared in the unit bm with the identifier i, an error is generated.

- NstType* **declaration_type**() const
  Returns the type of the variable declared.

- NstType* **declaration_type**(NstType& new_type)
  Replaces the type of the variable declared by new_type and returns the old one.

- NstObjectVariable* **object**() const
  Returns the NstObjectVariable object associated to the variable. This is a cast of the object returned by the method object() of the class NstDeclaration.

This example creates two variables in an unit:

```
% cat test_dv.cc
# include <libnstbase.H>
int main()
{
  NstIdentifier a ("a");
  NstIdentifier b ("b");
  NstIdentifier subr ("subr");
  NstUnitSubroutine subroutine(subr);
  NstDeclarationVariable av(subroutine, a, nst_integer_type);
  NstDeclarationVariable bv(subroutine, b, nst_complex_type);
  cout << "These two declarations:\n";
  cout << av << endl;
  cout << bv << endl;
  cout << "Are declared in subroutine " << *subroutine.name() << ":" << endl;
  cout << subroutine;
}
% test_dv
hese two declarations:
      integer*4 a
      complex*8 b
Are declared in subroutine subr:
      subroutine subr()
      integer*4 a
      complex*8 b
      end subroutine subr
```

---

class **NstDeclarationImplicit** : public **NstDeclaration**

---

int **type**() const
NstDeclarationImplicit* **isNstDeclarationImplicit**()

NstDeclarationImplicit(NstUnit& bm)

NstType* **implicit_type**() const
NstType* **implicit_type**(NstType& new_type)
char **first_letter**
char **last_letter**

This class permits to defines how the variables must be implicitly typed (`IMPLICIT` declaration). It inserts the declaration into the beginning of the declaration list of the unit.

- int **type**() const
  Returns the type NST_DECLARATION_IMPLICIT.

- NstDeclarationImplicit* **isNstDeclarationImplicit**()
  Returns this.

- **NstDeclarationImplicit**(NstUnit& bm)
  Creates a new instance of this class. Creates an `IMPLICIT NONE` declaration into the unit bm. The method NstObject* **object()** will return `NULL` because no definition is associated with this declaration.

- NstType* **implicit_type()** const
  Returns the type of the implicit declaration.

- NstType* **implicit_type**(NstType& new_type)
  Replaces the type of the implicit by new_type and returns the old one.

- char **first_letter**
  The first letter of the range defined in the implicit declaration.

- char **last_letter**
  The last letter of the range defined in the implicit declaration. The variables that have their first letter between first_letter and last_letter will have the implicit type defined.

This example creates an `implicit none` declaration in an unit:

```
% cat test_impl.cc
# include <libnstbase.H>
int main()
{
  NstIdentifier subr ("subr");
  NstUnitSubroutine subroutine(subr);
  NstDeclarationImplicit impl(subroutine);
  cout << subroutine;
}
% test_impl
      subroutine subr()
      implicit none
      end subroutine subr
```

---

class **NstDeclarationFormat** : public **NstDeclaration**

---

int **type()** const
NstDeclarationFormat* **isNstDeclarationFormat()**

NstParameterList **formats**
int **label**

This class permits to declare a `FORMAT` pattern. This instruction is considered as obsolete in Fortran 90. It specifies a format with a label that can be used in I/O operations.

- int **type()** const
  Returns NST_DECLARATION_FORMAT.

- NstDeclarationFormat* **isNstDeclarationFormat()**
  Return this.

- NstParameterList **formats**
  The list of formats.

- int **label**
  The label associated with the format that can be used in an IO operation.

---

class **NstDeclarationVarParam** : public **NstDeclaration**

---

int **type**() const
NstDeclarationFormat* **isNstDeclarationFormat**()

NstDeclarationVarParam(NstUnit& bm, NstIdentifier& i, NstType& t)

NstObjectVariable* **object**() const

This class permits the declaration of a formal parameter of a function or subroutine.

- int **type**() const
  Returns the type NST_DECLARATION_VAR_PARAM.

- NstDeclarationVarParam* **isNstDeclarationVarParam**()
  Returns this.

- **NstDeclarationVarParam**(NstUnit& bm, NstIdentifier& i, NstType& t)
  Creates a new instance of this class. Declares a new variable of type t with identifier i into the unit bm. This object is added to the list of declarations of the unit bm and the identifier is added to the list of the formal parameters of the unit. The unit must be a NstUnitFunction object or a NstUnitSubroutine object. A new NstObjectVariable object is created and stored in the symbol table of the unit bm. If there is already an entity declared in the unit bm with the identifier i, an error is generated.

- NstObjectVariable* **object**() const
  Returns the NstObjectVariable object associated to the variable. This is a cast of the object returned by the method object() of the class NstDeclaration.

This example creates a new formal parameter in a subroutine:

```
% cat test_varparam.cc
# include <libnstbase.H>
int main(int ac, char **av)
{
  NstIdentifier tutu ("toto");
  NstIdentifier n_id ("N");
  // New Subroutine
  NstUnitSubroutine stoto(tutu);
  // Declare new formal parameter in subroutine
  NstDeclarationVarParam dv(stoto, n_id, nst_integer_type);
  cout << stoto;
}
% test_varparam
      subroutine toto(N)
      integer*4 N
      end subroutine toto
```

---

class **NstDeclarationTemplate** : public **NstDeclaration**

---

int **type**() const
NstDeclarationTemplate* **isNstDeclarationTemplate**()

NstDeclarationTemplate(NstUnit& bm, NstIdentifier& i)

NstShapeList **dimensions**
NstObjectTemplate* **object**() const

This class defines the HPF declaration of a template (`TEMPLATE t`).

- int **type**() const
  Returns the type NST_DECLARATION_TEMPLATE.
- NstDeclarationTemplate* **isNstDeclarationTemplate**()
  Returns this.
- **NstDeclarationTemplate**(NstUnit& bm, NstIdentifier& i)
  Creates a new instance of this class. Declares a new template with identifier i into the unit bm. This object is added to the list of declarations of the unit bm. A new NstObjectTemplate object is created and stored in the symbol table of the unit bm. If there is already an entity declared in the unit bm with the identifier i, an error is generated.
- NstShapeList **dimensions**
  Stores the shape of the template.
- NstObjectTemplate* **object**() const
  Returns the NstObjectTemplate object associated with the declaration. This is a cast of the object returned by the method object() of the class NstDeclaration.

This example creates two templates in a program, one with one dimension and one with two dimensions:

```
% cat test_template.cc
# include <libnstbase.H>
int main()
{
  NstIdentifier t1 = "T1";
  NstIdentifier t2 = "T2";
  NstIdentifier pg = "temp";
  NstUnitProgram program(pg);
  NstDeclarationTemplate t1d(program, t1);
  NstDeclarationTemplate t2d(program, t2);
  NstShape one_hundred (100);
  t1d.dimensions.push_back(&one_hundred);
  t2d.dimensions.push_back(&one_hundred);
  t2d.dimensions.push_back(&one_hundred);
  cout << program;
}
% test_template
      program TEMP
!HPF$ TEMPLATE T1 (1:100)
!HPF$ TEMPLATE T2 (1:100,1:100)
      end program TEMP
```

---

class **NstDeclarationProcessors** : public **NstDeclaration**

---

int **type**() const
NstDeclarationTemplate* **isNstDeclarationTemplate**()

NstDeclarationProcessors(NstUnit& bm, NstIdentifier& i)

NstShapeList **dimensions**
NstObjectTopology* **object**() const

This class defines the HPF declaration of a topology (PROCESSORS)

- int **type**() const
  Returns the type NST_DECLARATION_PROCESSORS.
- NstDeclarationTopology* **isNstDeclarationTopology**()
  Returns this.
- **NstDeclarationProcessors**(NstUnit& bm, NstIdentifier& i)
  Creates a new instance of this class. Declares a new topology with identifier i into the unit bm. This object is added to the list of declarations of the unit bm. A new NstObjectTopology object is created and stored in the symbol table of the unit bm. If there is already an entity declared in the unit bm with the identifier i, an error is generated.
- NstShapeList **dimensions**
  Stores the shape of the topology.
- NstObjectTopology* **object**() const
  Returns the NstObjectTopology object associated with the declaration. This is a cast of the object returned by the method object() of the class NstDeclaration.

class **NstDeclarationDistribute** : public **NstDeclaration**

```
int type() const
NstDeclarationDistribute* isNstDeclarationDistribute()

NstDeclarationDistribute(NstUnit& u, NstObject& o, NstDistSpec& d, NstProcessor& p)

NstDistSpec* distribution() const
NstDistSpec* distribution(NstDistSpec& new_distribution)
NstProcessor* target() const
NstProcessor* target(NstProcessor& new_target)
```

This class describes the HPF specification of a distribution (DISTRIBUTE).

- int **type**() const
  Returns the type NST_DECLARATION_DISTRIBUTE.
- NstDeclarationDistribute* **isNstDeclarationDistribute**()
  Returns this.
- **NstDeclarationDistribute**(NstUnit& u, NstObject& o, NstDistSpec& d, NstProcessor& p)
  Creates a new instance of this class. This object is inserted into the list of declarations of the unit u and into the object o, that must represent a variable or a template. The distribution is specified by d and the target by p.
- NstDistSpec* **distribution**() const
  Returns the specification of the distribution.
- NstDistSpec* **distribution**(NstDistSpec& new_distribution)
  Replaces the specification of the distribution by new_distribution and returns the old one.
- NstProcessor* **target**() const
  Returns the target of the distribution.

- NstProcessor* **target**(NstProcessor& new_target)
  Replaces the target of the distribution by new_target and returns the old one.

This example distributes a template:

```
% cat test_distribute.cc
# include <libnstbase.H>
int main()
{
  // New program
  NstIdentifier pg = "temp";
  NstUnitProgram program(pg);
  // Declare a template
  NstIdentifier t1 = "T1";
  NstShape one_hundred (100);
  NstDeclarationTemplate t1d(program, t1);
  t1d.dimensions.push_back(&one_hundred);
  // Distribute the template
  NstDistSpec dspec(NST_DIST_SPEC_NODE, FALSE);
  NstDistFormat fblock(NST_DIST_FORMAT_ANY_BLOCK);
  dspec.mapping.push_back(&fblock);
  NstProcessor pd;
  NstDeclarationDistribute dd(program, *t1d.object(), dspec, pd);
  // Creates a new variable
  NstIdentifier aid = "a";
  NstTypeArray at(nst_real_type);
  at.shapes = t1d.dimensions;
  NstDeclarationVariable a(program, aid, at);
  // Creates a new alignment
  NstAlignSpec als(*t1d.object());
  NstDeclarationAlign align(program, *a.object(), als);
  // Unparse
  cout << program;
}
% test_distribute
      program temp
!HPF$ TEMPLATE T1 (1:100)
!HPF$ DISTRIBUTE T1 (BLOCK)
      real*4 a (1:100)
!HPF$ ALIGN a WITH T1
      end program temp
```

---

class **NstDeclarationAlign** : public **NstDeclaration**

---

int **type**() const
NstDeclarationAlign* **isNstDeclarationAlign**()

NstDeclarationAlign(NstUnit& u, NstObject& o, NstAlignSpec& as)

NstExpressionList **source**
NstAlignSpec* **target**() const
NstAlignSpec* **target**(NstAlignSpec& new_target)

This class describes the HPF specification of an alignment (`ALIGN`).

- int **type**() const
  Returns the type NST_DECLARATION_ALIGN.

- NstDeclarationAlign* **isNstDeclarationAlign**()
  Returns this.

- **NstDeclarationAlign**(NstUnit& u, NstObject& o, NstAlignSpec& as)
  Creates a new instance of this class. This object is inserted into the list of declarations of the unit u and into the object o, that must represent a variable or a template. The alignment is specified by as.

- NstExpressionList **source**
  Store the alignement of the source. To create a **NstVariableUsed** for an alignment, one must create a dummy **NstObjectVariable**. This can be done like this: imagine we want to create the alignment:

  ```
  !HPF$ ALIGN A(I,J) WITH B(I+1,J)
  ```

  We need two objects for I and J. They can be created like this:

  ```
  NstIdentifier i_id ("I");
  NstIdentifier j_id ("J");
  NstObjectVariable i_obj (i_id);
  NstObjectVariable j_obj (j_id);
  NstVariableUsed i_var(i_obj);
  NstVariableUsed j_var(j_obj);
  ```

- NstAlignSpec* **target**() const
  Returns the target of the alignment.

- NstAlignSpec* **target**(NstAlignSpec& new_target)
  Replaces the target of the alignment by new_target and returns the old one.

---

class **NstDeclarationDynamic** : public **NstDeclaration**

---

int **type**() const
NstDeclarationDynamic* **isNstDeclarationDynamic**()

NstDeclarationDynamic(NstUnit& u, NstObject& o)

This class describes the HPF declaration DYNAMIC.

- int **type**() const
  Returns the type NST_DECLARATION_DYNAMIC.

- NstDeclarationDynamic* **isNstDeclarationDynamic**()
  Returns this.

- **NstDeclarationDynamic**(NstUnit& u, NstObject& o)
  Creates a new instance of this class that is inserted into the unit u. Declares the object o as being dynamic: it can be realigned or redistributed.

---

class **NstDeclarationParameter** : public **NstDeclaration**

---

int **type**() const
NstDeclarationParameter* **isNstDeclarationParameter**()

NstDeclarationParameter(NstUnit& u, NstIdentifier& i, NstType& t, NstExpression& e)

NstExpression* **value**() const
NstExpression* **value**(NstExpression& new_value)
NstObjectVariable* **object**() const

This class describes the declaration of a parameter.

- int **type**() const
  Returns the type NST_DECLARATION_DYNAMIC.

- NstDeclarationParameter* **isNstDeclarationParameter**()
  Returns this.

- **NstDeclarationParameter**(NstUnit& u, NstIdentifier& i, NstType& t, NstExpression& e)
  Creates a new instance of this class that is inserted into the list of declarations of the unit u. Declares a parameter of identifier i with the type t and with value e (this expression is stored in the NstVarDescParameter object contained in the NstObjectVariable object associated with the parameter). A new NstObjectVariable object is created and stored in the symbol table of the unit u. If there is already an entity declared in the unit u with the identifier i, an error is generated.

- NstExpression* **value**() const
  Returns the value of the parameter.

- NstExpression* **value**(NstExpression& new_value)
  Replaces the value of the parameter by new_value and returns the old one.

- NstObjectVariable* **object**() const
  Returns the NstObjectVariable object associated to the parameter. This is a cast of the object returned by the method object() of the class NstDeclaration.

This example declares a parameter:

```
% cat test_param.cc
# include <libnstbase.H>
int main(int ac, char **av)
{
  NstIdentifier n_id ("N");
  NstIdentifier p_id ("PROG");
  NstInteger m = 1000;
  // New Subroutine
  NstUnitSubroutine stoto(p_id);
  // Declare new parameter in subroutine
  NstDeclarationParameter dv(stoto, n_id, nst_integer_type, m);
  cout << stoto;
}
% test_param
      subroutine PROG()
      integer*4 N
      PARAMETER (N = 1000)
      end subroutine PROG
```

---

class **NstDeclarationExternal** : public **NstDeclaration**

---

int **type**() const
NstDeclarationExternal* **isNstDeclarationExternal**()

NstDeclarationExternal(NstUnit& u, NstObject& o)

This class is used to declare an external subroutine.

- int **type**() const
  Returns the type NST_DECLARATION_EXTERNAL.
- NstDeclarationExternal* **isNstDeclarationExternal**()
  Returns this.
- **NstDeclarationExternal**(NstUnit& u, NstObject& o)
  Creates a new instance of this class, an external declaration of the object o into the list of declarations of u.

---

class **NstDeclarationIntrinsic** : public **NstDeclaration**

---

int **type**() const
NstDeclarationInstrinsic* **isNstDeclarationInstrinsic**()

NstDeclarationIntrinsic(NstUnit& u, NstObject& o)

This class is used to declare an intrinsic subroutine.

- int **type**() const
  Returns the type NST_DECLARATION_INTRINSIC.
- NstDeclarationIntrinsic* **isNstDeclarationIntrinsic**()
  Returns this.
- **NstDeclarationIntrinsic**(NstUnit& u, NstObject& o)
  Creates a new instance of this class, an intrinsic declaration of the object o into the list of declarations of u.

---

class **NstDeclarationList** : public **list<NstDeclaration*>**, public **NstTree**

NstTree* **isNstList**()
int **type**() const

NstDeclarationList()

NstDeclaration* **search**(NstIdentifier& id) const
NstDeclaration* **search**(const char* id) const
NstUnit* **in_unit**() const

This class represents a list of declarations.

- NstTree* **isNstList()**
  Returns this.

- int **type()** const
  Returns the type NST_DECLARATION_LIST.

- **NstDeclarationList()**
  Creates a new instance of this class, an empty list.

- NstDeclaration* **search**(NstIdentifier& id) const
  Searchs in the list the declaration with the name id. Returns NULL if the declaration is not found.

- NstDeclaration* **search**(const char* id) const
  Searchs in the list the declaration with the name id. Returns NULL if the declaration is not found.

- NstUnit* **in_unit()** const
  Returns the NstUnit object the list is in. Returns NULL if the list is not in an unit.

3.6 Types

---

class **NstType** : public **NstTree**

NstType* **isNstType**()
virtual NstTypeDummy* **isNstTypeDummy**()
virtual NstTypeInteger* **isNstTypeInteger**()
virtual NstTypeReal* **isNstTypeReal**()
virtual NstTypeComplex* **isNstTypeComplex**()
virtual NstTypeBoolean* **isNstTypeBoolean**()
virtual NstTypeArray* **isNstTypeArray**()
virtual NstTypeString* **isNstTypeString**()
virtual NstTypePointer* **isNstTypePointer**()

This class is the base class for representing fortran types.

- NstType* **isNstType**()
  Returns this.
- virtual NstTypeDummy* **isNstTypeDummy**()
  Returns NULL.
- virtual NstTypeInteger* **isNstTypeInteger**()
  Returns NULL.
- virtual NstTypeReal* **isNstTypeReal**()
  Returns NULL.
- virtual NstTypeComplex* **isNstTypeComplex**()
  Returns NULL.
- virtual NstTypeBoolean* **isNstTypeBoolean**()
  Returns NULL.
- virtual NstTypeArray* **isNstTypeArray**()
  Returns NULL.
- virtual NstTypeString* **isNstTypeString**()
  Returns NULL.
- virtual NstTypePointer* **isNstTypePointer**()
  Returns NULL.

---

class **NstTypeDummy** : public **NstType**

---

int **type**() const
NstTypeDummy* **isNstTypeDummy**()

NstTypeDummy()

This class represents a dummy type (no type).

- int **type**() const
  Returns the type NST_TYPE_DUMMY.

- NstTypeDummy* **isNstTypeDummy**()
  Returns this object.

- **NstTypeDummy**()
  Creates a new instance of this class.

---

class **NstTypeBase** : public **NstType**

NstExpression* **size**() const
NstExpression* **size**(NstExpression& new_size)

This is the base class for representing simple types that have a size: integers, reals, etc..

- NstExpression* **size**() const
  Returns the size of the type.

- NstExpression* **size**(NstExpression& new_size)
  Replaces the size of the type by new_size and returns the old one.

---

class **NstTypeInteger** : public **NstTypeBase**

---

int **type**() const
NstTypeInteger* **isNstTypeInteger**()

NstTypeInteger()
NstTypeInteger(NstExpression& t_size)

This class describes the type INTEGER. There is a global predefined object

NstTypeInteger **nst_integer_type**

that can be used instead of defining a new type. New objects for types should be created only if you want to create a type with a different size.

- int **type**() const
  Returns the type NST_TYPE_INTEGER.

- NstTypeInteger* **isNstTypeInteger**()
  Returns this object.

- **NstTypeInteger**()
  Creates a new instance of this class, a type with the default size (`sizeof (int)`).

- **NstTypeInteger**(NstExpression& t_size)
  Creates a new instance of this class, a type with size t_size.

---

class **NstTypeReal** : public **NstTypeBase**

---

int **type**() const
NstTypeReal* **isNstTypeReal**()

NstTypeReal()
NstTypeReal(NstExpression& t_size)

This class describes the type `REAL`. There is a global predefined type

NstTypeReal **nst_real_type**

that can be used instead of defining a new type. New objects for types should be created only if you want to create a type with a different size.

- int **type**() const
  Returns the type `NST_TYPE_REAL`.
- NstTypeReal* **isNstTypeReal**()
  Returns this object.
- **NstTypeReal**()
  Creates a new instance of this class, a type with the default size (`sizeof (float)`).
- **NstTypeReal**(NstExpression& t_size)
  Creates a new instance of this class, a type with size t_size.

---

class **NstTypeComplex** : public **NstTypeBase**

---

int **type**() const
NstTypeComplex* **isNstTypeComplex**()

NstTypeComplex()
NstTypeComplex(NstExpression& t_size)

This class describes the type `COMPLEX`. There is a global predefined type

NstTypeComplex **nst_complex_type**

that can be used instead of defining a new type. New objects for types should be created only if you want to create a type with a different size.

- int **type**() const
  Returns the type `NST_TYPE_COMPLEX`.
- NstTypeComplex* **isNstTypeComplex**()
  Returns this object.
- **NstTypeComplex**()
  Creates a new instance of this class, a type with the default size (`2*sizeof (float)`).
- **NstTypeComplex**(NstExpression& t_size)
  Creates a new instance of this class, a type with size t_size.

---

class **NstTypeBoolean** : public **NstTypeBase**

---

int **type**() const
NstTypeBoolean* **isNstTypeBoolean**()

NstTypeBoolean()
NstTypeBoolean(NstExpression& t_size)

This class describes the type LOGICAL. There is a global predefined type

NstTypeBoolean **nst_boolean_type**

that can be used instead of defining a new type. New objects for types should be created only if you want to create a type with a different size.

- int **type**() const
  Returns the type NST_TYPE_BOOLEAN.
- NstTypeBoolean* **isNstTypeBoolean**()
  Returns this object.
- **NstTypeBoolean**()
  Creates a new instance of this class, a type with the default size (`sizeof (int)`).
- **NstTypeBoolean**(NstExpression& t_size)
  Creates a new instance of this class, a type with size t_size.

---

class **NstTypeString** : public **NstTypeBase**

---

int **type**() const
NstTypeString* **isNstTypeString**()

NstTypeString()
NstTypeString(NstExpression& t_size)

This class describes the type CHARACTER.

- int **type**() const
  Returns the type NST_TYPE_STRING.
- NstTypeString* **isNstTypeString**()
  Returns this object.
- **NstTypeString**()
  Creates a new instance of this class, a type with the default size (`-1`).
- **NstTypeString**(NstExpression& t_size)
  Creates a new instance of this class, a type with size t_size.

---

class **NstTypeArray** : public **NstType**

---

int **type**() const
NstTypeArray* **isNstTypeArray**()

NstTypeArray(NstType& a_type)

NstType* **array_type**() const
NstType* **array_type**(NstType& new_type)
NstShapeList **shapes**

This class describes array types.

- int **type**() const
  Returns the type NST_TYPE_ARRAY.

- NstTypeArray* **isNstTypeArray**()
  Returns this object.

- **NstTypeArray**(NstType& a_type)
  Creates a new instance of this class, an array of type a_type.

- NstType* **array_type**() const
  Returns the type of the array.

- NstType* **array_type**(NstType& new_type)
  Replaces the type of the array by new_type and returns the old one.

- NstShapeList **shapes**
  Stores the shape of the array.

This is an example of the declaration of a two-dimensional array in a subroutine:

```
% cat test_tarray.cc
# include <libnstbase.H>
int main(int ac, char **av)
{
  // Identifiers
  NstIdentifier s_id = "DUMMY";
  NstIdentifier a_id = "A";
  // New Shape
  NstShape shape (100);
  // New array type
  NstTypeArray ar_type (nst_real_type);
  ar_type.shapes.push_back(&shape);
  // New subroutine
  NstUnitSubroutine subr(s_id);
  // Array declaration
  NstDeclarationVariable a_decl(subr, a_id, ar_type);
  // Unparsing
  cout << subr;
}
% test_tarray
      subroutine DUMMY()
      real*4 A (1:100,1:100)
      end subroutine DUMMY
```

class **NstTypePointer** : public **NstType**

int **type**() const
NstTypePointer* **isNstTypePointer**()

NstTypePointer(NstType& a_type)
NstType* **pointer_type**() const
NstType* **pointer_type**(NstType& new_type)

This class describes pointer types.

- int **type**() const
  Returns the type NST_TYPE_POINTER.
- NstTypePointer* **isNstTypePointer**()
  Returns this object.
- **NstTypePointer**(NstType& a_type)
  Creates a new instance of this class, a pointer of type a_type.
- NstType* **pointer_type**() const
  Returns the type of the pointer.
- NstType* **pointer_type**(NstType& new_type)
  Replaces the type of the pointer by new_type and returns the old one.

3.7 Shapes: arrays, templates, etc.

---

class **NstShape** : public **NstTree**

---

int **type**() const
NstShape* **isNstShape**()

NstShape()
NstShape(int tp, NstExpression& l)
NstShape(NstExpression& u)
NstShape(int u)
NstShape(NstExpression& l, NstExpression& u)
NstShape(int l, int u)

NstExpression* **lower**() const
NstExpression* **lower**(NstExpression& new_lower)
NstExpression* **upper**() const
NstExpression* **upper**(NstExpression& new_upper)

void **copy**(const NstShape& s, int deep = 0)
NstShape* **clone**(int deep = 0) const

This class represents shapes of arrays, templates, etc...

- int **type**() const
  Returns one of the following types:

NST_SHAPE_DEFERRED	(:)
NST_SHAPE_ASSUMED_SHAPE	(100:)
NST_SHAPE_ASSUMED_SIZE	(100:*)
NST_SHAPE_EXPLICIT	(1:100)

- NstShape* **isNstShape**()
  Returns this.

- **NstShape**()
  Creates an instance of this class with the type NST_SHAPE_DEFERRED.

- **NstShape**(int tp, NstExpression& l)
  Creates an instance of this class with the lower bound l with the type **tp** that can take the values NST_SHAPE_ASSUMED_SHAPE or NST_SHAPE_ASSUMED_SIZE.

- **NstShape**(NstExpression& u)
  Creates an instance of this class with the type NST_SHAPE_EXPLICIT. The lower bound is 1 and the upper bound is u.

- **NstShape**(int u)
  Creates an instance of this class with the type NST_SHAPE_EXPLICIT. The lower bound is 1 and the upper bound is u.

- **NstShape**(NstExpression& l, NstExpression& u)
  Creates an instance of this class with the type NST_SHAPE_EXPLICIT. The lower bound is l and the upper bound is u.

- **NstShape**(int l, int u)
  Creates an instance of this class with the type NST_SHAPE_EXPLICIT. The lower bound is l and the upper bound is u.
- NstExpression* **lower**() const
  Returns the lower bound expression.
- NstExpression* **lower**(NstExpression& new_lower)
  Replaces the lower bound expression with new_lower and returns the old one.
- NstExpression* **upper**() const
  Returns the upper bound expression.
- NstExpression* **upper**(NstExpression& new_upper)
  Replaces the upper bound expression with new_upper and returns the old one.
- void **copy**(const NstShape& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstShape* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

This example illustrates shapes:

```
% cat test_shapes.cc
# include <libnstbase.H>
int main()
{
  NstInteger hundred = 100;
  NstShape s1 (1000);
  NstShape s2 (NST_SHAPE_ASSUMED_SHAPE, hundred);
  NstShape s3 (NST_SHAPE_ASSUMED_SIZE, hundred);
  NstShape s4;
  NstShape s5 (5,67);
  cout << s1 << endl;
  cout << s2 << endl;
  cout << s3 << endl;
  cout << s4 << endl;
  cout << s5 << endl;
}
% test_shapes
1:1000
100:
100:*
:
5:67
```

---

class **NstShapeList** : public **list<NstShape*>**, public **NstTree**

---

int **type**() const
NstTree* **isNstList**()

NstShapeList()

void **copy**(const NstShapeList& e, int deep = 0)
NstShapeList* **clone**(int deep = 0) const

This class represents a list of shapes.

- int **type**() const
  Returns the type NST_SHAPE_LIST.
- NstTree* **isNstList**()
  Returns this.
- **NstShapeList**()
  Creates a new instance of this class, an empty list.
- void **copy**(const NstShapeList& e, int deep = 0)
  Copies the object **s** into this object, recursively or not according to **deep**.
- NstShapeList* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to **deep**.

3.8 Distributions, alignments

---

class **NstDistSpec** : public **NstTree**

---

int **type**() const
NstDistSpec* **isNstDistSpec**()

NstDistSpec(int dstp, bool id = FALSE)

NstDistFormatList **mapping**
bool **is_descriptive**

This class describes the specification of an HPF distribution.

- NstDistSpec* **isNstDistSpec**()
  Returns this.

- int **type**() const
  Returns one of the following types:

NST_DIST_SPEC_ANY	*
NST_DIST_SPEC_DEFAULT	*
NST_DIST_SPEC_REPL	(:REPLICATED)
NST_DIST_SPEC_NODE	*(BLOCK,BLOCK)

- **NstDistSpec**(int dstp, bool id = FALSE)
  Creates a new instance of this class. **dstp** can take the values above and **id** tells if the distribution is descriptive or not.

- NstDistFormatList **mapping**
  Stores the description of the distribution.

- bool **is_descriptive**
  Tells if the distribution is descriptive or prescriptive.

---

class **NstDistSpecList** : public **list<NstDistSpec*>**, public **NstTree**

NstTree* **isNstList**()
int **type**() const

NstDistSpecList()

void **copy**(const NstDistSpecList& e, int deep = 0)
NstDistSpecList* **clone**(int deep = 0) const

This class represents a list of distribution specifications.

- NstTree* **isNstList**()
  Returns this.

- int **type**() const
  Returns the type NST_DIST_SPEC_LIST.

- **NstDistSpecList**()
  Creates a new instance of this class, an empty list.

- void **copy**(const NstDistSpecList& e, int deep = 0)
  Copies the object **s** into this object, recursively or not according to deep.

- NstDistSpecList* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstDistFormat** : public **NstTree**

NstDistFormat* **isNstDistFormat**()
int **type**() const

NstDistFormat(int dt)
NstDistFormat(int dt, NstVariable& var)
NstDistFormat(int dt, NstExpression& exp)

NstExpression* **size**() const
NstExpression* **size**(NstExpression& new_expression)
NstVariable* **array**() const
NstVariable* **array**(NstVariable& new_array)

This class describes the specification of an HPF distribution format.

- NstDistFormat* **isNstDistFormat**()
  Returns this.

- int **type**() const
  Returns one of the following types:

NST_DIST_FORMAT_SERIAL	*
NST_DIST_FORMAT_ANY_BLOCK	BLOCK
NST_DIST_FORMAT_ANY_CYCLIC	CYCLIC
NST_DIST_FORMAT_ANY_GEN_BLOCK	GEN_BLOCK
NST_DIST_FORMAT_ANY_INDIRECT	INDIRECT
NST_DIST_FORMAT_ANY_DIM	ALL
NST_DIST_FORMAT_BLOCK	BLOCK, BLOCK(exp)
NST_DIST_FORMAT_CYCLIC	CYCLIC, CYCLIC(exp)
NST_DIST_FORMAT_GEN_BLOCK	GEN_BLOCK(var)
NST_DIST_FORMAT_INDIRECT	INDIRECT(var)

- **NstDistFormat**(int dt)
  Creates a new instance of this class with the type dt that can take the following values:

NST_DIST_FORMAT_SERIAL	*
NST_DIST_FORMAT_ANY_BLOCK	BLOCK
NST_DIST_FORMAT_ANY_CYCLIC	CYCLIC
NST_DIST_FORMAT_ANY_GEN_BLOCK	GEN_BLOCK
NST_DIST_FORMAT_ANY_INDIRECT	INDIRECT
NST_DIST_FORMAT_ANY_DIM	ALL

- **NstDistFormat**(int dt, NstVariable& var)
  Creates a new instance of this class with the type dt that can take the following values:

NST_DIST_FORMAT_GEN_BLOCK	GEN_BLOCK(var)
NST_DIST_FORMAT_INDIRECT	INDIRECT(var)

  and where var is the indirection array.

- **NstDistFormat**(int dt, NstExpression& exp)
  Creates a new instance of this class with the type dt that can take the following values:

NST_DIST_FORMAT_BLOCK	BLOCK, BLOCK(exp)
NST_DIST_FORMAT_CYCLIC	CYCLIC, CYCLIC(exp)

  and where exp is the size of the block.
  NstExpression* **size**() const
  Returns the size of the block.

  NstExpression* **size**(NstExpression& new_expression)
  Replaces the size of the block by new_expression and returns the old one.

  NstVariable* **array**() const
  Returns the indirection array.

  NstVariable* **array**(NstVariable& new_array)
  Replaces the indirection array by new_array and returns the old one.

---

class **NstDistFormatList** : public **list<NstDistFormat*>**, public **NstTree**

NstTree* **isNstList**()
int **type**() const

NstDistFormatList()

void **copy**(const NstDistFormatList& e, int deep = 0)
NstDistFormatList* **clone**(int deep = 0) const

This class represents a list of distribution formats.

- NstTree* **isNstList**()
  Returns this.

- int **type**() const
  Returns the type NST_DIST_FORMAT_LIST.

- **NstDistFormatList**()
  Creates a new instance of this class, an empty list.

- void **copy**(const NstDistFormatList& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstDistFormatList* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstProcessor** : public **NstTree**

---

int **type**() const

NstProcessor(int ptp = NST_PROCESSOR_DEFAULT_ARRAY)
NstProcessor(NstObjectTopology& o)

NstObjectTopology* **top_name**() const
NstObjectTopology* **top_name**(NstObjectTopology& new_object)
NstExpressionList **subscripts**

This class describes target processor for distribution or realignment.

- int **type**() const
Returns one of the following types:

NST_PROCESSOR_DEFAULT_ARRAY	
NST_PROCESSOR_ANY_ARRAY	`ONTO *`
NST_PROCESSOR_ARRAY	`ONTO P`

- **NstProcessor**(int ptp = NST_PROCESSOR_DEFAULT_ARRAY)
Creates an instance of this class with **ptp** that can take the following values:

  NST_PROCESSOR_DEFAULT_ARRAY
  NST_PROCESSOR_ANY_ARRAY

- **NstProcessor**(NstObjectTopology& o)
Creates an instance of this class of type NST_PROCESSOR_ARRAY, **o** being the topology.

- NstObjectTopology* **top_name**() const
Returns the topology.

- NstObjectTopology* **top_name**(NstObjectTopology& new_object)
Replaces the topology by **new_object** and returns the old one.

- NstExpressionList **subscripts**
This list defines a subset of the topology and is only pertinent if the object has the type:

  NST_PROCESSOR_ARRAY

---

class **NstAlignSpec** : public **NstTree**

---

int **type**() const
NstAlignSpec* **isNstAlignSpec**()

NstAlignSpec(NstObject& o)

NstObject* **top_name**() const
NstObject* **top_name**(NstObject& new_object)
NstExpressionList **subscripts**
bool **is_descriptive**

This class describes the specification of an alignment.

- NstAlignSpec* **isNstAlignSpec()**
  Returns this.

- int **type()** const
  Returns the type NST_ALIGN_SPEC.

- **NstAlignSpec**(NstObject& o)
  Creates a new instance of this class with the target alignment object o (an array or a template).

- NstObject* **top_name()** const
  Returns the object specifying the alignment.

- NstObject* **top_name**(NstObject& new_object)
  Replaces the object by new_object and returns the old one.

- NstExpressionList **subscripts**
  Specifies the alignment.

- bool **is_descriptive**
  Tells if the alignment is descriptive or prescriptive.

3.9 Statements and instructions

---

class **NstStatement** : public **NstTree**

NstStatement* **isNstStatement**()
virtual NstStatementContinue* **isNstStatementContinue**()
virtual NstStatementBasic* **isNstStatementBasic**()
virtual NstStatementIf* **isNstStatementIf**()
virtual NstStatementWhere* **isNstStatementWhere**()
virtual NstStatementRegion* **isNstStatementRegion**()
virtual NstStatementLoop* **isNstStatementLoop**()
virtual NstStatementNestor* **isNstStatementNestor**()
virtual NstStatementDo* **isNstStatementDo**()
virtual NstStatementWhile* **isNstStatementWhile**()
virtual NstStatementRepeat* **isNstStatementRepeat**()
virtual NstStatementForall* **isNstStatementForall**()

int **label**
int **line**

NstBranches **branches**

string **comment**
void **comment_all**()
void **uncomment_all**()
bool **is_commented**

NstUnit* **in_unit**() const
NstStatementList* **in_list**() const
void **attach_to_list**(NstStatementList& sl)
void **remove_from_list**()

void **copy**(const NstStatement& s, int deep = 0)
virtual NstStatement* **clone**(int deep = 0) const

This is the base class for describing statements. A statement is a control structure for describing loops, conditions, ...

- NstStatement* **isNstStatement**()
  Returns this.

- virtual NstStatementContinue* **isNstStatementContinue**()
  Returns NULL.

- virtual NstStatementBasic* **isNstStatementBasic**()
  Returns NULL.

- virtual NstStatementIf* **isNstStatementIf**()
  Returns NULL.

- virtual NstStatementWhere* **isNstStatementWhere**()
  Returns NULL.

- virtual NstStatementRegion* **isNstStatementRegion()**
  Returns NULL.

- virtual NstStatementLoop* **isNstStatementLoop()**
  Returns NULL.

- virtual NstStatementNestor* **isNstStatementNestor()**
  Returns NULL.

- virtual NstStatementDo* **isNstStatementDo()**
  Returns NULL.

- virtual NstStatementWhile* **isNstStatementWhile()**
  Returns NULL.

- virtual NstStatementRepeat* **isNstStatementRepeat()**
  Returns NULL.

- virtual NstStatementForall* **isNstStatementForall()**
  Returns NULL.

- int **label**
  Stores the label of the statement. If the object has not been created by a NstComputationUnit object, the label is set to 0 and is not unparsed.

- int **line**
  Stores the line number of the statement in the parsed source code. If the object has not been created by a NstComputationUnit object, the line number is set to 0.

- NstBranches **branches**
  This is a general list that permits to travel recursively around statements without knowing explicitely the type of the derived statement of NstStatement. This class has the following definition:

  NstBranches : public **list**<**NstStatementList***>

  For example, an NstStatementIf object has two branches, the then part and the else part. The arity of the statement is given by `branches().size()`. Therefore, it is easy to write a general travel around the statements:

```
void recursive_travel (NstStatementList& stats)
{
  for (NstStatementList::iterator i = stats.begin(); i != stats.end(); i++)
  {
    // Put your operation on *i here.
    NstBranches &b = (*i)->branches;
    for (NstBranches::iterator j = b.begin(); j != b.end(); j++)
      recursive_travel(*(*j));
  }
}
```

  This list if filled by the constructors of all derived classes of NstStatement.

- string **comment**
  If this string is not empty, it is put in the front of the statement during Fortran unparse.

- void **comment_all()**
  Comments the statement and all the statements inside it.

- void **uncomment_all**()
  Uncomments the statement and all the statements inside it.

- bool **is_commented**
  Tells if the statement is commented.

- NstUnit* **in_unit**() const
  Returns the NstUnit object where the statement is located. The object returned by this method depends on the method in_list(): if this method returns NULL, the method in_unit() too.

- NstStatementList* **in_list**() const
  This method returns a pointer to the list the statement is attached to. This method returns NULL if the statement is not attached to a list. See the method below for attaching an statement to a list. All the NstStatement objects created by the NstComputationUnit class are attached to their list, but a newly created object is not attached to a list.

- void **attach_to_list**(NstStatementList& sl)
  Attaches the statement to the list ul. This method will return an error if the statement is already in a list. Note that this method does not put the statement into the list, it just tells to the statement that it is in the list, so be careful! Example:

```
NstStatementContinue sc;
NstUnitProgram prog("toto");
NstStatementList dummy;
prog.statements.push_back(&sc);
dummy.push_back(&sc);
cout << sc.in_list(); // Return NULL
cout << sc.in_unit(); // Return NULL
sc.attach_to_list(prog.statements);
cout << sc.in_list(); // Return &prog.statements
cout << sc.in_unit(); // Return &prog
```

- void **remove_from_list**()
  Removes an statement from a list. Note that this method does not remove actually the statement from the list it is in, it just tells to the statement that it is not in the list anymore, so be careful! Example:

```
prog.statements.pop_back();
cout << sc.in_list(); // Return &prog.statements
cout << sc.in_unit(); // Return &prog
sc.remove_from_list();
cout << sc.in_list(); // Return NULL
cout << sc.in_unit(); // Return NULL
// sc is still in the list dummy
```

- void **copy**(const NstStatement& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- virtual NstStatement* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

class **NstStatementContinue** : public **NstStatement**

int **type**() const
NstStatementContinue* **isNstStatementContinue**()

NstStatementContinue()

void **copy**(const NstStatementContinue& s, int deep = 0)
NstStatement* **clone**(int deep = 0) const

This class represents the CONTINUE statement of Fortran.

- int **type**() const
  Returns the type NST_STATEMENT_CONTINUE.
- NstStatementContinue* **isNstStatementContinue**()
  Returns this object.
- **NstStatementContinue**()
  Creates a new isntance of this class.
- void **copy**(const NstStatementContinue& s, int deep = 0)
  Copies the object s into this object, deep is not taken into account because the copy cannot be recursive (it is a terminal object).
- NstStatement* **clone**(int deep = 0) const
  Returns a clone of this object, deep is not taken into account because the copy cannot be recursive (it is a terminal object).

This example creates a new CONTINUE statement with a label set to 100:

```
% cat test_continue.cc
# include <libnstbase.H>
int main()
{
  NstStatementContinue cont;
  cont.label = 100;
  cout << cont << endl;
}
% test_continue
 100  continue
```

---

class **NstStatementBasic** : public **NstStatement**

---

int **type**() const
NstStatementContinue* **isNstStatementContinue**()

NstStatementBasic(NstSimple& s)

NstSimple* **instruction**() const
NstSimple* **instruction**(NstSimple& new_simple)

void **copy**(const NstStatementBasic& s, int deep = 0)
NstStatement* **clone**(int deep = 0) const

This class encloses a instruction (action statement) of Fortran.

- int **type**() const
  Returns the type NST_STATEMENT_BASIC.
- NstStatementContinue* **isNstStatementContinue**()
  Returns this object.
- **NstStatementBasic**(NstSimple& s)
  Creates a new instance of this object, a basic statement containing the Fortran instruction s.
- NstSimple* **instruction**() const
  Returns the instruction enclosed in the statement.
- NstSimple* **instruction**(NstSimple& new_simple)
  Replaces the instruction enclosed in the statement by new_simple and returns the old one.
- void **copy**(const NstStatementBasic& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstStatement* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

This example creates a new basic statement containing a stop instruction with the parameter -1:

```
% cat test_basic.cc
# include <libnstbase.H>
int main()
{
  NstInteger mu = -1;
  NstSimple stop(mu, NST_SIMPLE_STOP);
  NstStatementBasic bs(stop);
  cout << bs << endl;
}
% test_basic
      stop -1
```

---

class **NstStatementIf** : public **NstStatement**

---

int **type**() const
NstStatementIf* **isNstStatementIf**()

NstStatementIf(NstExpression& e)

NstExpression* **condition**() const
NstExpression* **condition**(NstExpression& new_expression)
NstStatementList **then_part**
NstStatementList **else_part**

void **copy**(const NstStatementIf& s, int deep = 0)
NstStatement* **clone**(int deep = 0) const

This class represents the IF statement of Fortran.

- int **type**() const
  Returns the type NST_STATEMENT_IF.

- NstStatementIf* **isNstStatementIf**()
  Returns this object.

- **NstStatementIf**(NstExpression& e)
  Creates a new instance of this class, an IF statement with the boolean expression e.

- NstExpression* **condition**() const
  Returns the condition of the IF statement.

- NstExpression* **condition**(NstExpression& new_expression)
  Replaces the condition of the IF statement by new_expression and returns the old one.

- NstStatementList **then_part**
  The list of the statements in the THEN part of the statement.

- NstStatementList **else_part**
  The list of the statements in the ELSE part of the statement. The keyword ELSE is not unparsed if this list is empty.

- void **copy**(const NstStatementIf& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstStatement* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

This example creates a new IF statement with a statement CONTINUE in each part of the statement:

```
% cat test_if.cc
# include <libnstbase.H>
int main()
{
  NstBoolean vrai = TRUE;
  NstStatementContinue empty;
  NstStatementIf if_stat(vrai);
  if_stat.then_part.push_back(&empty);
  if_stat.else_part.push_back(&empty);
  cout << if_stat << endl;
}
% test_if
      if (.TRUE.) then
         continue
      else
         continue
      end if
```

class **NstStatementWhere** : public **NstStatement**

int **type**() const
NstStatementWhere* **isNstStatementWhere**()

NstStatementWhere(NstExpression& e)
NstExpression* **condition**() const
NstExpression* **condition**(NstExpression& new_expression)
NstStatementList **true_part**
NstStatementList **false_part**

void **copy**(const NstStatementWhere& s, int deep = 0)
NstStatement* **clone**(int deep = 0) const

This class represents the WHERE statement of Fortran.

- int **type**() const
  Returns the type NST_STATEMENT_WHERE.
- NstStatementWhere* **isNstStatementWhere**()
  Returns this object.
- **NstStatementWhere**(NstExpression& e)
  Creates a new instance of this class, a WHERE statement with the condition e.
- NstExpression* **condition**() const
  Returns the condition of the statement.
- NstExpression* **condition**(NstExpression& new_expression)
  Replaces the condition of the statement by new_expression and returns the old one.
- NstStatementList **true_part**
  The list of the statements in the TRUE part of the statement.
- NstStatementList **false_part**
  The list of the statements in the FALSE part of the statement. This list is not unparsed if it is empty.
- void **copy**(const NstStatementWhere& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstStatement* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstStatementRegion** : public **NstStatement**

---

int **type**() const
NstStatementRegion* **isNstStatementRegion**()

NstStatementRegion()

NstStatementList **body**

void **copy**(const NstStatementRegion& s, int deep = 0)
NstStatement* **clone**(int deep = 0) const

This class represents the TASK REGION statement of HPF.

- int **type**() const
  Returns the type NST_STATEMENT_REGION.
- NstStatementRegion* **isNstStatementRegion**()
  Returns this object.
- **NstStatementRegion**()
  Creates a new instance of this class, with an empty statements list.
- NstStatementList **body**
  The list of the statements in the region.
- void **copy**(const NstStatementRegion& s, int deep = 0)
  Copies the object **s** into this object, recursively or not according to **deep**.
- NstStatement* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to **deep**.

This example creates a region with a CONTINUE statement inside:

```
% cat test_task.cc
# include <libnstbase.H>
int main()
{
  NstStatementContinue empty;
  NstStatementRegion region;
  region.body.push_back(&empty);
  cout << region << endl;
}
% test_task
!HPF$ TASK_REGION
      continue
!HPF$ END TASK_REGION
```

---

class **NstStatementLoop** : public **NstStatement**

---

int **type**() const
NstStatementLoop* **isNstStatementLoop**()

NstStatementLoop()

NstStatementList **body**

void **copy**(const NstStatementLoop& s, int deep = 0)
NstStatement* **clone**(int deep = 0) const

This class represents an infinite loop in Fortran. An object of this class is unparsed like this:

```
do while (.true.)
  ! statements
enddo
```

- int **type**() const
  Returns the type NST_STATEMENT_LOOP.

- NstStatementLoop* **isNstStatementLoop**()
  Returns this object.

- **NstStatementLoop**()
  Creates a new instance of this class, with an empty statements list.

- NstStatementList **body**
  The list of the statements in the loop.

- void **copy**(const NstStatementLoop& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstStatement* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstStatementNestor** : public **NstStatement**

---

int **type**() const
NstStatementNestor* **isNstStatementNestor**()

NstStatementNestor()

NstStatementList **body**
bool **hide**

void **copy**(const NstStatementNestor& s, int deep = 0)
NstStatement* **clone**(int deep = 0) const

This class represents a region for Nestor: a set of statements delimited by a comment cNESTOR$ BEGIN and a comment cNESTOR$ END (see 2).

- int **type**() const
  Returns the type NST_STATEMENT_NESTOR.

- NstStatementNestor* **isNstStatementNestor**()
  Returns this object.

- **NstStatementNestor**()
  Creates a new instance of this class, with an empty statements list.

- NstStatementList **body**
  The list of the statements in the region.

- bool **hide**
  Hide the delimiters cNESTOR$ BEGIN and cNESTOR$ END if it is set to TRUE.

- void **copy**(const NstStatementNestor& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstStatement* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstStatementWhile** : public **NstStatement**

---

int **type**() const
NstStatementWhile* **isNstStatementWhile**()

NstStatementWhile(NstExpression& e)

NstExpression* **condition**() const
NstExpression* **condition**(NstExpression& new_expression)
NstStatementList **body**

void **copy**(const NstStatementWhile& s, int deep = 0)
NstStatement* **clone**(int deep = 0) const

This class represents the DO WHILE statement of Fortran.

- int **type**() const
  Returns the type NST_STATEMENT_WHILE.
- NstStatementWhile* **isNstStatementWhile**()
  Returns this object.
- **NstStatementWhile**(NstExpression& e)
  Creates a new instance of this class, with an empty statements list and the condition e.
- NstExpression* **condition**() const
  Returns the condition of the statement.
- NstExpression* **condition**(NstExpression& new_expression)
  Replaces the condition of the statement by new_expression and returns the old one.
- NstStatementList **body**
  The list of the statements in the while.
- void **copy**(const NstStatementWhile& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstStatement* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

```
% cat test_while.cc
# include <libnstbase.H>
int main()
{
  NstBoolean vrai = TRUE;
  NstStatementContinue empty;
  NstStatementWhile while_stat(vrai);
  while_stat.body.push_back(&empty);
  cout << while_stat << endl;
}
% test_while
      do while (.TRUE.)
         continue
      end do
```

---

class **NstStatementDo** : public **NstStatement**

---

int **type**() const
NstStatementDo* **isNstStatementDo**()

NstStatementDo(NstVariable& v, NstExpression& l, NstExpression& u)
NstStatementDo(NstVariable& v, NstExpression& l, NstExpression& u, NstExpression& s)

NstVariable* **index**() const
NstExpression* **lower_bound**() const
NstExpression* **upper_bound**() const
NstExpression* **step**() const
NstVariable* **index**(NstVariable& new_variable)
NstExpression* **lower_bound**(NstExpression& new_expression)
NstExpression* **upper_bound**(NstExpression& new_expression)
NstExpression* **step**(NstExpression& new_expression)
bool **independent**
int **nestor_flag**
NstVariableList **new_variables**
NstVariableList **reduction_variables**
NstStatementList **body**

void **copy**(const NstStatementDo& s, int deep = 0)
NstStatement* **clone**(int deep = 0) const

This class represents the DO statement of Fortran.

- int **type**() const
  Returns the type NST_STATEMENT_DO.
- NstStatementDo* **isNstStatementDo**()
  Returns this object.
- **NstStatementDo**(NstVariable& v, NstExpression& l, NstExpression& u)
  Creates a new instance of this class, with the loop index v, the lower bound l, the upper bound u.
- **NstStatementDo**(NstVariable& v, NstExpression& l, NstExpression& u, NstExpression& s)
  Creates a new instance of this class, with the loop index v, the lower bound l, the upper bound u and the step s.
- NstVariable* **index**() const
  Returns the loop index.
- NstVariable* **index**(NstVariable& new_variable)
  Replaces the loop index by new_variable and returns the old one.
- NstExpression* **lower_bound**() const
  Returns the lower bound.
- NstExpression* **lower_bound**(NstExpression& new_expression)
  Replaces the lower bound by new_expression and returns the old one.
- NstExpression* **upper_bound**() const
  Returns the upper bound.
- NstExpression* **upper_bound**(NstExpression& new_expression)
  Replaces the upper bound by new_expression and returns the old one.

- NstExpression* **step**() const
  Returns the step.
- NstExpression* **step**(NstExpression& new_expression)
  Replaces the step by new_expression and returns the old one.
- bool **independent**
  Unparses the HPF attribute INDEPENDENT if set to TRUE.
- int **nestor_flag**
  This flag is set to 1 by the node library if the directive SINGLE is put before the statement (see 2).
- NstVariableList **new_variables**
  List of variables in the HPF NEW clause. Only unparsed if the loop is independent.
- NstVariableList **reduction_variables**
  List of variables in the HPF REDUCTION clause. Only unparsed if the loop is independent.
- NstStatementList **body**
  The statements in the loop body.
- void **copy**(const NstStatementDo& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstStatement* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

This example creates a do loop:

```
% cat test_do.cc
# include <libnstbase.H>
int main()
{
  // Create a subroutine
  NstIdentifier td ("test_do");
  NstUnitSubroutine subroutine(td);

  // Declare a integer variable and get variable
  NstIdentifier i ("i");
  NstDeclarationVariable i_dv(subroutine, i, nst_integer_type);
  NstVariableLoop i_var (*i_dv.object());

  // Create bound expressions
  NstInteger un = 1;
  NstInteger dixmille = 10000;

  // Create a DO statement and insert in subroutine
  NstStatementDo do_stat (i_var, un, dixmille);
  subroutine.statements.push_back (&do_stat);
  do_stat.attach_to_list(subroutine.statements);

  // Declare: "integer a(1:10000)" in the subroutine
  NstIdentifier a ("a");
  NstTypeArray a_type (nst_integer_type);
  NstDeclarationVariable a_dv(subroutine, a, a_type);
  NstShape shape_dixmille (10000);
  a_type.shapes.push_back(&shape_dixmille);
```

```
  NstVariableUsed a_var(*a_dv.object());

  // Create a statement "a(i)=i" to insert in loop
  NstVariableIndexed a_indexed(a_var);
  a_indexed.indexes.push_back(&i_var);
  NstSimpleAssign assign (a_indexed, i_var);
  NstStatementBasic basic_stat (assign);
  do_stat.body.push_back(&basic_stat);
  basic_stat.attach_to_list(do_stat.body);

  // Unparse
  cout << subroutine << endl;
}
% test_do
      subroutine test_do()
      integer*4 i
      integer*4 a (1:10000)
      do i = 1,10000
         a(i) = i
      end do
      end subroutine test_do
```

class **NstStatementForall** : public **NstStatement**

int **type**() const
NstStatementForall* **isNstStatementForall**()

NstStatementForall(NstVariable& v, NstExpression& l, NstExpression& u)
NstStatementForall(NstVariable& v, NstExpression& l, NstExpression& u, NstExpression& s)

NstVariable* **index**() const
NstExpression* **lower_bound**() const
NstExpression* **upper_bound**() const
NstExpression* **step**() const
NstVariable* **index**(NstVariable& new_variable)
NstExpression* **lower_bound**(NstExpression& new_expression)
NstExpression* **upper_bound**(NstExpression& new_expression)
NstExpression* **step**(NstExpression& new_expression)
bool **independent**
int **nestor_flag**
NstVariableList **new_variables**
NstVariableList **reduction_variables**
NstStatementList **body**

void **copy**(const NstStatementForall& s, int deep = 0)
NstStatement* **clone**(int deep = 0) const

This class represents the FORALL statement of Fortran.

- int **type**() const
  Returns the type NST_STATEMENT_FORALL.
- NstStatementForall* **isNstStatementForall**()
  Returns this object.
- **NstStatementForall**(NstVariable& v, NstExpression& l, NstExpression& u)
  Creates a new instance of this class, with the loop index v, the lower bound l, the upper bound u.
- **NstStatementForall**(NstVariable& v, NstExpression& l, NstExpression& u, NstExpression& s)
  Creates a new instance of this class, with the loop index v, the lower bound l, the upper bound u and the step s.
- NstVariable* **index**() const
  Returns the loop index.
- NstVariable* **index**(NstVariable& new_variable)
  Replaces the loop index by new_variable and returns the old one.
- NstExpression* **lower_bound**() const
  Returns the lower bound.
- NstExpression* **lower_bound**(NstExpression& new_expression)
  Replaces the lower bound by new_expression and returns the old one.
- NstExpression* **upper_bound**() const
  Returns the upper bound.

- NstExpression* **upper_bound**(NstExpression& new_expression)
  Replaces the upper bound by new_expression and returns the old one.

- NstExpression* **step**() const
  Returns the step.

- NstExpression* **step**(NstExpression& new_expression)
  Replaces the step by new_expression and returns the old one.

- bool **independent**
  Unparses the HPF attribute cHPF$ INDEPENDENT if set to TRUE.

- int **nestor_flag**
  This flag is set to 1 by the node library if the directive SINGLE is put before the statement (see 2).

- NstVariableList **new_variables**
  List of variables in the HPF NEW clause. Only unparsed if the loop is independent.

- NstVariableList **reduction_variables**
  List of variables in the HPF REDUCTION clause. Only unparsed if the loop is independent.

- NstStatementList **body**
  The statements in the loop body.

- void **copy**(const NstStatementForall& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstStatement* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstStatementList** : public **list<NstStatement*>**, public **NstTree**

```
NstTree* isNstList()
int type() const

NstStatementList()

NstStatement* in_statement() const
NstUnit* in_unit() const
NstStatement* search_line(int l) const
NstStatement* search_label(int l) const
NstStatement* search(NstStatement* st) const
int check_list(int op = 0)
void release_statements()
void comment_all()
void uncomment_all()

void copy(const NstStatementList& s, int deep = 0)
```

This class represents a list of statements.

- NstTree* **isNstList**()
  Returns `this`.

- int **type**() const
  Returns the type NST_STATEMENT_LIST.

- **NstStatementList()**
  Creates a new instance of this class, an empty list.

- NstStatement* **in_statement()** const
  Returns the surrounding statement if this statement list is in a statement. Returns NULL elsewhere.

- NstUnit* **in_unit()** const
  Returns the surrounding unit if this statement list is in an unit. Returns NULL elsewhere.

- NstStatement* **search_line**(int l) const
  Returns the first statement in the list that is at or after the line l. Returns NULL elsewhere.

- NstStatement* **search_label**(int l) const
  Returns the statement in the list that has the label l. Returns NULL elsewhere.

- NstStatement* **search**(NstStatement* st) const
  Searchs the statement st in the list. Returns NULL if not found.

- void **release_statements()**
  Detachs all statements from the list (call the method remove_from_list() for each statement).

- void **comment_all()**
  Comments recursively all statements of the list.

- void **uncomment_all()**
  Uncomments recursively all statements of the list.

- void **copy**(const NstStatementList& s, int deep = 0)
  Copies all statements of the list s into this list, recursively or not according to deep. Each cloned statement of the list s is attached to the current list. If the copy is not recursive, the statements of s must not be attached to any list.

---

class **NstSimple** : public **NstTree**

---

virtual NstSimpleAssign* **isNstSimpleAssign()**
virtual NstSimpleCall* **isNstSimpleCall()**
virtual NstSimpleIO* **isNstSimpleIO()**
virtual NstSimpleGoto* **isNstSimpleGoto()**
virtual NstSimpleReturn* **isNstSimpleReturn()**
virtual NstSimpleExit* **isNstSimpleExit()**
virtual NstSimpleNullify* **isNstSimpleNullify()**
virtual NstSimpleAllocate* **isNstSimpleAllocate()**
virtual NstSimpleRealign* **isNstSimpleRealign()**
virtual NstSimpleRedistribute* **isNstSimpleRedistribute()**

void **copy**(const NstSimple& s, int deep = 0)
virtual NstSimple* **clone**(int deep = 0) const

This class is the base class for instructions (action statements). These instructions must be enclosed in a NstStatementBasic object.

- virtual NstSimpleAssign* **isNstSimpleAssign()**
  Returns NULL.

- virtual NstSimpleCall* **isNstSimpleCall()**
  Returns NULL.

- virtual NstSimpleIO* **isNstSimpleIO**()
  Returns NULL.

- virtual NstSimpleGoto* **isNstSimpleGoto**()
  Returns NULL.

- virtual NstSimpleReturn* **isNstSimpleReturn**()
  Returns NULL.

- virtual NstSimpleExit* **isNstSimpleExit**()
  Returns NULL.

- virtual NstSimpleNullify* **isNstSimpleNullify**()
  Returns NULL.

- virtual NstSimpleAllocate* **isNstSimpleAllocate**()
  Returns NULL.

- virtual NstSimpleRealign* **isNstSimpleRealign**()
  Returns NULL.

- virtual NstSimpleRedistribute* **isNstSimpleRedistribute**()
  Returns NULL.

- void **copy**(const NstSimple& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- virtual NstSimple* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstSimpleAssign** : public **NstSimple**

```
int type() const
NstSimpleAssign* isNstSimpleAssign()

NstSimpleAssign(NstVariable& l, NstExpression& r, int at = NST_SIMPLE_ASSIGN)

NstVariable* lvalue() const
NstVariable* lvalue(NstVariable& new_lvalue)
NstExpression* rvalue() const
NstExpression* rvalue(NstExpression& new_rvalue)

void copy(const NstSimpleAssign& s, int deep = 0)
NstSimple* clone(int deep = 0) const
```

This class represents assignments and pointer assignments.

- int **type**() const
  Returns one of the following types:

NST_SIMPLE_ASSIGN	`a = b + 2`
NST_SIMPLE_PTR_ASSIGN	`p => a`

- NstSimpleAssign* **isNstSimpleAssign**()
  Returns this object.

- **NstSimpleAssign**(NstVariable& l, NstExpression& r, int at = NST_SIMPLE_ASSIGN)
  Creates a new instance of this class, an assignment with the left value l and the right value r. The type of the assignment is at and can take the value NST_SIMPLE_ASSIGN or NST_SIMPLE_PTR_ASSIGN.

- NstVariable* **lvalue**() const
  Returns the left value of the assignment.

- NstVariable* **lvalue**(NstVariable& new_lvalue)
  Replaces the left value of the assignment by new_lvalue and returns the old one.

- NstExpression* **rvalue**() const
  Returns the right value of the assignment.

- NstExpression* **rvalue**(NstExpression& new_rvalue)
  Replaces the right value of the assignment by new_rvalue and returns the old one.

- void **copy**(const NstSimpleAssign& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstSimple* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstSimpleCall** : public **NstSimple**

---

int **type**() const
NstSimpleCall* **isNstSimpleCall**()

NstSimpleCall(NstObjectBaseProcedure& p)

NstObjectBaseProcedure* **procedure**() const
NstObjectBaseProcedure* **procedure**(NstObjectBaseProcedure& new_proc)
NstParameterList **parameters**

void **copy**(const NstSimpleCall& s, int deep = 0)
NstSimple* **clone**(int deep = 0) const

This class represents a procedure call.

- int **type**() const
  Returns the type NST_SIMPLE_CALL.

- NstSimpleCall* **isNstSimpleCall**()
  Returns this object.

- **NstSimpleCall**(NstObjectBaseProcedure& p)
  Creates a new instance of this class, a procedure call where p is the procedure.

- NstObjectBaseProcedure* **procedure**() const
  Returns the procedure call.

- NstObjectBaseProcedure* **procedure**(NstObjectBaseProcedure& new_proc)
  Replaces the procedure called by new_proc and returns the old one.

- NstParameterList **parameters**
  Parameters of the procedure call.

- void **copy**(const NstSimpleCall& s, int deep = 0)
  Copies the object **s** into this object, recursively or not according to **deep**.

- NstSimple* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to **deep**.

---

class **NstSimpleIO** : public **NstSimple**

```
int type() const
NstSimpleIO* isNstSimpleIO()

NstSimpleIO(NstIdentifier& i)

NstIdentifier* id() const
NstParameterList specs
NstParameterList items

void copy(const NstSimpleIO& s, int deep = 0)
NstSimple* clone(int deep = 0) const
```

This class handles IO operations of Fortran.

- int **type**() const
  Returns the type NST_SIMPLE_IO.

- NstSimpleIO* **isNstSimpleIO**()
  Returns this object.

- **NstSimpleIO**(NstIdentifier& i)
  Creates a new instance of this class, an IO instruction where **i** can take the following values:

  PRINT
  READ
  WRITE
  OPEN
  CLOSE
  REWIND
  BACKSPACE
  INQUIRE

- NstIdentifier* **id**() const
  Returns the identifier representing the IO instruction.

- NstParameterList **specs**
  Specifications of the IO operation.

- NstParameterList **items**
  Items of the IO operation.

- void **copy**(const NstSimpleIO& s, int deep = 0)
  Copies the object **s** into this object, recursively or not according to **deep**.

- NstSimple* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to **deep**.

---

class **NstSimpleGoto** : public **NstSimple**

---

int **type**() const
NstSimpleGoto* **isNstSimpleGoto**()

NstSimpleGoto(int lb)

int **label**

void **copy**(const NstSimpleGoto& s, int deep = 0)
NstSimple* **clone**(int deep = 0) const

- int **type**() const
  Returns the type NST_SIMPLE_GOTO.
- NstSimpleGoto* **isNstSimpleGoto**()
  Returns this object.
- **NstSimpleGoto**(int lb)
  Creates a new instance of this class, a jump to the label lb.
- int **label**
  The label where to go.
- void **copy**(const NstSimpleGoto& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstSimple* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstSimpleReturn** : public **NstSimple**

---

int **type**() const
NstSimpleReturn* **isNstSimpleReturn**()

NstSimpleReturn(NstExpression& e, int rt = NST_SIMPLE_RETURN)

NstExpression* **expression**() const
NstExpression* **expression**(NstExpression& new_exp)

void **copy**(const NstSimpleReturn& s, int deep = 0)
NstSimple* **clone**(int deep = 0) const

This class represents the RETURN, STOP, and PAUSE actions of Fortran.

- int **type**() const
  Returns one of the following types:

NST_SIMPLE_RETURN	`RETURN exp`
NST_SIMPLE_STOP	`STOP exp`
NST_SIMPLE_PAUSE	`PAUSE exp`

- NstSimpleReturn* **isNstSimpleReturn**()
  Returns this object.

- **NstSimpleReturn**(NstExpression& e, int rt = NST_SIMPLE_RETURN)
  Creates a new instance of this class, where rt can take the values

NST_SIMPLE_RETURN	`RETURN exp`
NST_SIMPLE_STOP	`STOP exp`
NST_SIMPLE_PAUSE	`PAUSE exp`

  and e is the expression of the instruction.

- NstExpression* **expression**() const
  Returns the expression.

- NstExpression* **expression**(NstExpression& new_exp)
  Replaces the expression by new_exp and returns the old one.

- void **copy**(const NstSimpleReturn& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstSimple* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstSimpleExit** : public **NstSimple**

---

int **type**() const
NstSimpleExit* **isNstSimpleExit**()

NstSimpleExit(NstIdentifier& lid, int at = NST_SIMPLE_EXIT)

NstIdentifier ***loopid**() const
NstIdentifier ***loopid**(NstIdentifier& new_lvalue)

void **copy**(const NstSimpleExit& s, int deep = 0)
NstSimple* **clone**(int deep = 0) const

This class represents the EXIT and CYCLE instructions of Fortran.

- int **type**() const
  Returns one of the following types:

NST_SIMPLE_EXIT	`EXIT id`
NST_SIMPLE_CYCLE	`CYCLE id`

- NstSimpleExit* **isNstSimpleExit**()
  Returns this object.

- **NstSimpleExit**(NstIdentifier& lid, int at = NST_SIMPLE_EXIT)
  Creates a new instance of this class, where rt can take the values

NST_SIMPLE_EXIT	`EXIT id`
NST_SIMPLE_CYCLE	`CYCLE id`

  and lid is the loop identifier.

- NstIdentifier ***loopid**() const
  Returns the loop identifier.
- NstIdentifier ***loopid**(NstIdentifier& new_lvalue)
  Replaces the loop identifier by new_lvalue and returns the old one.
- void **copy**(const NstSimpleExit& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstSimple* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstSimpleNullify** : public **NstSimple**

---

int **type**() const
NstSimpleNullify* **isNstSimpleNullify**()

NstSimpleNullify()

NstParameterList **parameters**

void **copy**(const NstSimpleNullify& s, int deep = 0)
NstSimple* **clone**(int deep = 0) const

This class represents the NULLIFY instruction of Fortran.

- int **type**() const
  Returns the type NST_SIMPLE_NULLIFY.
- NstSimpleNullify* **isNstSimpleNullify**()
  Returns this object.
- **NstSimpleNullify**()
  Creates a new instance of this class, a nullify instruction.
- NstParameterList **parameters**
  List of pointers to nullify.
- void **copy**(const NstSimpleNullify& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstSimple* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstSimpleAllocate** : public **NstSimple**

---

int **type**() const
NstSimpleAllocate* **isNstSimpleAllocate**()

NstSimpleAllocate(NstVariable& sv, int at = NST_SIMPLE_ALLOCATE)

NstVariable ***status**() const
NstVariable ***status**(NstVariable& new_stat)
NstParameterList **parameters**

void **copy**(const NstSimpleAllocate& s, int deep = 0)
NstSimple* **clone**(int deep = 0) const

This class represents the ALLOCATE and DEALLOCATE instructions of Fortran.

- int **type**() const
  Returns one of the following types:

NST_SIMPLE_ALLOCATE	`ALLOCATE(a,stat=s)`
NST_SIMPLE_DEALLOCATE	`DEALLOCATE(a,stat=s)`

- NstSimpleAllocate* **isNstSimpleAllocate**()
  Returns this object.

- **NstSimpleAllocate**(NstVariable& sv, int at = NST_SIMPLE_ALLOCATE)
  Creates a new instance of this class, an allocation/deallocation of the avriable sv, where st can take on of the types above.

- NstVariable ***status**() const
  Returns the status variable.

- NstVariable ***status**(NstVariable& new_stat)
  Replaces the status variable by new_stat and returns the old one.

- NstParameterList **parameters**
  List of the variables to allocate/deallocate.

- void **copy**(const NstSimpleAllocate& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstSimple* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstSimpleRealign** : public **NstSimple**

---

int **type**() const
NstSimpleRealign* **isNstSimpleRealign**()

NstSimpleRealign(NstVariable& v, NstAlignSpec& as)

NstVariable ***alignee**() const
NstVariable ***alignee**(NstVariable& new_var)
NstAlignSpec ***target**() const
NstAlignSpec ***target**(NstAlignSpec& new_as)

void **copy**(const NstSimpleRealign& s, int deep = 0)
NstSimple* **clone**(int deep = 0) const

This class represents the REALIGN instruction of HPF.

- int **type**() const
  Returns the type NST_SIMPLE_REALIGN.

- NstSimpleRealign* **isNstSimpleRealign**()
  Returns this object.

- **NstSimpleRealign**(NstVariable& v, NstAlignSpec& as)
  Creates a new instance of this class, a realign instruction of the array v with the alignment specification as.

- NstVariable ***alignee**() const
  Returns the alignee.

- NstVariable ***alignee**(NstVariable& new_var)
  Replaces the alignee with new_var and returns the old one.

- NstAlignSpec ***target**() const
  Returns the specification of the realignment.

- NstAlignSpec ***target**(NstAlignSpec& new_as)
  Replaces the specification of the realignment by new_as and returns the old one.

- void **copy**(const NstSimpleRealign& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstSimple* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstSimpleRedistribute** : public **NstSimple**

```
int type() const
NstSimpleRedistribute* isNstSimpleRedistribute()

NstSimpleRedistribute(NstVariable& v, NstDistSpec& ds, NstProcessor& p)

NstVariable* distributee() const
NstVariable* distributee(NstVariable& new_array)
NstDistSpec* distribution() const
NstDistSpec* distribution(NstDistSpec& new_dist)
NstProcessor* target() const
NstProcessor* target(NstProcessor& new_target)

void copy(const NstSimpleRedistribute& s, int deep = 0)
NstSimple* clone(int deep = 0) const
```

This class represents the REDISTRIBUTE instruction of HPF.

- int **type**() const
  Returns the type NST_SIMPLE_REDISTRIBUTE.

- NstSimpleRedistribute* **isNstSimpleRedistribute**()
  Returns this object.

- **NstSimpleRedistribute**(NstVariable& v, NstDistSpec& ds, NstProcessor& p)
  Creates a new instance of this class, a redistribution instruction of the array v with the distribution specification as onto processor p.

- NstVariable* **distributee**() const
  Returns the distributee.

- NstVariable* **distributee**(NstVariable& new_array)
  Replaces the distributee by new_array and returns the old one.

- NstDistSpec* **distribution**() const
  Returns the distribution specification.

- NstDistSpec* **distribution**(NstDistSpec& new_dist)
  Replaces the distribution specification by new_dist and returns the old one.

- NstProcessor* **target**() const
  Returns the target processor.

- NstProcessor* **target**(NstProcessor& new_target)
  Replaces the target processor by new_target and returns the old one.

- void **copy**(const NstSimpleRedistribute& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstSimple* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

3.10 Parameters

class **NstParameter** : public **NstTree**

NstParameter* **isNstParameter**()
virtual NstParameterVariable* **isNstParameterVariable**()
virtual NstParameterValue* **isNstParameterValue**()
virtual NstParameterNamed* **isNstParameterNamed**()
virtual NstParameterNone* **isNstParameterNone**()
virtual NstParameterFormat* **isNstParameterFormat**()

virtual int **is_variable_inside**(const NstVariable& v)

void **copy**(const NstParameter& e, int deep = 0)
virtual NstParameter* **clone**(int deep = 0) const

This class is the base class for parameters.

- NstParameter* **isNstParameter**()
  Returns this.

- virtual NstParameterVariable* **isNstParameterVariable**()
  Returns NULL.

- virtual NstParameterValue* **isNstParameterValue**()
  Returns NULL.

- virtual NstParameterNamed* **isNstParameterNamed**()
  Returns NULL.

- virtual NstParameterNone* **isNstParameterNone**()
  Returns NULL.

- virtual NstParameterFormat* **isNstParameterFormat**()
  Returns NULL.

- virtual int **is_variable_inside**(const NstVariable& v)
  This method searchs recursively the variable inside the parameter. Comparison is done between the value of the objects that are inside a variable. It returns 0 if the variable is not inside or 1 elsewhere.

- void **copy**(const NstParameter& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstSimple* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

class **NstParameterVariable** : public **NstParameter**

int **type**() const
NstParameterVariable* **isNstParameterVariable**()

NstParameterVariable(NstVariable& v)

NstVariable* **variable**() const
NstVariable* **variable**(NstVariable& new_var)

void **copy**(const NstParameterVariable& e, int deep = 0)
NstParameter* **clone**(int deep = 0) const

This class represents a parameter that is a variable.

- int **type**() const
  Returns the type NST_PARAMETER_VARIABLE.
- NstParameterVariable* **isNstParameterVariable**()
  Returns this object.
- **NstParameterVariable**(NstVariable& v)
  Creates a new instance of this class, a parameter that is a variable v.
- NstVariable* **variable**() const
  Returns the variable.
- NstVariable* **variable**(NstVariable& new_var)
  Replaces the variable by new_var and returns the old one.
- void **copy**(const NstParameterVariable& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstParameter* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstParameterValue** : public **NstParameter**

---

int **type**() const
NstParameterValue* **isNstParameterValue**()

NstParameterValue(NstExpression& v)

NstExpression* **value**() const
NstExpression* **value**(NstExpression& new_e)

void **copy**(const NstParameterValue& e, int deep = 0)
NstParameter* **clone**(int deep = 0) const

This class represents a parameter that is an expression.

- int **type**() const
  Returns the type NST_PARAMETER_VALUE.
- NstParameterValue* **isNstParameterValue**()
  Returns this object.

- **NstParameterValue**(NstExpression& v)
  Creates a new instance of this class, a parameter that has the value v.

- NstExpression* **value**() const
  Returns the value of the parameter.

- NstExpression* **value**(NstExpression& new_e)
  Replaces the value of the parameter by new_e and returns the old one.

- void **copy**(const NstParameterValue& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstParameter* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstParameterNamed** : public **NstParameter**

---

int **type**() const
NstParameterNamed* **isNstParameterNamed**()

NstParameterNamed(NstIdentifier& i, NstParameter& p)

NstIdentifier* **name**() const
NstIdentifier* **name**(NstIdentifier& new_i)
NstParameter* **parameter**() const
NstParameter* **parameter**(NstParameter& new_e)

void **copy**(const NstParameterNamed& e, int deep = 0)
NstParameter* **clone**(int deep = 0) const

This class represents a parameter of the form `name=...`, typically used for IO operations.

- int **type**() const
  Returns the type NST_PARAMETER_NAMED.

- NstParameterNamed* **isNstParameterNamed**()
  Returns this object.

- **NstParameterNamed**(NstIdentifier& i, NstParameter& p)
  Creates a new instance of this class, a named parameter with the name i and the parameter p.

- NstIdentifier* **name**() const
  Returns the name of the parameter.

- NstIdentifier* **name**(NstIdentifier& new_i)
  Replaces the name of the parameter by new_i and returns the old one.

- NstParameter* **parameter**() const
  Returns the parameter.

- NstParameter* **parameter**(NstParameter& new_e)
  Replaces the parameter by new_e and returns the old one.

- void **copy**(const NstParameterNamed& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstParameter* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstParameterNone** : public **NstParameter**

---

int **type**() const
NstParameterNone* **isNstParameterNone**()

NstParameterNone()

void **copy**(const NstParameterNone& e, int deep = 0)
NstParameter* **clone**(int deep = 0) const

This class represents no parameter.

- int **type**() const
  Returns the type NST_PARAMETER_NONE.
- NstParameterNone* **isNstParameterNone**()
  Returns this object.
- **NstParameterNone**()
  Creates a new instance of this class, a dummy parameter.
- void **copy**(const NstParameterNone& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstParameter* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstParameterFormat** : public **NstParameter**

---

int **type**() const
NstParameterFormat* **isNstParameterFormat**()

NstParameterFormat(string f)

string **format**

void **copy**(const NstParameterFormat& e, int deep = 0)
NstParameter* **clone**(int deep = 0) const

This class represents a parameter that is a format. It is used for IO operations.

- int **type**() const
  Returns the type NST_PARAMETER_FORMAT.
- NstParameterFormat* **isNstParameterFormat**()
  Returns this object.
- **NstParameterFormat**(string f)
  Creates a new instance of this class, a parameter defined by the format f.

- string **format**
  Stores the format.
- void **copy**(const NstParameterFormat& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstParameter* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstParameterList** : public **list<NstParameter*>**, public **NstTree**

NstTree* **isNstList**()
int **type**() const

NstParameterList()

void **copy**(const NstParameterList& e, int deep = 0)
NstParameterList* **clone**(int deep = 0) const

This class represents a list of parameters.

- NstTree* **isNstList**()
  Returns `this`.
- int **type**() const
  Returns the type NST_PARAMETER_LIST.
- **NstParameterList**()
  Creates a new instance of this class, an empty list.
- void **copy**(const NstParameterList& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstParameterList* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

3.11 Expressions

class **NstExpression** : public **NstTree**

NstExpression* **isNstExpression**()
virtual NstVariable* **isNstVariable**()
virtual NstInteger* **isNstInteger**()
virtual NstReal* **isNstReal**()
virtual NstBoolean* **isNstBoolean**()
virtual NstString* **isNstString**()
virtual NstComplex* **isNstComplex**()
virtual NstExpressionDummy* **isNstExpressionDummy**()
virtual NstExpressionBinary* **isNstExpressionBinary**()
virtual NstExpressionUnary* **isNstExpressionUnary**()
virtual NstExpressionCall* **isNstExpressionCall**()
virtual NstExpressionSlice* **isNstExpressionSlice**()

virtual int **is_variable_inside**(const NstVariable& v)

void **copy**(const NstExpression& e, int deep = 0)
virtual NstExpression* **clone**(int deep = 0) const

This class represents the base class for expressions.

- NstExpression* **isNstExpression**()
  Returns this.
- virtual NstVariable* **isNstVariable**()
  Returns NULL.
- virtual NstInteger* **isNstInteger**()
  Returns NULL.
- virtual NstReal* **isNstReal**()
  Returns NULL.
- virtual NstBoolean* **isNstBoolean**()
  Returns NULL.
- virtual NstString* **isNstString**()
  Returns NULL.
- virtual NstComplex* **isNstComplex**()
  Returns NULL.
- virtual NstExpressionDummy* **isNstExpressionDummy**()
  Returns NULL.
- virtual NstExpressionBinary* **isNstExpressionBinary**()
  Returns NULL.
- virtual NstExpressionUnary* **isNstExpressionUnary**()
  Returns NULL.

- virtual NstExpressionCall* **isNstExpressionCall**()
  Returns NULL.

- virtual NstExpressionSlice* **isNstExpressionSlice**()
  Returns NULL.

- virtual int **is_variable_inside**(const NstVariable& v)
  This method searchs recursively the variable inside the expression. Comparison is done between the value of the objects that are inside a variable. It returns 0 if the variable is not inside or 1 elsewhere.

- void **copy**(const NstExpression& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- virtual NstExpression* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstExpressionBinary** : public **NstExpression**

---

int **type**() const
NstExpressionBinary* **isNstExpressionBinary**()

NstExpressionBinary(int op, NstExpression& l, NstExpression& r)

int **operator_type**() const
NstExpression* **left**() const
NstExpression* **left**(NstExpression& new_e)
NstExpression* **right**() const
NstExpression* **right**(NstExpression& new_e)

void **copy**(const NstExpressionBinary& e, int deep = 0)
NstExpression* **clone**(int deep = 0) const

friend NstExpressionBinary& **operator+**(NstExpression& l, NstExpression& r)
friend NstExpressionBinary& **operator-**(NstExpression& l, NstExpression& r)
friend NstExpressionBinary& **operator***(NstExpression& l, NstExpression& r)
friend NstExpressionBinary& **operator/**(NstExpression& l, NstExpression& r)
friend NstExpressionBinary& **operator∧**(NstExpression& l, NstExpression& r)
friend NstExpressionBinary& **operator<**(NstExpression& l, NstExpression& r)
friend NstExpressionBinary& **operator>**(NstExpression& l, NstExpression& r)
friend NstExpressionBinary& **operator>=**(NstExpression& l, NstExpression& r)
friend NstExpressionBinary& **operator<=**(NstExpression& l, NstExpression& r)
friend NstExpressionBinary& **operator&&**(NstExpression& l, NstExpression& r)
friend NstExpressionBinary& **operator||**(NstExpression& l, NstExpression& r)
friend NstExpressionBinary& **operator==**(NstExpression& l, NstExpression& r)
friend NstExpressionBinary& **operator≠**(NstExpression& l, NstExpression& r)

This class represents binary expressions, like `a + b`.

- int **type**() const
  Returns the type NST_EXPRESSION_BINARY.

- NstExpressionBinary* **isNstExpressionBinary**()
  Returns this object.

- **NstExpressionBinary**(int op, NstExpression& l, NstExpression& r)
  Creates a new instance of this class, a binary expression with the left part l and the right part r. The operator op can take the following values:

Type	Fortran unparse	Operator redefined
NST_OPERAND_EQUAL	.eq.	operator==
NST_OPERAND_NOT_EQUAL	.ne.	operator≠
NST_OPERAND_LOWER	.lt.	operator<
NST_OPERAND_GREATER	.gt.	operator>
NST_OPERAND_GREATER_EQUAL	.ge.	operator>=
NST_OPERAND_LOWER_EQUAL	.le.	operator<=
NST_OPERAND_PLUS	+	operator+
NST_OPERAND_MINUS	-	operator-
NST_OPERAND_XOR	.xor.	
NST_OPERAND_OR	.or.	operator||
NST_OPERAND_CONCAT	//	
NST_OPERAND_TIMES	*	operator*
NST_OPERAND_DIVIDE	/	operator/
NST_OPERAND_AND	.and.	operator&&
NST_OPERAND_EQUIVALENT	.eqv.	
NST_OPERAND_NOT_EQUIVALENT	.neqv.	
NST_OPERAND_EXPONENTIAL	**	operator∧
NST_OPERAND_DEFINED	.identifier.	
NST_OPERAND_NOT	.not.	

- int **operator_type**() const
  The type of the binary operator (see above).

- NstExpression* **left**() const
  Returns the left part of the expression.

- NstExpression* **left**(NstExpression& new_e)
  Replaces the left part of the expression by new_e and returns the old one.

- NstExpression* **right**() const
  Returns the right part of the expression.

- NstExpression* **right**(NstExpression& new_e)
  Replaces the right part of the expression by new_e and returns the old one.

- void **copy**(const NstExpressionBinary& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstExpression* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

- friend NstExpressionBinary& **operator+**(NstExpression& l, NstExpression& r)
  These operators create a new object NstExpressionBinary with l the left part and r the right part of the expression. See the table above.

class **NstExpressionUnary** : public **NstExpression**

int **type**() const
NstExpressionUnary* **isNstExpressionUnary**()

NstExpressionUnary(int op, NstExpression& r)

int **operator_type**() const
NstExpression* **right**() const
NstExpression* **right**(NstExpression& new_e)

void **copy**(const NstExpressionUnary& e, int deep = 0)
NstExpression* **clone**(int deep = 0) const

friend NstExpressionUnary& **operator+**(NstExpression& r)
friend NstExpressionUnary& **operator-**(NstExpression& r)

This class represents unary expressions, like -a.

- int **type**() const
  Returns the type NST_EXPRESSION_UNARY.

- NstExpressionUnary* **isNstExpressionUnary**()
  Returns this object.

- **NstExpressionUnary**(int op, NstExpression& l, NstExpression& r)
  Creates a new instance of this class, a unary expression with the right part r. The operator op can take the following values:

Type	Fortran unparse	Operator redefined
NST_OPERAND_UNARY_PLUS	+	operator+
NST_OPERAND_UNARY_MINUS	-	operator-

- int **operator_type**() const
  The type of the unary operator (see above).

- NstExpression* **right**() const
  Returns the right part of the expression.

- NstExpression* **right**(NstExpression& new_e)
  Replaces the right part of the expression by new_e and returns the old one.

- void **copy**(const NstExpressionUnary& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstExpression* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

- friend NstExpressionUnary& **operator+**(NstExpression& l, NstExpression& r)
  These operators create a new object NstExpressionUnary with r the right part of the expression. See the table above.

class **NstExpressionDummy** : public **NstExpression**

int **type**() const
NstExpressionDummy* **isNstExpressionDummy**()

NstExpressionDummy()

void **copy**(const NstExpressionDummy& e, int deep = 0)
NstExpression* **clone**(int deep = 0) const

This class represents a dummy expression.

- int **type**() const
  Returns the type NST_EXPRESSION_DUMMY.
- NstExpressionDummy* **isNstExpressionDummy**()
  Returns this object.
- **NstExpressionDummy**()
  Creates a new instance of this class,
- void **copy**(const NstExpressionDummy& e, int deep = 0)
  Copies the object **s** into this object, recursively or not according to deep.
- NstExpression* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstExpressionCall** : public **NstExpression**

---

int **type**() const
NstExpressionCall* **isNstExpressionCall**()

NstExpressionCall(NstObjectBaseProcedure& op)

NstObjectBaseProcedure* **function**() const
NstObjectBaseProcedure* **function**(NstObjectBaseProcedure& new_ob)
NstParameterList **parameters**

void **copy**(const NstExpressionCall& e, int deep = 0)
NstExpression* **clone**(int deep = 0) const

This class represents a function call.

- int **type**() const
  Returns the type NST_EXPRESSION_CALL.
- NstExpressionCall* **isNstExpressionCall**()
  Returns this object.
- **NstExpressionCall**(NstObjectBaseProcedure& op)
  Creates a new instance of this class, a expression that is a call to the function represented by op.
- NstObjectBaseProcedure* **function**() const
  Returns the function called.
- NstObjectBaseProcedure* **function**(NstObjectBaseProcedure& new_ob)
  Replaces the function called by new_ob and returns the old one.

- NstParameterList **parameters**
  The parameters of the call.

- void **copy**(const NstExpressionCall& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstExpression* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstExpressionSlice** : public **NstExpression**

```
int type() const
NstExpressionSlice* isNstExpressionSlice()

NstExpressionSlice(NstExpression& lb, NstExpression& ub)
NstExpressionSlice(NstExpression& lb, NstExpression& ub, NstExpression& st)

NstExpression* lbound() const
NstExpression* lbound(NstExpression& new_e)
NstExpression* ubound() const
NstExpression* ubound(NstExpression& new_e)
NstExpression* step() const
NstExpression* step(NstExpression& new_e)

void copy(const NstExpressionSlice& e, int deep = 0)
NstExpression* clone(int deep = 0) const
```

This class represents a slice expression.

- int **type**() const
  Returns the type NST_EXPRESSION_SLICE.

- NstExpressionSlice* **isNstExpressionSlice**()
  Returns this object.

- **NstExpressionSlice**(NstExpression& lb, NstExpression& ub)
  Creates a new instance of this class, a slice expression with the lower bound lb, the upper bound ub.

- **NstExpressionSlice**(NstExpression& lb, NstExpression& ub, NstExpression& st)
  Creates a new instance of this class, a slice expression with the lower bound lb, the upper bound ub and the step st.

- NstExpression* **lbound**() const
  Returns the lower bound expression.

- NstExpression* **lbound**(NstExpression& new_e)
  Replaces the lower bound expression by new_e and returns the old one.

- NstExpression* **ubound**() const
  Returns the upper bound expression.

- NstExpression* **ubound**(NstExpression& new_e)
  Replaces the upper bound expression by new_e and returns the old one.

- NstExpression* **step**() const
  Returns the step expression.
- NstExpression* **step**(NstExpression& new_e)
  Replaces the step expression by new_e and returns the old one.
- void **copy**(const NstExpressionSlice& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstExpression* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstInteger** : public **NstExpression**

---

int **type**() const
NstInteger* **isNstInteger**()

NstInteger(int v)

int **value**

void **copy**(const NstInteger& e, int deep = 0)
NstExpression* **clone**(int deep = 0) const

This class represents an integer constant.

- int **type**() const
  Returns the type NST_INTEGER.
- NstInteger* **isNstInteger**()
  Returns this object.
- **NstInteger**(int v)
  Creates a new instance of this class, an integer constant with the value v.
- int **value**
  The value stored in the object.
- void **copy**(const NstInteger& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstExpression* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstReal** : public **NstExpression**

---

int **type**() const
NstReal* **isNstReal**()

NstReal(const string& v)
NstReal(const char* v)

string **value**

void **copy**(const NstReal& e, int deep = 0)
NstExpression* **clone**(int deep = 0) const

This class represents an real constant.

- int **type**() const
  Returns the type NST_REAL.
- NstReal* **isNstReal**()
  Returns this object.
- **NstReal**(const string& v)
  Creates a new instance of this class, a real constant with the value v.
- **NstReal**(const char* v)
  Creates a new instance of this class, a real constant with the value v.
- string **value**
  The value stored in the object.
- void **copy**(const NstReal& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstExpression* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstBoolean** : public **NstExpression**

---

int **type**() const
NstBoolean* **isNstBoolean**()

NstBoolean(bool v)

bool **value**

void **copy**(const NstBoolean& e, int deep = 0)
NstExpression* **clone**(int deep = 0) const

This class represents an boolean constant.

- int **type**() const
  Returns the type NST_BOOLEAN.
- NstBoolean* **isNstBoolean**()
  Returns this object.
- **NstBoolean**(bool v)
  Creates a new instance of this class, a boolean constant with the value v.
- bool **value**
  The value stored in the object.
- void **copy**(const NstBoolean& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstExpression* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstString** : public **NstExpression**

---

int **type**() const
NstString* **isNstString**()

NstString(const string& v)
NstString(const char* v)

string **value**

void **copy**(const NstString& e, int deep = 0)
NstExpression* **clone**(int deep = 0) const

This class represents an string constant.

- int **type**() const
  Returns the type NST_STRING.
- NstString* **isNstString**()
  Returns this object.
- **NstString**(const string& v)
  Creates a new instance of this class, a string constant with the value v.
- **NstString**(const char* v)
  Creates a new instance of this class, a string constant with the value v.
- string **value**(
)The value stored in the object.
- void **copy**(const NstString& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstExpression* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstComplex** : public **NstExpression**

---

int **type**() const
NstComplex* **isNstComplex**()

NstComplex(const string& r, const string& i)
NstComplex(const char* r, const char* i)

string **real**
string **img**

void **copy**(const NstComplex& e, int deep = 0)
NstExpression* **clone**(int deep = 0) const

This class represents an complex constant.

- int **type**() const
  Returns the type NST_COMPLEX.

- NstComplex* **isNstComplex**()
  Returns this object.

- **NstComplex**(const string& r, const string& i)
  Creates a new instance of this class, a complex constant with the real value r and the imaginary value i.

- **NstComplex**(const char* r, const char* i)
  Creates a new instance of this class, a complex constant with the real value r and the imaginary value i.

- string **real**
  The real part.

- string **img**
  The imaginary part.

- void **copy**(const NstComplex& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstExpression* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstExpressionList** : public **list<NstExpression*>**, public **NstTree**

NstTree* **isNstList**()
int **type**() const

NstExpressionList()

void **copy**(const NstExpressionList& e, int deep = 0)
NstExpressionList* **clone**(int deep = 0) const

This class represents a list of expressions.

- NstTree* **isNstList**()
  Returns this.

- int **type**() const
  Returns the type NST_EXPRESSION_LIST.

- **NstExpressionList**()
  Creates a new instance of this class, an empty list.

- void **copy**(const NstExpressionList& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstExpressionList* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

3.12 Variables

class **NstVariable** : public **NstExpression**

int **type**() const
NstVariable* **isNstVariable**()

virtual NstVariableDummy* **isNstVariableDummy**()
virtual NstVariableUsed* **isNstVariableUsed**()
virtual NstVariableLoop* **isNstVariableLoop**()
virtual NstVariableIndexed* **isNstVariableIndexed**()

void **copy**(const NstVariable& e, int deep = 0)
virtual NstVariable* **clone**(int deep = 0) const

This class is the base class for variables.

- int **type**() const
  Returns the type NST_VARIABLE.
- NstVariable* **isNstVariable**()
  Returns this object.
- virtual NstVariableDummy* **isNstVariableDummy**()
  Returns NULL.
- virtual NstVariableUsed* **isNstVariableUsed**()
  Returns NULL.
- virtual NstVariableLoop* **isNstVariableLoop**()
  Returns NULL.
- virtual NstVariableIndexed* **isNstVariableIndexed**()
  Returns NULL.
- void **copy**(const NstVariable& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- virtual NstVariable* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

class **NstVariableDummy** : public **NstVariable**

int **type**() const
NstVariableDummy* **isNstVariableDummy**()

NstVariableDummy()

void **copy**(const NstVariableDummy& e, int deep = 0)
NstVariable* **clone**(int deep = 0) const

This class represents dummy variables.

- int **type**() const
  Returns the type NST_VARIABLE_DUMMY.

- NstVariableDummy* **isNstVariableDummy**()
  Returns this object.

- **NstVariableDummy**()
  Creates a new instance of this class, a dummy variable.

- void **copy**(const NstVariableDummy& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstVariable* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstVariableLoop** : public **NstVariable**

---

int **type**() const
NstVariableLoop* **isNstVariableLoop**()

NstVariableLoop(NstObjectVariable& o)

NstObjectVariable* **object**() const

void **copy**(const NstVariableLoop& e, int deep = 0)
NstVariable* **clone**(int deep = 0) const

This class represents an access to a loop index variable (read or write).

- int **type**() const
  Returns the type NST_VARIABLE_LOOP.

- NstVariableLoop* **isNstVariableLoop**()
  Returns this object.

- **NstVariableLoop**(NstObjectVariable& o)
  Creates a new instance of this class, an access to the loop index variable represented by o.

- NstObjectVariable* **object**() const
  Returns the accessed variable.

- void **copy**(const NstVariableLoop& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstVariable* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstVariableUsed** : public **NstVariable**

---

int **type**() const
NstVariableUsed* **isNstVariableUsed**()

NstVariableUsed(NstObjectVariable& o)

NstObjectVariable* **object**() const

void **copy**(const NstVariableUsed& e, int deep = 0)
NstVariable* **clone**(int deep = 0) const

This class represents an access to a variable (read or write).

- int **type**() const
  Returns the type NST_VARIABLE_USED.
- NstVariableUsed* **isNstVariableUsed**()
  Returns this object.
- **NstVariableUsed**(NstObjectVariable& o)
  Creates a new instance of this class, an access to the variable represented by o.
- NstObjectVariable* **object**() const
  Returns the accessed variable.
- void **copy**(const NstVariableUsed& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.
- NstVariable* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstVariableIndexed** : public **NstVariable**

---

int **type**() const
NstVariableIndexed* **isNstVariableIndexed**()

NstVariableIndexed(NstVariable& v)

NstVariable* **variable**() const
NstVariable* **variable**(NstVariable& new_var)
NstExpressionList **indexes**

void **copy**(const NstVariableIndexed& e, int deep = 0)
NstVariable* **clone**(int deep = 0) const

This class represents an access to an indexed variables (read or write), e.g `a(i,j,k+1)`.

- int **type**() const
  Returns the type NST_VARIABLE_INDEXED.
- NstVariableIndexed* **isNstVariableIndexed**()
  Returns this object.
- **NstVariableIndexed**(NstVariable& v)
  Creates a new instance of this class, an indexed variable v.

- NstVariable* **variable**() const
  Returns the indexed variable.

- NstVariable* **variable**(NstVariable& new_var)
  Replaces the indexed variable by new_var and returns the old one.

- NstExpressionList **indexes**
  The indexes.

- void **copy**(const NstVariableIndexed& e, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstVariable* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

---

class **NstVariableList** : public **list<NstVariable*>**, public **NstTree**

NstTree* **isNstList**()
int **type**() const

NstVariableList()

void **copy**(const NstVariableList& s, int deep = 0)
NstVariableList* **clone**(int deep = 0) const

This class represents a list of variables.

- NstTree* **isNstList**()
  Returns `this`.

- int **type**() const
  Returns the type NST_VARIABLE_LIST.

- **NstVariableList**()
  Creates a new instance of this class, an empty list.

- void **copy**(const NstVariableList& s, int deep = 0)
  Copies the object s into this object, recursively or not according to deep.

- NstVariableList* **clone**(int deep = 0) const
  Returns a clone of this object, recursive or not according to deep.

3.13 Symbol table, definitions

---

class **NstObject** : public **NstTree**

NstObject* **isNstObject**()
virtual NstObjectBaseProcedure* **isNstObjectBaseProcedure**()
virtual NstObjectVariable* **isNstObjectVariable**()
virtual NstObjectTemplate* **isNstObjectTemplate**()
virtual NstObjectTopology* **isNstObjectTopology**()

NstIdentifier* **identifier**() const
NstIdentifier* **identifier**(NstIdentifier& id)
NstObjectBaseProcedure* **in_procedure**() const

This is the base class for storing informations about variables, subroutines, templates, etc... These objects are stored in the symbol table and are created automatically when a new variable, subroutine, function, or template is created. This class and child classes have no public constructors because they are only created by NstDeclaration objects.

- NstObject* **isNstObject**()
  Returns this.
- virtual NstObjectBaseProcedure* **isNstObjectBaseProcedure**()
  Returns NULL.
- virtual NstObjectVariable* **isNstObjectVariable**()
  Returns NULL.
- virtual NstObjectTemplate* **isNstObjectTemplate**()
  Returns NULL.
- virtual NstObjectTopology* **isNstObjectTopology**()
  Returns NULL.
- NstIdentifier* **identifier**() const
  Returns the name of the object (the name of the function, subroutine, ...).
- NstIdentifier* **identifier**(NstIdentifier& id)
  Replaces the current name with id and returns the old one.
- NstObjectBaseProcedure* **in_procedure**() const
  If the subroutine or function is contained into another unit, returns the unit containing it. Returns NULL elsewhere.

---

class **NstObjectBaseProcedure** : public **NstObject**

---

int **type**() const
NstObjectBaseProcedure* **isNstObjectBaseProcedure**()

const NstSymbolTable& **symbols**() const
NstUnit* **declaration**() const
int **kind**

This class permits to create objects that are associated with subroutines, functions, block data constructions or modules. An object created with this class contains the symbol table and the declaration of the unit associated with the object. Note that an unit also stores a pointer to this object.

- int **type**() const
  Returns one of the following types:

NST_OBJECT_EXTERNAL	`EXTERNAL s`
NST_OBJECT_PROCEDURE	`SUBROUTINE sub (...)`
NST_OBJECT_FUNCTION	`FUNCTION f (...)`
NST_OBJECT_BLOCK_DATA	`BLOCK DATA b`
NST_OBJECT_MODULE	`MODULE m`

- NstObjectBaseProcedure* **isNstObjectBaseProcedure**()
  Returns this object.

- const NstSymbolTable& **symbols**() const
  Returns the symbol table of the unit. The symbol table is a list of **NstObject** objects.

- NstUnit* **declaration**() const
  Returns the declaration of the unit associated with the object. Note that the **NstUnit** object has a method **object()** that returns the object associated with the unit (this object!).

- int **kind**
  This attribute is equal to 0 if the unit associated with the object is an user routine, to 1 if it is a dummy routine or to 2 if it is a module routine.

---

class **NstObjectVariable** : public **NstObject**

---

int **type**() const
NstObjectVariable* **isNstObjectVariable**()

NstObjectVariable(NstIdentifier& idf)

NstDeclarationVariable* **declaration**() const
NstVarDesc* **description**() const
NstDeclaration* **distribution**() const

This class defines an object that is associated with each declared variable. This object is stored in the symbol table of the object corresponding to the unit where the variable has been declared.

- int **type**() const
  Returns the type NST_OBJECT_VARIABLE (`INTEGER a`).

- NstObjectVariable* **isNstObjectVariable**()
  Return this object.

- **NstObjectVariable**(NstIdentifier& idf)
  Creates a dummy variable with no declaration, no type. It can be used for the declaration of a symbolic variable for an alignment.

- NstDeclarationVariable* **declaration()** const
  Returns the declaration of the variable associated with the object. Note that the NstDeclarationVariable object has a method object() that returns this object.

- NstVarDesc* **description()** const
  Returns the description of the variable (intent, target, ...).

- NstDeclaration* **distribution()** const
  Returns the HPF distribution of the variable if it exists. Returns `NULL` elsewhere.

---

class **NstObjectTemplate** : public **NstObject**

---

int **type()** const
NstObjectTemplate* **isNstObjectTemplate()**

NstDeclarationTemplate* **declaration()** const
NstDeclaration* **distribution()** const

This class defines an object that is associated with each declared template. This object is stored in the symbol table of the object corresponding to the unit where the template has been declared.

- int **type()** const
  Returns the type `NST_OBJECT_TEMPLATE` (`TEMPLATE t`).

- NstObjectTemplate* **isNstObjectTemplate()**
  Returns this object.

- NstDeclarationTemplate* **declaration()** const
  Returns the declaration of the template associated with the object. Note that the NstDeclarationTemplate object has a method object() that returns this object.

- NstDeclaration* **distribution()** const
  Returns the HPF distribution of the template if it exists. Returns `NULL` elsewhere.

---

class **NstObjectTopology** : public **NstObject**

---

int **type()** const
NstObjectTopology* **isNstObjectTopology()**

NstDeclarationProcessors* **declaration()** const

This class defines an object that is associated with each declared topology. This object is stored in the symbol table of the object corresponding to the unit where the topology has been declared.

- int **type()** const
  Returns the type `NST_OBJECT_TOPOLOGY` (`PROCESSORS P(6)`).

- NstObjectTopology* **isNstObjectTopology()**
  Returns this object.

- NstDeclarationProcessors* **declaration()** const
  Returns the declaration of the template associated with the object. Note that the NstDeclarationProcessors object has a method object() that returns this object.

---

class **NstSymbolTable** : public **list<NstObject>**, public **NstTree**

NstTree* **isNstList**()
int **type**() const

NstSymbolTable()

NstObject* **search**(NstIdentifier& id) const
NstObject* **search**(const char* id) const

This class represents a symbol table, a list of NstObject objects.

- NstTree* **isNstList**()
  Returns `this`.

- int **type**() const
  Returns the type NST_OBJECT_LIST.

- **NstSymbolTable**()
  Creates a new instance of this class, an empty list.

- NstObject* **search**(NstIdentifier& id) const
  Searchs in the list an object with the identifier id. Returns NULL if not found. A global variable search_status contains the location of the object after a call to this method. This variable can contain the following values:

SEARCH_LOCAL	Found in local table
SEARCH_INTRINSIC	Found in intrinsic table
SEARCH_EXTERNAL	Found in external table
SEARCH_NOTFOUND	Not found
SEARCH_NOTLOCAL	Found in an upper table

- NstObject* **search**(const char* id) const
  Searchs in the list an object with the identifier id. Returns NULL if not found. A global variable search_status contains the location of the object after a call to this method. See above for the possible values.

---

class **NstVarDesc** : public **NstTree**

---

virtual NstVarDescDummy ***isNstVarDescDummy**()
virtual NstVarDescParameter ***isNstVarDescParameter**()
virtual NstVarDescLocal ***isNstVarDescLocal**()

This class is the base class for description of variable attributes, like the intent, the expression associated with a parameter, etc... This class has no public contructor because objects of this class and descendents are only created by NstObject objects.

- virtual NstVarDescDummy ***isNstVarDescDummy**()
  Returns NULL.

- virtual NstVarDescParameter ***isNstVarDescParameter**()
  Returns NULL.

- virtual NstVarDescLocal ***isNstVarDescLocal**()
Returns NULL.

---

class **NstVarDescDummy** : public **NstVarDesc**

---

int **type**() const
NstVarDescDummy ***isNstVarDescDummy**()

int **intent**
int **dynamic**
bool **is_optional**
bool **is_target**

This class defines informations for a variable.

- int **type**() const
Returns the type NST_VARDESC_DUMMY.

- NstVarDescDummy ***isNstVarDescDummy**()
Returns this object.

- int **intent**
This is the intent of the variable. This attribute can have the following values:

NstIntentNo	0
NstIntentIn	1
NstIntentOut	2
NstIntentInOut	3

- int **dynamic**
This is the status of the variable in terms of dynamicity. This attribute can have the following values:

NstArrayIllegal	-1
NstArrayFixedSize	0
NstArrayAutomatic	1
NstArrayAllocatable	2
NstArrayAssumedShape	3
NstArrayAssumedSize	4

- bool **is_optional**
Tells if the variable is optional.

- bool **is_target**
Tells if the variable is declared as being a target for a pointer.

---

class **NstVarDescParameter** : public **NstVarDesc**

---

int **type**() const
NstVarDescParameter ***isNstVarDescParameter**()

NstExpression* **expression**() const
NstExpression* **expression**(NstExpression& e)

This class defines informations for a parameter variable.

- int **type**() const
  Returns the type NST_VARDESC_PARAMETER.

- NstVarDescParameter ***isNstVarDescParameter**()
  Returns this object.

- NstExpression* **expression**() const
  Returns the expression of the parameter.

- NstExpression* **expression**(NstExpression& e)
  Replaces the current expression of the parameter with e and returns the old expression.

---

class **NstVarDescLocal** : public **NstVarDesc**

---

int **type**() const
NstVarDescLocal ***isNstVarDescLocal**()

int **save**
int **dynamic**
bool **is_target**

- int **type**() const
  Returns the type NST_VARDESC_LOCAL.

- NstVarDescLocal ***isNstVarDescLocal**()
  Returns this object.

- int **save**
  Tells if the variable has the save attribute.

- int **dynamic**
  This is the status of the variable in terms of dynamicity. This attribute can have the following values:

NstArrayIllegal	-1
NstArrayFixedSize	0
NstArrayAutomatic	1
NstArrayAllocatable	2
NstArrayAssumedShape	3
NstArrayAssumedSize	4

- bool **is_target**
  Tells if the variable is declared as being a target for a pointer.

3.14 Miscellaneous functions

3.14.1 IO functions

- int **set_indent_step**(int nid)
  Set the indentation step to nid for output. By default, this indentation step is set to 3. Returns the old indentation step.

- int **nst_petit_unparse**(ostream& fortran, ostream& petit, const NstStatement &statement)
  Unparses a statement on a stream in the Petit language. The declarations needed are added to the beginning of the file. Note that the Petit language is a subset of Fortran. All the statements cannot be unparsed in Petit mode. This function returns 1 if the code cannot be unparsed. In this case, a warning is written on error stream to tell where the problem resides. Note that the unparse mode is not modified by this function. statement is unparsed in Fortran on the stream fortran (this pass is needed to get all the declarations), and in Petit on the stream petit.

- const string& **nst_temp_dir**()
  Returns the temporary directory where nestor stores its temporary files. By default, this directory is `/tmp`.

- const string& **nst_temp_dir**(const string& ntp)
  Replaces the temporary directory where nestor stores its temporary files by ntp and returns the old one.

- string **nst_tmp_file**(const string& postfix, const string& end)
  This functions creates a string of the form

  ```
  <nst_temp_dir()>/nst_<nst_tmp_number><postfix>.<end>
  ```

  For example, the call:

  ```
  string t = "truc";
  string e = "txt";
  cout << nst_tmp_file(t, e) << endl;
  ```

  will display:

  ```
  /tmp/nst_0truc.txt
  ```

- unsigned long **nst_tmp_number**()
  Each time this function is called, a global number is returned and incremented.

- ifstream& **nst_eatwhite**(ifstream& is)
  This function removes all consecutive spaces until it reaches a non-space character from the input stream is.

- ifstream& **nst_eatsome**(ifstream& is, int number)
  This function removes number spaces from the input stream is.

- ifstream& **nst_endofline**(ifstream& is)
  This function removes all character from is until the end of the current line.

- char **nst_sure_get**(ifstream& is)
  This function ensures that a character will be read. Returns an error if there are no character left in the input stream is. It returns the character read.

- string **nst_stream_to_file**(istream& from)
  This function reads the input stream from until EOF and put its content into a temporary file. It returns the name of the temporary file.

References

[1] T. Brandes, S. Chaumette, M.-C. Counilh, A. Darte, J.C. Mignot, F. Desprez, and J. Roman. HPFIT: A Set of Integrated Tools for the Parallelization of Applications Using High Performance Fortran: Part I: HPFIT and the TransTOOL Environment. *Parallel Computing*, 23(1-2):71–87, 1997.

[2] Thomas Brandes. *ADAPTOR Programmer's Guide - Version 5.0.* German National Research Institute for Computer Science, March 1997. `http://www.gmd.de/SCAI/lab/adaptor/`.

[3] PIPS Team. PIPS (Interprocedural Parallelizer for Scientific Programs). World Wide Web document, URL: `http://www.cri.ensmp.fr/~pips/index.html`.

[4] Stanford Compiler Group. SUIF compiler system. World Wide Web document, URL: `http://suif.stanford.edu/suif/suif.html`.

A Class types of the Base library

Type of the class	Name of the class
NST_IDENTIFIER	NstIdentifier
NST_COMPUTATION_UNIT	NstComputationUnit
NST_UNIT_PROGRAM	NstUnitProgram
NST_UNIT_SUBROUTINE	NstUnitSubroutine
NST_UNIT_FUNCTION	NstUnitFunction
NST_UNIT_MODULE	NstUnitModule
NST_UNIT_BLOCK_DATA	NstUnitBlockData
NST_UNIT_LIST	NstUnitList
NST_DECLARATION_DYNAMIC	NstDeclarationDynamic
NST_DECLARATION_TEMPLATE	NstDeclarationTemplate
NST_DECLARATION_PROCESSORS	NstDeclarationProcessors
NST_DECLARATION_IMPLICIT	NstDeclarationImplicit
NST_DECLARATION_FORMAT	NstDeclarationFormat
NST_DECLARATION_DISTRIBUTE	NstDeclarationDistribute
NST_DECLARATION_ALIGN	NstDeclarationAlign
NST_DECLARATION_VARIABLE	NstDeclarationVariable
NST_DECLARATION_VAR_PARAM	NstDeclarationVarParam
NST_DECLARATION_PARAMETER	NstDeclarationParameter
NST_DECLARATION_INTRINSIC	NstDeclarationIntrinsic
NST_DECLARATION_EXTERNAL	NstDeclarationExternal
NST_DECLARATION_LIST	NstDeclarationList
NST_TYPE_INTEGER	NstTypeInteger
NST_TYPE_REAL	NstTypeReal
NST_TYPE_BOOLEAN	NstTypeBoolean
NST_TYPE_COMPLEX	NstTypeComplex
NST_TYPE_DUMMY	NstTypeDummy
NST_TYPE_STRING	NstTypeString
NST_TYPE_ARRAY	NstTypeArray
NST_TYPE_POINTER	NstTypePointer
NST_SHAPE_EXPLICIT	NstShape
NST_SHAPE_ASSUMED_SHAPE	NstShape
NST_SHAPE_DEFERRED	NstShape
NST_SHAPE_ASSUMED_SIZE	NstShape
NST_SHAPE_LIST	NstShapeList
NST_DIST_SPEC_ANY	NstDistSpec
NST_DIST_SPEC_DEFAULT	NstDistSpec
NST_DIST_SPEC_REPL	NstDistSpec
NST_DIST_SPEC_NODE	NstDistSpec
NST_DIST_SPEC_LIST	NstDistSpecList
NST_DIST_FORMAT_SERIAL	NstDistFormat
NST_DIST_FORMAT_ANY_BLOCK	NstDistFormat
NST_DIST_FORMAT_ANY_CYCLIC	NstDistFormat
NST_DIST_FORMAT_ANY_GEN_BLOCK	NstDistFormat
NST_DIST_FORMAT_ANY_INDIRECT	NstDistFormat
NST_DIST_FORMAT_ANY_DIM	NstDistFormat
NST_DIST_FORMAT_BLOCK	NstDistFormat

continued on next page

continued from previous page

Type of the class	Name of the class
NST_DIST_FORMAT_CYCLIC	**NstDistFormat**
NST_DIST_FORMAT_GEN_BLOCK	**NstDistFormat**
NST_DIST_FORMAT_INDIRECT	**NstDistFormat**
NST_DIST_FORMAT_LIST	**NstDistFormatList**
NST_PROCESSOR_ANY_ARRAY	**NstProcessor**
NST_PROCESSOR_DEFAULT_ARRAY	**NstProcessor**
NST_PROCESSOR_ARRAY	**NstProcessor**
NST_ALIGN_SPEC	**NstAlignSpec**
NST_STATEMENT_CONTINUE	**NstStatementContinue**
NST_STATEMENT_BASIC	**NstStatementBasic**
NST_STATEMENT_IF	**NstStatementIf**
NST_STATEMENT_WHERE	**NstStatementWhere**
NST_STATEMENT_WHILE	**NstStatementWhile**
NST_STATEMENT_LOOP	**NstStatementLoop**
NST_STATEMENT_REPEAT	**NstStatementRepeat**
NST_STATEMENT_FORALL	**NstStatementForall**
NST_STATEMENT_REGION	**NstStatementRegion**
NST_STATEMENT_NESTOR	**NstStatementNestor**
NST_STATEMENT_DO	**NstStatementDo**
NST_STATEMENT_LIST	**NstStatementList**
NST_SIMPLE_IO	**NstSimpleIO**
NST_SIMPLE_GOTO	**NstSimpleGoto**
NST_SIMPLE_RETURN	**NstSimpleReturn**
NST_SIMPLE_PAUSE	**NstSimpleReturn**
NST_SIMPLE_STOP	**NstSimpleReturn**
NST_SIMPLE_EXIT	**NstSimpleExit**
NST_SIMPLE_CYCLE	**NstSimpleExit**
NST_SIMPLE_ALLOCATE	**NstSimpleAllocate**
NST_SIMPLE_DEALLOCATE	**NstSimpleAllocate**
NST_SIMPLE_NULLIFY	**NstSimpleNullify**
NST_SIMPLE_REALIGN	**NstSimpleRealign**
NST_SIMPLE_REDISTRIBUTE	**NstSimpleRedistribute**
NST_SIMPLE_ASSIGN	**NstSimpleAssign**
NST_SIMPLE_PTR_ASSIGN	**NstSimpleAssign**
NST_SIMPLE_CALL	**NstSimpleCall**
NST_EXPRESSION_DUMMY	**NstExpressionDummy**
NST_EXPRESSION_SLICE	**NstExpressionSlice**
NST_EXPRESSION_BINARY	**NstExpressionBinary**
NST_EXPRESSION_UNARY	**NstExpressionUnary**
NST_EXPRESSION_CALL	**NstExpressionCall**
NST_VARIABLE_DUMMY	**NstVariableDummy**
NST_VARIABLE_USED	**NstVariableUsed**
NST_VARIABLE_LOOP	**NstVariableLoop**
NST_VARIABLE_INDEXED	**NstVariableIndexed**
NST_VARIABLE_LIST	**NstVariableList**
NST_CONSTANT_BOOL	**NstBoolean**
NST_CONSTANT_INT	**NstInteger**
NST_CONSTANT_STRING	**NstString**

continued on next page

continued from previous page	
Type of the class	Name of the class
NST_CONSTANT_REAL	**NstReal**
NST_CONSTANT_COMPLEX	**NstComplex**
NST_EXPRESSION_LIST	**NstExpressionList**
NST_PARAMETER_NONE	**NstParameterNone**
NST_PARAMETER_FORMAT	**NstParameterFormat**
NST_PARAMETER_VAR	**NstParameterVariable**
NST_PARAMETER_VALUE	**NstParameterValue**
NST_PARAMETER_NAMED	**NstParameterNamed**
NST_PARAMETER_LIST	**NstParameterList**
NST_OBJECT_VARIABLE	**NstObjectVariable**
NST_OBJECT_TEMPLATE	**NstObjectTemplate**
NST_OBJECT_TOPOLOGY	**NstObjectTopology**
NST_OBJECT_EXTERNAL	**NstObjectBaseProcedure**
NST_OBJECT_PROCEDURE	**NstObjectBaseProcedure**
NST_OBJECT_FUNCTION	**NstObjectBaseProcedure**
NST_OBJECT_BLOCK_DATA	**NstObjectBaseProcedure**
NST_OBJECT_MODULE	**NstObjectBaseProcedure**
NST_OBJECT_LIST	**NstSymbolTable**
NST_VARDESC_DUMMY	**NstVarDescDummy**
NST_VARDESC_PARAMETER	**NstVarDescParameter**
NST_VARDESC_LOCAL	**NstVarDescLocal**

Table 1: Type of the classes of the base library.

Index