The new LooPo scanner and parser

Günz Robert
Kollnberg 2
94538 Fürstenstein

e-Mail: guenz@fmi.uni-passau.de

February 5, 1996

Contents:

1 THE NEW LOOPO SCANNER AND PARSER 2

1.1 INTRODUCTION 2
1.2 DIFFERENCES BETWEEN THE NSP AND THE OSP 2
1.3 EXTENSIONS 2
1.3.1 CONSTANT statement 2
1.3.2 Comments 2
1.3.3 Procedure and function declarations 2
1.3.4 Procedure and function calls 3
1.3.5 Conditional Expressions 3
1.3.6 WHILE 3
1.3.7 More complex arith. expressions 3
1.4 THE CLASS DESCRIPTIONS 5
1.4.1 LaNode 5
1.4.2 LaExpr 9
1.4.3 LaDep 10
1.4.4 SymbolTable 11
1.4.5 loopoParser 13
1.4.6 Parser 15
1.5 GRAMMAR 16
1.6 THE NSP AND THE TOOLS BISON++ AND FLEX++ 20
1.6.1 Note 20
1.7 AN EXAMPLE: 20
1.7.1 Source program: 20
1.7.2 SymbolTable: 21
1.7.3 MainProgram: 22
1.7.4 Functions: 25
1.7.5 Dependencies: 25
1.7.6 Statistics: 25

1 The new LooPo scanner and parser

1.1 Introduction

LooPo is a tool for loop parallelisation being based on the polytop model [Len93]. For that purpose the user gives LooPo the source program, he wants to parallelize, as input. The syntax of the source program is not bound to a special programming language, but different notations like C or FORTRAN can be used and the different notations can even be mixed within a program statement by statement. The statements, allowed in a source program, are assignments, function and procedure calls, skip-statements and for/while loops. The scanner and parser is one module of LooPo and its function is to create a parsetree and symbol table from the source program. This information is used by other LooPo modules during parallelisation process. Because the original scanner and parser was a simple recursive descent parser and the syntax of the source program was extended to the statements mentioned above (and some more), a new scanner and parser was written. To construct this new scanner and parser the tools *bison*++ and *flex*++ were used. Therefore it is sufficient to provide the grammar of the source language in a BNF-like format and to provide some semantical actions connected to single grammar rules and *bison*++ will create the correspondending parser with this information. This is somewhat simplified, but in fact you needn't spend time on the implementation of the parser itself, but on the definition of the grammar of the source language.
For abbreviation in the rest of this paper the new scanner and parser is called **NSP** ond the old one **OSP**.

1.2 Differences between the NSP and the OSP

The NSP is an extension of the OSP except the following. The automatically detection of the source language is not supported so far. One consequence of this is, that statement by statement the programming language can change without causing an error or a warning. Furthermore the handling of expressions like a(x) has slighty changed. Within FORTRAN for-loops and FORTRAN while-loops these expressions are treated as arrays, otherwise as functions.

1.3 Extensions

1.3.1 CONSTANT statement

This statement allows to define a variable explicitly as a structure parameter (also called blob for abbreviation). This was necessary, because on principle the NSP considers a simple identifier as a zero dimensional array. To avoid this mechanism you must use the CONSTANT statement.

Examples:

```
DO i=1,n
  skip
END DO
```

n will be treated as an array

```
CONSTANT n;
DO i=1,n
  skip
END DO
```

now n is introduced as a blob

1.3.2 Comments

Comments can be inserted into the source program. They must be enclosed by /* and */, like in C-notation. The use of */ within the comment is not allowed.

Examples:

```
/* The following program consists
   only of the 'skip' statement      */

skip;
```

```
/* you can use /* within the comment,
   but not '*/'
*/
```

This will cause an error.

1.3.3 Procedure and function declarations

Now it is possible to define procedures and functions on your own, which then can be called in the main program. Beside the automatically provided functions min, max, ceil, floor and return, new functions and procedures can be added as you like, either complete declarations or only prototypes (or even both). Note, that no 'overloading' is supported. Moreover, declaring functions and procedures, it is necessary to specify the

input/output behaviour of the formal parameters, i.e. whether they are used for input, output or both. The key words provided for that are IN, OUT and INOUT (according to the notation of FORTRAN 90). If in FORTRAN-notation the specification of the variables is omitted, they are assumed to be of type INOUT.

Examples: (on the left the C-notation and on the right the correspondending FORTRAN-notation)

```
int sqr(IN x)
{
 return x*x;
}
```

```
FUNCTION sqr(x)
 INTEGER x,sqr
  sqr=x*x
END FUNCTION sqr
/* x is of type INOUT */
```

```
void sqr2(IN x, OUT res)
{
 res=x*x;
}
```

```
SUBROUTINE sqr2(x, res)
 INTEGER,INTENT(IN)::x
 INTEGER,INTENT(OUT)::res
  res=x*x
END SUBROUTINE sqr2
```

1.3.4 Procedure and function calls

Because there is a opportunity of declaring/defining functions and procedures, it is also possible to call these functions and procedures of course. Procedures can be called at the same sites where assignments can be called, functions at any position within an arith. or cond. expression.

Examples:

```
a=5;
while( sqr(a)<10 ) { a=a+1; }
c=sqr(4*sqr(10));
sqr2(10,d);
```

1.3.5 Conditional Expressions

A condtional expression has the following form:

```
cond  :  expr
      |  expr { < | <= | == | = | >= | > } expr
      |  NOT cond
      |  cond { AND | OR } cond
      |  '(' cond ')'
```

where expr is an arith. expression

Note, that arith. expressions are not limited to boolean functions in this case. Every kind of arith. expression is possible, even integers. Then the same behaviour as in C is adopted (expr equals 0: FALSE, otherwise: TRUE).

1.3.6 WHILE

The NSP can now process while-loops written in the languages FORTRAN and C.

Examples: (Note: the C-notation and FORTRAN-notation do not represent the same)

```
a=0;
while(a<100)
{
  a=a+1;
}
```

```
FUNCTION sqr(x)
 INTEGER sqr,x
END FUNCTION sqr

c=0;
DO WHILE ( sqr(c) <= 121 )
  c=c+1
END DO
```

1.3.7 More complex arith. expressions

The OSP is limited to very simple arith. expressions. Expressions of the form $(c_1*b_1+...+c_n*b_n)/c_{n+1}$ are the most complex ones (with c_i a constant number, b_i a index or blob). The NSP has no such limitations. Quite

complex expressions can be used both in assignments and in loop bounds. The so called lambda expressions, the OSP is limited to, are supported in a special way (see **Class LaEpr**), because many of the transformation algorithms used for parallelisation work more efficiently on this kind of structure.

Examples:

```
a := 5 * f( b[ x,g[a] ],2 ) + ( g(a) + f( a,5 ) ) / b[a,a]

CONSTANT n;
DO i = 1, ( 5 * n + 3 ) / 6 + ( n * 8 + 1 ) / 7
/* the upper bound of the loop will be translated in a LaExpr */
     skip
END DO
```

1.4 The class descriptions

On the following pages a short survey of the used classes is given. For that purpose the internal structure and the element-functions are described.

1.4.1 LaNode

Objects of **class LaNode** (short *LaNodes*) are used by the NSP to construct the parsetree and the list of function and procedure declarations detected in the source program.

1.4.1.1 Elements and functions

The class LaNode (lamda node) includes:

private components:

The exact decription of some elements is shown in the next section, because the meaning can change depending on the type *typ* of the *LaNode*.

int	name;	
int	typ;	Assign | Expr | Skip | Forloop | Whileloop | If | Fctdecl | Proccall
int	state;	Read (an array used on the right hand side (RHS) of an assignment) | Write (an array used on the left hand side (LHS) of an assignment) | Const | Index | Blob | Function | FunctionCall | Proto | Various
int	stmnt;	number of the statement
int	object_id;	identifier of the LaNode
int	target_stmnt;	together with *stmnt* a more exact numbering of statements is possible
int	op;	BRACKET | PLUS | MINUS | TIMES | DIV | MOD | NOT | AND | OR
LaExpr	*laExpr;	pointer to a *LaExpr*, if a *LaNode* subtree of type **Expr** rooted at this *LaNode* represents a lambda expression
Entry	symbol;	pointer to an element in the symbol table only used temporarily during parsing a function/procedure declaration by the parser itself
LaNode	*low;	lower bound of a loop
LaNode	*up;	upper bound of a loop
LaNode	*step;	stepwidth of a loop
LaNode	*indices;	pointer to a list of *LaNodes* of type **Expr** for the indices of an array variable
LaNode	*parent;	pointer to the *LaNode* representing the next outer loop
LaNode	*children;	
LaNode	*next;	pointer to the next program construct on the same hierarchy level
LaNode	*cond;	pointer to a *LaNode* representing the condition of **Whileloop** or **If** statements
LaNode	*elsechild;	
LaNode	*arguments;	pointer to a *LaNode* representing the arguments of a function/procedure

public components:

~LaNode( );	destructor
LaNode(LaNode *ln);	constructor creating a copy of the given LaNode *ln*
LaNode(int t);	constructor which only sets the type *t* of the *LaNode*

static LaNode *set_skip( );

creates a *LaNode* of type **Skip**.

static LaNode *set_assignment(int name, LaNode *indices, LaNode *expr);

creates a *LaNode* of type **Assign.** The LHS (is an array) has the name *name* and the indices *indices.* The RHS of the assignment is given by *expr.*

static LaNode *set_expr_for_NUM(int num);

creates a *LaNode* of type **Expr** representing a constant with value *num*.

static LaNode *set_expr_for_INDEX(int name);

creates a *LaNode* of type **Expr** representing an index with number *name* in the symbol table.

static LaNode *set_expr_for_BLOB(int name);

creates a *LaNode* of type **Expr** representing a blob with number *name* in the symbol table.

static LaNode *set_expr_for_ARRAY(int name, LaNode *indices);

creates a *LaNode* of type **Expr** representing an array with number *name* in the symbol table and indices *indices*.

static LaNode *set_expr_for_FCT(int name, LaNode *arg);

creates a *LaNode* of type **Expr** representing a function call of a function with number *name* in the symbol table and arguments *arg*.

static LaNode *set_parenthesis(LaNode *expr);

sets brackets around the expression represented by the LaNode *expr*, but only if there are not already brackets around this expression. Therefor a new *LaNode* of type **Expr** is created with entry *op* = BRACKET pointing to the LaNode *expr*.

static LaNode *set_unary_expr(int op, LaNode *expr);

creates a *LaNode* of type **Expr** representing an unary expression with operator *op* (MINUS | NOT) and expression *expr* the operator *op* is applied to.

static LaNode *set_binary_expr(int op, LaNode *expr1, LaNode *expr2);

creates a *LaNode* of type **Expr** representing a binary expression with operator *op* (MINUS | PLUS | ...) and expressions *expr1* and *expr2* the operator *op* is applied to.

static LaNode *set_forloop(int name, LaNode* low, LaNode* up, LaNode* step, LaNode* body, int state);

creates a *LaNode* of type **Forloop** for the index with number *name*. The LaNode *body* represents the loopbody and *state* is one of the values Do, DoAll or DoAcross.

static LaNode *set_if(LaNode *cond, LaNode *then_, LaNode *else_);

creates a *LaNode* of type **If** with condition *cond*, then-branch *then_* and else-branch *else_* .

static LaNode *set_while(int name, LaNode* cond, LaNode* low, LaNode* step, LaNode* body, int state);

creates a *LaNode* of type **Whileloop** for the index with number *name* and condition *cond.* The LaNode *body* represents the loopbody of the while loop and *state* is one of the values Do, DoAll or DoAcross.

static LaNode *set_fct_argument(int name, Entry sym);

creates a *LaNode* of type **Expr** and state *Proto.* The number *name* of an identifier and the pointer *sym* to this identifier in the symbol table is stored in such a *LaNode.* This information is only used temporarily during parsing a function/procedure declaration by the parser itself.

static LaNode *set_fctdecl(int name, LaNode *body, LaNode *arg);

creates a *LaNode* of type **Fctdecl** representing a function/procedure declaration for a function/procedure with number *name* in the symbol table and formal parameters represented by *arg*. If the *body* of the function/procedure is NULL, this *LaNode* represents only a prototype, not really a declaration.

static LaNode *set_proccall(int name, LaNode *arg);

creates a *LaNode* of type **Proccall** representing a procedure call for a procedure with number *name* in the symbol table and arguments *arg.*

LaNode *change_RHS_in_assign(LaNode *expr);

changes the entry representing the RHS of an assignment to the LaNode *expr.*

LaExpr *ToLaExpr(int indices, int blobs);

converts a LaNode-subtree of type **Expr** in a correspondending *LaExpr,* if possible. To normalize the vector length in this *LaExpr*, the maximum number of indices and blobs are passed.

LaNode *link_node(LaNode *node);

connects the given LaNode *node* to the list of LaNodes, whose head is the *LaNode* the function link_node is applied to and returns a pointer to this *LaNode* again.

int LengthOf( );

returns the length of a LaNode list

void UpdateArguments(LaNode *arg);

this function is used by the parser object itself. If a function is introduced by a prototype and later in the source code a explicitly declaration of this function follows, then this element function is used to replace the argument entry by *arg.*

void InsertBody(LaNode *body);

inserts the loopbody *body* into a loop.

void SetParentTo(LaNode *parent_, int dip);

sets the entry parent of a LaNode to *parent_* recursive down in a LaNode tree. The user calls this function with dip=0 to avoid setting the parent entry in the root of the LaNode tree.

void SetStmntTo(int stmnt_);

sets the entry stmnt of a LaNode to *stmnt_* recursive down in a LaNode tree.

void	ReplaceIndices(Vect oldIndName, LaNode *newInd); replaces *LaNodes* of type **Expr** and state *Index* having one of the names given in *oldIndName* with the correspondending entries in *newInd* which is a list of *LaNode* trees.
void	Save(ostream &out); saves a whole LaNode tree to stream *out.*
void	Load(istream &in); loads a whole LaNode tree from stream *in*. The format of the input file is checked.
<type>	Get<entry>( ); returns the content of the element <entry> of a LaNode, e.g GetArguments( ), GetState( ) .
void	Set<entry>(<type> set); changes the content of the element <entry> in a LaNode to *set,* e.g. SetStep(LaNode*), SetStmnt(int).
int	GetLevel( ); returns the loop depth of a statement.
int	GetNotation( ); returns always FORTRAN, because no language detection is carried out by the parser object.
LaNode	*SearchLoop(int i); searchs for a loop with name *i*. If such a loop can be found, a pointer to this loop is returned, otherwise NULL is returned.
void	shrink( ); tries to reduce the size of a LaNode tree. This procedure traverses the parsetree and replaces subtrees by semantical equivalent *LaNode*-trees. For example, *LaNode*(TIMES , *LaNode*(Const, 1), *LaNode*(Index, i)) will be replaced by *LaNode*(Index, i).

1.4.1.2 The different types

LaNodes can be classified into several types, namely Assign, Forloop, If, Skip, Whileloop, Fctdecl, Proccall and Expr. Because the elements of class LaNode really used by the different *LaNodes* depend on its type, only the used entries are mentioned in the description of the different LaNode-types below. The contents of the unused entries are zero and are simply omitted for abbreviation and clearness.

1.4.1.2.1 Assign

typ	**Assign**
name	number of the left hand side variable in the symtab
object_id	identifier of the LaNode
target_stmnt	together with *stmnt* a more exact numbering is possible
stmnt	number of the statement
indices	pointer to the indices
state	*Write*
children	pointer to the first right hand side expression
next	pointer to the next program construct on the same hierarchy level
parent	pointer to the *LaNode* representing the next outer loop

1.4.1.2.2 Forloop

typ	**Forloop**
name	number of the index variable in the symtab
object_id	identifier of the LaNode
state	*Do* | *DoAll* | *DoAcross*
low	pointer to the lower bound
up	pointer to the upper bound
step	pointer to the loop stride
children	pointer to the loop body
next	pointer to the next program construct on the same hierarchy level
parent	pointer to the *LaNode* representing the next outer loop

1.4.1.2.3 If

typ	**If**
object_id	identifier of the LaNode
target_stmnt	together with *stmnt* a more exact numbering is possible
stmnt	number of the statement
cond	pointer to the condition
children	pointer to the then branch
elsechild	pointer to the else branch
next	pointer to the next program construct on the same hierarchy level
parent	pointer to the *LaNode* representing the next outer loop

1.4.1.2.4 Skip

typ	**Skip**
object_id	identifier of the LaNode
target_stmnt	together with *stmnt* a more exact numbering is possible
stmnt	number of the statement
next	pointer to the next program construct on the same hierarchy level
parent	pointer to the *LaNode* representing the next outer loop

1.4.1.2.5 Whileloop

typ	**Whileloop**
name	number of the (artificial) index in the symtab
object_id	identifier of the LaNode
state	*Do* | *DoAll* | *DoAcross*
cond	pointer to the condition
low	pointer to the lower bound
step	pointer to the loop stride
children	pointer to the loop body
next	pointer to the next program construct on the same hierarchy level
parent	pointer to the *LaNode* representing the next outer loop

1.4.1.2.6 Fctdecl

typ	**Fctdecl**
name	number of the function/procedure name in the symtab
object_id	identifier of the LaNode
state	0 | *Various* *Various* means that the number of arguments can change, cp. printf( ) in C
children	NULL, then it is only a prototyp otherwise pointer to the function body
arguments	pointer to the first formal parameter
next	pointer to the next function/procedure declared in the source code

1.4.1.2.7 Proccall

typ	**Proccall**
name	number of the procedure name in the symtab
object_id	identifier of the LaNode
target_stmnt	together with *stmnt* a more exact numbering is possible
stmnt	number of the statement
arguments	pointer to the first argument
next	pointer to the next program construct on the same hierarchy level
parent	pointer to the *LaNode* representing the next outer loop

1.4.1.2.8 Expr

typ	**Expr**
object_id	identifier of the LaNode
target_stmnt	together with *stmnt* a more exact numbering is possible
stmnt	number of the statement
laExpr	pointer to a *LaExpr*, if the LaNode-subtree rooted at this LaNode represents an lambda expression

next	pointer to the next *LaNode* **Expr** in an expression list, e.g. argument lists, indices lists
parent	pointer to the *LaNode* representing the next outer loop

The meaning of a *LaNode* of type **Expr** can be separated into some subcases. Therefore the description is refined once more. The entries above, whose meanings don't change, must be combined with the entries of one of the following cases to obtain a valid *LaNode* of type **Expr**.

1.	name	constant number
	state	*Const*
2.	name	number of the variable in the symtab
	state	*Index* | *Blob*
3.	name	number of the variable in the symtab
	state	*Read* (thus it is an array)
	indices	pointer to the indices of the array (a list of *LaNodes* of type **Expr**)
4.	name	number of the function name in the symtab
	state	*FunctionCall*
	arguments	pointer to the arguments of the function (a list of *LaNodes* of type **Expr**)
5.	name	0
	state	0
	op	BRACKET | PLUS | MINUS | TIMES | DIV | MOD | NOT | AND | OR
	children	pointer to the first operand
	elsechild	pointer to the second operand or NULL, if op is BRACKET or op is unary minus
6.	name	number of the variable in the symtab
	state	*Proto*
	symbol	pointer to the correspondending entry in the symtab this is only used temporarily while parsing function/procedure declaration and will not occur in the output written to disk

1.4.1.3 Note

You can get a good view of the parsetree by transforming in into an .info file. Let *tree.out* be the output containg the parsetree created by the NSP, then the call **ii** *tree.out tree.info* will create the correspondending info-file. You can find **ii** (**i**nto **i**nfo) in the ./scanpars2 directory of the LooPo distribution (perhaps you must create it by *gmake -f mkinf*). Then you can look at *tree.info* by an info reader, for example *axinfo* providing additionally a graphical view of the info-file and in this way of the parsetree.

1.4.2 LaExpr

A lambda expression represents an arith. expression of the form:

$$(\Sigma c_i * I_i + \Sigma d_k * B_k + C) / D$$

(with c_i, d_k, C and D are constants, I_i are indices and B_k are blobs)

Such an expression can be stored in an object of **class LaDep** (short *LaDep*). Because the indices get numbers beginning with one and these numbers are contigous, a vector for the indices can be created containing in the j-th component the constant multiplied with the index having the number j. In the 0-th component the length of the vector (i.e. the total number of the indices is stored). The same is valid for the blobs too.
After loading the parsetree from disk, the function *LaExprIn( )* is applied to the root of this tree to convert subtrees of type **Expr** into *LaExprs*, whenever it is possible.

1.4.2.1 Elements and functions

typedef int* Vect;

The class LaExpr (lamda expression) includes:

private components:

Vect	coef;	vector of index variable coefficients
int	c0;	constant
Vect	blobcoef;	vector of blob coefficients
int	denom;	denominator
LaExpr	*next;	pointer to the next *LaExpr* in an expression list

LaExpr *Mult(LaExpr *op);

multiplies a *LaExpr* with the LaExpr *op*. If the result is also a lambda expression, a pointer to a new created *LaExpr* containing this result is returned. Otherwise NULL is returned.

LaExpr *Div(LaExpr *op);

divides a *LaExpr* by the LaExpr *op*. If the result is also a lambda expression, a pointer to a new created *LaExpr* containing this result is returned. Otherwise NULL is returned.

LaExpr *Add(int sign, LaExpr *op);

adds to a *LaExpr* the LaExpr *op* multiplied with *sign* (1, -1) and returns a pointer to a new created *LaExpr* containing the result.

static LaExpr *Const_(int const);

creates a *LaExpr* representing the constant *const*.

static LaExpr *Index_(int name, int indices);

creates a *LaExpr* representing the the index with number *name* in the symbol table. *indices* is used to normalize the length of the index vector coef.

static LaExpr *Blob_(int name, int blobs);

creates a *LaExpr* representing the the blob with number *name* in the symbol table. *blobs* is used to normalize the length of the blob vector blobcoef.

public components:

~LaExpr( ); destructor
LaExpr( ); constructs a *LaExpr* and sets all entries to NULL, except of denom (= 1)
LaExpr(LaExpr *lexpr);
constructor creating a new list of *LaExprs* by copying the given list represented by *lexpr*.
LaExpr(Vect *coef, int c0, Vect *blobcoef , int denom);
constructor setting the entries coef, c0, blobcoef and denom.

LaNode *ToLaNode(int *coef_name);

creats a LaNode-tree of type **Expr** correspondending to the *LaExpr*. The name of an **Expr**-LaNode in this tree for an index with number i is coef_name[i], if coef_name is not NULL, otherwise i again.

LaExpr *Norm(int index, int blob);

If in a *LaExpr* the entry coef is NULL , a 0-vector of length *index* is created and replaces the entry coef, and if blobcoef is NULL, a 0-vector of length *blob* is created and replaces the entry blobcoef.

void	Print( );	displays a lambda expression on stdout
Vect	GetCoef( );	returns the coefficient vector
void	SetCoef(Vect coef_);	sets the coef vector to *coef_*
int	GetC0( );	returns the constant
void	SetC0(int const);	sets c0 to *const*
Vect	GetBlobCoef( );	returns the blob coefficient vector
void	SetBlobCoef(Vect blob_);	sets the blob vector to *blob_*
int	GetDenom( );	returns the denominator
void	SetDenom(int den);	sets the denominator to *den*
LaExpr	*GetNext( );	returns the next *LaExpr* in an expression list
void	SetNext(LaExpr *next_);	sets the next pointer to *next_*
void	Save(ostream &out);	saves a *LaExpr*-list to stream *out*
void	Load(istream &in);	loads a *LaExpr*-list from stream *in*

The next function is not a member function of class LaExpr, but it is used to create *LaExprs* and therefore is stated here.

void LaExprIn(LaNode *tr, int indices, int blobs)

converts LaNode-subtrees of type **Expr** within the tree *tr* into *LaExprs*, whenever it is possible. *indices* and *blobs* are used to normalize the length of the coef and blobcoef vectors of the *LaExprs*. Results of the conversion are stored in the correspondending *laExpr*-entries of the *LaNodes*.

1.4.3 LaDep

Objects of **class LaDep** (short *LaDeps*) are used to store dependencies. Each *LaDep* contains one dependency and the (user-defined) dependencies in the source program are organized as a *LaDep* list (if there are more than one).

The structure of a dependency entry in the source program is the following:

dep *dimension matrix* [**denom** *denominator*] *statements*

dimensions	is two integers to represent the number of rows and columns in the matrix.
matrix	is rows*columns integers representing the matrix for the function that determines the source of a dependence, i.e. where a variable was referenced the last time.
denominator	is an integer. This entry is optional and allows the use of rational matrices in this way.
statements	is two integers representing the source and the destination dependent statement

A short example:

```
CONSTANT m,n;
DO i = 1, n
    DO j = max( i, m ) , n
        A( i,j ) = A( i-1, j-2 ) + 5
    END DO
END DO
DO k = 1, m
    B(k) = 3 * k
END DO
```

The following dependency entry shows the dependence for the first statement. The first two integers tell that it is a 2 by 6 matrix. The matrix itself consists of three parts: indices, blobs and offset, e.g.

```
i  j  k  m n
1 0 0  0  0   -1
0 1 0  0  0   -2
indices  blobs  offset
```

The last two integers tell that the first statement is both, the source and the destination dependent statement.

```
dep 2 6
1 0 0 0 0 -1
0 1 0 0 0 -2 1 1
```

1.4.3.1 Elements and functions:

The class LaDep includes:

private components:

```
int      rows;      number of rows of the matrix
int      columns;   number of columns of the matrix
Vect     matrix;    has rows * columns entries and is stored in linear order
int      denom;     denominator of the matrix, to allow rational entries in this way
int      stmnt1;    first  dependent statement
int      stmnt2;    second dependent statement
LaDep*next;         pointer to the next dependency ( in a dependency list )
```

public components:

LaDep( );
creats an 'empty'*LaDep*, all entries are NULL, except the *denom* (= 1)

LaDep(int rows, int columns);
initializes a *LaDep* and allocates memory for the matrix, according to the number of *rows* and *columns*.

```
int    GetRows( );                  returns the number of rows of the matrix
int    GetColumns( );               returns the number of columns of the matrix
int    GetMatrixElem( int i, int j ); returns the matrix element at row i and column j
int    GetDenom( );                 returns the denominator of the matrix
int    GetStmnt1( );                returns the first dependent statement
int    GetStmnt2( );                returns the second dependent statement
int    GetNext( );                  returns a pointer to the next dependency
void   Save( ostream &out );        saves a list of dependencies to the stream out
void   Load( istream &in );         Loads a list of dependecies from the stream 'in'
```

1.4.4 SymbolTable

The **class SymbolTable** provides the necessary objects for the NSP to handle identifiers and to distinguish local variables in function/procedure declarations and global variables. Each function and procedure gets its own part of the symbol table, a so called **frame**. Such a **frame** is implemented as a binary search tree (not

balanced) and identifiers are inserted by their name (the string read by the scanner). An entry in that tree can be a list of nodes (**class node** is described below) to allow renaming of loop index variables. The numbering of indices and blobs is „frame-overlapping“, i.e. the numbers are simply incremented irrespective of the **frame** the identifier is inserted into.

1.4.4.1 Elements and functions

First the auxiliary class **node** is presented.

typedef class node *Entry;
The class node includes:

private components:

char	*lexptr;	pointer to the lexem string
int	basictype;	the type of an identifier, e.g. int, void if *type* is **Function**, basictype is the return-type of the function
int	token;	token of the entry
int	type;	Index | Blob | Array | Function
int	number;	the systematic number of the corresponing variable
int	indices;	number of indices of the corresponding variable, if type is **Array**
int	level;	level of the loop of an index variable
int	valid_;	used to avoid nested loops with the same index variable
int	intent;	IN | OUT | INOUT
Entry	next;	pointer to replacement node to allow renaming of a loop index variable
Entry	leftchild;	left child in the binary tree
Entry	rightchild;	right child in the binary tree

public components:

	~node( );	destructor
	node( );	initialization function
char	*LexemOf( );	returns a pointer to the lexem string
int	TokenOf( );	returns the token
int	TypeOf( );	returns the type
int	NumberOf( );	returns the number
int	IndicesOf( );	returns the number of indices of a array variable
int	LevelOf(int n);	returns the level of an index variable with number *n*
Entry	NextOf( );	returns the replacement node
void	SetIndices(int n);	sets the number of indices of a array variable to *n*
void	SetNext(int nu, int le);	adds a replacement for an identifier. The new number and level are *nu* and *le*.
void	Update(int tok, int typ, int nu, int le);	updates the entries token, type, number and level.
void	Save(ostream &out);	saves a node to stream *out.*
void	valid( );	sets the entry *valid_* to avoid nested loops with the same index varible
void	invalid( );	resets the entry *valid_* => the name (lexptr) of the index variable can be used again in a following loop
int	isvalid( );	indicates wheather the name (lexptr) of the index variable is allowed to be used
void	SetIntent(int i);	sets the entry *intent* to i (IN | OUT | INOUT)
void	SetBasictype(int ba);	sets the entry *basictype* type to *ba,* e.g. int, void
int	GetBasictype( );	returns the content of *basictype*
int	GetIntent( );	returns the content of *intent*

Now the class **SymbolTable** is presented.

#define StrMax 999
The class SymbolTable includes:

private components:

Entry	root;	root entry of the binary tree
framelist	list;	contains a list of different frames (the global frame is 1, each defined procedure and function get a own frame, in which their variable-indentifiers are stored)
char	lexemes[StrMax];	memory area for the lexeme strings of size StrMax
int	lastchar;	pointer to the last used field in lexemes (lexemes[lastchar])
Entry	FindHelp(int i, int t, Entry x);	help function for the Find function
Entry	LoadHelp(istream &in, Entry y, int fr);	help function for the Load function

public components:

	SymbolTable( );	constructor for an empty symbol table

void RemoveFrame(int fr); removes the frame *fr* in the framelist *list*
void RenameFrame(int old, int new_); renames the frame *old* to *new_*
Entry Insert(char *s, int tok, int fr);
inserts a new entry with name *s* and token *tok* into frame *fr.* A pointer to this entry is returned.
Entry Retrieve(char *s, int fr);
searchs for an entry with name *s* in the frame *fr.* If an entry could be found, a pointer to this entry is returned, otherwise NULL is returned.
Entry Find(int nu, int type);
searches for an entry with number *nu* and type *type* in all frames. If such an entry is found (there can be only one entry, because the numbering is „frame-overlapping“), a pointer to this entry is returned, otherwise NULL is returned.
void Save(ostream &out); saves the symbol table to stream *out*
void Load(istream &in); loads a symbol table from stream *in*
void SetNewSymTab(SymbolTable &symtab); sets the framelist *list* to the one given by *symtab.*

1.4.5 loopoParser

The class loopoParser generated by bison++ using the grammar.y file is presented next.

```
#define GLOBAL   1

class loopoParser
{
    enum        YY_loopoParser_ENUM_TOKEN { . . . }
  public:
    int         yyparse( void );        // the parse function
    virtual void  yyerror( char *msg );
    virtual int   yylex( void );
    yy_loopoParser_stype yylval;        // it's an union-type for the parser stack ( created by the bison++ )
    int         yynerrs;                // counts the number of syntax errors ( used internal )
    int         yychar;                 // current token received from the scanner

    // the following functions are documented after this class description
    void        read_matrix( LaDep *dep );
    void        def_blob( char *s );
    LaNode      *array( char *s, int tok, LaNode *l );
    int         number_of( int n );
    void        func_end( LaNode *fun, LaNode *body, LaNode *arg );
    void        c_prototype( int type, char *s, LaNode *arg );
    Entry       argument( char *s, int tok, int len, int frame_ );
    int         index( char *s );
    int         index_( char *s );
    LaNode      *c_function( int type, char *s, LaNode *arg );
    LaNode      *array_index_fctcall( char *s, LaNode *l, int r );
    int         c_loop( int index1, int index2, int op, int index3, int incr );
    void        functionCall( Entry symentry, LaNode *argument );
    void        f_proto( LaNode *arg );
    int         f_proto_func( char *s );
    LaNode      *procedure( char *s, LaNode *expr );

                loopoParser( )
                {
                    frame( GLOBAL ),  Functions(NULL),   Parsetree(NULL),
                    Dep(NULL),        level(0),          succeed(1),
                    funcVSarray(0),   number_index(0),   number_array(0),
                    number_blob(0),   number_proto(0),   number_function(0),
                    statement(0),     lineno(1)
                }
```

```
SymbolTable     symtab;           // symbol table of the parser
    int         frame;            // number of the frame identfiers will be inserted into
    LaNode      *Functions;       // pointer to the first function in the function list
    LaNode      *Parsetree;       // pointer to the root of the parsetree
    LaDep       *Dep;             // pointer to the first element in the dependency list
    int         level;            // shows the current loop depth
    int         succeed;          // signals, if the the parse process was seccessful
    int         funcVSarray;      // controls the detection of fuctions and arrays of the form a(...)
    int         number_index;     // current number of indices detected while parsing
    int         number_array;     // current number of arrays detected while parsing
    int         number_blob;      // current number of blobs detected while parsing
    int         number_proto;     // number of formal parameters
    int         number_function;  // current number of functions detected while parsing
    int         token;            // current token returned by yylex( )
    int         statement;        // current number of statements parsed so far
    int         lineno;           // current line number
    RING        states;           // a kind of ring buffer used to store the parser states
}
```

void loopoParser::read_matrix(LaDep *dep)

reads the denominator and the matrix of a dependency and stores its entries in the given LaDep object pointed to by 'dep'.

void loopoParser::def_blob(char *s)

inserts the variable with name 's' into the symbol table as a constant.

LaNode *loopoParser::array(char *s, int tok, LaNode *indices)

inserts the variable with name 's' into the symbol table as an array and returns the pointer to a new created LaNode of type **Assign** correspondending to this array and its indices ‘indices’.

int loopoParser::number_of(int type)

returns the number of variables with type 'type' detected so far while parsing. Additionally the number of statements can be counted here too.

void loopoParser::functionCall(Entry symentry, LaNode *argument)

tests whether the variable given with 'symentry' is a function and if so, whether the number of arguments is correct.

LaNode *loopoParser::array_index_fctcall(char *s, LaNode *exprlist, int rule)

If the variable with name 's' is not in the symbol table yet, it will be inserted as array, index, blob or function dependent on the argument 'rule'. Otherwise the information in the symbol table is used to control whether the variable is applied correctly (semantic check). The argument 'exprlist' is only used, if 's' is an array or function.

int loopoParser::index(char *s)

is only used with c-forloops and tests whether 's' is an index and (if so) whether 's' is already used in a nesting loop. If 's' is not in the symbol table yet, 's' will be inserted as index. The number of the index with name ‘s’ will be returned as result.

int loopoParser::index_(char *s)

is only used with c-forloops and is similar to the function 'loopoParser::index( )' above.

int loopoParser::c_loop(int index1, int index2, int op, int index3, int incr)

tests whether the number of the indices 'index1', 'index2' and 'index3' are the same and whether the increment 'incr' fit to the upper bound of the loop.

Entry loopoParser::argument(char *s, int tok, int len, int frame_)

inserts a formal parameter with name ‘s’ into the symbol table frame ‘frame’ while parsing a function or procedure declaration. ‘len’ is the number of indices, if the parameter is an array. Double use of a parameter in a parameter list is detected.

void loopoParser::c_prototype(int type, char *s, LaNode *arg)
LaNode* loopoParser::c_function(int type, char *s, LaNode *arg)

create a LaNode of type Fctdecl for a function or procedure with name ‘s’, return-type ‘type’ and formal parameters ‘arg’. If a prototype for this function already exists, a consistence check is

carried out. Otherwise it will be appended to the function list. (the loopoParser element 'Functions' points to the first function in this list).

LaNode *loopoParser::procedure(char *s, LaNode *argument)

creates a LaNode of type Proccall for a procedure with name 's' and actual parameters 'arg'. If there is no such procedure in the funcction list, a correspondending entry will be created and inserted.

void loopoParser::func_end(LaNode *fun, LaNode *body, LaNode *arg)

completes a LaNode of type Fctdecl for a function or procedure. It test, whether all formal parameters were really used within the function body and transfers information stored so far only in the symbol table into the argument-LaNodes 'arg'.

void loopoParser::f_proto(LaNode *arg)

test, whether all formal parameters in a FORTRAN function or procedure declaration were defined. If not, the formal parameter is assumed to be an integer used for input and output.

int loopoParser::f_proto_func(char *s)

looks for a function with name 's' in the symbol table. If such a function is found, the correspondending return-type, like INTEGER, is returned, otherwise this function will be inserted into the symbol table having then a return-type INTEGER by default. This function is called after parsing a FORTRAN function declaration or prototype.

1.4.6 Parser

The class Parser inherited from the class loopoParser is shown in this section.

```
class Parser : public loopoParser
{
  loopoScanner  Scanner;     // the LooPo scanner object
  LaNode        *init_function(char *name_str, int name_int, int ret_type);
                             // auxiliary function to initialize the functions min, max, ceil, floor and return
  RING          tokens;      // a ring buffer which stores the received tokens
 public:
  void    init();                          // function to initialize the functions min, max, ceil, floor and return
  void    ScannerInit() { Scanner.init(); }
          /* resets the Scanner to call the parser again with a new source program. It must be called after
          SetInputStream(.). */
  int     yylex();                         // returns the next token read by the scanner
  void    SetInputStream(istream &in);     // forces the scanner to read from the given stream in (default is cin).
  void    yyerror(char *s);                // called if the parser detects a syntax error
  void    Load(istream &in);               // loads the symbol table, parse tree and defined functions
  void    Save(ostream &out);              // saves the symbol table, parse tree and defined functions
  LaNode  *GetRoot();                      // returns the root of the parsetree
  LaDep   *GetDeps();                      // returns the beginning of the dependency list
  int     GetNumInd( );                    // returns the number of indices found
  int     GetNumBlob( );                   // returns the number of blobs found
  int     GetNumStmnt( );                  // returns the number of statements found
  int     GetNumArray( );                  // returns the number of arrays found
  void    SetRoot(LaNode *parsetree);      // sets a new parse tree in the Parser object
  void    SetDep(LaDep *dependence);       // sets new dependecies in the Parser object
  void    SetVar(int n_coef);              // sets the number of indices to n_coef.
  void    SetParam(int n_blob);            // sets the number of blobs to n_blob.
  void    SetNumStmnt( int n_stat );       // sets the number of statements to n_stat.
  void    SetNumArray( int n_array );      // sets the number of arrays to n_array.
};
```

1.5 Grammar

The grammar of the new source language of the NSP is presented in this section. It is almost the same used for bison++, only a few changes are made to enhance the legibiltity and clearness. Of course, all *semantical actions* attached to the rules are omitted. Further there is no distinction between different kinds of identifiers, only **ID** is used. The tokens **IDPAR**, **IDCOM** and **IDID** in the orginal bison++ file were only introduced into the grammar to avoid shift/reduce conflicts. Additionally some *epsilon*-rules were deleted which were required only to invoke *semantical actions* attached to these rules.

```
program         : definitions statements end_program
                | statements end_program
                | statements definitions
                | definitions end_program
                | definitions statements definitions
                | end_program
                ;
end_program     : dep definitions
                | dep
                |
                ;
definitions     : definitions definition
                | definition
                ;
definition      : c_prototype
                | f_prototype
                | c_function
                | f_function
                ;
c_prototype     : c_type ID '(' c_parameters ')' ';'
                | ID '(' c_parameters ')' ';'
                ;
c_parameters    : c_para_list
                | NOTYPE
                ;
c_para_list     : c_para_list ',' c_para
                | c_para
                ;
c_para          : intent ID argument
                | intent argument
                ;
argument        : ID
                | ID '[' ']'
                | ID '[' numlist ']'
                ;
c_function      : c_type ID '(' c_parameters ')' c_fct_body
                | ID '(' c_parameters ')' c_fct_body
                ;
c_type          : ID
                | NOTYPE
                ;
c_fct_body      : '{' lokals c_fct_stats '}'
                ;
c_fct_stats     : c_fct_stats statement
                | c_fct_stats RETURN expr ';'
                | c_fct_stats RETURN ';'
                |
                ;
lokals          : lokals list_of_lokals ';'
                |
                ;
list_of_lokals  : ID arg_list
                ;
```

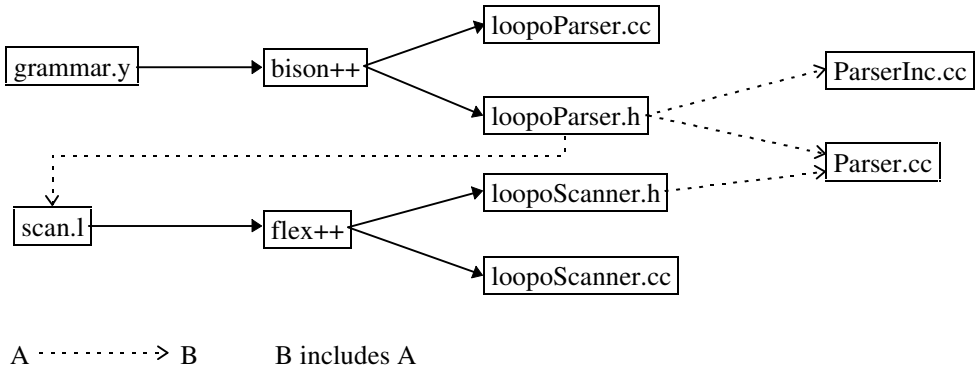
```
arg_list        : arg_list ',' arg
                | arg
                ;
arg             : ID
                | ID '[' numlist ']'
                ;
numlist         : numlist ',' NUM
                | NUM
                ;
f_prototype     : SUBROUTINE fctname '(' f_parameters ')'
                    para_def_io
                  END SUBROUTINE fctname
                | FUNCTION   fctname '(' f_parameters ')'
                    para_def
                  END FUNCTION fctname
                ;
f_parameters    : f_para_list
                |
                ;
f_para_list     : f_para_list ',' ID
                | ID
                ;
para_def        : para_def ID f_arg_list
                |
                ;
para_def_io     : para_def_io def_io
                |
                ;
def_io          : ID ',' INTENT '(' intent ')' DOPPELDOPPEL f_arg_list
                | ID f_arg_list
                ;
f_arg_list      : f_arg_list ',' f_arg
                | f_arg
                ;
intent          : IN
                | OUT
                | INOUT
                ;
f_arg           : ID
                | ID '(' numlist ')'
                | arg
                ;
f_function      : SUBROUTINE fctname '(' f_parameters ')'
                    para_def_io   statements
                  END SUBROUTINE fctname
                | FUNCTION fctname   '(' f_parameters ')'
                    para_def   statements
                  END FUNCTION fctname
                ;
fctname         : ID
                ;
statements      : statements statement
                | statement
                ;
bloblist        : bloblist ',' ID
                | ID
                ;
```

```
statement       : skip
                | assignment
                | proccall
                | loop
                | CONSTANT bloblist ';'
                ;
skip            : SKIP
                | SKIP ';'
                ;
assignment      : array EQ expr
                | array EQ expr ';'
                | array ASSIGN expr ';'
array           : ID
                | ID '[' exprlist ']'
                | ID '(' exprlist ')'
                ;
proccall        : CALL procedure
                | procedure ';'
                | procedure
                ;
procedure       : ID '(' exprlist ')'
                | ID '(' ')'
                ;
c_body          : '{' statements '}'
                | '{' '}'
                | statement
                ;
index           : ID
                ;
index_          : ID
                ;
op              : LE | LEQ | GE | GEQ | EQ | EQEQ
                ;
loop            : c_loop
                | f_loop
                | s1_loop
                | s2_loop
                ;
c_loop          : FOR '(' index EQ expr ';' index_ op expr ';' c_step ')'  c_body
                | C_WHILE '(' cond ')'  c_body
                ;
c_step          : index_ PLUSPLUS
                | index_ MINUSMINUS
                | index_ EQ index_ PLUS  NUM
                | index_ EQ index_ MINUS NUM
                ;
f_loop          : DO index EQ expr ',' expr f_step f_s1_body END DO
                | DO WHILE '(' cond ')' f_s1_body END DO
                | DO index EQ expr WHILE '(' cond ')' f_s1_body END DO
                ;
f_step          : ',' sign NUM
                |
                ;
sign            : PLUS
                | MINUS
                |
                ;
f_s1_body       : statements
                |
                ;
```

```
s2_loop          : FOR index EQ expr LEFT expr s2_step statements
                 ;
s2_step          : RIGHT sign NUM
                 |
                 ;
s1_loop          : FOR index ASSIGN expr TO expr s1_step DOs f_s1_body OD
                 ;
s1_step          : STEP sign NUM
                 |
                 ;
exprlist         : exprlist ',' expr
                 | expr
                 ;
expr             : '(' expr ')'
                 | expr PLUS  expr
                 | expr MINUS expr
                 | expr TIMES expr
                 | expr  DIV  expr
                 | expr  MOD  expr
                 | PLUS  expr
                 | MINUS expr
                 | array_index_fctcall
                 | NUM
                 ;
array_index_fctcall
                 : ID '[' exprlist ']'
                 | ID '(' exprlist ')'
                 | ID '(' ')'
                 | ID
                 ;
dep              : dep DEP NUM NUM matrix NUM NUM
                 | DEP NUM NUM matrix NUM NUM
                 ;
matrix           |
                 ;
cond             : '(' cond ')'
                 | expr
                 | cond OR  cond
                 | cond AND cond
                 | NOT cond
                 | compare
                 ;
compare          : expr op expr
                 ;
```

1.6 The NSP and the tools bison++ and flex++

The NSP is a LALR(1) parser generated by the compiler-compiler bison++. To read characters from the input stream a scanner generated by flex++ is used. Both tools create C++ classes for the parser and scanner respectively. The files *grammar.y*, *scan.l*, *ParserInc.cc* and *Parser.cc* contain the description of the NSP. The grammar of the new source language is in *grammar.y* and the definition of the scanner belonging to this language is in *scan.l*. Most of the element functions of class loopoParser generated by bison++ were put into *ParserInc.cc*. In *Parser.h* and *Parser.cc* you can find the definition of the *class Parser*. Note, that the NSP (or to be more exact the *class Parser*) is inherited from the *class loopoParser*. Within the *class Parser* there is an element of type *class loopoScanner* used to get the token stream.



1.6.1 Note

The bison++ from a bison++ distribution and the bison++ from a bison distribution are different, though the names are the same. We need a bison++ from a bison++ distribution, otherwise the file *grammar.y* can't be parsed. The same is valid for flex++ analogously. Therefore the LooPo distribution contains the files loopoParser.cc, loopoParser.h and loopoScanner.cc, loopoScanner.h, which would be created by bison++ and flex++ respectively.

1.7 An Example:

1.7.1 Source program:

```
SUBROUTINE exit( n )
  INTEGER n
END SUBROUTINE exit

CONSTANT m,n;
DO i = 1, min( 10, m )
     DO j = 1, i + n + 5
          a( i, j ) = a( i-1, j ) + a( i-2, j-2 ) + c
     END DO
END DO
exit(1)

dep 2 5
1 0 0 0 -1
0 1 0 0  0
1 1

dep 2 5
1 0 0 0 -2
0 1 0 0 -2
1 1
```

1.7.2 SymbolTable:

Frame 1

lexem	min
token	262
type	Funct.
number	-1
indices	-1
level	0
intent	0
basictype	INT
left child / next / right child	

lexem	max
token	262
type	Funct.
number	-2
indices	-1
level	0
intent	0
basictype	INT
left child / next / right child	

lexem	return
token	262
type	Funct.
number	-5
indices	-1
level	0
intent	0
basictype	VOID
left child / next / right child	

lexem	ceil
token	262
type	Funct.
number	-3
indices	-1
level	0
intent	0
basictype	INT
left child / next / right child	

lexem	n
token	259
type	Blob
number	2
indices	-1
level	0
intent	0
basictype	0
left child / next / right child	

lexem	a
token	262
type	Array
number	1
indices	2
level	0
intent	0
basictype	0
left child / next / right child	

lexem	floor
token	262
type	Funct.
number	-4
indices	-1
level	0
intent	0
basictype	INT
left child / next / right child	

lexem	c
token	262
type	Array
number	2
indices	0
level	0
intent	0
basictype	0
left child / next / right child	

lexem	exit
token	262
type	Funct.
number	1
indices	-1
level	0
intent	0
basictype	VOID
left child / next / right child	

lexem	m
token	259
type	Blob
number	1
indices	-1
level	0
intent	0
basictype	0
left child / next / right child	

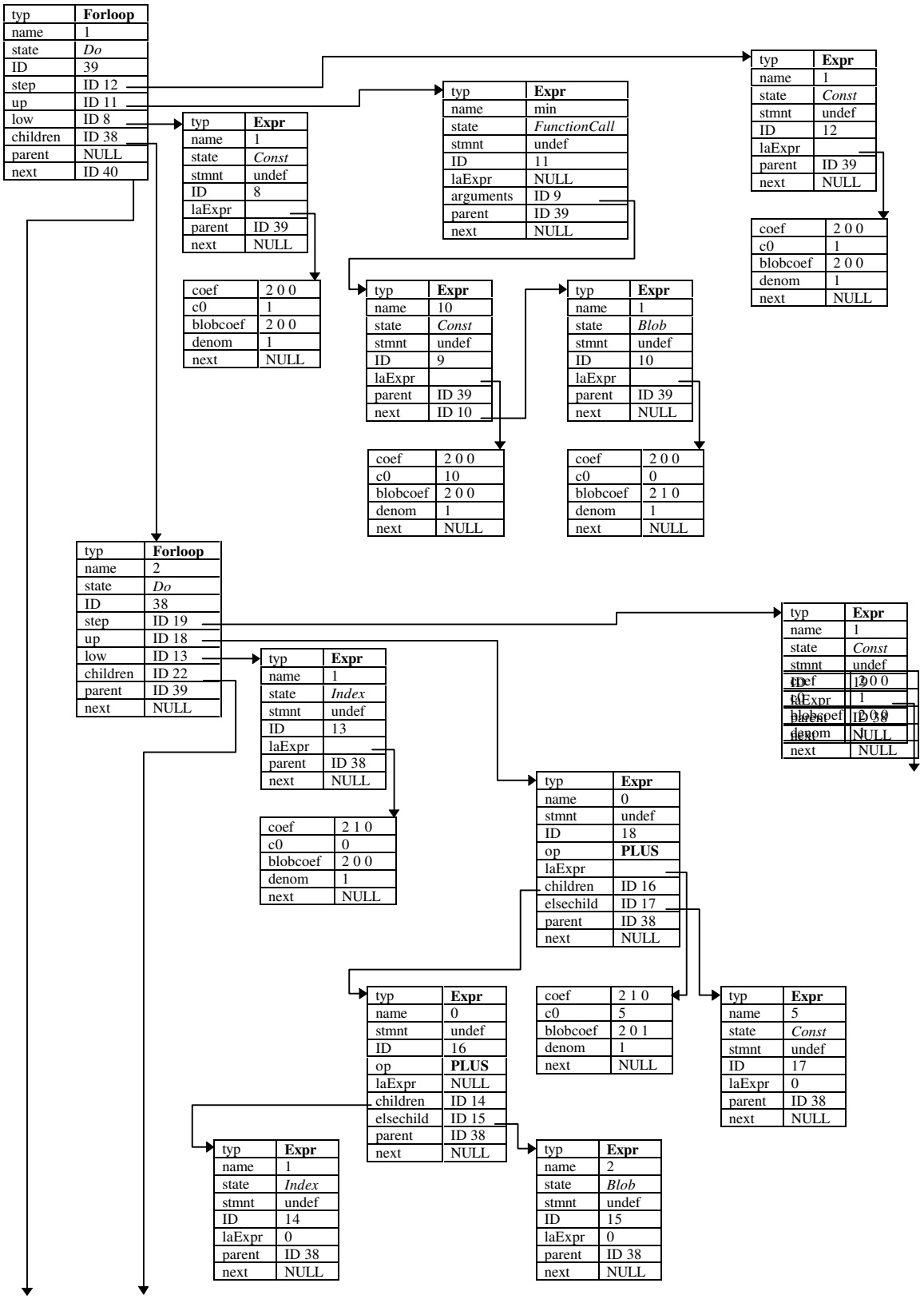
lexem	i
token	259
type	Index
number	1
indices	-1
level	0
intent	0
basictype	0
left child / next / right child	

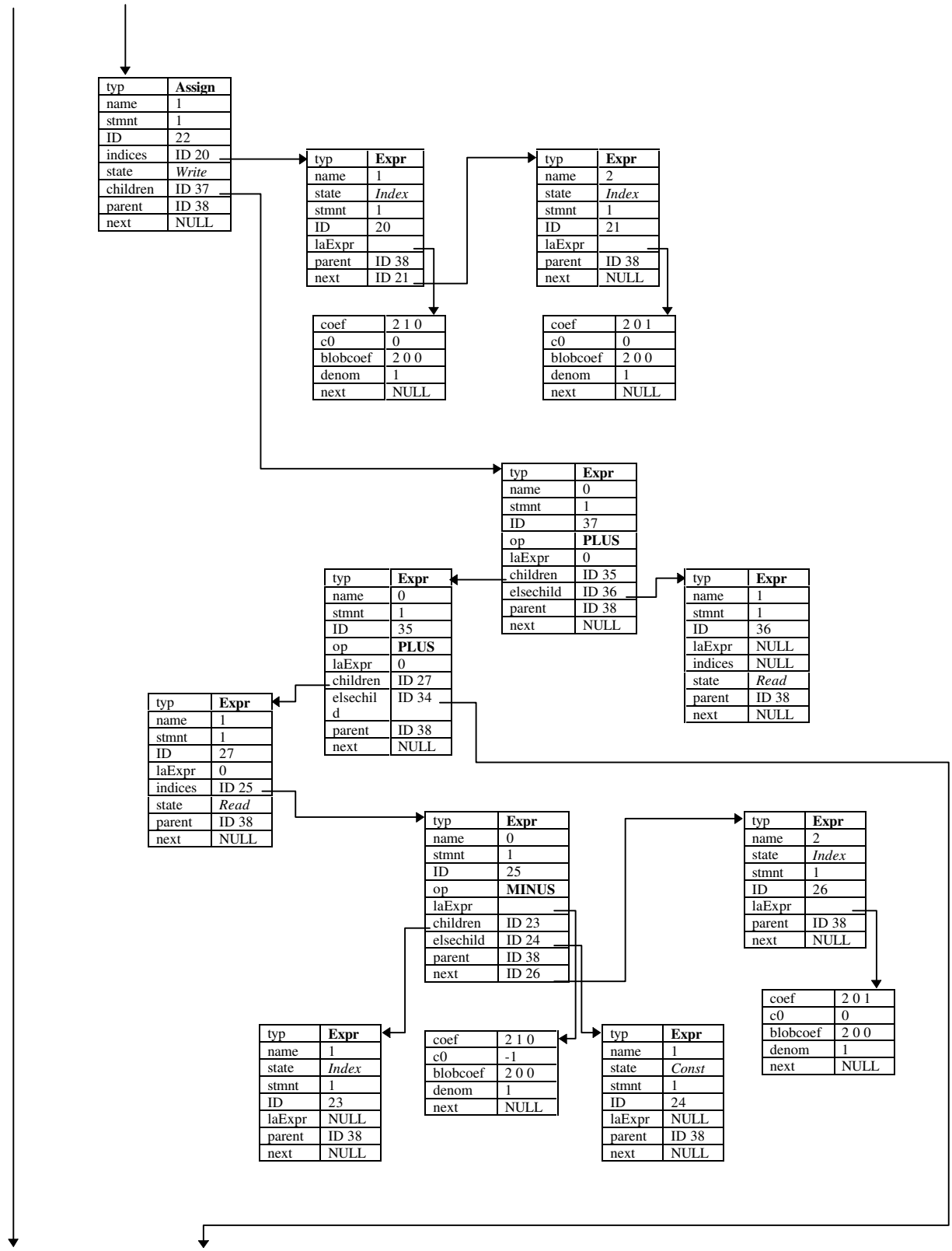
lexem	j
token	259
type	Index
number	2
indices	-1
level	1
intent	0
basictype	0
left child / next / right child	

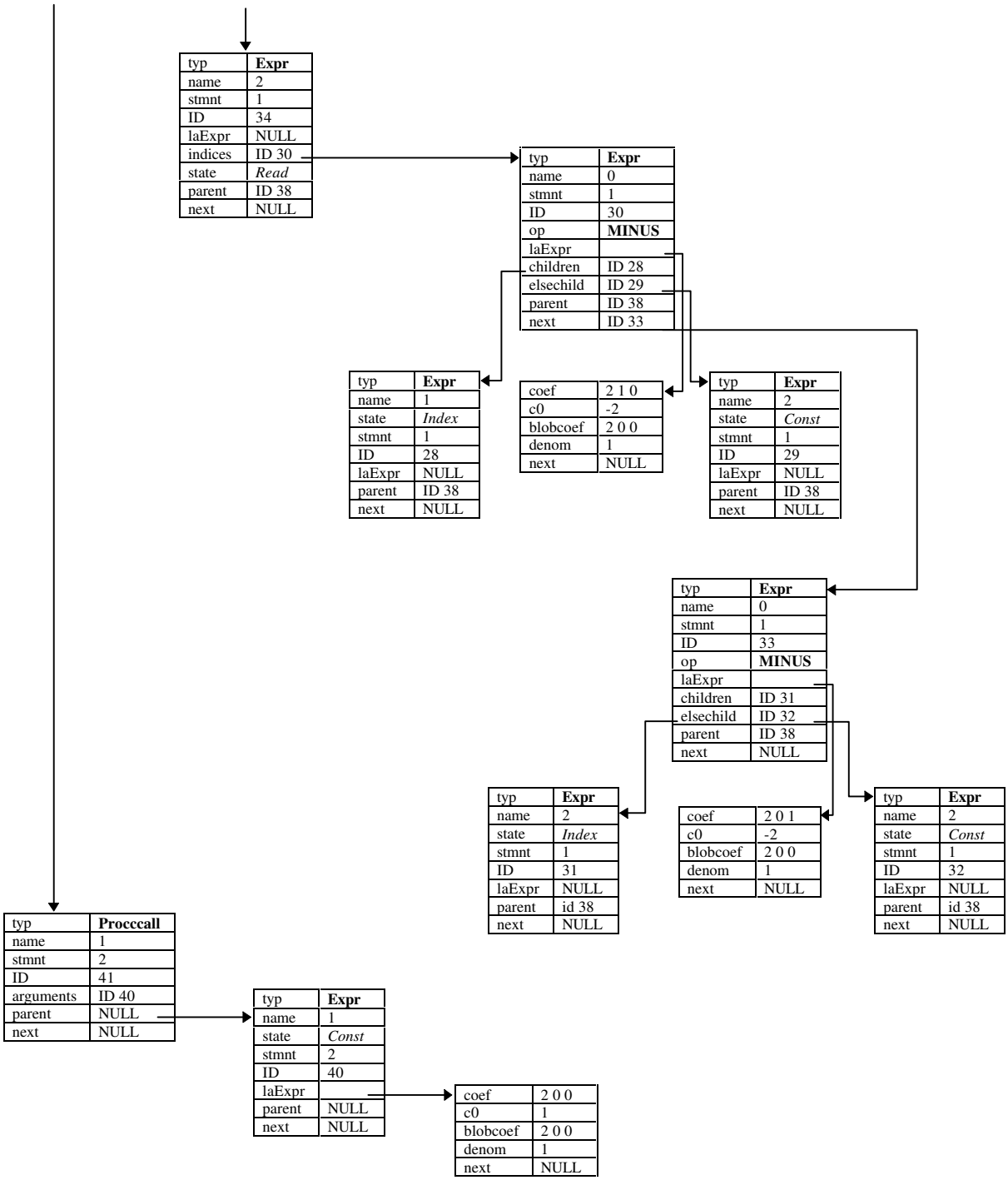
Frame 2

lexem	n
token	261
type	0
number	1
indices	0
level	0
intent	0
basictype	INT
left child / next / right child	

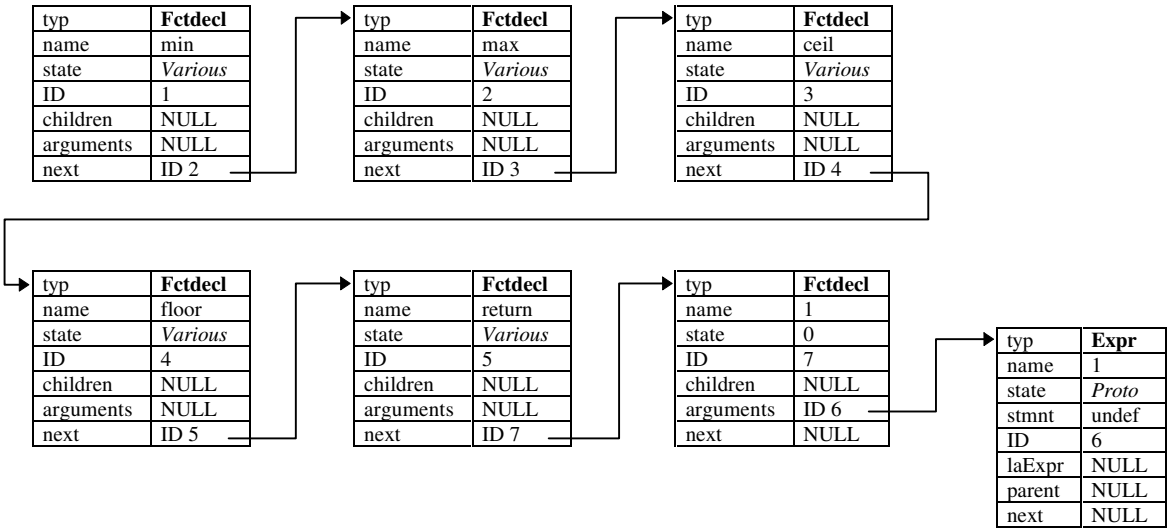
1.7.3 MainProgram:



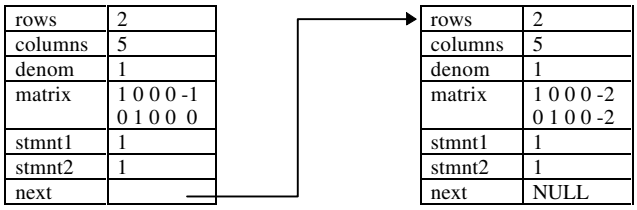

typ Assign
name 1
stmnt 1
ID 22
indices ID 20
state Write
children ID 37
parent ID 38
next NULL
typ Expr
name 1
state Index
stmnt 1
ID 20
laExpr
parent ID 38
next ID 21
typ Expr
name 2
state Index
stmnt 1
ID 21
laExpr
parent ID 38
next NULL
coef 2 1 0
c0 0
blobcoef 2 0 0
denom 1
next NULL
coef 2 0 1
c0 0
blobcoef 2 0 0
denom 1
next NULL
typ Expr
name 0
stmnt 1
ID 37
op PLUS
laExpr 0
children ID 35
elsechild ID 36
parent ID 38
next NULL
typ Expr
name 0
stmnt 1
ID 35
op PLUS
laExpr 0
children ID 27
elsechil d ID 34
parent ID 38
next NULL
typ Expr
name 1
stmnt 1
ID 36
laExpr NULL
indices NULL
state Read
parent ID 38
next NULL
typ Expr
name 1
stmnt 1
ID 27
laExpr 0
indices ID 25
state Read
parent ID 38
next NULL
typ Expr
name 0
stmnt 1
ID 25
op MINUS
laExpr
children ID 23
elsechild ID 24
parent ID 38
next ID 26
typ Expr
name 2
state Index
stmnt 1
ID 26
laExpr
parent ID 38
next NULL
coef 2 0 1
c0 0
blobcoef 2 0 0
denom 1
next NULL
typ Expr
name 1
state Index
stmnt 1
ID 23
laExpr NULL
parent ID 38
next NULL
coef 2 1 0
c0 -1
blobcoef 2 0 0
denom 1
next NULL
typ Expr
name 1
state Const
stmnt 1
ID 24
laExpr NULL
parent ID 38
next NULL


typ Expr
name 2
stmnt 1
ID 34
laExpr NULL
indices ID 30
state Read
parent ID 38
next NULL
typ Expr
name 0
stmnt 1
ID 30
op MINUS
laExpr
children ID 28
elsechild ID 29
parent ID 38
next ID 33
typ Expr
name 1
state Index
stmnt 1
ID 28
laExpr NULL
parent ID 38
next NULL
coef 2 1 0
c0 -2
blobcoef 2 0 0
denom 1
next NULL
typ Expr
name 2
state Const
stmnt 1
ID 29
laExpr NULL
parent ID 38
next NULL
typ Expr
name 0
stmnt 1
ID 33
op MINUS
laExpr
children ID 31
elsechild ID 32
parent ID 38
next NULL
typ Expr
name 2
state Index
stmnt 1
ID 31
laExpr NULL
parent id 38
next NULL
coef 2 0 1
c0 -2
blobcoef 2 0 0
denom 1
next NULL
typ Expr
name 2
state Const
stmnt 1
ID 32
laExpr NULL
parent id 38
next NULL
typ Procccall
name 1
stmnt 2
ID 41
arguments ID 40
parent NULL
next NULL
typ Expr
name 1
state Const
stmnt 2
ID 40
laExpr
parent NULL
next NULL
coef 2 0 0
c0 1
blobcoef 2 0 0
denom 1
next NULL

1.7.4 Functions:



1.7.5 Dependencies:



1.7.6 Statistics:

number of indices 2
number of blobs 2
number of stmnts 2
number of arrays 2
last LaNode ID 41