The LooPo scanner and parser

Frank Schuler
Spitalhofstr. 22a
94032 Passau
Phone (08 51) 75 27 93
e-Mail schuler@fmi.uni-passau.de

January 12, 1995

Abstract

The LooPo scanner and parser converts a source program containing loops, statements and dependencies into a tree and a list structure to be used by loop transformation algorithms.

Contents

1 Introduction

1.1 Purpose

The LooPo scanner and parser is a part of the LooPo loop parallelization tool. The scanner and parser converts a source program containing possibly different notations of loops, statements and dependencies. The loops and statements are converted into a parse tree while the dependencies are converted into a list. These structures are provided to the following phases of the loop parallelization process. Therefore the user can input a source program into the loop parallelization tool the way it is, i. e. in *C*, *FORTRAN* or some more notations.

1.2 Invocation

As mentioned above the LooPo scanner and parser is embedded in the LooPo environment. Therefore it is called via the LooPo shell by pressing the *Scanner/Parser* bottom. Since the whole LooPo tool runs under a consistent graphical user interface it is not possible to start the scanner and parser from the command line.

2 The source language

A source program contains two major parts. These are the loops with statements part and the dependence part. Thereby loops can be imperfectly nested but the stepwidth is ignored so far and can be omited.

2.1 The loops and statements part

The following four notations of loops are available:

1. The *C* notation, e. g.

   ```
   for (i = 0; i < n; i++) {
     }
   ```

2. The new *FORTRAN* notation (not *FORTRAN 66*), e. g.

   ```
   DO i = 0, n, 1
   END DO
   ```

3. The *"Systoliker 1"* notation, e. g.

   ```
   for i = 0 <- n -> 1
   ```

   Since a deliminator for the loop body is missing with this notation, only perfectly nested loops are provided.

4. The *"Systoliker 2"* notation, e. g.

   ```
   for i := 0 to n step 1 do
   od
   ```

Instead of simple expressions for the loop boundaries, minima and maxima are available, e. g.

```
DO i = max (1, n), min(m, l, 100)
END DO
```

A formal description of the input language is given in section 2.4.

2.2 The dependence part

The dependence part is a feature for the user to add artificially some dependencies, e. g. to manipulate the possible sets of transformations without having to modify the loops and statements part of the source program. Furthermore, this is the only way to handle dependencies in the parallelization process, as long as the dependence analysis is not yet implemented. The structure of a dependency entry is the following:

`dep` *dimension matrix statements*

- *dimension* is two integers to represent the number of rows and columns in the matrix.
- *matrix* is *rows* $\cdot$ *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.
- *statements* is two integers representing the source and the destination dependent statement.

2.3 An example source program

/* This is the loops with statements part. */

```
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
```

/* This dependence part 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 the three parts: indices, blobs (also called structure parameters or size parameters) and offset, e. g.

$$\begin{array}{ccccccc} i & j & k & n & m & & \\ 1 & 0 & 0 & 0 & 0 & -1 & . \\ \underbrace{0 \quad 1 \quad 0}_{indices} & & & \underbrace{0 \quad 0}_{blobs} & & \underbrace{-2}_{offset} & \end{array}$$

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
```

2.4 A formal definition of the source language

```
Prog      →  Loop { Dep }.
Dep       →  'dep' Num Num { Num } Num Num.
Loop      →  Stmnt | FLoop | CLoop | S1Loop | S2Loop.
Stmnt     →  ArrayId ( '=' | ':=' ) ( Num | ArrayId | IndexId)
             { ( '+' | '-' | '*' | '/' ) ( Num | ArrayId | IndexId) } Loop.
FLoop     →  'DO' IndexId ' =' Limit ',' Limit [ ',' Num ]
             Loop 'END DO' Loop.
CLoop     →  'for (' IndexId ' =' Limit ';' IndexId '<' Limit ';'
             IndexId ( '++' | '=' IndexId '+' Num) ') {' Loop '}' Loop.
S1Loop    →  'for' IndexId ' =' Limit '<-' Limit '->' Num Loop.
S2Loop    →  'for' IndexId ' :=' Limit 'to' Limit [ 'step' Num ]
             'do' Loop 'od' Loop.
ArrayId   →  Id [ '[' ExprList ']' | '(' ExprList ')' ].
IndexId   →  Id.
Limit     →  ( 'min' | 'max' ) '(' ExprList ')'.
ExprList  →  Expr { ',' Expr }.
Expr      →  Term | '(' Term ') /' Num.
Term      →  [ '-' ] Factor { ( '+' | '-' ) Factor }.
Factor    →  [ '-' | '+' ] ( Num | Num '*' Id | Id ).
Id        →  ( 'a' ... 'z' | 'A' ... 'Z' ) { 'a' ... 'z' | 'A' ... 'Z' | '0' ... '9' }.
Num       →  '0' ... '9' { '0' ... '9' }.
```

3 The implementation

In order to allow easy maintenance the LooPo scanner and parser is divided into four major and two minor modules. These are:

- major modules
  - main module
  - symbol table
  - scanner
  - parser
- minor modules
  - initialization module
  - error module

Some of the used data structures and methods to decompose loops and statements are based on those described in *The Lamda Loop Transformation Toolkit (User's Reference Manual)* [1]. Since these structures and methods were not powerful enough for the given problem they were extended in some respects and translated from *C* to *C++*.

The symbol table is based on an implementation and evaluation of binary search trees described in [2]. As above, the structures were extended and translated from *Modula–2* to *C++*.

Finally, the scanner and parser itself is based on principles described in [3].

3.1 The symbol table

The symbol table provides the necessary objects for the scanner to handle identifiers. Its definition and implementation files are `symtab.h` and `symtab.cc` respectively.

3.1.1 The definition

Two types are defined. `Entry` is a pointer to a `node` object and `SymbolTable` is an object itself. The `SymbolTable` object provides methods to insert and retrieve identifiers whereas the `node` object allows access to the properties of an identifier.

```
#define StrMax 999

typedef class node *Entry;

/*
** object node with friend object SymbolTable including
**  private components:
**   lexptr: pointer to the lexem string
**   token: token of the entry
**   create: flag to indicate a newly created entry
**   type: can be Index, Blob or Array
**   number: the systematic number of the corresponding variable
**   indices: number of indices of the corresponding variable
**   level: the level of the loop of an index variable
**   next: pointer to replacement node
**   leftchild: left child in the binary tree
**   rightchild: right child in the binary tree
**  public components:
**   node: initialization function
**   LexemOf: returns a pointer to the lexem string
**   TokenOf: returns the token
**   IsNew: indicates whether a node is newly created
**   TypeOf: returns the type
**   NumberOf: returns the number
**   IndicesOf: returns the number of indices of a array variable
**   LevelOf: returns the level of an index variable
**   NextOf: returns the replacement node
**   SetIndices: sets the number of indices of a array variable
**   SetNext: connects a replacement node
**   Update: updates some node contents
**   Save: saves a node to disk
*/

class node {
  friend class SymbolTable;
  char *lexptr;
  int token, create, type, number, indices, level;
  Entry next, leftchild, rightchild;
 public:
  node(char *s, int tok, Entry l, Entry r) :
    lexptr(s),
    token(tok),
    create(1),
    indices(-1),
    level(0),
    next(NULL),
    leftchild(l),
    rightchild(r) {
    };
```

```
  char *LexemOf();
  int TokenOf();
  int IsNew();
  int TypeOf();
  int NumberOf();
  int IndicesOf();
  int LevelOf(int n);
  Entry NextOf();
  void SetIndices(int i);
  void SetNext(int n, int l);
  void Update(int tok, int t, int n, int l);
  void Save(ostream &out);
};

/*
** object SymbolTable including
**  private contents:
**   root: root entry of the binary tree
**   lexemes: memory area for the lexem strings of size StrMax
**   lastchar: pointer to the last used field in lexemes
**   FindHelp: help function for the Find function
**   LoadHelp: help function for the Load function
**   GetStr: loads a string from disk
**   GetNum: loads a number from disk
**  public components:
**   SymbolTable: initialization function
**   Insert: inserts an entry
**   Retrieve: finds an entry given the lexem
**   Find: finds an entry given the number and type
**   Save: saves a symbol table to disk
**   Load: loads a symbol table from disk
*/

class SymbolTable {
  Entry root;
  char lexemes[StrMax];
  int lastchar;
  Entry FindHelp(int i, int t, Entry x);
  Entry LoadHelp(istream &in, Entry y);
  void GetStr(istream &in, char *s);
  int GetNum(istream &in);
 public:
  SymbolTable() : root(NULL), lastchar(-1) {
  };
  Entry Insert(char *s, int tok);
  Entry Retrieve(char *s);
  Entry Find(int i, int t);
  void Save(ostream &out);
  void Load(istream &in);
};
```

3.1.2 The implementation

The symbol table is implemented as a binary search tree. An entry in that tree can be a list of nodes to allow renaming of loop index variables.

3.2 The scanner

The scanner is divided into an initialization and a scan module. It interacts with the parser as producer and consumer.

3.2.1 The initialization module

The initialization module consists of the two files `init.h` and `init.cc`. It initializes the symbol table with the key words of the source language. These are `for`, `to`, `step`, `do`, `od`, `DO`, `END`, `max`, `min` and `dep`.

3.2.2 The scan module

The definition and implementation files of the scan module are `scanner.h` and `scanner.cc` respectively. It scans the source program and returns the corresponding stream of tokens to the parser.

```
/*
** end of file flag
*/

#define EOS '\0'

/*
** maximum buffer size for the lexem buffer
*/

#define BSize 128

/*
** definitions of token values
*/

#define None -1          /* no token */
#define Num 256          /* number */

#define For 257          /* lexem 'for' */
#define Assign 258       /* lexem ':=' */
#define To 259           /* lexem 'to' */
#define Step 260         /* lexem 'step* */
#define Do 261           /* lexem 'do' */
#define Od 262           /* lexem 'od' */
#define Id 263           /* identifier */
#define Done 264         /* end of file */
#define Le 265           /* lexem '<' */
#define Leq 266          /* lexem '<=' */
#define Eq 267           /* lexem '=' */
#define EqEq 268         /* lexem '==' */
#define Geq 269          /* lexem '>=' */
#define Ge 270           /* lexem '>' */
```

```
#define Left 271        /* lexem '<-' */
#define Right 272       /* lexem '->' */
#define PlusPlus 273    /* lexem '++' */
#define MinusMinus 274  /* lexem '--' */
#define Plus 275        /* lexem '+' */
#define Minus 276       /* lexem '-' */
#define Times 277       /* lexem '*' */
#define Div 278         /* lexem '/' */
#define DO 279          /* lexem 'DO' */
#define END 280         /* lexem 'END' */
#define Max 281         /* lexem 'max' */
#define Min 282         /* lexem 'min' */
#define Dep 283         /* lexem 'dep' */

#define DoAll 284
#define DoAcross 285

/*
** object Scanner including
**  private components:
**   lexbuf: lexem buffer of size BSize
**  public components:
**   symtab: symbol table
**   symentry: pointer to symbol table entry according to recognized identifier
**   tokenval: token of recognized lexem
**   lineno: source code line counter
**   in: input stream
**   out: output stream for error message
**   Scanner: initialization function
**   LexAn: recognizes next lexem
*/

class Scanner {
  char lexbuf[BSize];
 public:
  SymbolTable symtab;
  Entry symentry;
  int tokenval, lineno;
  istream in;
  ostream out;
  Scanner() : symentry(NULL), tokenval(None), lineno(1) {
  };
  int LexAn();
};
```

3.3 The parser

The stream of tokens from the scanner is received and the corresponding parse tree is constructed. The parser works with the recursive descent method and is therefore easy to extend to provide support for additional source language structures. The definition and implementation files of the parse module are `parser.h` and `parser.cc` respectively.

```
/*
```

```
** maximum number of index variables or blobs
*/

#define MaxId 63

/*
** the two different types of nodes in the parse tree
*/

#define IsLoop 0
#define IsStat 1

/*
** the seven possible states of identifiers. The first five, i. e. Read,
** Write, Const, Index and Blob are used to identify the state of a variable
** in an expression. Write stands for an array variable on the left side of
** an assignment statement, Read for an array variable on the right side.
** Const denotes a constant. Index and Blob indicate index variables and
** structure parameters respectively. To avoid redefinitions, the last four
** also define the type of identifiers in the symbol table.
*/

#define Read 0   /* array variable on the right hand side of an assignment */
#define Write 1  /* array variable on the left hand side of an assignment */
#define Const 2  /* constant */
#define Index 3  /* index variable or index identifier */
#define Blob 4   /* structure parameter */
#define Array 5  /* array identifier in the symbol table */
#define Key 6    /* key word */

/*
** the four different types of loop notations
*/

#define Fortran 0
#define C 1
#define Systoliker1 2
#define Systoliker2 3

/*
** definition of a lambda vector
*/
```

/* see also [1] */

```
typedef int *Vect;

/*
** object LaExpr (lamda expression) including:
**  private components:
**   coef: vector of index variable coefficients
**   c0: constant
**   blobcoef: vector of blob coefficients
**   denom: denominator
**   next: pointer to the next lambda expression in an expression list
```

```
**  public components:
**   LaExpr: initialization function
**   Print: displays a lambda expression
**   Transfer: saves a lambda expression
**   GetCoef: returns the coefficient vector
**   GetC0: returns the constant
**   GetBlobCoef: returns the blob coefficient vector
**   GetDenom: returns the denominator of a vector
**   GetNext: returns the next lambda expression in the expression list
*/

class LaExpr {
  friend class Parser;
  Vect coef;
  int c0;
  Vect blobcoef;
  int denom;
  LaExpr *next;
 public:
  LaExpr(int *c, int a, int *b, int d);
  void Print();
  void Transfer(ostream &out);
  Vect GetCoef();
  int GetC0();
  Vect GetBlobCoef();
  int GetDenom();
  LaExpr *GetNext();
};

/*
** object LaNode (lamda node) including:
**  private components:
**    name: number of the index variable, blob or array variable
**   typ: type of the lamda node:
**         - IsLoop
**         - IsStat
**   low: lower bound of the loop
**   up: upper bound of the loop
**   step: stepwidth of the loop
**   notation: notation of the loop in the source code:
**              - FORTRAN
**              - C
**              - Systoliker1
**              - Systoliker2
**   stmnt: number of the statement
**   indices: lamda expression for the indices of an array variable
**   state: status of a variable:
**           - Read
**           - Write
**           - Const
**           - Index
**           - Blob
**           - Array
**           - Key
```

```
**   op: operand following the term
**   parent: pointer to the parent node
**   children: pointer to the children node:
**              - first inner statement for a loop
**              - next term for an expression
**   next: pointer to the next node with loop or statement of the same level
**  public components:
**   LaNode: initialization function
**   GetName: returns the name of a node
**   GetTyp: returns the type a node
**   GetLow: returns the lower bound of a loop
**   GetUp: returns the upper bound of a loop
**   GetStep: returns the stepwidth of a loop
**   GetStmnt: returns the statement number
**   GetParent: returns the parent pointer
**   GetChildren: returns the children pointer
**   GetNext: returns the next pointer
*/

class LaNode {
  friend class Parser;
  int name, typ;
  LaExpr *low, *up;
  int step, notation;
  int stmnt;
  LaExpr *indices;
  int state, op;
  LaNode *parent, *children, *next;
 public:
  LaNode(int t) : typ(t), indices(NULL), parent(NULL), children(NULL), next(NULL) {
  }
  int GetName();
  int GetTyp();
  LaExpr *GetLow();
  LaExpr *GetUp();
  int GetStep();
  int GetStmnt();
  LaExpr *GetIndices();
  int GetOp();
  int GetState();
  LaNode *GetParent();
  LaNode *GetChildren();
  LaNode *GetNext();
};

/*
** object LaDep representing a dependency.
**
** private components:
**  - rows: number of rows of the matrix
**  - columns: number of columns of the matrix
**  - matrix: the matrix itself stored in linear order
**  - stmnt1: first dependent statement
**  - stmnt2: second dependent statement
```

```
**  - next: pointer to the next dependency
**
** public functions:
**  - LaDep: initialization function
**  - GetRows: returns the number of rows of the matrix
**  - GetColumns: returns the number of columns of the matrix
**  - GetMatrixElem: returns the element at a certain location in the matrix
**  - GetStmnt1: returns the first dependent statement
**  - GetStmnt2: returns the second dependent statement
**  - GetNext: returns the next dependency
*/

class LaDep {
  friend class Parser;
  int rows, columns;
  Vect matrix;
  int stmnt1, stmnt2;
  LaDep *next;
 public:
  LaDep(int n, int m);
  int GetRows();
  int GetColumns();
  int GetMatrixElem(int, int);
  int GetStmnt1();
  int GetStmnt2();
  LaDep *GetNext();
};

/*
** object Parser including
**  private components:
**   lookahead: lookahead to the next token
**   coef: cache memory for index variable coefficients
**   c0: cache memory for a constant
**   blobcoef: cache memory for blob coefficients
**   denom: cache memory for a denominator
**   numarray: number of the array variables found so far
**   numstmnt: number of the statements found so far
**   root: root of the parse tree
**   deps: start of the dependence list
**   ExpList: parses a list of expressions
**   Expt: parses an expression
**   Term: parses a term
**   Factor: parses a factor
**   ArrayId: parses an array identifier
**   IndexId: parses an index identifier
**   Limit: parses the upper and lower bound of a loop
**   Loop: parses a loop
**   Depend: parses a dependency
**   Match: matches a token and gets the next one
**   SaveDeps: saves the dependence list
**   LoadDeps: loads the dependence list
**   TraverseHelp: help function to traverse and save the parse tree
**   LoadHelp: help function to load the parse tree
```

```
**   GetNum: reads an integer number ignoring comments
**   LoadExpr: loads a lamda expression
**  public components:
**   scan: LooPo scanner
**   Parser: Initialization function
**   Parse: parses a source code program
**   Traverse: traverses and saves a parse tree
**   Load: loads a parse tree
**   GetRoot: returns the root of the parse tree
**   GetDeps: returns the beginning of the dependence list
**   GetNumInd: returns the number of indices found
**   GetNumBlob: returns the number of blobs found
*/

class Parser {
  ostream output;
  int lookahead, coef[MaxId], c0, blobcoef[MaxId], denom, numarray, numstmnt;
  LaNode *root;
  LaDep *deps;
  LaExpr *ExprList(int &i);
  void Expr();
  void Term();
  void Factor();
  LaNode *ArrayId(LaNode *parent, int state);
  void IndexId(LaNode *parent, LaNode *n, int level);
  void Limit(int typ, LaNode *n);
  LaNode *Loop(LaNode *parent, int level);
  LaDep *Depend();
  void Match(int t);
  void SaveDeps(LaDep *d, ostream &out);
  LaDep *LoadDeps(istream &in);
  void TraverseHelp(LaNode *l, ostream &out);
  LaNode *LoadHelp(LaNode *parent, istream &in);
  int GetNum(istream &in);
  void LoadExpr(istream &in, int n);
 public:
  Scanner scan;
  Parser();
  void Parse(istream &in, ostream &out);
  void Traverse(ostream &out);
  void Load(istream &in);
  LaNode *GetRoot();
  LaDep *GetDeps();
  int GetNumInd();
  int GetNumBlob();
};
```

4 An example

To illustrate the structures constructed by the parser the following example will be considered.

4.1 The source program

```
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
```

/* These dependencies are just an arbitrary example and have nothing to do with the above loops and statements part. The intention is only to show the shape of an example dependence list. */

```
dep 2 3
1 2 3
4 5 6 1 2

dep 3 2
 1  4
 9 16
25 36 3 6
```

4.2 The parse tree

The parse tree constructed by the LooPo parser for the above source program is shown in Figure 1 where pointers are represented by arrows.

4.3 The dependence list

The optional dependence part of the example source program is converted into the list structure shown in Figure 2.

5 Retrieving the stored information

5.1 How to use the provided tools

An example of how to use the provided tools for traversing and analyzing the parse tree and the dependence list structure is given in `analyse.cc`. It is more or less a reverse dual to the recursive descent parser. Since all the source code has a lot of comments, there are only two more things to add:

- The first component of a vector of coefficients contains the number of coefficients following.
- A matrix is stored as a sequence of entries, i. e.

  $$a_{11}, a_{12}, \ldots, a_{1m}, a_{21}, a_{22}, \ldots, a2m, \ldots, a_{n1}, a_{n2}, \ldots, a_{nm}$$

  where n is the number of rows and m is the number of columns.

The source code of `analyse.cc` is shown in the next section.

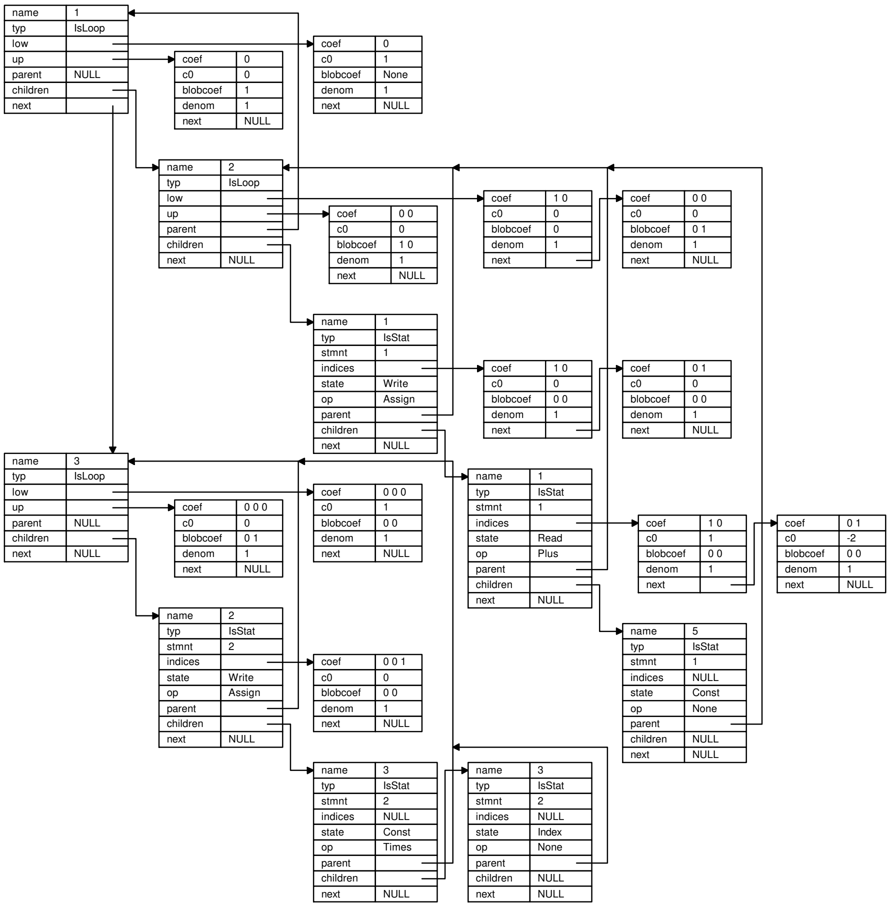

Figure 1: The parse tree

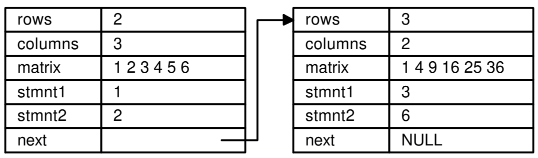

Figure 2: The dependence list

5.2 Source code of `analyse.cc`

```
/*
** The function factor prints an factor of the form a*<identifier> and
** indicates whether it actually printed something.
** a is the multiple, i and type are the number and type of the
** identifier, out is the output stream to write to and s is the
** symbol table to look for the identifier.
*/

int factor(int a, int i, int type, int out, SymbolTable s)
{
  if (a > 0) {
    if (out) {
      cout << "+";
    }
    if (a > 1) {
      cout << a << "*";
    }

/*
** finds the lexem string for identifier number i of type type
*/

    cout << s.Find(i, type)->LexemOf();
    return 1;
  } else {
    if (a < 0) {
      if (a == -1) {
        cout << "-";
      } else {
        cout << a << "*";
      }
      cout << s.Find(i, type)->LexemOf();
    } else {
      return out;
    }
  }
}

/*
** The function term prints a term of the form
** <factor> + <factor> + ... + <factor>.
** v is the vector of multiples, type indicates an index variable or
** structure parameter vector, out is the output stream and s the
** symbol table.
*/

int term(Vect v, int type, int out, SymbolTable s)
{
  int i;

/*
** call factor for all components of vector v (remember, the number of
```

```
** components is stored in v[0])
*/

  for (i = 1; i <= v[0]; i++) {
    out = factor(v[i], i, type, out, s);
  }
  return out;
}

/*
** The procedure expression prints a whole lambda expression.
*/

void expr(LaExpr *e, SymbolTable s)
{
  Vect blobcoef;
  int c0, out;

/*
** get the constant with GetC0(),
** the index coefficients with GetCoef()
** and the blob coefficients with GetBlobCoef()
*/

  c0 = e->GetC0();
  out = term(e->GetCoef(), Index, 0, s);
  out = term(e->GetBlobCoef(), Blob, out, s);
  if (c0 > 0) {
    if (out) {
      cout << "+";
    }
    cout << c0;
  } else {
    if (c0 < 0) {
      cout << c0;
    } else {
      if (!out) {
        cout << c0;
      }
    }
  }
}

/*
** The procedure limit prints a lower or upper limit corresponding to
** type.
*/

void limit(LaExpr *e, int type, SymbolTable s)
{

/*
** a limit can be a single expression or a list of expressions
*/
```

```
  if (e->GetNext() != NULL) {
    if (type == 0) {
      cout << "max(";
    } else {
      cout << "min(";
    }
    while(e != NULL) {
      expr(e, s);
      if (e->GetNext() != NULL) {
        cout << ", ";
      }
      e = e->GetNext();
    }
    cout << ")";
  } else {
    expr(e, s);
  }
}

/*
** The procedure indices prints the list of indices for an array variable.
*/

void indices(LaExpr *e, SymbolTable s)
{

/*
** the indices of an array variable are stored as a list of expressions
*/

  cout << "(";
  while(e != NULL) {
    expr(e, s);
    if (e->GetNext() != NULL) {
      cout << ", ";
    }
    e = e->GetNext();
  }
  cout << ")";
}

/*
** void analyse(LaNode *n, int level, Symboltable s)
**
** analyses the parse tree at node n and level level.
*/

void analyse(LaNode *n, int level, SymbolTable s)
{
  LaNode *m;
  int i;

  while (n != NULL) {
```

```
/*
** a node can be either a loop node (IsLoop) or a statement node (IsStat)
*/

    if (n->GetTyp() == IsLoop) {
      for (i = 0; i < level; i++) {
        cout << "  ";
      }

/*
** get the number of an index variable and find the lexem string for it
*/

      cout << "DO " << s.Find(n->GetName(), Index)->LexemOf() << " = ";

/*
** get the lower limit
*/

      limit(n->GetLow(), 0, s);
      cout << ", ";

/*
** get the upper limit
*/

      limit(n->GetUp(), 1, s);
      cout << endl;

/*
** children of a loop node are other loop or statement nodes whitch are one
** level higher
*/

      analyse(n->GetChildren(), level + 1, s);
      for (i = 0; i < level; i++) {
        cout << "  ";
      }
      cout << "END DO" << endl;
    } else {
      if (n->GetTyp() == IsStat) {
        for (i = 0; i < level; i++) {
          cout << "  ";
        }

/*
** get the number of an array variable and then its lexem string
*/

        cout << s.Find(n->GetName(), Array)->LexemOf();
        indices(n->GetIndices(), s);
        m = n;
```

```
/*
** the children of a statement are the following terms
*/

        while (m->GetChildren() != NULL) {
          switch (m->GetOp()) {
          case Assign:
            cout << " = ";
            break;
          case Plus:
            cout << " + ";
            break;
          case Minus:
            cout << " - ";
            break;
          case Times:
            cout << " * ";
            break;
          case Div:
            cout << " / ";
            break;
          default:
            cout << endl << "Wrong operand error!" << endl;
          }
          m = m->GetChildren();
          switch (m->GetState()) {
          case Read:
            cout << s.Find(m->GetName(), Array)->LexemOf();
            indices(m->GetIndices(), s);
            break;
          case Index:
            cout << s.Find(m->GetName(), Index)->LexemOf();
            break;
          case Blob:
            cout << s.Find(m->GetName(), Blob)->LexemOf();
            break;
          case Const:
            cout << m->GetName();
            break;
          default:
            cout << endl << "Wrong type error!" << endl;
          }
        }
        cout << endl;
      } else {
        cout << endl << "Unknown type error!" << endl;
      }
    }

/*
** go to the next node on the same level
*/

    n = n->GetNext();
```

```
  }
}

/*
** void ScanDep(LaDep *d)
**
** analyses the dependence list.
*/

void ScanDep(LaDep *d)
{
  int i, j;

  while (d != NULL) {
    cout << "dependency from statement " << d->GetStmnt1() << " to ";
    cout << "statement " << d->GetStmnt2() << " with matrix:" << endl;
    for (i = 0; i < d->GetRows(); i++) {
      for (j = 0; j < d->GetColumns(); j++) {
        cout.width(3);
        cout << d->GetMatrixElem(i, j);
      }
      cout << endl;
    }
    cout << endl;
    d = d->GetNext();
  }
}
```

References

[1] Li, Wei/Pingali, Keshav, *The Lambda Loop Transformation Toolkit (User's Reference Manual)*, University of Rochester Technical Report 311/Cornell University Technical Report 94-1431, 1994.

[2] Kingston, Jeffrey H., *Algorithms and Data Structures*, Addison–Wesley Co., Inc., 1990.

[3] Aho, Alfred V./Sethi, Ravi/Ullman, Jeffrey D., *Compilerbau*, Addison–Wessley Co., Inc., 1992.