A Prototype Restructuring Compiler

Aart J.C. Bik

May 6, 1992

INF/SCR-92-11

Preface

This document presents techniques and methods used to implement a prototype source to source restructuring compiler, that enables the user to define program transformations which the restructuring compiler will use, with a transformation language. This is in contrast with most conventional restructuring compilers, that have a fixed set of internal transformations, not accessible for users. By giving the user explicit control over the set of transformations of the restructuring compiler and over the application of these transformations, more insight in the behavior of compilers can be obtained, leading to the development of better restructuring compilers. Some results achieved by using this compiler on some FORTRAN 77 programs are given, together with some issues for future research.

I would like to thank dr. H.A.G. Wijshoff for his support and advice during the implementation of this compiler. His constructive input during all phases of the implementation and testing, was of great help. I would also like to thank drs. A.H. Snippe and ir. A.J. Niessen for their suggestions during the writing of this article.

Aart J.C. Bik

Contents

1 Introduction

The goal of restructuring compilers is to transform serial program code, written for single processor machines, into parallel or vector code, without changing the semantics of the original program. By applying certain transformations to the serial source code the restructuring compiler tries to achieve the best possible usage of the vector or parallel characteristics of the target machine. Because the bigger part of the runtime is spent in loops, the aim of most transformations is the eventual vectorization or concurrentization of loops. Some program transformations can help in achieving better resulting code rather than the straightforward conversion of loops into vector or parallel code, by transforming the structure of loops first (e.g. resulting in stride-1 vector instructions or a parallel loop with more statements in its body). An overview of these transformations and their goals, can be found in [Bik91].

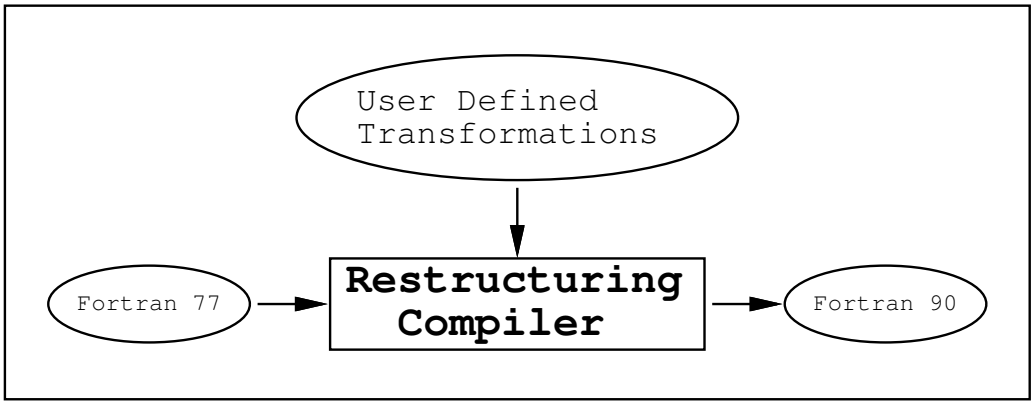
Although some restructuring compilers perform well (e.g. KAP and VAST), there is still some scepticism about the quality of the resulting code. Most of that scepticism is probably the result of the following two facts. First, it is often unclear which transformations a restructuring compiler applies, for what reason, and in which order. Also in most cases, programmers using restructuring compilers, cannot guide the compiler towards a certain direction, to force the best exploitation of potential parallelism. Second, most transformations defined in restructuring compilers are only program transformations. More parallelism can be achieved, if the restructuring compiler is able to perform data structure transformations as well as program transformations. Therefore, it is useful to examine if it is possible to solve both problems by using more advanced restructuring compilers. A start has been made by considering the first problem.

This article presents the implementation and use of a prototype restructuring compiler in which the user can define transformations on the program, and in which the user has explicit control over the application of these transformations. The transformations can be specified in simple transformation language which can define transformations on program patterns and with the ability to associate conditions with every transformation. The initial version of this transformation language must be powerful enough to express the basic transformations (vectorization and concurrentization) as well as other optimizing transformations, and serves to gain more insight in the required nature of such languages. Hopefully, the study of the behavior of the prototype compiler using this simple transformation language, will lead to more advanced languages and resulting restructuring compilers.

Since many existing applications requiring high speed computations are written in serial FORTRAN 77, it is very useful to have a restructuring compiler that can apply vectorization and concurrentization to this language. All the investments taken in developing this serial software can be saved by automatically converting this software into parallel code instead of recoding all the programs by hand. Therefore the source language of the prototype restructuring compiler is chosen to be a dialect of FORTRAN 77, although all the techniques and methods used to implement this compiler can aid the implementation of a compiler for any FORTRAN-like 3^{rd} generation language. Naturally such a

restructuring compiler is not only helpful in converting existing software, but can also be of great assistance in developing new parallel programs.

To keep the resulting code as portable as possible, it is decided to generate FORTRAN 90. This language is almost identical to FORTRAN 77 but has vector-instructions (using the array-section notation) and the possibility to indicate that all the iterations of a DO-loop can be executed in parallel (using the construction `DOALL`). So the prototype compiler is a so-called source to source restructuring compiler, which translates a (serial) FORTRAN 77 dialect into (parallel/vector) FORTRAN 90. Note the fact that if the set of user-defined transformations is empty, the compiler only functions as a syntax converter of a FORTRAN 77 dialect into (serial) FORTRAN 90 code. The conversion into vector or parallel code must be explicitly defined using the transformation language. The general idea of the prototype compiler using user-defined transformations can be presented as shown in the following picture:



General Idea

Figure 1: Compiler

Because the prototype compiler is a source to source compiler, only the **front end** of a traditional compiler for a dialect of FORTRAN 77 is necessary, consisting of the phases lexical analysis, syntax analysis, and semantic analysis. These three phases are followed by a data dependence analysis phase, in which all the static data dependences of the program are computed. These data dependences can be used in the following transformation phase in which the user-defined transformation are applied to the FORTRAN 77 code, transforming it into FORTRAN 90 code. Finally an unparsing phase writes the resulting FORTRAN 90 code to a text file, so it can be used as input for a FORTRAN 90 compiler. These different phases of the prototype compiler are shown in the following picture.

First the lexical analysis phase and the syntax analysis phase (parsing) will be discussed. After that, the semantic analysis phase, the data structure used for storing the program after some initial transformations (**constant folding** and the conversion of subscript expressions into *normal form*), and data depen-

dence analysis is discussed, preceded by some extensions on the existing theory about data dependences. The transformation language of this prototype compiler will be introduced next, followed by the methods used to apply these transformations on the code in the program data structure. The presentation of the implementation of unparsing the code in memory into a FORTRAN 90 text file, is the next topic of discussion. Finally, the resulting FORTRAN 90 code of some FORTRAN 77 programs is presented to demonstrate and discuss the performance of this prototype compiler, resulting in some conclusions and issues for future research.

The prototype compiler is implemented in C, and the source code can be found in the appendices.

1.1 Interactive Environment

The prototype compiler can be called from within the UNIX shell with the command **f2f**. An interactive environment is entered then, from which commands can be given after the prompt '=>' to read in programs, apply transformations, etc. The available commands are listed when the command **help** is used. All these commands and their meanings are explained in a quick reference manual that can be found in appendix A, but most commands will also be presented during the discussion about the implementation of the prototype compiler in this article. In the same appendix the different system responses [1] are given. The implementation of this environment can be found in module *env.c*, listed in appendix T.

Information about the program can be obtained by examining the symbol table of a program or the assumed data dependences. During the transformation phase, the fragments of a program that match on a user defined transformation are presented to the user together with the result from applying the transformation. The user can decide then, if the transformation must be applied. The compiler also offers the possibility of aborting the transformation phase so the program obtained so far can be examined, together with its symbol table and data dependences. This kind of interactive processing of programs has been chosen to gain more insight in the behavior of this prototype compiler.

[1] e.g. an error is generated when the user tries to read a non-existing file.

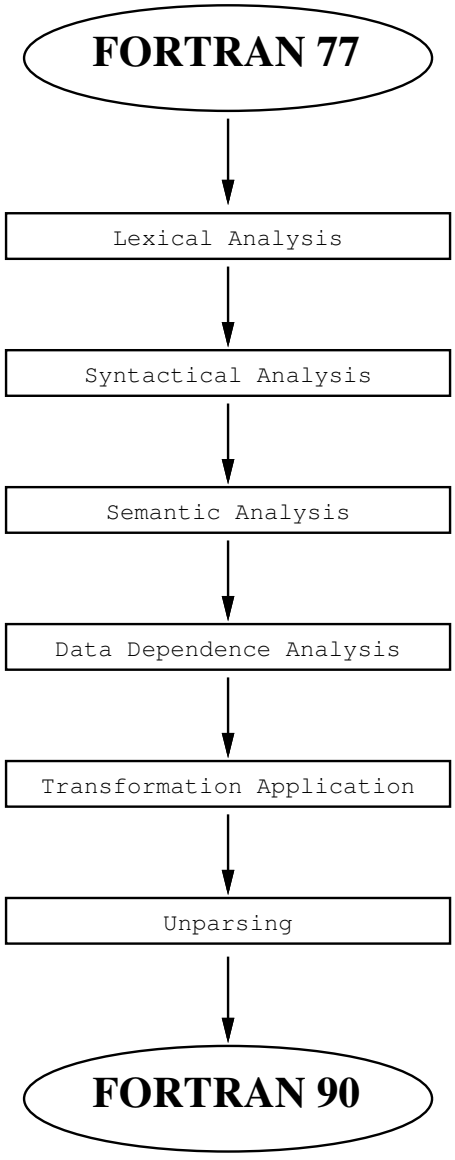


Figure 2: Phases of the Prototype

2 The Source Language: a FORTRAN 77 dialect

Because lexical analyzing and parsing standard FORTRAN 77 is a difficult task, certainly if automatically generated lexical analyzers and parsers are used, and because interprocedural data dependence analysis is not the topic of study during the implementation of this prototype restructuring compiler, a certain dialect of FORTRAN 77 will be defined as input language, which does not support intrinsic or user-defined functions or subroutines, GOTO statements, input and output statements and CHARACTER data, and which has different lexical conventions than standard FORTRAN 77.

The lexical conventions of the dialect are discussed first, together with the presentation of the implementation of the lexical analyzer, followed by a discussion of the implementation of the syntactical analysis phase.

2.1 Lexical Analysis

The task of a lexical analyzer is to tokenize the input text. Strings that match a certain pattern defined in the lexical analyzer, usually as a regular expression, are passed to the next phase (parsing) as tokens. These matching strings will be referred to as the lexemes of a certain token. The following constructs will be represented by one particular token: every keyword and every operator, identifiers, integer constants, real constants, labels and every punctuation symbol. To avoid a certain shift-reduce conflict [2] on having seen a single label (integer constant), in fact, a label and its associated `CONTINUE` statement are represented as one single token `LCONTINUE`. Because in FORTRAN 77 every statement must begin at a new line, a **newline** separating two statements is also defined as a punctuation symbol with its own token (it cannot simply be ignored as is the case in most programming languages). The compound operators are the arithmetic exponentiation operator '`**`', the logical operators (e.g. `.NEQV.`), and the relational operators (e.g. `.NE.`).

The keywords `CONTINUE`, `DO`, `ELSE`, `END`, `ENDIF`, `IF`, `INTEGER`, `LOGICAL`, `PARAMETER`, `PROGRAM`, `REAL`, `STOP`, and `THEN` are reserved and can not be used as user-defined identifiers. User-defined identifiers must always begin with a letter possibly followed by a sequence of letters or digits. The special characters '$*$', '/', '+', '-', '(', ')', '.', ':', ',' and '=' can not be used in identifiers. Only the first six characters of identifiers are used (so the identifiers `LONGID` and `LONGID2` will be seen as the same identifier). Both uppercase and lowercase letters may be used, although the keywords must be spelled in a consequent manner (so both `END` and `end` are allowed, but `End` will be seen as an user-defined identifier. Note that since the lexical analyzer is lower and upper case distinct, the identifiers `I` and `i` are seen as different identifiers. No spaces or tabs may appear inside an identifier or keyword. An exception forms the keyword `END IF`, included in the definition, since it is used very often. Note the difference with standard FORTRAN 77, in which all spaces and tabs are ignored first during a so-called *scanning* phase, before the actual lexical analysis is done. Spaces or tabs between different constructs are allowed and ignored, but although this

[2] This topic is disussed in more detail in subsection 2.2.

means that spaces and tabs are not really part of the language, keywords, identifiers, labels and operators *must* be separated in an unambiguous way. So, `DO10I=1,10` will generate an error, because the identifier `DO10I` is recognized, but `IF(I.EQ.3)A=5` is a legal statement, because the characters '(', ')', '.' and '=' can never be part of an identifier. In this fashion, it is not necessary to scan the text ahead to determine if an assignment statement `DO10I=1.10` or a DO-loop `DO10I=1,10` is encountered. Empty lines between statements are allowed, so the **newline** on such lines is not passed to the parser as a statement separator, but simply ignored.

Labels must appear in the first 5 columns, and no tabs before a label are allowed, because the lexical analyzer explicitly searches for the beginning of a line, followed by a maximum of 4 spaces and digits in sequence up to and including the 5^{th} column. Every line that starts with a 'C', 'c' or '$*$' character is treated as a comment line and is therefore ignored. A continuation is indicated by any character in the sixth column, not preceded by a label, tab or anything else than 5 spaces. The allowed number of continuations in sequence is unlimited. Note that the **newline** before a continuation must not be seen as a statement separating **newline**, but must be ignored. Because comment lines or blank lines can appear between that **newline** and the continuation, as the next example demonstrates, the lexical analyzer must scan ahead on the input, before deciding if a statement separating **newline** must be passed to the parser, or if that **newline** must be ignored.

```
          R = R + 200 +
C This is a comment line before a continuation

C and this is a second comment line after a blank line
     +            200 * 10
```

So, a **newline** before a continuation matches the following regular expression,

```
\n/({comment}|{emptyline})*{continuation},
```

where `{comment}`, `{emptyline}` and `{continuation}` are the regular expressions for respectively comments, blank lines, and a continuation. The regular expression $r.e._1/r.e._2$ matches only on a string matching on $r.e._1$ if it is followed by a string that matches $r.e._2$, without actually scanning that second string. The reason that the comments and blank lines between the **newline** and the continuation are not scanned and ignored too during the look ahead phase, is that the number of these continuations must be known to the lexical analyzer, to administrate the current line number (see below), *and* that in case that no continuation follows after the (incomplete) expression `R = R + 200 +`, the syntax error must be reported to be at that line. Since it is not possible to administrate the number of times a regular expression $r.e.$ is matched in a regular expression of the form $(r.e.)*$, it is necessary to scan the input text ahead, before a decision can be taken, which is a rather expensive operation.

Lines without a label can start at any position in this dialect, although it is wise to follow the same convention as in standard FORTRAN 77. Text after the

72th column is not ignored, although the unparsing phase that will be discussed in section 9, will use a continuation before a line exceeds the 72th column, in order to enable the user to present the resulting FORTRAN 90 code to another compiler.

Constants of type REAL are recognized as strings that match the following regular expression, where `{digit}` is the regular expression for a single digit.

```
{digit}+\.{digit}*(("E"|"e")("+"|"-")?{digit}+)?
```

So both scientific and the decimal point notation are allowed. Any sequence of digits without a decimal point or exponentiation operator are seen as constants of type INTEGER.

The lexical analyzer has been generated using the LEX-tool. Whenever a pattern described using a regular expression has been recognized, an action defined in C will be executed. For most patterns this action is to return the appropriate token to the parser. For some patterns, however, some extra or different actions must be specified. Whenever a comment, a blank line, or a **newline** before a continuation appears on the input, only the variable *f_line* must be increased, which keeps track of the current line number in the FORTRAN 77 source file. This variable can be used if a warning or an error is generated by the parser, to inform the user where in the program that error or warning occurred.

If a single newline separating two statements has been scanned, the only action executed is to return the token for a **newline**. The current line number *f_line* must be increased in the code of the parser itself, because if this increasing was done by the lexical analyzer (as is the case for comments, empty lines and **newline**s before continuations), the next line number would be reported in cases where the error is detected on seeing a statement separating **newline**, where it is not expected, as in the following example.

```
PARAMETER ( N = 100
```

The missing ')' can only be detected if the **newline** is passed to the parser. Since associated actions in the lexical analyzer have already been executed before passing tokens to the parser, the next line number would have been used in the resulting error message. To prevent this wrong behavior, the parser increases the line number if the **newline** is expected. Therefore, the additional rule *newline* → `'\n'` is necessary, with the increment of the line number as associated action. Every occurrence of a single `'\n'` in the grammar rules of the parser, must be substituted with the non-terminal *newline*.

Whenever a constant of type INTEGER or REAL has been recognized on the input, an attribute associated with the token (terminal) must be set to the representation of the type of that constant. An auxiliary variable *currentval* is set to the value of that constant (because different types of constant are possible, the variable *currentval* is implemented as a union in C), using the *sscanf* [3] function of C to assign the correctly typed value to the variable *currentval*.

[3] This function can also handle constants in scientific notation.

Note that this variable just acts as an extra attribute of constant terminals, and the parser will use this value only in the rules using these constant terminals.

In order to be able to store information about variables (e.g. type and dimension) a symbol table must be maintained. This symbol table is a data structure containing all the information needed by the compiler and must be visible to all different compilation phases. It has an entry for every variable used in the program. Because no limitations on the number of variables used in a program exists, the symbol table is implemented using the technique of dynamic memory allocation. Whenever more entries than allocated at a particular moment are needed, more memory is allocated at run-time. By doing so the only limitation on the number of variables that may be used in a program, results from global memory constraints. The functions needed for (re)allocating memory for the symbol table can be found in the module *symbol.c* which is described in the section about the symbol table and is listed in appendix J.

The task of the lexical analyzer regarding symbol table management is to create new entries for every new identifier in the program text. So, if an identifier has been recognized by the lexical analyzer, a function from the module *symbol.c* is called, in which the symbol table is scanned to determine if the lexeme (matching string) has been used before. If this is not the case, a new entry is created in the symbol table in which the lexeme will be stored. The attribute associated with the token of an identifier is set to the entry in the symbol table in both cases, before that token is passed to the parser.

If a character is scanned, that is no element of the standard FORTRAN 77 character set, the character itself is returned to the parser as a token (the representation of every token chosen by the tool used to generate the parser, does not interfere with the representation of characters, so this can be done without causing any problems). This is done to prevent the code generated by the LEX-tool from dumping that character to standard output (which is done whenever a string does not match any pattern). In that fashion the error can be detected and handled in the parser.

The LEX definitions can be found in the module *fscanner.l*, which is listed in appendix G.

2.2 Syntax Analysis

The parsing of the program is done according to the context-free grammar of a FORTRAN 77 dialect shown in the following picture:

```
program          →  PROGRAM ID newline program_list END newline
program_list     →  specification newline program_list
                 |  stmtlist
                 ;
specification    →  PARAMETER '(' par_list ')'
                 |  INTEGER defvar_list
                 |  REAL defvar_list
                 |  LOGICAL defvar_list
                 ;
par_list         →  par ',' par_list
                 |  par
                 ;
par              →  ID '=' expr
                 ;
defvar_list      →  defvar ',' defvar_list
                 |  defvar
                 ;
defvar           →  ID
                 |  ID '(' subscriptlist ')'
                 ;
subscript_list   →  subscript ',' subscript_list
                 |  subscript
                 ;
subscript        →  expr ':' expr
                 |  expr
                 ;
stmt_list        →  LABEL stmt newline stmt_list
                 |  stmt newline stmt_list
                 |  ϵ
                 ;
```

```
stmt        →  DO internlab scalarvar '=' expr ',' expr newline
               stmt_list LCONTINUE
            |  DO internlab scalarvar '=' expr ',' expr ',' expr newline
               stmt_list LCONTINUE
            |  IF '(' expr ')' singlestmt
            |  IF '(' expr ')' THEN newline stmt_list ENDIF
            |  ELSE IF '(' expr ')' THEN
            |  ELSE
            |  singlestmt
            ;
singlestmt  →  STOP
            |  var '=' expr
            ;
internlab   →  NUM_INT
            ;
var         →  scalarvar
            |  arrayvar
            ;
scalarvar   →  ID
            ;
arrayvar    →  ID '(' expr_list ')'
            ;
expr_list   →  expr ',' expr_list
            |  expr
            ;
expr        →  expr '+' expr
            |  expr '-' expr
            |  expr '*' expr
            |  expr '/' expr
            |  expr EXP expr
            |  expr EQ expr
            |  expr NE expr
            |  expr GE expr
            |  expr GT expr
            |  expr LE expr
            |  expr LT expr
            |  expr EQV expr
            |  expr NEQV expr
            |  expr AND expr
            |  expr OR expr
            |  NOT expr
            |  '(' expr ')'
            |  '-' expr
            |  var
            |  constant
            ;
```

constant	→	NUM_REAL
	|	NUM_INT
	|	NUM_BOOL
	;	
newline	→	'\n'
	;	

The parser is generated using the YACC-tool. The YACC definitions can be found in the module $fparser.y$ which is listed in appendix H. Because it must be possible to compile very large programs, the run-time memory allocation mode for the resulting parser is set by defining the flag $\_\_RUNTIME\_YYMAXDEPTH$.

Associated with every grammar rule in YACC, pieces of C-code can be defined that can be used for checking the semantics of a program, and building a program data structure. So, the tasks that must be performed after the lexical analysis phase, parsing, checking the semantics and building the program data structure for the program, are in fact executed in an interleaved way.

Actions that occur at the end of a grammar rule are executed when the reduce action of that rule in the resulting parser is performed, that is whenever that rule has been recognized. Actions occurring within grammar rules are executed whenever they are encountered during the parsing phase. Note that these actions are reduced as if they were grammar rules that yield the empty string, so in referencing attribute values using the YACC-method of $\$i$, where i indicates the number of the right hand side (non-)terminal, they cannot be ignored. All these associated actions and their different tasks will be discussed in the following sections.

The function *yyerror* is called with a string containing an error message whenever the routines generated by YACC detect an error, to enable YACC-users to prompt appropriate error messages. In this case the line number in the FORTRAN file under consideration and the error-message are printed. An error flag is also set to enable the function that calls the YACC-generated routines, to detect that an error has occurred. If such an error is generated during parsing, the parsing phase stops immediately. The semantic errors in the source file detected during parsing, or warnings generated (possibly caused by some inaccuracy in the coding of the program) are reported using some other functions, but will not lead to the termination of the parsing phase in order to report more errors or warnings about a program in a single pass.

First the differences between standard FORTRAN 77 and the dialect accepted by the prototype compiler are discussed. Note the fact that every loop must be closed with a `CONTINUE` statement (the test if the label has the correct value is done during the semantic checking phase). So the following program is incorrect, although it is correct in standard FORTRAN 77:

```
      DO 10 I = 1, 10
10      A(I) = 20.0
```

Also note that a single `CONTINUE` statement with a label, that does not close a DO-loop [4] is not allowed.

Now it is clear, why a label and a corresponding `CONTINUE` must be represented as one token. If separate tokens for the label and the keyword `CONTINUE` are passed to the parser, the parser can not decide on having seen the token for a label only, if a new statement, belonging to the loop body follows on input, or a closing `CONTINUE` statement. This is caused by the fact that YACC generates parsers for LALR(1) grammars, so only one token is used as lookahead.

Subroutines, user-defined functions, and intrinsic functions are not supported in this prototype. Therefore, subroutine calls are not allowed (`CALL` is not recognized as a keyword) and function calls are parsed as if they were array references (probably generating errors, if the number of actual parameters does not correspond with the declared or assumed dimension of the variable name used in the function call).

The precedence and associativity of the operators must be defined, in order to enable YACC to create the correct parser. Left associative are the '.OR.', '.AND.', '+', '−', '∗' and '/' operators. Right associative operators are the '∗∗' and the unary minus. The logical and relative operators in FORTRAN 77 may not associate with themselves. The precedence between the operators is defined in the following order, indicated by < (the operators grouped together have the same precedence).

`.EQV.`, `.NEQV.`
<
`.OR.`
<
`.AND.`
<
`.NOT.`
<
`.EQ.`, `.NE.`, `.GE.`, `.GT.`, `.LE.`, `.LT.`
<
+, −, ∗, / <, ∗∗
<
(unary) −

During the unparsing phase of the program it is explicitly stated how the operators have been parsed using bracket notation. So the assignment statement `R = R + 10 * S` will be listed in the resulting FORTRAN 90 program (if it has not been transformed by some transformation, naturally) as the statement `R = (R + (10 * S))`, in which the expression at the right-hand-side has been annotated with brackets.

Some (non)terminals have an attribute of type *int* associated with them. This attribute contains the integer representation of the type for the nonterminals *expr* and *simple_expr*. For *internlab* and the terminals `LABEL` and

[4] This construction is sometimes used to indicate that a label may be branched to. Since GOTOs are not allowed in the dialect, there is no need for this construction.

LCONTINUE, the attribute holds the integer value of the corresponding label. For the terminals indicating a constant value, NUM_INT, NUM_REAL, NUM_BOOL the attribute contains the representation of the type of the constant. The attribute of the terminal *var* holds the entry in the symboltable of the associated variable. The attribute of the non-terminal *expr_list* holds a merged type, that will be introduced in section 3 on semantic analysis.

2.3 An Example of Syntax Analysis

The following simple example illustrates how a syntax error is reported when the following program is parsed by the compiler.

```
PROGRAM DOIT

INTEGER I
DO 10 I = 1, 10
STOP

END

I = 200
```

The compiler reports a syntax error when this program is read using the command **readprg** within the interactive environment. This is caused by the fact that the DO-loop has not been closed using a CONTINUE statement before the end of the program, which is detected when the token for END, is passed to the parser.

```
=> readprg doit.f

PROGRAM DOIT

*** syntax error in FORTRAN file: line <7>

Terminated
```

The parsing phase is terminated immediately after detecting the error, so the (illegal) assignment statement after the keyword END will not be reported to the user.

3 Semantic Analysis

As stated before, checking the semantics of the FORTRAN 77 dialect is done while the program is being parsed. Whenever certain grammar rules have been (partially) recognized, certain associated actions are executed. In this section the actions that check the semantics are discussed. It is clear that only static checking can be done, since no executable code is generated by this compiler, in which dynamic checks could have been inserted.

Since the compiler must check if the source program under consideration follows the semantic conventions of the language used, it is necessary to exactly define these conventions in advance. Therefore, in the following sections the conventions are given first before the implementation of different kinds of semantic checks are discussed.

3.1 Type Checking

The types REAL, INTEGER and LOGICAL are supported by the FORTRAN 77 dialect of the prototype compiler. Because no input or output statements can be used in this dialect, the use of CHARACTER types has not been implemented.

In standard FORTRAN 77 the type of a (sub-)expression is determined using certain typing rules. The same rules have been chosen to hold for the dialect language. Every constant and variable has a type. Either the type of a variable is given explicitly in a type declaration statement, or it is determined according the implicit data typing rules of FORTRAN 77. These rules state that all variables starting with the letters I through N are considered to be of type INTEGER, and that all other variables are considered to be of type REAL.

The following tables show the resulting type of a (sub-)expression for every operator of FORTRAN 77 when that operator is applied to operand(s) of particular types. The fact that the tables of binary operators are symmetric, reflects the fact that the resulting type does not alter if the types of the left- and right-operand are interchanged.

The arithmetic operators in FORTRAN 77 are the '$*$', '$/$', '$+$', '$-$', and the '$**$'. The typing rules of these operators are shown in the following table. Note that the expression `3 / 4` is of type INTEGER, according to the FORTRAN 77 rules. So the resulting value is 0 (instead of 0.75).

	INTEGER	REAL	LOGICAL
INTEGER	INTEGER	REAL	INTEGER
REAL	REAL	REAL	REAL
LOGICAL	INTEGER	REAL	INTEGER

Arithmetic Operators

The relational operators are `.EQ.`, `.NE.`, `.GE.`, `.GT.`, `.LE.`, and `.LT.`. The resulting value, after applying these operators, is always of type LOGICAL, so it is independent of the type of its operands, which is stated in the next table.

	INTEGER	REAL	LOGICAL
INTEGER	LOGICAL	LOGICAL	LOGICAL
REAL	LOGICAL	LOGICAL	LOGICAL
LOGICAL	LOGICAL	LOGICAL	LOGICAL

Relational Operators

Logical binary operators are the `.EQV.`, `.NEQV.`, `.OR.`, and `.AND.`, and cannot be applied to operands of type REAL. So whenever these operators have REAL typed operands, an error must be generated. If one of the operands is of type INTEGER, the type of the resulting expression is also INTEGER. In that case, a specific logical operation on the binary representation of the integer is executed, resulting in an expression of type INTEGER. This logical operation will be discussed in section 4.3 where the data structure of expressions is discussed. The typing rules for binary logical operators are summarized in the next table.

	INTEGER	REAL	LOGICAL
INTEGER	INTEGER	not allowed	INTEGER
REAL	not allowed	not allowed	not allowed
LOGICAL	INTEGER	not allowed	LOGICAL

Binary Logical Operators

One unary logical operator is available in FORTRAN 77: `.NOT.`. This operator cannot be applied on an operand of type REAL. When it is applied to a operand of type INTEGER, a logical operation to the binary representation of the integer is executed, resulting in an expression of type INTEGER, see also section 4.3.

	-
INTEGER	INTEGER
REAL	not allowed
LOGICAL	LOGICAL

Unary Logical Operators

The only unary arithmetic operator, the unary '$-$' results in an expression of the same type as the type of the operand. So, even when it is applied on an expression of type LOGICAL the resulting expression is also of type LOGICAL.

	-
INTEGER	INTEGER
REAL	REAL
LOGICAL	LOGICAL (!)

Unary Arithmetic operator

Now the actions for type checking, associated with the grammar rules can be presented. The implementation of most of the following functions can be found in module *fsup.c*, listed in appendix I.

If a declaration statement has been partially parsed (on encountering one of the keywords INTEGER, REAL or LOGICAL before any executable statements

has been seen) a variable *currenttype* is set to the representation of the type introduced. By doing so, the type of every identifier that is parsed afterwards in that declaration statement, can be found by examining this variable. The dimension for array variables can be determined by counting the number of following subscript ranges in the declaration, so it is known once all the subscript range definitions of this variable have been parsed. Naturally, the dimension of scalar (or simple) variables is 0. Every subscript range may be a single expression of type INTEGER or an expression of the form `Minval : Maxval`, in which `Minval` and `Maxval` are expressions of type INTEGER, indicating the allowed minimum and maximum value of subscript expressions of the associated dimension. If only one expression with value N of type INTEGER and value is given, the lower bound is assumed to be 1, so the allowed subscript range will be `1 : N`. It must be possible to evaluate these expressions at compile-time. If this is not possible, a warning is generated (indicating that the lower- or the upper bound cannot be computed) and the corresponding value is assumed to be 0, the evaluated value is recorded otherwise. If the lower bound is greater than the upper bound, an error is generated. So, for example, the declaration `REAL A(R)`, where R is a non-parameter variable (so its value is undefined at compile-time), will result in the generation of a warning reporting the fact that the upper bound cannot be computed, followed by an error message that the lower bound (assumed to be 1 in this case) is greater than the upper bound (assumed to be 0). If the expressions in the subscript ranges are not of type INTEGER, a warning is generated and the values are converted to INTEGER typed values. Because most compilers for standard FORTRAN 77 allow other typed expressions as well (without any complaints to the user) no error will be generated. However, because type conversion occurs in that situation, and the language definition states that only expressions of type INTEGER are allowed, a warning seems appropriate. If the dimension exceeds 7 an error is generated, because standard FORTRAN 77 only allows multidimensional arrays up to 7 dimensions. Subscript ranges belonging to one array variable are linked into a single list, so all these subscript ranges can be represented by one single pointer. Further memory for storing subscript ranges can be used more efficiently, since memory is allocated dynamically instead of statically allocating the space required for storing the maximum allowed number of subscript ranges.

The type, dimension and the subscript range list of a variable are stored in the symbol table using functions of module *symbol.c*, if that variable has not been declared before. If the variable has been declared earlier in the program, and the type and dimension found in the symbol table are the same as those recently seen, only a warning is generated. In that case the new subscript ranges will be stored in the symbol table, so they overwrite [5] the old subscript ranges without checking if the bounds for every dimension are identical to the old bounds (following the standard FORTRAN 77 convention). But, if the type or the dimension in the recent declaration is different from older declarations, an error is generated.

If a PARAMETER statement has been parsed, every following parameter

[5] The old subscript range list, stored in the symbol table, must be deleted from memory.

definition in the parameter list is processed to store the associated value of the variable in the symbol table. If the variable has been declared as an array before, an error is generated, since it is not allowed to use array variables in PARAMETER statements. The variable is marked in the symbol table as a parameter otherwise, by setting its dimension to -1 (since only scalar variables may be parameters, this will not lead to any ambiguity). If the variable has not been declared before, its type is determined and recorded in the symbol table according to the implicit data typing rules of FORTRAN 77, and a warning reports that type to the user. The expression associated with this variable is evaluated (if that cannot be done at compile-time, a warning reports this fact, and the value is assumed to be 0, 0.0, or .FALSE. for expressions of type [6] INTEGER, REAL and LOGICAL respectively). The resulting value is also stored in the symbol table, in order to be able to use this value in every following occurrence of the parameter variable. If the type of the value stored is not the same as the type of the variable, type conversion is applied first, before the value is stored, and a warning is generated. So if, e.g. the parameter statement `PARAMETER (R = I)` appears in a program, where I is a variable of type INTEGER, and R of type REAL, two warnings are generated. One indicating the fact that the expression I cannot be evaluated (it is assumed to be 0), and one that reports the type conversion of 0 to 0.0.

Because all the associated expressions have been stored in memory during the creation of the program data structure, and their values can always be found in the symbol table after they have been stored in the symbol table, the intermediate data structures of these expressions can be deleted as soon as these values are saved to free the memory occupied by these data structures. It can be stated here that if the expression cannot be evaluated at compile-time, this can be detected by the fact that the intermediate representation of the expression is not the representation of a constant. This is because during the building of the program data structure of expressions, a technique called **constant folding** will be applied, see section 4.3. In fact, the same remarks can be made for the expressions in the subscript ranges.

On encountering a grammar rule of the form $expr \rightarrow expr$ **operand** $expr$, the type of the resulting expression (attribute on the left-hand-side nonterminal) is determined using the rules given at the beginning of this section, and passed to the synthesized attribute of the left-hand-side nonterminal *expr*. The types of both right-hand-side expressions can be found in the attributes of the corresponding nonterminals. A warning is generated if operands of type LOGICAL are used in arithmetic or relational operators, or if expressions of type INTEGER are used as operands of logical operators. An error is generated if an expression of type REAL appears as operand of a logical operator. Most compilers do not generate warnings at all in the first cases, but again, regarding the language definition, the generation of warnings seems appropriate.

For the unary operator `.NOT.` the resulting type is also determined, possibly leading to an error or warning messages, and stored in the attribute of the left-

[6] Note the fact that although the expression cannot be evaluated, its type can always be determined at compile-time.

hand-side nonterminal. The unary operator '−' can be handled very easily, because the type of the operand can be passed immediately to the left-hand-side nonterminal.

If a variable occurrence is encountered in an expression, it is tested whether its dimension in that occurrence corresponds to its dimension stored in the symbol table. If that is not the case, this is reported in an error message. The dimension of the occurrence of that variable can be determined by counting the number of following subscript expressions, and is 0 if the variable is used as a scalar variable. The counting of the number of subscript expressions is done using a stack of integers, because subscript expressions may also contain subscripted variables, and the number of subscripts in those subscripted variables must be tested too. So, before the subscript expressions of a variable are parsed, 0 is pushed on the stack. For every following subscript expression, the top of the stack must be increased by one. After all the subscript expressions of a variable has been parsed, the dimension of its current occurrence can be found by simply popping the integer stack. In this fashion it is possible to account for the dimension of array variables appearing in subscript expression too, since the accounting is done at one level higher on the stack. The implementation of the stack procedures can be found in module *mem.c*. This integer stack will be used for other purposes as well. Again, the technique of dynamically allocating memory for the integer stack has been chosen, since the number of indirect references is theoretically unbounded (as in `A(A(A(...A(1) ...)))`, where A is a one dimensional array variable. It is also tested if all the subscript expression are of type INTEGER. This is done by 'merging' the types of every subscript list, which only returns type INTEGER if all the expressions in a subscript list are of type INTEGER. This merged type is stored into the attribute of the nonterminal *expr_list*. If this attribute does not contain the representation of the type INTEGER, a warning is generated, since type conversion during run-time will be performed in evaluating the value of the subscript expressions.

It is also tested if all constant subscript expressions (those that can be evaluated at compile-time) are within the bounds of the ranges defined in the array declaration. A warning is generated if this is not the case. No analysis will be performed on non-constant subscript expressions.

If a variable has not been declared yet, it is assumed to be a scalar variable of a type which is deduced using the implicit data typing rules of FORTRAN 77, and a warning is generated together with this type. This information is stored in the symbol table just as with a declaration statement, since it is unwanted to repeat all the warnings with every following occurrence of this variable. Note that no standard FORTRAN 77 compiler generates a warning in this case, but doing so enables programmers, using this restructuring compiler, to check if the assumed types of these variables, are as expected.

On parsing an assignment statement, it is tested if the types on the left- and right-hand-side of that assignment are the same. A warning is generated if these types are different. Again, this can be helpful in debugging a program. For example, a classic error is to expect the assignment of `3 / 4` to a REAL variable, to be 0.75. Because the right-hand-side is of type INTEGER in this case, the potential wrong type conversion to 0 can be detected. It is also tested

if no parameter or loop-control variable is assigned to. Since this is illegal, an error must be generated if this is the case. In order to be able to detect this case, the environment containing current loop-control variables must be maintained.

On recognizing the header of a DO-loop, a test is performed that checks if the loop index has not been used as a loop index of a surrounding DO-loop too and if the index has not been defined as a parameter, resulting in an error if it has. If the types of the loop index and the lower bound, upper bound and possible stride expressions are not the same, a warning is generated. If no explicit stride is given, the type INTEGER (for the implicit 1) is used for the stride expression in this test. So a warning is given if the following fragment if presented to the compiler:

```
          ...
          REAL R
          DO 10 R = 3.0, 4.0
            ...
10        CONTINUE
```

If the implicit stride (1) of this loop is changed into an explicit stride (1.0) the warning disappears. If the bounds and the stride can be determined at compile-time, it is checked if this loop will be executed during run-time. If this is not the case (e.g. `DO 10 I = 20, 10`), a warning is generated.

On encountering logical-IF, general-IF or WHILE statements, the type of the condition is tested to be LOGICAL. If it is not, an error is generated, since it is not allowed in FORTRAN 77 to use other typed expressions as a condition.

3.2 Label Related Checks

When a label has been parsed, a check is performed if this label has not been used before. If this is the case, the duplicate use of the label is reported in an error message. In order to be able to detect this error, all the labels used before must be stored. This is done using dynamically allocated memory. Since the expected number of labels in a program written in this FORTRAN 77 dialect is usually small (they are only useful in CONTINUE statements), it is more memory efficient to initially allocate space for a few labels, and only allocate more memory if required (since the bound on the number of different labels possible in a program is 99999, statically allocating memory is in fact not very practical).

Every DO-loop must be closed with a CONTINUE statement and an associated label, that is the same as in the DO-loop header. The first test is performed by the syntax analyzer, but the second test is a semantic test. Therefore, on encountering a DO-loop header, the label after the keyword `DO` is stacked. To maintain the environment of current loop-control variables, which is also necessary to perform the tests mentioned in the previous section, the symbol table entry of the loop-control variable is stacked as well. Because the nesting depth in most programs is limited, this (stack-)environment can be implemented using a static array. If the nesting depth becomes greater than the allowed maximum

depth, an error is generated and the program terminates. The maximum nestings depth is set to 25. If a CONTINUE statement has been parsed, a test is performed whether the associated label of this CONTINUE statement is expected. An error is generated otherwise.

3.3 Flow of Control Checks

If a statement follows after a STOP statement, a warning must be generated to inform the user that that statement cannot be reached. Because this can only be detected during the building of the program data structure, the implementation of this test is discussed in section 4.3.

Because ELSE or ELSE IF statements are parsed as 'normal' statements, it is necessary to check if they only occur within general IF statements. Therefore, it is recorded if a general IF header has been parsed by incrementing a variable $ifnest$, which indicates the nesting depth of general IF statements. Whenever an ELSE or ELSE IF statement has been parsed, the semantic analyzer checks if $ifnest \geq 0$. Besides that, the occurrence of an ELSE statement is recorded using the integer stack, to prevent other ELSE or ELSE IF statements from occurring after this single ELSE.

3.4 Warnings and Errors

Whenever error or warning messages are reported, the current line number in the program text is also given to enable the user to easily locate the problem. In some cases, the name of a variable involved in an error or warning must also be reported (e.g. `- Warning: Duplicate declaration -> I (line 10)`). Because most of the warnings that can be generated by this prototype compiler are non-standard, the generation of warnings is turned off by default. The generation of warnings can be enabled by using the command **warnings** within the shell and can be disabled again with the command **nowarnings**.

It is strongly advised to enable the generation of warnings during the phase of developing or debugging a program until all the warnings generated have been eliminated or are clearly understood.

3.5 An Example of Semantic Analysis

The warnings and errors that are generated when the following program is presented to the prototype compiler, are discussed here to illustrate the features presented earlier in this section.

```
PROGRAM TYPES

PARAMETER ( N = 100, M = 200.0 )
INTEGER   I, J, I
REAL      R, A( M ), B( 300.0 ), B
LOGICAL   B2
PARAMETER ( B2 = (.FALSE. .AND. 30) )
```

```
          DO 10 I = 1, 2 * N, - 1.0
            A( I ) = 100
  15      CONTINUE

          IF (3) THEN
            R = 2.0
          ELSE
            R = 3.0
          ELSE
            R = 4.0
          ENDIF

          STOP
  15      R = A(201)

          END
```

The warnings are a result of the fact that the variables N and M are not declared before their use in the PARAMETER statement, the fact that the real number 200.0 is used in the PARAMETER definition of M, the duplicate, but compatible declaration of I, the use of an INTEGER expression in one of the operands of the logical operator `.AND.` which also results in the warning about an INTEGER expression in the PARAMETER statement defining B2, the use of an expression of type REAL in the stride of the DO-loop, the assignment of an INTEGER typed value to A, the use of the subscript 201 in the array A that has only 200 elements, and the fact that the last statement cannot be reached. The error messages are generated because the CONTINUE statement at line 11 does not have the correct label (it has label 15 while label 10 is expected), because an integer is used as a condition at line 13, an ELSE follows after another ELSE at line 17, and because label 15 is set again at line 22.

```
=> readprg errors.f

PROGRAM ERRORS
- Warning: Parameter not declared -> N (line 3)
   type INTEGER assumed
- Warning: Parameter not declared -> M (line 3)
   type INTEGER assumed
- Warning: Other type in value of Parameter -> M (line 3)
- Warning: Duplicate declaration -> I (line 4)
- Warning: Subscripts are not of type INTEGER in declaration (line 5)
- Error: Variable redeclared -> B (line 5)
- Warning: INTEGER operand in logical operator (line 7)
- Warning: Other type in value of Parameter -> B2 (line 7)
- Warning: Different types in DO-loop (line 9)
- Warning: Other type in assignment to -> A (line 10)
- Error: Incorrect CONTINUE label (line 11)
- Warning: DO-loop will not be executed (line 11)
```

```
- Error: Condition is not of type LOGICAL (line 13)
- Error: ELSE(IF) after ELSE (line 17)
- Error: Label has already been set (line 22)
- Warning: Subscript out of bound (line 22)
- Warning: Statement after STOP cannot be reached (line 24)

Terminated
```

Note that all semantic errors in this program can be detected and reported during a single pass, which saves time while debugging a program.

4 Building the Program Data Structure

During the parsing phase, the program and program related information must be stored into memory. The data structures used and the methods to build these are presented in this section.

Section 4.1 discusses the symbol table, in which information about all variables used in the program is stored. The implementation of symbol table functions can be found in module *symbol.c*, listed in appendix J. Sections 4.2 and 4.3 present the way in which the program itself is stored. The implementation of functions used to create the program data structure of programs can be found in module *struct.c*. This section discusses the implementation issues in more detail than the previous sections did, since it is also intended to be a documentation about the language dependent choices made, and these are less straightforward than those made while implementing the previous phases.

All the type information about the program data structure and the symbol table can be found in module *prgtype.h*, which is listed in appendix F, and which must be included in those modules.

4.1 The Symbol Table

For every identifier a new entry is created in the symbol table. The attributes that belong to every entry are enumerated below.

- *f_symbol*[*i*]: This attribute is an *integer* index in a dynamic array of characters. In this array the characters of the name of every identifier (lexeme) are stored, using the next available space in that array. Every identifier is terminated with a '`\0`' character, so that it can be printed to output as a string. In this fashion no memory is wasted for allocating the maximum allowed number of characters (6) per entry, because the strings of successive identifiers are stored consecutively. If the size of the dynamic array of characters is not sufficient, more memory is (re)allocated. This technique is illustrated in the following picture, in which the symbol table is shown after the identifiers `I`, `LONGID`, and `RE` are stored consecutively.

  The reason an *integer* index has been chosen instead of a pointer to a (character) memory position, is that if more character memory is reallocated, the absolute positions in memory of the characters already stored, may change. There is no need for changing these pointers, if the relative addressing technique is used.

- *f_dim*[*i*]: This attribute contains an integer that holds the dimension of the identifier. If it is a scalar variable this dimension is 0. To indicate that the identifier is used as a (scalar) parameter, this integer is set to -1. The value of the parameter can be found in another attribute.

- *f_type*[*i*]: This attribute holds an integer containing the representation of the type of the identifier. The represented type can be INTEGER, REAL or LOGICAL.

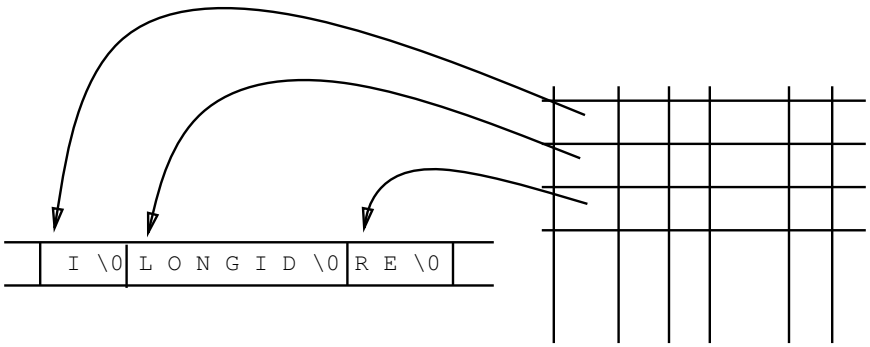


Figure 3: Storing Lexemes in the Symbol Table

- $f\_val[i]$: This attribute contains the value of the identifier if it is used as a parameter. This value is always of the type given in the entry $f\_type[i]$, since type conversion has been applied on values of a different type in parameter definitions, as is stated in the previous section. This attribute is implemented as a union which can contain values of the three different types.

- $f\_dimlist[i]$: This attribute is a pointer to a list containing the allowed subscript ranges per dimension. Because a list structure is used instead of allocating memory for the maximum allowed number of subscript ranges (7) in advance, a more efficient use of memory results. This list can be linked together after the array declaration has been parsed. The bounds of subscript range are always values of type INTEGER, since type conversion has been applied on the expressions that are not of type INTEGER in array declarations before storing these ranges.

Whenever during the parsing phase of a program more entries in the symbol table are necessary than is allocated initially, more memory will be allocated. So, the number of identifiers that can be stored in the symbol table is only bound by the run-time available memory of the machine on which the compiler is currently executed.

After the program has been correctly parsed and no semantic errors have been detected, the information stored in the symbol table is written in a readable format into the text file *program.sym*. The user can examine this information with the command **symtb**, which shows this file using the UNIX command

less.

The modules using the symbol table can insert new identifiers, update and retrieve information stored in the symbol table by using interface functions only. So, the implementation of the symbol table is only known in module *symbol.c*. The functions for allocation and maintenance of the symbol table memory are also local to this module in order to hide the implementation issues from other modules.

4.1.1 An Example of a Symbol Table

The following program contains declaration statements, parameter statements and one assignment statement in which a variable occurs that has not been declared.

```
PROGRAM SYMBOL

INTEGER   I, J, K, N
REAL      R
LOGICAL   LO1, LO2
PARAMETER ( N = 100, K = 2.0 * 100 )
PARAMETER ( R = 50.0 )
PARAMETER ( LO1 = (.TRUE. .AND. .FALSE.) )
REAL      A(N), B(2 * N), C(300.0,20)

P = 3.0

END
```

The resulting symbol table is shown below. The identifier `SYMBOL` is the only identifier without a type, because it is only used as identifier in the program header. So, the implicit data typing rules have not been applied. The values of all the expressions in the parameter statements are computed and if necessary converted to the right type. According to the implicit data typing rules, the identifier `O` is of type REAL, and this information is stored in the symbol table. The bounds of the subscript ranges can be arbitrary expression provided that they can be evaluated at compile-time, and are computed and stored in the symbol table, after possibly type conversion has been applied.

```
=> symtb

Symbol table:

Id      Dim           Type          Value        Bounds

SYMBOL  -             Undefined     -
I       Scalar        INTEGER       -
J       Scalar        INTEGER       -
K       Parameter     INTEGER       200
```

```
N       Parameter     INTEGER       100
R       Parameter     REAL          50.0
LO1     Parameter     LOGICAL       .FALSE.
LO2     Scalar        LOGICAL       -
A       1             REAL          -            ( 1:100 )
B       1             REAL          -            ( 1:200 )
C       2             REAL          -            ( 1:300 1:20 )
P       Scalar        REAL          -
```

4.2 Declaration Statements

All the information contained in the declaration and parameter statements will be stored in the symbol table. Since all the occurrences of parameter variables in the rest of program will be replaced with its value found in the symbol table, there is no need to store the parameter statement explicitly. The declaration statements can be reproduced from the information found in the symbol table. So neither declaration nor parameter statements need to be stored. Note that since the symbol table will be used to generate declaration statements for the resulting FORTRAN 90 code, variables which were defined implicitly in the original program, will have an explicit declaration statement in the resulting program. This generation of declaration statements will be discussed in section 9, which covers the implementation of the unparsing phase of the program in memory.

4.3 Executable Statements and Expressions

All executable statements must be stored in memory in order to be able to apply data dependence analysis and transformations to the code. First the general way of creating the data structure is presented. After that, the structures for specific statements are discussed. Remember that the functions for creating the data structure of a program are also executed during the parsing of a program.

The main data structure that is created for a program is a list of statements. Every element of this statement list consists of a representation for one particular statement and a pointer to the next statement in the program. This structure is shown in the following picture.

During the parsing phase an expression stack and a statement stack are maintained. The element of these stacks are pointers to the data structure of respectively expressions and statements. Whenever an expression or an executable statement has been parsed, its data structure is created and a pointer to this structure is pushed on the stack. If these data structures are needed again to build a bigger data structure from earlier created data structures which are still on the stack (e.g. a statement list is created from the successive statements in the program, the left-hand-side and right-hand-side expressions are needed to build an assignment statement), they can be popped of the stack. When the bigger data structure has been built, a pointer to it is pushed back on the stack. In this fashion the total data structure can be built in an incremental way. The only pointer on the statement stack after the parsing phase has terminated cor-

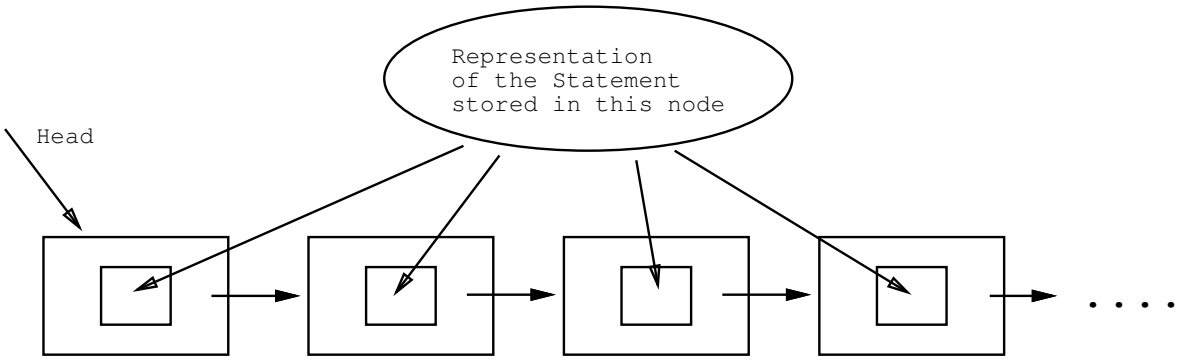


Figure 4: Main Data Structure

rectly, is the pointer to the first statement in the statement list, which is also the first statement in the program. This pointer is saved in a variable (*head*) which can be used as entry to the program data structure.

During the creation of a statement list from the separate statements, the flow of control check, mentioned in section 3.3, can be executed. It is tested if no statement appears after a STOP statement. The only exception on this rule is, that an ELSE or ELSE IF 'statement' (remember that they are treated as normal statements when building their data structure) may appear after a STOP statement, since the flow of control can never enter more than one branch of the same IF statement. Therefore, whenever other statements after a STOP statement are detected, a warning is generated, and the following statements are scanned and deleted from memory up to the first ELSE or ELSE IF 'statement'. This last statement and its following body are linked then to the previous statement.

The expression stack is empty after a correct program has been parsed, since all the expressions have been used in creating the symbol table [7] or executable statements. Since the data structure of expressions is a part of the executable statements data structure, this is discussed first.

Every expression is represented as a structure *expr_node*. The first field of this structure hold the kind of expression that is represented in this particular node (*kind*). The second field is a union *u* which can contain different attributes, depending on the kind of expression represented. The attributes for every kind of expression are discussed below.

4.3.1 Constants

The type and value of the constant are stored in the fields *type* and *val*. The value is stored in a union field, which can contain one value of type INTEGER,

[7] Expressions are also used while building the symbol table, see section 3.1.

REAL or LOGICAL. Note that the type must be stored in order to be able to retrieve the right type of value.

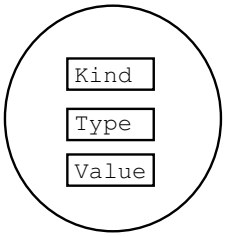


Figure 5: Constant Data Structure

4.3.2 Variables

The symbol table entry of the variable is stored in the field *entry* (so all information stored in the symbol table can be obtained by using the functions of module *symbol.c*) and a pointer to a subscript list is stored in the field *dim_list*. These consecutive node of this subscript list contain a pointer to the corresponding subscript expression of the variable (*head*), and a pointer to the rest of the subscript list (*tail*). The use of a list instead of static memory, needs no more explanation by now. The semantic analysis phase has already determined that the number of following subscript expressions equals the dimension of that variable. The pointer is set to NULL in case of a scalar variable. The subscript list of array variables can be created by popping the correct number of subscript expression from the expression stack (since they have already been parsed, they can be found there) and linking them together. Together with a pointer to the subscript expression used in the program, a pointer to the so-called *normal form* of a subscript expression is stored (*normexpr*) in every node of the subscript list. This *normal form* expresses a subscript expressions in terms of surrounding loop-control variables ($a_0 + a_1 i_1 + ...a_n i_n$, where the $i_k, 1 \leq k \leq n$ are loop-control variables), and is necessary to simplify the data dependence analysis phase. If this *normal form* cannot be computed (because the expression is too complicated [8], or variables that are no loop-control variables are used in the expression), this pointer is set to NULL. Whenever a user wants to see the program in memory using the command **showprg**, every subscript expression that can be transformed into *normal form*, is shown using angle-brackets (< ... >) to inform the user about the knowledge that the compiler has about that subscript expression. This enables him to rewrite subscript expressions that are too complicated for the compiler, avoiding the possible assumption of too many data dependences. Note that the rewriting of subscript expressions to their *normal form* which will be used in data dependence analysis, may result in subscript expressions in which the evaluation order may change. Since a special test on REAL typed constants has been included (only if all REAL constants in the subscript expression have an empty fractional part, the subscript expression

[8] If operators different from '+', '−', '∗' or the unary '−' are applied on the loop-control variable inside the subscript expression, the expression cannot be brought into *normal form*.

can be rewritten into *normal form*), and because the data dependence analysis phase will take the use of loop-control variables that are not of type INTEGER into account, this will not result in the assumption of wrong data dependences. The following demonstration of the conversion into *normal form* illustrates the power of this rewriting, since the resulting subscript expression are more suitable for automatic analysis. The assignment labels that appear in the resulting program are also given in the original fragment.

```
          PROGRAM NORMAL
          INTEGER I, J
          REAL    A(10,10), R
    S1:   A(R,2) = 100.0
          DO 2 I = 1, 10
            DO 1 J = 1, 10
    S2:       A(I,J) = A(I + 0.5,2.0*J + 1.0)
    S3:       A(2*I + 3*J + 2 + I - 2,-J + 6+ 10*I+ 3*I + J-2) = 10.0
  1         CONTINUE
  2       CONTINUE
          END
```

The first subscript expression of assignment statement S_1 cannot be converted into *normal form*, since it is not used as an loop index. The use of a REAL constant with a non-empty fractional part (0.5) results in the inability to rewrite the first subscript expression in the reference to A on the right-hand-side of the assignment S_2. The REAL constants in the second subscript do not prohibit the conversion into *normal form*, since they have an empty fractional part. The complicated expressions appearing in S_3 of the original program can be simplified to the *normal form*s shown below.

```
...
S1: A(R, <2> ) = 100.0
L1: DO I = 1, 10, 1
  L2: DO J = 1, 10, 1
    S2: A( <I> , <J> ) = A((I + 0.500000), <2*J+1> )
    S3: A( <3*I+3*J> , <13*I+4> ) = 10.0
  ENDDO
ENDDO
...
```

The original subscript expressions, however, will be used again in generating the resulting FORTRAN 90 program, so the *normal form*s will only be used in the data dependence analysis. The following picture illustrates the storage of a variable.

If the variable has been defined as a parameter, the data structure for a constant with value as found in the symbol table is created instead. The direct use of the associated constant instead of the variable, is called **constant folding**.

After a variable has been parsed, its symbol table entry can be found in the attribute of the nonterminal *var*. All other information can be retrieved

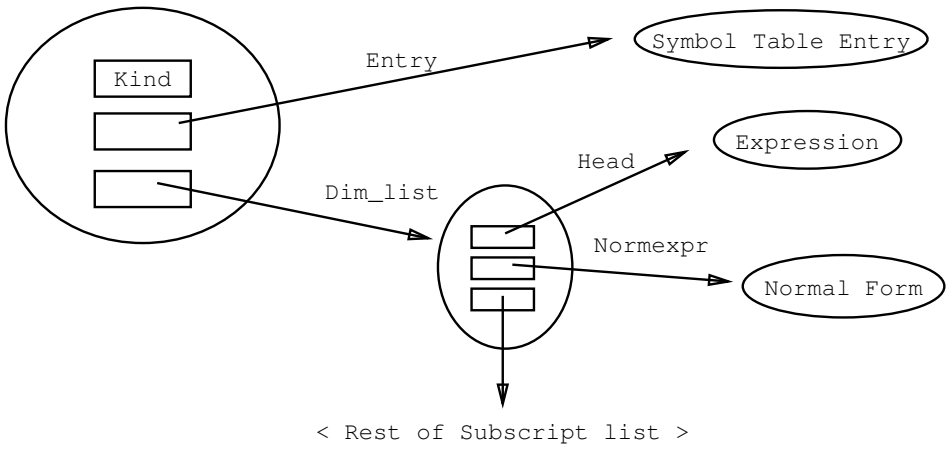


Figure 6: Variable Data Structure

from the symbol table, using that entry. The representation is created and the pointer to this data structure is pushed on the expression stack.

4.3.3 Operators

Pointers to the argument expression(s) of operands are stored in the fields $arg1$ and $arg2$. Both pointers are needed in the case of binary operators, but for an unary operator, only one pointer is needed, so the other pointer is set to NULL. The kind of operator represented is explicitly stored in the *kind* field of the data structure for this expression.

If an expression is encountered during the creation of the data structure, which can be evaluated at compile-time in the same way as it would be evaluated at run-time, the representation of the resulting constant is created instead [9]. This is a more general form of **constant folding**. So, the expression `3.0 * 2.0 + A` can be transformed into `6.0 + A`, `A + 2.0 * 3.0` into `A + 6.0`, but the expression `3.0 * A * 2.0` cannot be evaluated any further, since the subexpression `3.0 * A` will be evaluated first at run-time. Changing the order in which subexpression are evaluated may change the semantics of a program, as round-off error may accumulate differently in expressions of type REAL. The same procedure is used for evaluating expressions of type INTEGER and LOGICAL. By incrementally evaluating expressions while building the corresponding data structures, composite constant expressions are transformed into the representation of the resulting constant value. The resulting value will be converted to the type of the resulting expression first if required.

Most operations performed are straightforward but a few require some explanation. Whenever the '/' operator has a zero right operand (0, 0.0 or .FALSE.) a division by zero error is generated. Whenever the `.EQV.`, `.NEQV.`, `.AND.`, `.OR.`, or `.NOT.` are applied resulting in an integer expression, the following operations on the binary representation of both operands (o_1 and o_2) are applied respectively: $(-1)\hat{}(o_1\hat{}o_2)$, $o_1\hat{}o_2$, $o_1\&o_2$, $o_1|o_2$, and $(-1)\hat{}o_1$ (in which

[9] Note that the representation of the operands must be deleted from memory.

`^`, `|`, and `&` are the operators for bitwise *exclusive or*, bitwise *or*, and bitwise *and* respectively). If the operation cannot be applied on its operands (e.g., `.AND.` with operands of type REAL), an error has already been generated, so the result does not need to be computed.

Because this technique is also applied on the expressions that appear in PARAMETER statements, the compiler can easily determine if the associated expressions can be evaluated at compile-time, since in that case a constant expression is on top of the stack as a single parameter definition has been parsed. The storage of an operator is illustrated in the following picture.

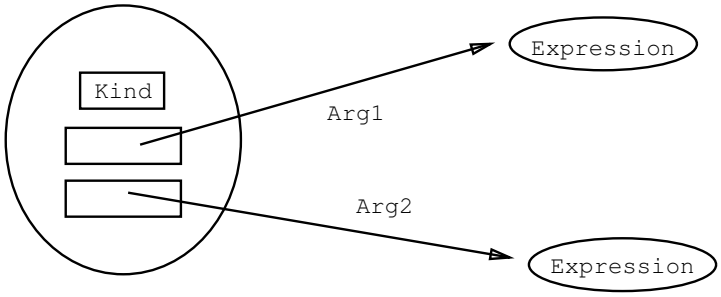


Figure 7: Operator Data Structure

Statements are stored in *stmt_node* structures. This structure contains a field which indicates the type of statement stored in that particular structure (*kind*), a pointer to the representation of the next statement in the program (*next*) and a union *u* which can contain different attributes, depending on the kind of statement stored.

All assignment statements, IF statements and DO-loops are numbered in lexical order. Assignment statements are numbered as S_i ($i \geq 1$), to enable the referencing of particular assignment statements in the presentation of the data dependences in that program [10]. For the same reason, logical-IF and general-IF statements are numbered as C_i ($i \geq 1$) and DO-loops as L_i ($i \geq 1$). This numbering of particular statements is shown when the program is listed using the command **showprg** during a session, but naturally they do not appear in the resulting FORTRAN 90 program text. In order to administrate this numbering, three different variables are reset to zero before a program is processed and are increased by one before stored in the data structure of the particular statements. Whenever an assignment statement, an IF statement, or a DO-loop has been parsed the corresponding counter is incremented. The following fragment demonstrates the way in which the numbering is presented to the user when the command **showprg** is given. Note that the assignment statements inside the conditional statements have their own numbers S_1 and S_3.

```
C1: IF (N .GT. 10) S1: N = 10
L1: DO I = 1, N, 1
```

[10] A more detailed presentation of data dependences will be used than the presentation on assignment statements only, see section 5.

```
      S2: A(I) = B(I) + 10.0
      C2: IF (I .GT. 5) S3: C(I) = C(I) + 1.0
    ENDDO
```

4.3.4 DO-loops

The attributes of the data structure for DO-loops consist of a pointer to the loop-control variable (*index*), three pointers to the lower, upper bound and stride expressions (*expr*1, *expr*2 and *expr*3), a pointer to the data structure (statement list) representing the body of this loop (*body*), and the number of the loop (*loopno*). If no explicit stride is given, an integer constant 1 with *expr*3 pointing to it, is created. In this fashion, only DO-loops with explicit strides appear in the data structure for DO-loops (and in the resulting program). This decision has been made, to prevent the need for two different patterns in the transformation language, which often results in many different transformations [11]. which all, in fact, describe the same transformation.

Because during data dependence analysis it must be possible to reach following statements not in a loop body, from within that loop body, the statement list representing the body of a loop does not end with a NULL pointer, but with a special marker, called LINK_UP. The next field of this marker points to the representation of its surrounding DO-loop. In this fashion statements after this DO-loop can be reached by following the next pointer of the DO-loop pointed to by the marker.

Whenever a DO-loop has been parsed, all the expressions necessary to create its data structure can be found on the expression stack, and the representation of its loop body can be found by popping the pointer on top of the statement stack, since these data structures have already been created in the actions of recently used grammar rules.

An extra field *ext* is used to indicate if it concerns a serial DO-loop or a parallel DOALL-loop. Since only serial DO-loops are allowed in the FORTRAN 77 dialect, this field is always set to the representation of serial loops and can only be set to the representation of parallel loops during the transformation phase. The data structure of a DO-loop is shown in the following picture.

4.3.5 Assignment Statements

The attributes of an assignment statement are *lhs* and *rhs*, pointing to respectively the left-hand-side expression (a variable) and the right-hand-side of this assignment. A third field *stmtnod* contains the number of this assignment statement. The two expressions needed to create this data structure can be found on the expression stack, once an assignment statement has been parsed. The data structure is illustrated in the following picture.

[11] Consider for example, a pattern with two DO-loops. If the pattern must specify if an explicit stride is present, 4 patterns result.

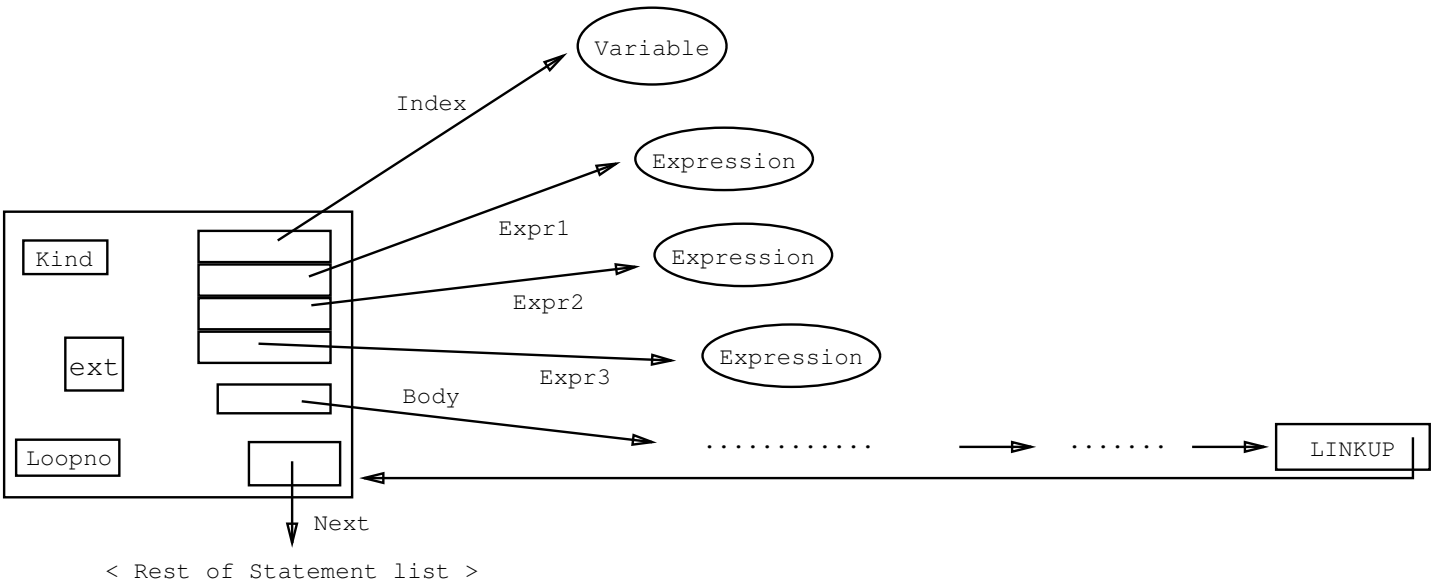


Figure 8: DO-loop Data Structure

4.3.6 Logical-IF and General-IF Statements

The attributes of both the logical-IF and the general-If statements are a pointer to the expression acting as condition (*condition*), a pointer to the data structure (statement list) for the body of this IF statement (*body*), and the number of its condition *condno*. The kind of IF statement stored (Logical-IF or General-IF) is explicitly recorded in the *kind* field. The statement list representing the body of IF statements is closed with a special LINKIF_UP marker for the same reason mentioned before.

The associated condition can be found on the expression stack and a pointer to the data structure of the body can be found on the statement stack, once an IF statement has been parsed. If this condition can be evaluated at compile-time, and is .TRUE. in the case of a logical-IF statement, only the associated statement is stored in the program data structure, and a warning is generated to report this form of **constant folding** to the user. If the value if .FALSE., the whole logical-IF statement is stored, because otherwise no trace of the original program structure is left behind.

Remember that the ELSE and ELSE IF branches are only implicit in the data structure, as the representations of ELSE and ELSE IF are stored as 'normal' statements between the other statements in the statement list of the loop body. The data structure is presented in the following picture.

4.4 STOP statements

Since STOP statements do not have any attributes, they can be stored in a node without any additional information.

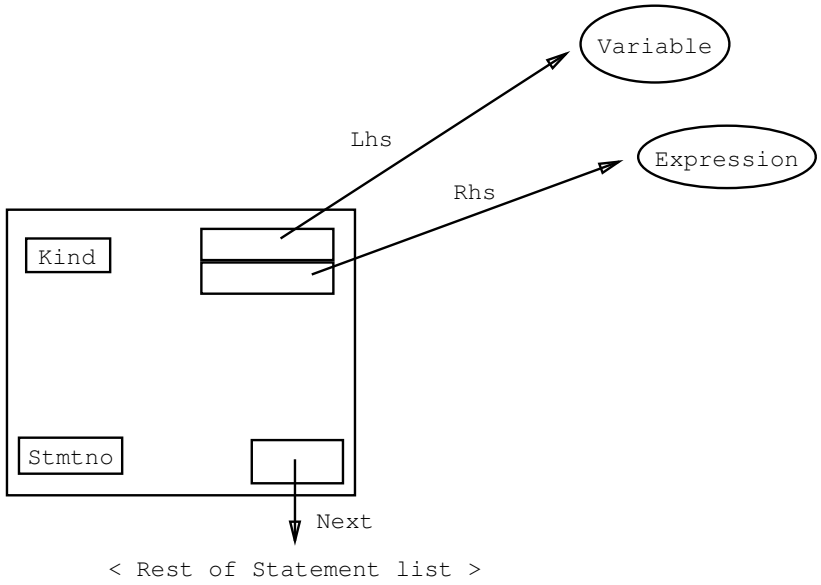


Figure 9: Assignment Statement Data Structure

4.5 Memory Management

In module *mem.c*, the implementation of general memory management functions can be found, together with the implementation of the routines for the four different stacks (the integer stack, the expression pointers stack, the DO-loop management stack and the statement pointers stack) and the functions responsible for releasing [12] the memory occupied by the program data structure elements. If a syntax or semantic error is detected, a garbage collecting function of this module can be called, which will release all the memory occupied by all the (partial [13]) data structures stored. When a new program is read, the program in memory must be deleted first which is also done by functions from module *mem.c*. This module is listed in appendix K.

4.6 Some examples

Although the listing of a program in memory does not show all the concepts discussed in this section, it reflects the way in which a program has been stored. Therefore the result of the command **showprg** is given below when the following FORTRAN 77 program has been read.

```
PROGRAM INT

INTEGER I, N
PARAMETER (N = 200 + 100 * 2)
```

[12] Remember that during the creation of the symbol table, some intermediate data structures can be deleted.

[13] If the parsing phase does not terminate normally, the partial data structures have not been linked together yet.

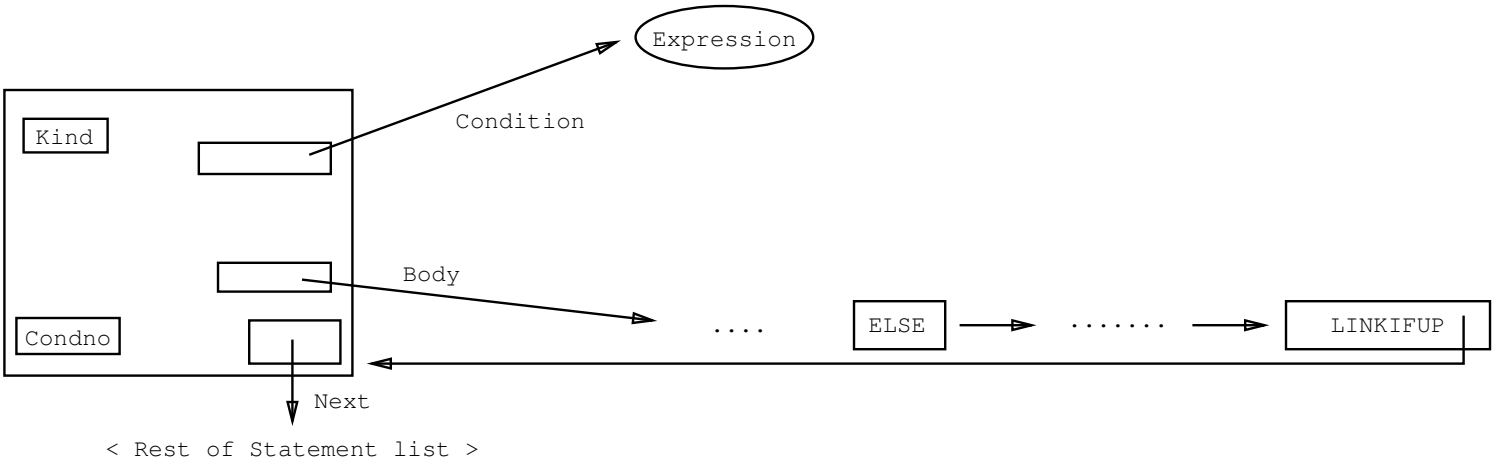


Figure 10: IF Statement Data Structure

```
      REAL   A(N + 5), B(N + 10), R

      R = 10.0 - 5.0 / 5.0

      DO 10 I = 1, N + 5
        IF (.TRUE.) A(I) = 20.0 * 2.0 + R
        B(I) = 10.0 + 10.0 + 10.0 + 10.0 * R + 5.0
10    CONTINUE

      IF (.FALSE. .EQV. .FALSE.) THEN
        R = 3.0 + 6.0 * 2.0
        STOP
        R = 3.0
        R = 4.0
      ELSE
        R = 4.0 / 2.0 + 10
      ENDIF

      STOP

      END
```

In the resulting program, the expressions have been partially evaluated, as far as is allowed without changing the evaluation order, which would occur during run-time. The implicit stride of 1 is made explicit. The .TRUE. value in the logical-IF statement has been used to eliminate the condition from the statement, but has not been eliminated in the general-IF, because that would require the elimination of all other branches. The statements in the first branch of the general-IF that appear after STOP have been deleted. Why the layout of some statements is different from the one in the original program, is explained in section 9.

```
=> readprg int.f

PROGRAM INT
- Warning: IF (.TRUE.) ignored (line 10)
- Warning: statement after STOP cannot be reached (line 21)

Computing Data Dependences

=> showprg

      PROGRAM INT

      INTEGER I
      REAL    A(1:405)
      REAL    B(1:410)
      REAL    R

      S1: R = 9.000000
      L1: DO I = 1, 405, 1
        S2: A( <I> ) = (40.000000 + R)
        S3: B( <I> ) = ((30.000000 + (10.000000 * R)) + 5.000000)
      ENDDO
      C2: IF (.TRUE.) THEN
        S4: R = 15.000000
        STOP
      ELSE
        S5: R = 12.000000
      END IF
      STOP

      END
```

5 Data Dependences

Information of data dependences reflecting the flow of data in a specific program, is essential to the restructuring compiler. Without this information it is not possible to determine if a certain transformation preserves the semantics of the original fragment. Before the computation of data dependences is discussed, the basic concepts of data dependences are summarized, together with some minor extensions to the existing theory, in order to be able to express more than only data dependences between assignment statements, and to define some concepts more precisely. Further discussion of data dependences can be found in [Pol88], [Bik91], [ZC90] and [Wol89].

Two kinds of statements can be distinguished: *scalar statements* and *indexed statements*. *Indexed statements* are statements that appear inside a loop body, so they are under explicit control of a loop-control variable, or vector statements which are under implicit control. The number of loops surrounding an indexed statement is called the degree of that statement. The notation for an indexed assignment statement of degree k is $S_i(I_1,...,I_k)$. A similar notation can be used for IF statements and even for DO-loops inside other loops. The variable I_j indicates the j^{th} loop-control variable. An instance of an indexed statement is obtained by substituting every loop-control variable by the appropriate value. In a simple loop for example, with one loop-control variable that runs from 1 to N, there are N instances of every (indexed) statement inside that loop. This is illustrated in the next example.

```
L1: DO  I = 1, 3, 1
  S1: A(I) = B(I)
  C1: IF ( B(I) > 10 ) ...
ENDDO
```

In this example, $S_1(I)$ and $C_1(I)$ are indexed statements. $S_1(1)$, $S_1(2)$, and $S_1(3)$ are the instances of the indexed assignment statement, indicating `A(1) = B(1)`, `A(2) = B(2)`, and `A(3) = B(3)` respectively. In the following presentation of some concepts, only assignment statements are considered. The same concepts, however, can be defined on the other kind of statements.

For an indexed statement S_l of degree k, with first instance $S_l(1,...,1)$ and last instance $S_l(N_1,...,N_k)$, the number of different instances N is given in the following formula.

$$N = \prod_{r=1}^{k} N_r$$

Note the difference between an indexed statement and its instances. If the loop-control variables of a particular indexed statement are known or do not matter at that moment, S_1 can be used to refer to $S_1(I)$.

Scalar statements are statements that are not under explicit or implicit control of a loop-control variable. Scalar statements can be considered to be indexed statements of degree 0, so it can be stated that these statements only have one instance.

The order of execution in a program is a relation defined on the *instances* of statements. It reflects the order in which instances of statements are executed during the sequential execution of the program. $S_l(\vec{i}) <_O S_m(\vec{j})$ indicates that the instance of S_l belonging to iteration $\vec{i}$ will be executed before the instance of S_m belonging to iteration $\vec{j}$. The notation $S_l(\vec{i}) \leq_O S_m(\vec{j})$ is used if it is possible that $S_l(\vec{i})$ and $S_m(\vec{j})$ are the same instances of one statement ($l = m$ and $\vec{i} = \vec{j}$). So if S_l and S_m have the same degree k, $S_l(i_1,...,i_k) <_O S_m(j_1,...,j_k)$ if $(i_1,...,i_k) <$ [14] $(j_1,...,j_k)$ holds. If these two vectors are the same ($\vec{i} = \vec{j}$ holds), $S_l <_O S_m$ holds if S_l precedes S_m in the program ($i < j$). If these statements do not have the same degree, only the loop-control variables in common are considered, so the iteration vectors $\vec{i}$ and $\vec{j}$ used in the comparison have the same dimension. For example, the execution order between two scalar statements, in which S_l lexically precedes S_m ($l < m$), is denoted by $S_l <_O S_m$.

The definitions of the set IN (all the variables that are read by an assignment statement S, a conditional statement C or a DO-loop statement L) and the set OUT (the variables that receive a value from an assignment statement S or a DO-loop statement L [15]), are trivially extended for instances of indexed statements. The set $IN(S_l(i_1,...,i_k))$, for example, consists of all the variable instances that are read by $S_l(i_1,...,i_k)$. This is illustrated in the following example.

```
L1: DO I = 1, N, 1
  S1: A(I) = B(I) + C(I) + D(Z)
ENDDO
```

$OUT(S_1(1))$ = { A(1) } and IN($S_1(1)$) = {B(1), C(1), D(Z), Z}. The variable Z is read in order to compute the subscript expression of the array `D`. For reasons discussed below, it is decided to exclude the loop-control variable I from the IN set. The set $IN(L_1)$ = { N } and $OUT(L_1)$ = { I } (since the degree of L_1 is 0, it has only once instance).

The data dependence relation is defined on scalar statements first, and this definition will be extended on indexed statements afterwards. For the sake of simplicity, only assignment statements are considered.

5.1 Data dependences on Scalar Statements

Four types of data dependences between scalar statements can exist, **Flow Dependences**, **Anti Dependences**, **Output Dependences** and **Input Dependences**.

[14] Lexicographic Ordering, in which the $i_r <j_r$, for $1 \leq r \leq k$, on the elements indicates that the value i_r occurs before j_r in the iteration space of the loop-control variable I_r. If the stride of the loop is positive, it is the conventional '<' relation.

[15] The loop-control variable receives a certain value as the loop has been executed (the first value that exceeds the upper bound if the stride is positive). Because all the instances of statements inside the loop body are under control of this variable, these instances will not have that variable as element in their IN set, for reasons that will be discussed later on, and since it is forbidden to assign to a loop-control variable, it also cannot be an element of OUT set. It is not important if the OUT set of a DO-loop statement can be seen inside its body, since this decision results in the fact that no data dependences can hold between a DO-loop and the statements inside its body.

5.1.1 True Dependence or Flow Dependence

A true dependence holds between two assignment statements S_i and S_j, if $S_i <_O S_j$ and $OUT(S_i) \cap IN(S_j) \neq \emptyset$ and $\forall_{S_k}\ S_i <_O S_k <_O S_j : OUT(S_k) \cap (OUT(S_i) \cap IN(S_j)) = \emptyset$. This relation is denoted by $S_i \delta S_j$. Note that the last condition reflects the fact that a direct flow dependence only exists between a statement writing a particular value and the statement reading this value. In the following example $S_1 \delta S_2$ holds, caused by the fact that $S_1 <_O S_2$ and $OUT(S_1) \cap IN(S_j) = \{ A \}$. The dependence is said to be caused by the occurrence of variable `A` in both statements [16].

```
S1: A = <expression>
S2: <variable> = A
```

Note the need for strictness in the execution order ($<_O$ and <u>not</u> $\leq_O$), since a flow dependence can never occur in the same statement.

5.1.2 Anti Dependence

An anti dependence between two assignment statements S_i and S_j holds, if $S_i \leq_O S_j$ and $IN(S_i) \cap OUT(S_j) \neq \emptyset$ and $\forall_{S_k}\ S_i \leq_O S_k <_O S_j : OUT(S_k) \cap (IN(S_i) \cap OUT(S_j)) = \emptyset$. An anti dependence is denoted by $S_i \overline{\delta} S_j$. Note the fact that $\leq_O$ is used in the definition now, because an anti dependence can occur in one statement as illustrated in the following example where the anti dependence $S_1 \overline{\delta} S_1$ caused by the two occurrences of variable `A`, holds.

```
S1: A = A + <expression>
```

A more complicated example is given in the following fragment, in which the anti dependence $S_1 \overline{\delta} S_2$ holds, caused by the fact that variable $Z \in IN(S_1)$ and $Z \in OUT(S_2)$.

```
S1: <variable> = A( Z )
S2: Z = <expression>
```

5.1.3 Output Dependence

An output dependence $S_i \delta^o S_j$ holds if $S_i <_O S_j$ and $OUT(S_i) \cap OUT(S_j) \neq \emptyset$ and $\forall_{S_k}\ S_i <_O S_k <_O S_j : OUT(S_k) \cap (OUT(S_i) \cap OUT(S_j)) = \emptyset$. So in the following fragment the output dependence $S_1 \delta^o S_2$ holds, because `A`$\in OUT(S_1)$ and `A`$\in OUT(S_2)$.

```
S1: A = <expression>
S2: A = <expression>
```

[16] In general, it can be said that the data dependences, defined in this section, are caused by the occurrence of variables in the corresponding intersections.

5.1.4 Input Dependence

If $S_i <_O S_j$ and $IN(S_i) \cap IN(S_j) \neq \emptyset$ and $\forall_{S_k}\ S_i \leq_O S_k <_O S_j : OUT(S_k) \cap \{variable\}$ [17] $= \emptyset$, then these two statements are involved in an input dependence, denoted by $S_i \delta^i S_j$. So, in the following example the input dependences $S_1 \delta^i S_2$ holds, because A$\in IN(S_1)$ and A$\in IN(S_2)$, and $S_3 \delta^i S_4$ holds, because Z$\in IN(S_3)$ and Z$\in IN(S_4)$,

```
S1:  <variable> = A
S2:  <variable> = A
S3:  <variable> = B( Z )
S4:  <variable> = C( Z )
```

An assignment statement S_j is **indirectly dependent** on another statement S_i, if there exist statements S_1,...,S_k, such that the following dependences hold: $S_i \delta^* S_1$, $S_1 \delta^* S_2$,..., $S_{k-1} \delta^* S_k$, $S_k \delta^* S_j$, where δ^* can denote any kind of data dependence. So, in the following example the dependences $S_1 \delta^o S_2$ and $S_2 \delta S_3$ hold. Therefore, S_3 is indirectly dependent of S_1. Note that formally $S_1 \delta S_3$ does not hold, reflecting the fact that S_3 only uses the value of S_2. However, during the data dependence computation, discussed in section 6, this data dependence will also be determined, but since an indirect dependence exists, there is no real need for filtering these virtual dependences out.

```
S1: A = <expression>
S2: A = <expression>
S3: <variable> = A
```

In any kind of dependence $S_i \delta^* S_j$, S_i is said to be the **source** of the dependence, and S_j is the **sink** of the dependence.

5.2 Data Dependences on Indexed Statements

The data dependence relations defined on scalar statements, can be extended to hold on indexed statements as well.

> A **static dependence** exists between two indexed statements if a dependence as defined earlier exists between *at least* one pair of instances of the two statements.

Since scalar statements can be seen as indexed statements, with only one instance, the definition can be applied on both kinds of statements now.

It is important to realize that a static dependence between two indexed statements is formed by underlying dependences between statement instances. Therefore, it is possible that one static dependence is caused by more than one dependence between pairs of statement instances. If it is clear which level of data dependence is meant, the additional 'static' is often omitted.

[17] Note the difference with the other conditions, since the intersection of two IN sets in not necessarily a singleton set. The *variable* that must be taken, is the variable possibly causing the dependence.

Note the need for the extension of the definition of the IN and OUT sets on indexed statement instances, in order to be able to focus on the variables used in one particular statement instance. The intersection of two sets of indexed statement instances consists of a set of variable instances, which can be scalar variables or array elements.

A static dependence between two indexed statements is denoted by the same notation as above, with an additional direction flag for every loop-control variable of the loops surrounding *both* statements, indicating the direction in the iteration space in which the dependence between underlying instances hold. These loops will be referred to as the *common nest* from now on. The direction flag is dependent on which iterations the instances of the indexed statements that are responsible for the static dependence, are executed in. The following example illustrates the use of a direction flag.

```
L1: DO I = 1, N, 1
  S1: A(I) = B(I)
  S2: C(I) = A(I)
ENDDO
```

The only static flow dependence that holds in this loop is denoted by $S_1\delta_{=}S_2$, because a flow dependence holds between every two instances of S_1 and S_2 belonging to the same iteration of the loop. That is why the direction flag '=' is used. Note that the loop-control variable is omitted in the notation of the static dependence between the indexed statements.

It is also possible that static dependences hold when dependences between instances of indexed statements of different iterations in a loop hold. In the next example, the static dependence $S_1\delta_{<}S_2$ holds.

```
L1: DO I = 2, N, 1
  S1: A(I) = B(I)
  S2: C(I) = A(I - 1)
ENDDO
```

The '<'-direction is used because all the underlying dependences hold between an instance of S_1 of one particular iteration and an instance of S_2 of a *later* iteration. This static dependence indicates that the instances of this DO-loop can be executed in any order, as long as the execution of every instance of S_1 precedes the execution of the instances of S_2 of later iterations. $S_1(1)$ must precede $S_2(2)$, for example. Thus `A(1) = B(1)` must be executed before `C(2) = A(1)`, because otherwise `C(2)` would not receive the value of `B(1)`, which happens according to the semantics of serial execution of this DO-loop.

A static anti dependence is illustrated in the next example, in which $S_2\overline{\delta}_{<}S_1$ holds.

```
L1: DO I = 1, N, 1
  S1: A(I) = B(I)
  S2: C(I) = A(I + 1)
ENDDO
```

This fact can be easier noticed if some of iterations the loop are enrolled, as shown in the following table.

Index	Instance
I = 1	A(1) = B(1) C(1) = A(2)
I = 2	A(2) = B(2) C(2) = A(3)
...	....

The execution of $S_2(1)$ must precede $S_1(2)$ to preserve the correct value of `C(1)` afterwards. So, there exists a static dependence between these two indexed statements, because $S_2(1) \leq_O S_1(2)$ and the intersection between the corresponding IN and OUT sets is nonempty. Because `I = 1` belongs to an earlier iteration than `I = 2`, the direction '<' is used. Dependences that hold between statement instances of different iterations are called cross iteration or loop carried dependences.

If the iteration space of the original loop was traversed in reverse order (i.e. `DO I = N, 1, -1`), the data dependence between statement instances would turn around, resulting in the static flow dependence $S_1\delta_< S_2$. Again the direction '<' is used because the dependence holds between for example, the statement instances of `I = 1` and `I = 2` (and the first one is executed before the second). So, note that a '>' direction [18] can never occur as direction flag of a static dependence in a *single* loop.

The sequence of directions is called the **dependence direction vector**. An example of a data dependence vector of length two is given below.

```
L1: DO I = 1, N, 1
  L2: DO J = 2, N, 1
    S1: A(I,J) = A(I,J - 1)
  ENDDO
ENDDO
```

The only static dependence in this fragment is denoted by $S_1\delta_{=,<}S_1$. This kind of dependence is called a self dependence, because the static dependence holds between different instances of the *same* statement.

The fact that only direction flags for the loop-control variables in the *common nest* are given, is illustrated below. The *common nest* of S_1 and S_2 is empty in this example.

```
L1: DO I = 1, N, 1
  S1: A(I) = <expression>
ENDDO
L2: DO I = 1, N, 1
  L3: DO J = 1, N , 1
    S2: B(I,J) = A(I)
  ENDDO
ENDDO
```

[18] Which can occur in nested loops.

Since the values written in the first DO-loop are read in the second DO-loop, a flow dependences between the instances of S_1 and S_2 exists. It is, however, not possible to annotate the resulting static dependence $S_1 \delta S_2$, with a direction flag (for example a single '=' to reflect the fact that the dependences holds between statement instances belonging to iterations in which `I` has the same value), since all the iterations of the first DO-loop are executed before the iterations of the DO-loops L_2 and L_3, and direction flags are only used to indicate the direction in *one* iteration space of nested DO-loops.

If a static data dependence holds between two indexed statement instances $S_1(i_1, ..., i_k)$ and $S_2(j_1, ..., j_k)$, both of degree k, then the r^{th} distance ϕ_r is defined as: $\phi_r = (j_r - i_r)$. The k-tuple $< \phi_1, ..., \phi_k >$ is the **dependence distance vector**. The true distance Φ_{ij} between the two statement instances is the total number of iterations between the execution of the two instances. It is easy to see that the following equation holds if the m^{th} index runs from 1 to N_m.

$$\Phi_{ij} = \sum_{r=1}^{k} \phi_r \prod_{m=r+1}^{k} N_m$$

The true distance is equal to the first distance for dependences between two statement instances in simple loops. Note that the distances between different instances that belong to the same static data dependence do not have to be constant. The dependence direction vector can be obtained from the dependence distance vector, by transforming every element of the tuple into the appropriate direction flag ($0 \rightarrow$ '=', positive numbers $\rightarrow$ '<' and negative numbers $\rightarrow$ '>'). Strictly spoken, the dependence direction vector belongs to the dependence between two instances of statements and not to the static dependence. So, again, a static dependence with a certain dependence direction vector can have more than one instance.

Now it will be discussed, why it is decided that the loop-control variable is not an element of the IN set of statement instances that are controlled by this variable. Consider the next example.

```
S1: I = <expression>
S2: A(I) = <expression>
L1: DO I = 1, 10, 1
  S3: B(I) = <expression>
  S4: C(I) = <expression>
ENDDO
S5: I = <expression>
```

The following static data dependences hold: $S_1 \delta S_2$ (caused by the fact that S_1 writes the variable `I` that S_2 uses in the subscript expression), $S_1 \delta^O L_1$, $S_2 \overline{\delta} L_1$ and $L_1 \delta^O S_5$ (note that $S_1 \delta^O S_5$ and $S_2 \overline{\delta} S_5$ do not hold because the variable I is written to in the meanwhile).

If the variable `I` was taken as part of the IN set of S_3 a lot of meaningless dependences would result. The static dependence $S_3 \delta^i_= S_4$ only reflects the fact

that the variable `I` is used by the two statement instances of the same iteration (and formally, input dependences across iterations do not hold, because the variable `I` is written to before each new iteration). Static dependences between L_1 and S_3 and S_4 are caused by `I`. $L_1\delta_{=}S_3$ and $L_1\delta_{=}S_4$ are caused by the fact that statement instances of one particular iteration uses values of the loop-control variable of that same iteration and $S_3\overline{\delta}_{<}L_1$ and $S_4\overline{\delta}_{<}L_1$ are caused by the fact that the loop-control variable is used in the iterations and written to before the next iteration. So, if `I` is part of the IN set, it seems that dependences across iterations exist, although the iterations of the loop in the example can be executed in any order. The last resulting static dependences are those between L_1 of different iterations. $L_1\delta^{O}_{<}L_1$ holds because the variable is set before every iteration (and if the loop-control variable setting is implemented as `I = I + stride`, followed by a bounds test, even the dependences $L_1\delta_{<}L_1$ and $L_1\overline{\delta}_{=}L_1$ hold). Note that another problem is caused by the fact that L_1 is at level 0, although dependences in which L_1 is involved hold at level 1, so a direction flag has to be added to the dependences.

Therefore, to get rid of all these problems, it is decided that loop-control variables are not an element of the IN set of statement instances inside its body. Note that (as stated in footnote 15), since a loop-control variable can never be written to, it will never be part of a OUT set of statement instances in its loop body as well. Intuitively this is a right decision, because every iteration can be treated as having its own local loop-control variable, with a particular value [19]. As long as every statement instance receives the right value of its loop-control variable and after all iterations the loop-control variable receives the same value as in the original execution of the DO-loop, the data dependences inside the loop body caused by the loop-control variable may be ignored. More formally, for every statement instance the value of I is substituted by its value, so in $IN(S(I)) = \{I,...\}$, the resulting IN set per statement instance contains a constant, which in general is removed from IN sets: $IN(S(i_1) = \{i_1,...\})$. The result of this decision is that the only global effect of a loop as a whole is the final assignment to its loop-control variable. If a programmer uses the fact that the loop-control variable has a certain value after the execution of a DO-loop, this is reflected by a flow dependence between the DO-loop and the statement using the variable. Care must be taken, that this flow dependence will never be violated by any transformation. Another result of this decision is that information about input dependences between statement instances of the same iteration is lost. However, since input dependences can be violated without changing the semantics of a program, and the only reason for computing input dependences is to get more insight in the reuse of data, this does not cause a real problem, since it is a well-known fact that the value of a loop-control variable is frequently used in every iteration.

The only exception on the elimination of loop-control variable from the IN sets of the statements in the body is made for for the determination of dependences between two DO-loops. Naturally, the resulting dependences have

[19] In fact, DOALL-loops are implemented that way, and vector instructions are under implicit control of a loop variable

a direction vector with as length the *common_nest* of the statements involved. This is done to enable the user to define that in certain transformations the value of a loop-control variable may not be used by DO-loops that are deeper in the nest (this is necessary for e.g. loop interchanging). Thus, in the following fragment the extra dependences $L_1 \delta L_2$ (no direction!) and $L_2 \delta^i_< L_2$ caused by `I` are concluded, resulting from the fact that `I` $\in IN(L_2)$ (naturally the dependence $L_2 \delta^O_< L_2$ also holds, because `J` $\in OUT(L_2)$).

```
L1: DO I = 1, 100
  L2: DO J = I, 100
    ...
  ENDDO
ENDDO
```

6 Data Dependence Analysis

Now that the concepts of data dependences have been introduced, the way in which the restructuring compiler determines the data dependences in a given program can be presented. First the data structure for storing data dependences is discussed.

6.1 Data Dependence Table

All static data dependences found in a program must be stored because during the transformation phase, these dependences are needed in the evaluation of the conditions of each transformation. Therefore the dependences are saved in a table, with the following information per entry.

- The source statement of the static dependence (S_i, C_i or L_i).
- The sink statement of the static dependence (S_j, C_j or L_j).
- The kind of data dependence (flow, anti, output or input).
- The symbol table entry of the variable which is in the intersection of the IN or OUT sets causing this dependence.
- The number of underlying dependences (if this number can be determined) or a '?' indicating that the exact number is unknown.
- The dependence direction vector of the static dependence.

Because the number of dependences in a program can be arbitrary large, and to make efficient use of memory, the table has been implemented using dynamic memory. The separate directions of the data direction vector are stored in a way, similar to the method presented in section 4.1, for storing the lexemes of identifiers.

This table is maintained by the functions defined in module *deptb.c*, which is listed in appendix L. The contents of the data dependence table is written in a readable format to the file *program.dep* after all static data dependences have been computed, together with some information about the number of static data dependences found. This table can be examined using the command **showdep**, after the program has been read, or the transformation phase is terminated. The user can select which static dependences must be generated in this file by using the command **dep**. This command selects in turn among the following possibilities of dependence generation.

- All static dependences;
- Only Flow, Anti and Output dependences
- Only static dependences between assignment statements
- Only Flow, Anti and Output dependences between assignment statements

As an example, consider the dependences of the following program fragment.

```
...
L1: DO I = 2, 100, 1
  L2: DO J = 1, 100, 1
    S1: A( <I> , <J> ) = <expression>
    S2: R = A( <I-1> , <J+1> )
  ENDDO
ENDDO
S3: R = <expression>
...
```

In this fragment the static dependences $L_2\delta^o_{<}L_2$, $S_1\delta_{<,>}S_2$, $S_2\delta^o_{=,<}S_2$, $S_2\delta^o_{<,*}S_2$, and $S_2\delta^o S_3$ hold, which are presented to the user as follows.

```
Dependence                    Variable  Number

L2 d-outp  <     L2              J       98
S1 d-flow  <>    S2              A       9702
S2 d-outp  =<    S2              R       9801
S2 d-outp  <*    S2              R       98
S2 d-outp        S3              R       1

Number of input  dependences : 0
Number of output dependences : 4
Number of flow   dependences : 1
Number of anti   dependences : 0

Total number of  dependences : 5
```

Note that the dependence $S_2\delta^o S_3$ does not have any direction flag, since it holds for every instance inside the loop body, to the (single) instance outside the loop body. Because the common nest is 0, no data direction vector is present. The static dependence $S_2\delta^o_{<,*}S_2$ holds, because the intermediate writes of later iterations are not taken into consideration, so the '$*$' is used, since it holds for the directions '$<$', '$=$', and '$>$'.

6.2 Data Dependence Computation

Since the restructuring compiler will use data dependences to decide which transformations can be applied without changing the semantics of a program, it is extremely important that the methods used for data dependence computation are conservative, i.e. if it is not certain if a particular data dependence will actually hold during run time, it must be assumed. However, if too many data dependences are assumed, potential parallelism may be lost. Therefore, it is very important to find the data dependences in a program as accurately as possible.

The routines performing data dependence analysis can be found in module *dep.c*, which is listed in appendix M. The implementation of this task is a bit overwhelming, because a lot of different tasks are performed concurrently

(traversing the program data structure, maintaining information about the environment, subscript expression analysis e.d.). Therefore, the algorithms used are discussed at an abstract level, ignoring a lot of straight-forward to implement details. The analysis techniques themselves, however, are discussed in great detail.

For the sake of simplicity the implementation of data dependence computations compares every statement of the program with all the other statements in the program, and determines if a data dependence may hold. The disadvantage of comparing each statement with all other statements is that dependences that formally do not hold, may be determined. But since the resulting dependences are a transitive closure of the indirect dependence relation, these dependences will not limit the transformations possible, but will only slow down the condition evaluation. One reason that these dependences are not filtered out is that it cannot always easily be determined if this filtering is valid, since this requires extensive subscript analysis inside DO-loop bodies. The second reason is that conditional statements may complicate this filtering process.

For instance, in the following fragment the approximation [20] tests used to see if the subscript expressions can be equal, may report that $f(I_1) = g(I_2)$, $g(I_3) = h(I_4)$ and $f(I_5) = h(I_6)$, for different (unknown) values I_j in the iteration space, resulting in the assumption of output dependences on the statement instances, but it is unclear without a more accurate analysis of the actual values, for which instances the dependence formally exists (i.e. the array element is not written to between the two instances).

```
L1: DO I = 1, 100, 1
  S1: A( f(I) ) = <expression>
  S2: A( g(I) ) = <expression>
  S3: A( h(I) ) = <expression>
ENDDO
```

For example, for $f(I) = g(I) = h(I) = I$, only the output dependences $S_1\delta^o_{=}S_2$, and $S_2\delta^o_{=}S_3$ hold, although $S_1\delta^o_{=}S_3$ will be generated by the method used in this prototype compiler. Because the three statements reference the same element of `A` in every iteration, it would be valid to filter this last dependence out. Note that S_1 and S_3 must still be compared in order to detect this, and to conclude that no cross iteration dependence between these statements holds.

If $f(I) = 2$, $g(I) = 4$, and $h(I) = I$, both static dependences $S_1\delta^o_{=}S_3$ and $S_2\delta^o_{=}S_3$ hold, each with only one instance belonging to *different* iterations. In this case the first dependence cannot be filtered out, because there is no intermediate write between the statement instances that cause this static dependence. Because the intersection of the outsets of S_1 and S_2 is always empty, it can be concluded that no dependence between these statements hold. Again all pairs of statements must be considered in order to be able to detect the cross iteration dependences $S_2\delta^oS_2$, $S_1\delta^oS_1$, $S_3\delta^oS_1$. The dependence $S_1\delta^oS_3$ would

[20] The test used in this compiler will report if subscript expressions can be equal, and if this is the case the direction in the iteration space is determined, but the actual values for which these subscripts expressions are equal are not computed.

also be generated (because $S_1(1)$ and $S_3(2)$ both use `A(2)`), although formally it does not hold, since `A(2)` is written to by $S_2(2)$.

So, because the filtering can only be done with extensive subscript analysis (it must be determined which elements are written to in every instance) and the filtering cannot help to reduce the number of statement pairs that must be considered (since also cross iteration dependences must be accounted for), it is decided to generate all dependences. Note that the first problem does not occur for dependences on scalar variables, but in order to keep the generation of data dependences consistent, it is decided to ignore intermediate writes to scalar variables too in the dependence computation, although some information will be used in the computation of the number of underlying instances.

If conditional statements are present, it cannot be predicted if certain dependences will really exist in all cases, as is demonstrated in the following example. Whether $S_1 \delta^o S_3$ holds, or $S_1 \delta^o S_2$ and $S_2 \delta^o S_3$ hold, depends on the value of the variable L1 of type LOGICAL.

```
S1: R = <expression>
C1: IF (L1) S2: R = <expression>
S3: R = <expression>
```

Therefore, some restructuring compilers compute also the so-called control dependence relation (δ^C), in order to be able to compute the data dependences more precisely. This dependence, however, will not be used in this prototype compiler, although it accounts for the flow of control in the different branches of one single IF statement. In the following fragment, no dependence can hold, since only one branch of the IF statement will be executed during run time.

```
C1: IF (L1) THEN
  S1: R = <expression>
C2: ELSE IF (L2) THEN
  S2: R = <expression>
ELSE
  S3: R = <expression>
ENDIF
```

Concluding it can be stated that the prototype compiler computes all data dependences in a program without inspection of the second condition in the definitions of dependences presented in sections 5.1.1 - 5.1.4, and without extensive control flow analysis.

The actual implementation is done using two pointers to statements. The first one shifts over all statements of a program indicating the statement currently under consideration, while the second one iterates over all the statements, which follow that current statement, before the first pointer is shifted. In this fashion, all statement pairs are considered. Note at this point that, although the statement pointed to by the first pointer, precedes the statement pointed to by the second pointer in the program, this does not mean that all instances of that statement precede the instances of the second statement at run-time. For all statement pairs considered in this way, the corresponding IN and OUT sets

are compared, and if non empty intersections are possible, the resulting static data dependences are assumed, the direction in the iteration space is determined and they are added in the data dependence table. The global algorithm is given below.

for every statement s_1 **in the program**
do
$OUT_1 = OUT$-set(s_1);
$IN_1 = IN$-set(s_1);
$compare\_self()$;
for every statement s_2 **after** s_1
do
$OUT_2 = OUT$-set(s_2);
$IN_2 = IN$-set(s_2);
$compare()$;
end do
end do

Remember that all variables used in subscript expressions are also elements of the IN set of the corresponding statement. The environment (i.e. nesting depth and loop-control variables) of both statements s_1 and s_2 must be maintained for testing purposes. Therefore, interleaved with the algorithm above, the environments of s_1 and s_2 are maintained. A variable *mynest* indicates the nesting depth of s_1, while *hisnest* holds the nesting depth of s_2. A variable $minnest \leq \min(mynest, hisnest)$ keeps track of the *common nest* depth. The loop-control variables of the environment of both s_1 and s_2 are also administrated, together with their bounds and stride, if these values can be evaluated and can be type converted to INTEGER values (if the original expressions are of type REAL, the evaluation is only successful, if they have an empty fractional part) at compile-time. If these values cannot be determined or are of type REAL with a non-empty fractional part, it is recorded that the values are unknown. Why this is done, is discussed in section 6.3.

The $compare\_self()$ function determines if an anti self dependence exists, or if cross-iteration self dependences of any kind hold, when $mynest > 0$. The function $compare()$ determines all the dependences possible between two different statements

In these two functions, the elements of the appropriate IN and OUT sets are compared with each other in a pair-wise fashion. Now two actions must be performed. First, it must be tested if two variables can be in the intersection, and therefore, are involved in a dependence. Then, if this is the case, the direction in the iteration space [21] in which the dependence holds must be determined, possibly resulting in the conclusion that a dependence cannot hold. Finally, if a dependence is still possible, the correct resulting static dependences must be determined. These three tasks are discussed in sections 6.3, 6.4, and 6.5.

[21] This is not exactly the same as the direction flag of the dependence direction vector of the resulting static dependence, although there is a close relation.

6.2.1 Optimizations

The following optimizations for the algorithm presented have been implemented. If the IN-set of an IF statements only consists of constants (e.g. `IF (3.GT.4) THEN`), the examination of all following statements can be skipped (since $IN = \emptyset$ and $OUT = \emptyset$). The same optimization can be implemented for the `ELSE IF` statements inside a general-IF statement.

6.3 Dependence Test between Two Variables

It can be easily determined if a dependence holds between two scalar variables. The two identifiers are compared and if these are equal, a dependence exists [22].

A dependence between two array variables possibly exists if the corresponding identifiers are equal (so the same array is referenced), **and** certain tests on the subscript expressions hold. These tests are applied separately on all corresponding subscript expressions that were transformed into their *normal form* of both variables as is illustrated by the following algorithm.

```
d1 = dim_list of first array variable
d2 = dim_list of second array variable;
assumed = true;
while (d1 ≠ NULL and assumed) do
    if (d1 -> normexpr ≠ NULL) and (d2 -> normexpr ≠ NULL)
then
        assumed = apply_tests();
    end if
    d1 = d1 -> tail;
    d2 = d2 -> tail;
end do
```

The occurrence of a dependence, can be found by examining the boolean variable *assumed* after this algorithm has been executed. The function *apply_tests*() also tries to determine the direction per loop-control variable in the iteration space in which the instances are executed. If conflicting directions for one loop-control variable are found to hold over different subscript expressions, it can directly be concluded that no dependence between the current array variables is possible (see update table at the end of section 6.4).

Now it is already clear why it is so important to convert subscript expressions into the *normal form*, because subscript expressions without a *normal form* do not participate in the test. So if `Z1` and `Z2` are variables, which are not used as a loop-control variable, a dependence between `A(Z1)` and `A(Z2)` is assumed, because both subscript expressions cannot be converted into *normal form*, reflecting the fact that the values of these variables cannot be determined at compile-time. If the inability to convert these subscript expressions into *normal form* is only the result of complicated subscript expressions [23], too many assumed dependences may result.

[22] Aliasing is not allowed.

[23] Take as an extreme example *A(Z1-Z1+I)*, with `I` as a loop-control variable in which the subscript expression cannot be converted into normal form

The tests applied subsequently in *apply_tests*() on the two subscript expressions in *normal form* are enumerated below. If one of these tests fails, it can directly be concluded that no dependence can exists without further evaluation of following tests. Since it must be determined if it is possible that the two subscript expressions have the same value, the problem can be rewritten into a single linear diophantine equation as shown below (note that since only one dimension is considered at the time, collapsing [24] is not applied).

$$a_0 + a_1 * i_1 + \ldots + a_k * i_k = b_0 + b_1 * j_1 + \ldots + b_l * j_l$$

$$\text{with k} = mynest, \text{ l} = hisnest$$

$$\forall_{1 \leq p \leq k} : L_p \leq i_p \leq U_p, L_p \in \mathbf{Z} \wedge U_p \in \mathbf{Z}$$

$$\forall_{1 \leq p \leq l} : L'_p \leq j_p \leq U'_p, L'_p \in \mathbf{Z} \wedge U'_p \in \mathbf{Z}$$

$$\forall_{0 \leq p \leq k} : a_p \in \mathbf{Z}$$

$$\forall_{0 \leq p \leq l} : b_p \in \mathbf{Z}$$

Note that, if some of the loop-control variables are in common ($minnest > 0$) then, $\forall_{1 \leq p \leq minnest} : L_p = L'_p \wedge U_p = U'_p$.

Constant Subscript Expression Test: If both subscript expressions are constant expressions, it can directly be determined if a dependence exists by testing if $b_0 - a_0 = 0$. So in the following example no data dependence is assumed between S_1 and S_2, since 2 - 1 $\neq$ 0.

```
S1: A(1) = <expression>
S2: A(2) = <expression>
```

G.C.D. Test: This test is based on the following well-known observation.

$$\exists\ (k_1,\ldots,k_n) \in \mathbf{Z}^n \text{ such that } c_1 * k_1 + \ldots c_n * k_n = c_0 \text{ with } \forall_{1 \leq p \leq n} : c_p \in \mathbf{Z}$$
$$\Longleftrightarrow$$
$$\text{GCD}(c_1,\ldots,c_n) \mid c_0.$$

Therefore, it is tested if all the loop-control variables used in the two subscripts expressions are of type INTEGER and if GCD($\mid a_1 \mid,\ldots,\mid a_k \mid, \mid b_1 \mid,\ldots,\mid b_l \mid$) $\mid$ ($\mid b_0 - a_0 \mid$). Remember that all a_p and b_p are of type INTEGER, since only subscript expressions in *normal form* are considered. In the following fragment, no data dependence between S_1 and S_2 is assumed, since GCD(2,2) $\not\mid$ ($\mid 1-2 \mid$).

```
L1: DO I = 1, 10, 1
  S1: A(2 + 2 * I) = <expression>
  S2: A(1 + 2 * I) = <expression>
ENDDO
```

[24] This technique is applied if all subscripts of all dimensions are examined at once.

If the stride and the lower bound are known at compile time, the loop can be normalized, often resulting in a better application of the GCD test as is illustrated below, in which the GCD test only can conclude that no dependence holds in the normalized fragment (GCD(2,2) $\not|$1).

```
DO I = 1, 5, 2                    DO I = 0, 4, 1
  A(I)       = <expression> >>>     A(1 + 2 * I) = <normalized expression>
  <variable> = A(I + 1)             <normalized variable> = A(2 + 2 * I)
ENDDO                             ENDDO
```

Therefore, if the lower bound and stride are known for the pth loop in the nesting of the first variable, $a_p * i_p$ is converted into $a_p * low_p + stride_p * i'_p$ in which i'_p has new bounds that are not important for the GCD test. The same conversion can be done for $b_p * j_p$. For all loops in the common nest for which $a_p = b_p, 1 \leq p \leq minnest$, only the value of the stride must be known at compile-time, since the offsets on both sides of the equation will be equal ($a_p * low_p = b_p * low_p$).

Bounds Test: The following observation can be used to test if real solutions for the equation exist.

$$\exists\ (c_1,\ldots,c_n) \in \mathbf{R}^n \text{ such that } c_1 * k_1 + \ldots c_n * k_n = c_0 \text{ with } \forall_{1\leq p\leq n} : k_p \in [L''_p, U''_p]$$

$$\Leftrightarrow$$

$$c_0 \in [\sum_{p=1}^{n} c_p^+ L''_p - c_p^- U''_p, \sum_{p=1}^{n} c_p^+ U''_p - c_p^- L''_p]$$

$$c_i^+ = \begin{cases} c_i & \text{if } c_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$c_i^- = \begin{cases} -c_i & \text{if } c_i < 0 \\ 0 & \text{otherwise} \end{cases}$$

Therefore, it is tested if $b_0 - a_0 \in$ [LOW, HIGH], in which LOW and HIGH are the minimum and maximum value of the expression $a_1 * i_1 + \ldots a_k * i_k - b_1 * j_1 - \ldots - b_l * j_l$. This test can be used on loop-control variables of any type, but it is very important that the coefficients and the bounds are of type INTEGER to prevent the inability to detect data dependences caused by run-time round off errors in the subscript expressions. Consider the following fragment in which the bounds of a DO-loop are of type REAL.

```
INTEGER I
...
L1: DO I = 1.5, 10.0, 1.0
  S1: A(I) = <expression>
  S2: A(1) = <expression>
ENDDO
```

If the bounds test was also applied on REAL typed bounds, the conclusion would be that there is no output dependence $S_1 \delta^o_{=} S_2$, since $1 \notin [1.5,10.0]$. However, since at run time the value 1.5 is rounded to 1 in the subscript evaluation, a dependence does hold. Therefore, the values of bounds with non-empty fractional part are considered to be unknown. So, using REAL typed values in DO-loops may result in the assumption of too many data dependences, since the bounds test will not be evaluated. Naturally, the bounds of the loop-control variables corresponding to a zero coefficient may be of any type or unknown value, since it is not used in the bounds test.

The Constant Subscript Expression Test is a special case of the bounds test, but since it can also be performed in loops with bounds that are unknown at compile-time, it is included as extra test. The following example demonstrates the use of the bounds test between different loops. No data dependence is assumed since $0 \notin$ [-19,-1].

```
L1: DO I = 1, 10, 1
  S1: A(I) = <expression>
ENDDO
L2: DO I = 11, 20, 1
  S1: A(I) = <expression>
ENDDO
```

Because the upper bound of a DO-loop is not necessarily greater than the lower bound (e.g. `DO I = 10, 1, -1`), and since the bounds tests needs the valid interval for every loop-control variable $\forall_{1 \leq p \leq n} : k_p \in [L''_p, U''_p]$, the maximum of the bounds is assigned to U''_p and the minimum to L''_p in the implementation of this test.

6.4 Direction Determination

If after these three tests, a dependence still may exist, a function is called that tries to determine the order in which the instances of statements involved in the possible dependence reference the same elements of the current array. This is done, because the tests only state if a dependence is possible, and do not use the direction in the iteration space in the constraints, a technique described in [Ban88] (see section 6.9 for a comparison between that technique and the method used in this prototype compiler).

The computation of the direction in the prototype compiler is done by recording the direction in the iteration space for every loop-control variable in the common nest. This direction can only be determined for two subscript expressions that are linear functions of the same loop-control variable in the common nest. Since references to array variables with more than 1 dimension the subscript expressions are considered in sequence, this may result in an intermediate iteration direction vector, because usually the subscript expressions of corresponding dimensions are linear functions of the same loop-control variable. From this intermediate iteration direction vector, that only indicates the order in which the array is referenced by the different instances, the resulting static dependence and its dependence direction vector can be determined.

How this direction is determined is illustrated using the following example, in which the two subscript expressions shown belong to the same dimensions, and `c1`, `s1`, `c2`, and `s2` are constants.

```
        ...
          DO 10 I = LOW, HIGH, STRIDE
            ...
              S1: ... A(...,c1 + s1 * I,...) ...  <- f1
              S2: ... A(...,c2 + s2 * I,...) ...  <- f2
            ...
 10       CONTINUE
        ...
```

All of the following conclusions can be drawn if `STRIDE`>0. If `s1` = `s2` > 0 (the resulting straight curves of these functions are parallel, and have positive slope) the direction flag for `I` in a dependence between S_1 and S_2 is '=' if `c1` = `c2`, '>' if `c1` < `c2` and '<' otherwise, as is illustrated in the following picture, for `c2` > `c1`. If the slopes are negative, the opposite [25] directions must be taken instead.

The '>' direction reflects the fact that the reference to A of a particular element in an instance of S_1 *follows* on the reference to the same element of A done by an instance of S_2, in the iteration space of the loop-control variable under consideration.

If the resulting straight curves are not parallel, the intersection point is determined (the value for which the outcome of the two linear functions are equal) and the interval in which the values of the loop-control variable are. This interval can be one of the following types.

- *left*, which means that all the values are less than the intersection point.
- *right*, which means that all the values are greater than the intersection point.
- *leftoverlap*, which means that all the values are smaller than or equal to the intersection point.
- *rightoverlap*, which means that all the values are less than or equal to the intersection point.
- *equal*, which means that the loop is only executed once, and the value of the loop-control variable is equal to the intersection point. This case is very rare.
- *overlap*, which means that the values appear 'at both sides' of the intersection point.

The possible intervals are presented in the following picture for the case that `s1` > `s2` > 0.

This interval can easily be determined with knowledge of the values of the stride and bounds in the DO-loop, and the intersection point (IP), using the

[25] See table at the end of this section.

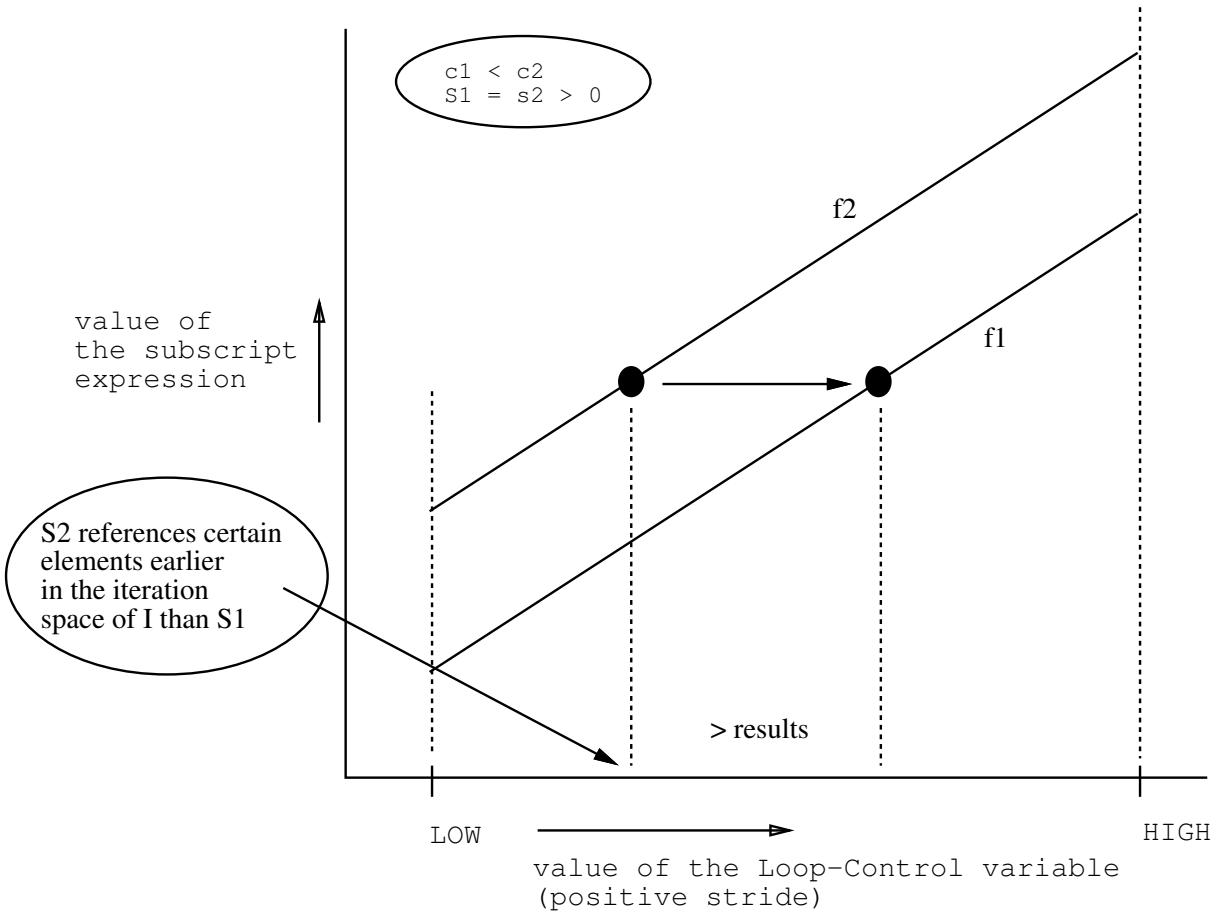


Figure 11: Dependence Direction

following algorithm (STRIDE≠0). Again only bounds of type INTEGER are considered to be known at compile-time, since round-off errors may cause unexpected values during run-time.

if (values of LOW and HIGH are known and both equal to IP) **then**
  return *equal*;
else if (value of STRIDE is known) **then**
  if (STRIDE > 0) **then**
    if (value of LOW is known **and** LOW ≥ IP) **then**
      return (LOW > IP) ? *right* : *rightoverlap*;
    else if (value of HIGH is known **and** HIGH ≤ IP) **then**
      return (HIGH < IP) ? *left* : *leftoverlap*;
    else
      return *overlap*;
    end if
  else if (STRIDE < 0) **then**
    if (value of LOW is known **and** LOW ≤ IP) **then**
      return (LOW < IP) ? *left* : *leftoverlap*;
    else if (value of HIGH is known **and** HIGH ≥ IP) **then**

```
            return (HIGH > IP) ? right : rightoverlap;
        else
            return overlap;
        end if
    end if
else if (values of LOW and HIGH are known) then
    if (LOW ≤ IP) and (HIGH ≤ IP) then
        return (LOW < IP and HIGH < IP) ? left : leftoverlap;
    else if (LOW ≥ IP) and (HIGH ≥ IP) then
        return (LOW < IP and HIGH < IP) ? right : rightoverlap;
    else
        return overlap;
    end if
else
    return overlap;
end if
```

In the interval *right*, if both slopes are positive, '<' can be concluded if 0 < `s2` < `s1` and '>' otherwise. If both slopes are negative '<' results if `s1` < `s2` < 0, or '>' otherwise, as is illustrated in the following picture:

If the slopes have opposite signs, it can be concluded that no dependence holds, since the functions are divergent. This can be seen in the following picture:

Analog conclusions can be drawn in the interval *left*. In the interval *leftoverlap* and *rightoverlap* the directions ≤ and ≥ result, instead of < and > respectively. In the interval *equal*, only = results. Finally, in the interval *overlap* no conclusion about the direction can be drawn.

If `STRIDE` < 0, the opposite direction holds, since the iteration space is traversed in reversed order. If the value of `STRIDE` cannot be determined at compile time, the direction and its opposite must be assumed. These two computations are shown in the following table (a '$*$' indicates the direction '=', '<' or '>' and a '+' indicates the direction '<' or '>'). Since the '≤' and '≥' are not available as single character, they are represented as '[' and ']' respectively and printed as such in the resulting static dependence list.

	$*$	$+$	$<$	$>$	$=$	$\leq$	$\geq$
Opposite direction	$*$	$+$	$>$	$<$	$=$	$\geq$	$\leq$
Both directions	$*$	$+$	$+$	$+$	$=$	$*$	$*$

Once the direction in the iteration space has been determined, it is used to update a temporary data structure, which contains the computed direction so far, for all loop-control variables in the *common nest*. Initially [26] this data structure contains only '$*$'s, indicating the fact that no directions are known, and it is updated as every dimension of the two array variables is considered. If a direction has already been recorded, which is in conflict with the new direction, it can be concluded that no dependence holds. The following table shows the resulting direction as function of the previously determined direction.

[26] i.e. before all subscript expressions of every dimension are compared in a pairwise fashion.

Update	$*$	$+$	$<$	$>$	$=$	$\leq$	$\geq$
$*$	$*$	$+$	$<$	$>$	$=$	$\leq$	$\geq$
$+$	$+$	$+$	$<$	$>$	**conflict**	$<$	$>$
$<$	$<$	$<$	$<$	**conflict**	**conflict**	$<$	**conflict**
$>$	$>$	$>$	**conflict**	$>$	**conflict**	**conflict**	$>$
$=$	$=$	**conflict**	**conflict**	**conflict**	$=$	$=$	$=$
$\leq$	$\leq$	$<$	$<$	**conflict**	$=$	$\leq$	$=$
$\geq$	$\geq$	$>$	**conflict**	$>$	$=$	$=$	$\geq$

So if, for example in the following fragment, the direction in the iteration space of `I` for dependences between elements of the array variable A in instances of S_1 and S_2 respectively, is determined, a conflict is detected.

```
DO I = 1, N, 1
  S1: A(I,I) = ...
  S2: ...    = A(I+1,I-1)
ENDDO
```

Since only the loop-control variable `I` is in the *common nest*, the initial data structure is $\boxed{*}$. After `I` has been compared with `I+1`, the direction $>$ is stored, since instances of S_2 reference array elements before S_1 in that dimension, resulting in the data structure $\boxed{>}$. Since the comparison of `I` and `I-1` results in the update with the $<$-direction, it is concluded that no dependence holds.

The process for two loop-variables and an array of dimension two is illustrated below.

```
DO I = 1, N, 1
  DO J = 1, N, 1
    S1: A(I,J) = A(I+1,J-1)
  ENDDO
ENDDO
```

Initially the data structure is $\boxed{*,*}$. After the first dimension has been considered it is updated to $\boxed{>,*}$, and finally the result is $\boxed{>,<}$.

6.5 Resulting Dependences

Once it is determined that two variables are possibly involved in a dependence, and the intermediate iteration direction vector has been computed, all resulting static dependences and their dependence direction vectors must be gathered.

The following algorithm determines all the resulting static dependences between scalar variables (note that array references with constant subscript expressions are considered as scalar variables [27]). The function add_dependence adds a static dependence between the statements given in the first two parameters to the dependence table. The type of this dependence is given in the third

[27] The same algorithm is used to generate all the dependences between statements that possibly reference the same elements of an array, but have no loops in common (i.e. $minnest = 0$). In this case only (1) is executed.

parameter and its dependence direction vector in the fourth parameter.

case (set-kind of v_1,set-kind of v_2)
(In,In) : dep1 = INPUT; dep2 = INPUT;
(In,Out) : dep1 = ANTI; dep2 = FLOW;
(Out,In) : dep1 = FLOW; dep2 = ANTI;
(Out,Out) : dep1 = OUTPUT; dep2 = OUTPUT;
end case
if ((sets do not belong to same statement)
or ((In,Out) is considered))
and (dependence possible in flow of control) **then**
add_dependence(s1,s2,dep1,($\underbrace{=,...,=}_{minnest}$)); (1)
end if
for i = 0, minnest - 1 **do**
add_dependence(s2,s1,dep2, ($\underbrace{=,...,=}_{minnest-1-i}$, <, $\underbrace{*,...,*}_{i}$)) (2)
end do

The static dependence added in (1) holds between instances of the same iteration of every (possibly zero) surrounding DO-loop. Therefore, it is explicitly tested if this dependence is possible according to the flow of control, as demonstrated in the last example of section 6.2. The test of (1) is necessary to preserve the strictness demands on the execution order in the definitions, discussed in sections 5.1.1 - 5.1.4. If (In,Out) has been chosen in the **case** statement, an anti self dependence results. The dependences added in (2) are all the dependences that hold between statement instances of different iterations. For example, in the following fragment the data dependences $S_1\delta^o_{=,=,=,<}S_1$ indicating the dependences that hold between instances of different iterations of the `L` loop, $S_1\delta^o_{=,=,<,*}S_1$ for iterations over the `K` loop (and later iterations of the `L` loop), $S_1\delta^o_{=,<,*,*}S_1$, and $S_1\delta^o_{<,*,*,*}S_1$ hold. It is important to realize that, in the case of a flow, anti or output dependence, in fact a '$\geq$' could have been concluded, since the writes done in every iteration mask the reads and writes of later iterations, so the cross iteration dependence formally only holds over one iteration of the innermost loop. Since intermediate writes of other statement instances (or even of instances of the same statement caused by variables not under consideration yet) are ignored in other cases, it is decided here to consequently generate the '$*$' direction. In the computation of the number of underlying dependences, however, these writes will not be ignored. For input dependences the '$*$' direction is correct, since no write (caused by instances of the two variables under consideration) will interfere.

```
DO I = 1, 10, 1
  DO J = 1, 10, 1
    DO K = 1, 10, 1
      DO L = 1, 10, 1
        S1: A(10) = <expression>
      ENDDO
    ENDDO
```

```
  ENDDO
ENDDO
```

The computation of static dependences between array variables, in which loop-control variables appear inside the subscript expressions, require the use of the intermediate iteration direction vector. This computation is presented in the following algorithm.

i = position of the first non-'=' in the iteration direction vector;
if i cannot be found **then**
  /* iteration direction vector = (=,...,=) */
  if (dependence possible in flow of control) **then**
    process('='); (1)
  end if
else
  dir = direction at position i;

$$/ * (\underbrace{=, ..., =}_{start}, dir, \underbrace{...}_{rest}) * /$$

  case *dir*
    '<' : *process*('<');
    '>' : *process*('>');
    '+' : *process*('<'); *process*('>');
    '≤' : *process*('<');
          set *dir* to '=' and apply algorithm again
    '≥' : *process*('>');
          set *dir* to '=' and apply algorithm again
    '∗' : *process*('<'); *process*('>')
          set *dir* to '=' and apply algorithm again
  end case
end if

Case (1) handles dependences between instances of the same iteration, so again the flow of control test must be performed. The function *process*() determines the actual dependence vector and the kind of the resulting dependence. This is done by the next algorithm, in which *dir* is the parameter passed to this function.

if *dir* = '>' **then**
  invert intermediate iteration direction vector (1)
end if
store the intermediate iteration direction vector
in the dependence direction vector
dep = table_lookup(*dir*, (set kind of s_1 , set kind of s_2)); (2)
if (sets do not belong to the same statement
  or self dependence is allowed) (3)
  if *dir* = '>' **then**
    add_dependence(s_2,s_1,dep,dependence direction vector)
  else

add_dependence(s_1,s_2,dep,dependence direction vector)
end if
end if

The table used in (2) is given below. This table also contains the information if a self dependence is allowed, which is used in the condition (3). For (In,In) and (Out,Out) self dependences are not allowed if *dir* is '=', since the resulting dependence can never hold between the same instances of one iteration. The reason that *self* dependences are never allowed for (Out,In) is that these dependences are also computed for the (In,Out) case, and it is not necessary to conclude the same self dependence twice. Self dependences are also not allowed for the '>' direction in the (Out,Out) case, since they are always the reversed versions of the one handled by the '<' direction, because the OUT set is always a singleton set. This is unfortunately not the case for (In,In) sets, since the IN set can contain more elements and all different pairs must be compared because the directions that can be determined may differ. Therefore, the input dependences belonging to the same pair of variables are generated twice.

dep	(In,In)	(In,Out)	(Out,In)	(Out,Out)
$<$	INPUT, allowed	ANTI, allowed	FLOW	OUTPUT, allowed
$>$	INPUT, allowed	FLOW, allowed	ANTI	OUTPUT
$=$	INPUT	ANTI, allowed	FLOW	OUTPUT

Because the intermediate iteration direction vector only reflects the order in which the instance references to the same array elements appear in the execution, it must be turned around (e.g. $\boxed{>,<,+,=,*,\leq}$ becomes $\boxed{<,>,+,=,*,\geq}$), to obtain the right dependence direction vector. This is illustrated with the following loop.

```
          DO 10 I = 1, 5
            DO 5 J = 1, 5
              S1: A(I,J)     = <expression>
              S2: <variable> =  A(I,J + 1)
  5         CONTINUE
 10       CONTINUE
```

The resulting iteration direction vector in this loop is $\boxed{=,>}$, since S_1 writes the elements of `A` after S_2 has read them. The resulting dependence, however, is $S_2\overline{\delta}_{=,<}S_1$, since dependences always reflect the flow of data in the execution order.

The '$*$', '$+$', '$\leq$' and $\geq$ after the first '$<$' or '$>$' (which has been converted into a '$<$'), are not further expanded, as can be seen in the algorithm, since the first direction ($\neq$ '=') determines the order of execution in which the dependence holds and therefore its kind. Expanding these directions would only introduce more dependences of the same kind, as in the following case. If it is determined for example, that in a particular dependence the direction vector in the iteration space is $\boxed{<,*}$ between an IN-set of S_1 and an OUT-set of S_2, a static anti dependence results, since the direction in the iteration space is '$<$', so statement instances of S_1 read elements before instances S_2 overwrite them. Expanding the

second direction would only result in the list $S_1\overline{\delta}_{<,<}S_2$, $S_1\overline{\delta}_{<,>}S_2$, and $S_1\overline{\delta}_{<,=}S_2$, which contains no more information than the single representation $S_1\overline{\delta}_{<,*}S_2$. If the constraints on the loop-control variables would have been used, as is suggested in section 6.9, expanding could be useful, since some of the directions might not be possible.

6.6 Improved Data Dependence Computation

In some numerical programs, more than one loop-control variable is used per subscript expressions. This is the case, for example, if a loop-control variable is used as an offset in subscript expressions with other-loop control variables, as is illustrated in the next fragment.

```
          DO 10 BASE = 0, MAXBASE, BASESTEP
            DO 5 I = 1, K
              X(BASE + I) = <expression>
              <variable>  = X(BASE + I + 1)
 5          CONTINUE
10        CONTINUE
```

The algorithm discussed in section 6.3 is not able to determine the direction of `I` in the iteration space during a single execution of the first DO-loop, so without any precautions, too many dependences between the first and the second statement in the loop-body are assumed: $S_1\delta_{=,<}S_2$, $S_2\overline{\delta}_{=,<}S_1$, and $S_1\delta_{=,=}S_2$, while in fact only $S_2\overline{\delta}_{=,<}S_1$ holds. Another problem is that the algorithm is also not able to determine the direction in the iteration space of the variable `I`, in dependences between statement instances belonging to different iterations of `BASE`. So, $S_1\delta_{<,*}S_2$ and $S_2\overline{\delta}_{<,*}S_1$ result, while in fact only $S_1\delta_{<,>}S_2$ and $S_2\overline{\delta}_{<,\geq}S_1$ hold.

The following two sections discuss methods to overcome both problems. The subscript expressions of the two variables under consideration are again presented to the algorithm in terms of their surrounding loop-control variables, as is illustrated below for a one dimensional array `A`.

$$...A(a_0 + a_1 * i_1 + ... + a_k * i_k)...$$

$$...$$

$$...A(b_0 + b_1 * j_1 + ... + b_l * j_l)...$$

$$k = mynest \wedge l = hisnest$$

Since it is very important to have precise information about the directions of dependences after a prefix of '='-directions, as will be discussed in section 8.2, the first problem is considered to be more important than the second one. This is reflected in the solutions given.

6.6.1 Improved for Prefix of '='-directions

For every array variable which uses more than one loop-control variable in the subscript expression of a particular dimension, an extra function is called to determine the dependences that hold whenever a prefix of r loop-control variables in the *common nest* is kept invariant. This can be done for $1 \leq r \leq minnest$. By using the fact that $\forall_{1 \leq p \leq r} : i_p = j_p$, the equation resulting from the general form given above, can be rewritten into the following form.

$$a_0 + (a_1 - b_1) * i_1 + ... + (a_r - b_r) * i_r + a_{r+1} * i_{r+1} + ... + a_k * i_k$$

$$=$$

$$b_0 + b_{r+1} * j_{r+1} + ... + b_l * j_l$$

$$1 \leq r \leq minnest$$

$$k = mynest \wedge l = hisnest$$

The three tests and direction determination, mentioned in section 6.3 are applied to this resulting equation, for all possible r.

The results for a certain value of r, obtained by these tests and the directions that can be determined are used in the generation of the static dependences that hold in the same iteration of the first r loop-control variables, i.e. when the data direction vector is of the following form.

$$(\underbrace{=, ..., =}_{r}, \underbrace{dir_{r+1}, ..., dir_{minnest}}_{minnest-r})$$

So the algorithm of section 6.5 must be adapted to use the new directions if a dependence with a prefix of '='-directions is generated.

Since sometimes, less loop-control variables are considered (which is the case if $a_p = b_p$ for certain $1 \leq p \leq minnest$), the probability that the direction can be determined [28] increases, and since coefficients may change, the tests can draw the conclusion that dependences cannot hold if the first r loop-control variable are kept invariant, in contrast with the earlier conclusion of these tests.

If $a_p = b_p$, the p^{th} coefficient becomes 0 reflecting the fact that both subscript expressions have the same offset during one iteration of the p^{th} loop-control variable, which can be seen in the following figure for $p = 2$. Since the tests consider $b_0 - a_0$, and during one iteration the offset can be seen as belonging to both constants, these offsets can be ignored,

$$a_0 + \underbrace{a_1 * i_1 + a_2 * i_2}_{offset} + a_3 * i_3 + ... + a_k * i_k$$

and

[28] This is because the direction can only be determined for two linear functions.

$$b_0 + \underbrace{b_1 * j_1 + b_2 * j_2}_{offset} + b_3 * j_3 + ... + b_l * j_l$$

$$k = mynest \wedge l = hisnest$$

$$because\ a_1 = b_1 \wedge i_1 = j_1 \wedge a_2 = b_2 \wedge i_2 = j_2$$

This technique has been implemented by, in the case that $minnest \geq 1$, initializing $minnest$ data structures as follows. For $r = 1$ $\boxed{=,*,...,*}$ (representing the direction vector if i_1 is kept invariant), for $r = 2$ $\boxed{=,=,*,...,*}$ and so on until $r = minnest$ with as data structure $\boxed{=,...,=}$ (representing the direction vector if i_1,...,$i_{minnest}$ are kept invariant). After one dimension has been considered, the directions in the iteration space that already have been found are combined with the first data structure, by updating this data structure with these directions with the rules found in the update table of the previous section. Then a_1 is set to a_1 - b_1, and b_1 to 0, after which the tests are applied as stated above for $r = 1$, with all the updates done in the data structure belonging to $r = 1$. After that, the data structure belonging to $r = 2$ is combined with the data structure of $r = 1$, and the tests are applied for $r = 2$. This process repeats until either $r = minnest$ [29], or it is detected that a dependence is not possible when the first r loop-control variables are kept invariant. In that case this is recorded, and used to prevent the generation of all corresponding dependences. After one dimension has been processed, the next dimension is considered, starting again with the data structure of the original algorithm, and using the data structures found for all r.

An optimization can be added, since the three tests and direction determination need not be executed if $b_p = 0$, as the coefficient of the p^{th} loop-control variable does not change in that case compared to the coefficients used in the tests that are executed earlier.

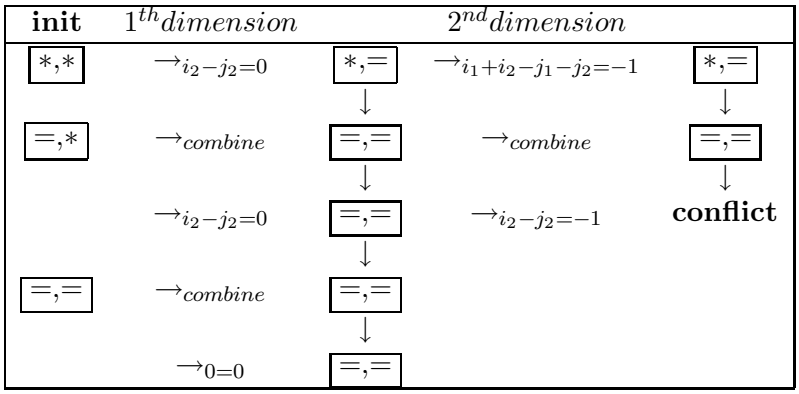
The use of this technique eliminates the incorrect assumption of the data dependences of the `BASE` example, since the equation becomes $a_2 * i_2 - b_2 * j_2 = b_0 - a_0$ if `BASE` (i.e. i_1 and j_1) is kept invariant, and the '>' direction can be determined, resulting in the elimination of $S_1\delta_{=,<}S_2$ and $S_1\delta_{=,=}S_2$.

To illustrate the implementation, the consecutive data structures are shown, for the program fragment given below, when array `A` in S_1 and S_2 is considered.

```
L1: DO I = 1, 100, 1
  L2: DO J = 1, 100, 1
    S1: A( <J> , <I+J+1> ) = <expression>
    S2: <variable>         = A( <J> , <I+J> )
  ENDDO
ENDDO
```

[29] The direction determination will never add new information in this case. The application of the three tests, however, is useful, since the last coefficient also changes.

The first row corresponds to the direction of the original algorithm, while the following rows correspond to $r = 1$ and $r = 2$, i.e. keeping `I` and `J` invariant, respectively. The table reflects the execution in 'column wise' order. The conflict is detected because a '$<$' direction must be placed in the data structure at the second position.



Only the static dependences $S_1\delta_{<,=}S_2$ and $S_2\overline{\delta}_{<,=}S_1$ result, since $S_1\delta_{=,=}S_2$ cannot hold according to the conflict. The method of the next section will also eliminate the assumption of the non-existing static dependence $S_2\overline{\delta}_{<,=}S_1$.

In the following fragment, it can be detected that no dependences are possible between S_1 and S_2, if the value of loop-control variable `I` is kept invariant, because the GCD-test fails as a result of the elimination of one coefficient, as shown below.

```
L1: DO I = 1, 100, 1
  L2: DO J = 1, 100, 1
    L3: DO K = 1, 100, 1
      S1: A( <I+2*J+2*K> ) = <expression>
      S2: <variable>       = A( <I+2*J+2*K+1> )
    ENDDO
  ENDDO
ENDDO
```

Since $a_1 = b_1$, the variable i_1 and j_1 disappear.

$$i_1 + 2 * i_2 + 2 * i_3 = j_1 + 2 * j_2 + 2 * k_2 + 1$$

$$\rightarrow_{i_1=j_1}$$

$$2 * i_2 + 2 * i_3 = 2 * j_2 + 2 * k_2 + 1$$

So the only dependences between S_1 and S_2 that result are $S_1\delta_{<,*,*}S_2$ and $S_2\delta_{<,*,*}S_1$. Without the improved technique, the dependences $S_1\delta_{=,=,=}S_2$, $S_1\delta_{=,=,<}S_2$, $S_2\overline{\delta}_{=,=,<}S_1$, $S_1\delta_{=,<,*}S_2$, and $S_2\overline{\delta}_{=,<,*}S_2$ would also have been assumed.

An example in which the altering (to a non-zero value) of one coefficient enables the GCD test to conclude that no static output dependence between S_1 and S_2 is possible is given below.

```
L1: DO I = 1, 100, 1
  L2: DO J = 1, 100, 1
     S1: A( <I+2*J-3> )     = <expression>
     S2: A( <-1*I+4*J+8> ) = <expression>
  ENDDO
ENDDO
```

If I is kept invariant, the equation can be rewritten as follows.

$$i_1 + 2 * i_2 + j_1 - 4 * j_2 = 11$$
$$\rightarrow_{i_1 = j_1}$$
$$2 * i_1 + 2 * i_2 - 4 * j_2 = 11$$

And the last equation has no integer solution, since GCD(2,2,4) = 2 does not divide 11.

6.6.2 Improved for Prefix of one '<' or '>' direction

In some cases, more directions in the iteration space can be determined in the case that the first direction is known to be '<' or '>'. This is the case when two subscripts are of the following form ($a_1 = b_1$, $a_1 \neq 0$, $a_r = b_r$, $a_r \neq 0$ and $\forall_{p \neq 0 \wedge p \neq 1 \wedge p \neq r} a_p = 0$ and $|stride_1| \geq 1$).

$$a_0 + a_1 * i_1 + a_r * i_r = b_0 + a_1 * j_1 + a_r * j_r$$

$$with\ 2 \leq r \leq minnest$$

The conclusions for $a_0 > 0$ and $a_r > 0$ are illustrated below for a '<' direction for the first loop-control variable, i.e. $j_1 - i_1 \geq 1$ holds in case of a positive stride ($stride_1 \geq 1$), since the minimal distance in the iteration space is 1.

$$a_r(i_r - j_r) = \underbrace{a_1 * (j_1 - i_1)}_{\geq a_1} + b_0 - a_0$$

So $i_r > j_r$ can be concluded for $b_0 - a_0 > a_1$ and $i_r \geq j_r$ for $b_0 - a_0 = a_1$. For a starting '>' direction and in case of a positive stride, i.e. $i_1 - j_1 \geq 1$ holds, $i_r < j_r$ for $b_0 - a_0 < a_1$ and $i_r \leq j_r$ for $b_0 - a_0 = a_1$ results. For all other cases (positive/negative signs of a_0 and a_r) analog conclusions can be drawn. In case of a negative stride for the 1^{st} loop-control variable ($stride_1 \leq -1$) the '<' and '>' case must be switched. Note that the conclusions (e.g. $i_r > j_r$) is defined in terms of the 'normal' relation on numbers. So, in case of a positive stride for the p^{th} loop-control variable ($stride_p > 0$), the same direction in the iteration space can be concluded, while the opposite direction must be taken in case of a negative stride ($stride_p < 0$).

Therefore, two data structures are maintained: one for a beginning '<' direction and one for a '>' direction, which are initialized to $\boxed{<,*,...,*}$ and $\boxed{>,*,...,*}$ respectively, before two array variables are considered. If $minnest > 1$, this structure is updated with the directions found so far, and a function is

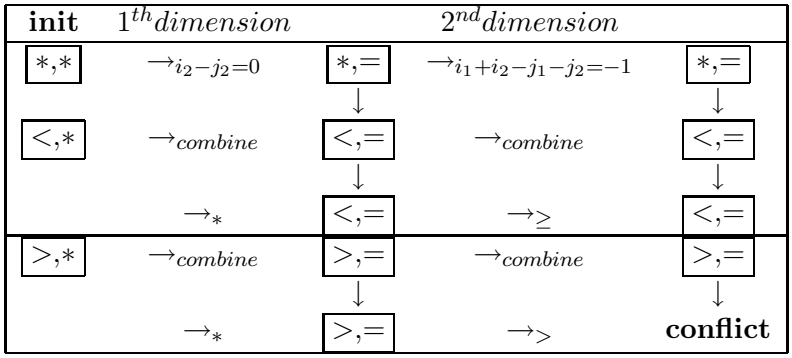
called which determines if more conclusions can be drawn using the method of this section. If that is the case, these directions are updated and recorded in the corresponding data structure. This updating and the calling of the function is done for every dimension, using the same data structure. If conflicting directions are detected, this is recorded to prevent the corresponding dependences from being generated. The algorithm of section 6.5 must be extended to use these directions while generating data dependences with a beginning '<' [30] direction.

Note that another improvement would be to maintain data structures for every possible combination of '<' and '>' directions at all places, but this would only work in limited cases, unless a more extensive analysis of the subscript expressions is implemented. The ad hoc solution of this section has been added because it works in case of simple subscript expressions, and it does not require much computational effort. This technique solves the problem with the `BASE` loop-control variable introduced at the beginning of this section, since it is able to determine the second direction of the flow and anti dependence: $S_1\delta_{<,>}S_2$ and $S_2\overline{\delta}_{<,\geq}S_1$.

To illustrate the implementation, the data structures of the original program and for the '<' and '>' directions are shown for the following fragment (that was also considered in the previous section), in which static dependences caused by references to array `A` between S_1 and S_2 are determined.

```
L1: DO I = 1, 100, 1
   L2: DO J = 1, 100, 1
      S1: A( <J> , <I+J+1> ) = <expression>
      S2: <variable>         = A( <J> , <I+J> )
   ENDDO
 ENDDO
```

In the data structure for a starting '<' direction, examination of the first subscripts does not add any additional information ('$*$' is updated), while examination of the second subscripts result in the updating with '$\geq$'. The dependence $S_1\delta_{<,=}S_2$ results.



The conflict is detected in case of a starting '>' direction, because a '>' direction must be placed at the second position in the data structure, which already contains an '=' direction. Therefore, the dependence $S_2\overline{\delta}_{<,=}S_1$ is not assumed.

[30] Remember that the first '>' direction in the iteration space is reversed in the resulting dependence.

6.7 Computation of Number of Underlying Dependences

In some cases it can be determined how many dependences between statement instances are responsible for one static dependence. One must keep in mind, that this number is computed without checking intermediate writes of any *other* statement, without writes to (the same) variables not under consideration when two variables are handled (but possibly in the statements under consideration), and without the use of extensive knowledge of the flow of control in conditional statements. Intermediate writes to the current two variables of the instances of the statements involved in the static dependence, however, will be taken into consideration in the computation.

6.7.1 Scalar Like Variables

The number of underlying dependences for a static dependence caused by the use of scalar variables or array variables with constant subscript expressions, can easily be computed for two statements inside the same nest ($mynest = minnest$ **and** $hisnest = minnest$).

```
L1: DO I = 1, N, 1
  S1: R = <expression>
  S2: <variable> = R
ENDDO
```

The flow dependence $S_1 \delta S_2$ has N different instances, and the cross iteration anti dependence $S_2 \delta_< S_2$ has N-1 different instances, ignoring all variable occurrences in <expression> and <variable>, but accounting for the intermediate writes in every iteration caused by the considered variables. In general, the number Num of underlying instances of statements inside the same nest can be determined for the case that the direction vector is $(\underbrace{=, ..., =}_{minnest})$, in which case

$$Num = \prod_{i=1}^{minnest} N_i$$

where N_i is the number of iterations of the ith loop, or for direction vectors

$$(\underbrace{=, ... =}_{r}, <, \underbrace{*, ..., *}_{minnest-1-r}),$$

with

$$Num = (\prod_{i=1}^{r} N_i) * (N_{r+1} - 1), \quad dep \neq INPUT$$

because the number of times an iteration is crossed, is one less than the number of iterations. In the last case the determined number cannot be used for input dependences, since formally these hold between *all* different pairs taken from all instances. In the case of a flow, input or anti dependence, every intermediate write formally prohibits the flow to instances of later iterations, and those writes are taken into consideration, in contrast with possible writes of instances of other statements or caused by variables not under consideration.

If N_i cannot be determined at compile-time, the number of underlying dependences is set to unknown. If a DO-loop is not executed at all ($N_i = 0$) the static dependence with 0 underlying dependences is not recorded, since it does not really hold.

If the statements do not appear inside the same nest, some extra information must be used in the computation. Consider the following examples.

```
DO I = 1, 10, 1                DO I = 1, 10, 1
  S1: A = <expression>           S1: <variable> = A
ENDDO                          ENDDO
S2: <variable> = A             S2: A = <expression>
```

In the left fragment, the flow dependence between S_1 and S_2 has only one underlying instance, since formally no dependences holds between $S_1(1)$ through $S_1(9)$ and S_2. In the right fragment, however, 10 anti dependences are responsible for the static anti dependence $S_1 \delta S_2$, since no intermediate writes are done (assuming <variable> is not `A`). So the kind of dependence must be used in the computation of the number. In general it can be stated that for the direction vector $(\underbrace{=, ..., =}_{minnest})$, the following equations hold.

$$Num = (\prod_{i=1}^{minnest} N_i) * (\prod_{i=minnest+1}^{hisnest} N_i''), \quad dep = FLOW$$

$$Num = (\prod_{i=1}^{minnest} N_i) * (\prod_{i=minnest+1}^{mynest} N_i'), \quad dep = ANTI$$

$$Num = (\prod_{i=1}^{minnest} N_i), \quad dep = OUTPUT$$

In these equations, N_i, with $1 \leq i \leq minnest$ indicates the number of iterations of the *common nest*, N_i', with $minnest + 1 \leq i \leq mynest$, and N_i'', with $minnest + 1 \leq i \leq hisnest$ the number of iterations of the loops only surrounding the first and second statement respectively. Because the number of underlying instances for input dependences is usually a very large number (it is an accumulated product), it is only determined for the case that the nesting of one of the two statements is empty. The formula is given below for the case that the first statement has a nesting depth of 0.

$$Num = (\prod_{i=1}^{hisnest} N_i''), \quad dep = INPUT$$

In case the direction vector of a dependence has the form $(\underbrace{=,...,=}_{r},<,\quad \underbrace{*,...,*}_{minnest-1-r}\quad)$, the following equations hold.

$$Num = (\prod_{i=1}^{r} N_i) * (N_{r+1} - 1) * (\prod_{i=minnest+1}^{mynest} N'_i), \quad dep = FLOW$$

$$Num = (\prod_{i=1}^{r} N_i) * (N_{r+1} - 1) * (\prod_{i=minnest+1}^{hisnest} N''_i), \quad dep = ANTI$$

$$Num = (\prod_{i=1}^{r} N_i) * (N_{r+1} - 1), \quad dep = OUTPUT$$

6.7.2 Indexed Variables

The number of underlying dependences in which array variables are involved can be determined with the techniques found in [ETW92]. The implementation of these techniques in this prototype compiler can handle the cases in which the number of variables in the diophantine equation (for which the coefficient $\neq 0$) is less than 5 and for which all subscript expressions are in *normal form* and the bounds and stride values of all surrounding DO-loops are known.

First the set of equations that results in case of multi dimensional arrays is collapsed into a single equation by using the fact that an element `A(I,J,K)` of array `A` can be found at address $offset + I + J * d_1 + K * d_1 * d_2$ [31], in which d_i indicates the size of the i^{th} dimension. So, the following set can be collapsed into the equation below.

$$\begin{cases} a_{10} + a_{11} * i_1 + ... + a_{1k} * i_k = b_{10} + b_{11} * j_1 + ... + b_{1l} * j_l \\ \vdots \\ a_{m0} + a_{m1} * i_1 + ... + a_{mk} * i_k = b_{m0} + b_{m1} * j_1 + ... + b_{ml} * j_l \end{cases}$$

$$\Longleftrightarrow$$

$$a_{10} + ... + d_1 * ... * d_{m-1} * a_{m0} +$$

$$(a_{11} + ... + d_1 * ... * d_{m-1} * a_{m1}) * i_1 + ...$$

$$+(a_{1k} + ... + d_1 * ... * d_{m-1} * a_{mk}) * i_k$$

$$=$$

[31] Arrays are stored in column major order in FORTRAN.

$$b_{10} + ... + d_1 * ... * d_{l-1} * b_{l0} +$$

$$(b_{11} + ... + d_1 * ... * d_{l-1} * b_{m1}) * j_1 + ...$$

$$+(b_{1l} + ... + d_1 * ... * d_{l-1} * b_{ml}) * j_l$$

So the problem can be rewritten into an equation of the following form, for certain n, in which the coefficients that are 0 have been filtered out.

$$c_1 * i_1 + ...c_n * i_n = c_0$$

$$lower_p \leq i_p \leq upper_p, in\ which\ i_p\ has\ stride_p$$

After that the corresponding loops are normalized resulting in new coefficients $c'_p = c_p * stride_p$ for $1 \leq p \leq n$, lower bound (0) and upper bound ($upper' = lower_p/stride_p$) and adaptation of the constant for every p ($c - c_p * lower_p$). In this way, the stride of the original DO-loops is accounted for in the number determination since all the resulting DO-loops have stride 1. Since the method described in [ETW92] requires that all the coefficients are > 0 all coefficients that are < 0 are negated. Because this requires the bounds to be negated and reversed and since the resulting lower bounds must normalized again to 0, the adaptation of the constant ($c + c_i * upper'_i$) is also performed [32].

Because coefficients that are 0 cannot be used in the formulae found in [ETW92], they have been filtered out first. However, every possible value of associated loop-control variable will add an extra solution to all solutions of the equation. Therefore, the following number is computed, which will be used in the generation of the final answer.

$$Extra\_Solutions = \Pi_{p,coefficient\ p\ has\ been\ filtered\ out} N'''_p\ [33]$$

As last step, all coefficients and the constant are divided by GCD(c_1,...,c_n). If the constant cannot be divided by this GCD, it can be concluded that no dependence holds (this is an application of the GCD test on the collapsed equation).

The number of solutions for the resulting equation is determined with the techniques found in [ETW92]. In this prototype compiler it has been implemented for all $n \leq 4$. So this number can be determined for all dependences between statements for which the total number of loop-control variables used in the subscript expressions ≤ 4 (so the number of loop-control variables that have a coefficient with value 0 do not add to that number n). This means that for many cases the number can be determined as is illustrated for some simple examples in the following table (I_p are loop-control variables and c_q are constants).

[32] The reason that the first two steps found in [ETW92] (make all $a_i > 0$ and make all $M_i =$ can be performed in one step now, is that the loop has been normalized first.

[33] This can be N_i, N'_i or N''_p introduced in the previous section.

	n = 1	...	n = 4
1 dimension	$A(I_1) \leftrightarrow A(c_1)$ $A(c_1) \leftrightarrow A(I_1)$ $\vdots$		$A(I_1 + I_2 + I_3 + I_4) \leftrightarrow A(c_1)$ $A(I_1 + I_2 + I_3) \leftrightarrow A(I_4)$ $\vdots$
2 dimensions	$A(I_1,c_1) \leftrightarrow A(c_2,c_3)$ $\vdots$		$A(I_1,I_2) \leftrightarrow A(I_3,I_4)$ $\vdots$
$\vdots$			

The following observations are done in [ETW92].

$$\Delta_{upper'_1,...,upper'_n}(c_1, ..., c_n; c_0)$$

(the number of solutions for the bounded equation)
be expressed in terms of

$$\Delta_\infty(c_1, ..., c_n; c_0)$$

(the number of solutions for the unbounded case)
using the fact that Δ_∞ is the coefficient of x^{c_0} in the series development of

$$\frac{1}{(1 - x^{c_1})...(1 - x^{c_n})}$$

for which formulae can be found

Therefore, the number of solutions for the bounded case can also be determined. After this number has been computed it is multiplied with *Extra_solutions*. This final number is stored with every static dependence that results. So, for example, if the number of elements of array `A` used in S_1 as well as in S_2 can be determined, but the direction in the iteration space is unknown, that number is stored with $S_1 \delta_= S_2$ *and* $S_2 \overline{\delta}_< S_1$, indicating the fact that the total number of dependences that are responsible for the two static dependences is known.

```
L1: DO I = 1, 100, 1
  S1: A( g(I) ) = <expression>
  S2: <variable> = A( h(I) )
ENDDO
```

6.8 Examples of Data Dependence Computation

To illustrate the dependence computation discussed in this section, some small programs and the computed static dependences are given. The subscript expressions in these programs are shown in *normal form*.

```
L1: DO I = 1, 20, 1
  S1: A( <I> ) = B( <-1*I+41> )
  S2: B( <I> ) = C( <I> )
ENDDO
```

```
L2: DO J = 21, 40, 1
  S3: A( <J> ) = B( <-1*J+41> )
  S4: B( <J> ) = C( <J> )
ENDDO
```

These two loops are the result of the transformation Index Set Splitting, which splits the bounds of a DO-loop in such a fashion, that dependences between instances of different iterations are converted into cross DO-loop dependences. The data dependence computation correctly determines that no cross iteration dependences exist. The number of underlying dependences *between* both DO-loops can be determined

```
Dependence                Variable  Number

S1 d-anti        S4           B       20
S2 d-flow        S3           B       20
```

The following example demonstrates that normalizing DO-loops first is necessary for the correct determination of the underlying dependences. The dependence table is shown without statistics.

```
L1: DO I = 1, 100, 2
  S1: A( <I> ) = 100.000000
ENDDO
L2: DO I = 1, 100, 1
  S2: A( <I> ) = 200.000000
ENDDO
```

```
Dependence                  Variable  Number

L1 d-outp        L2             I       1
S1 d-outp        S2             A       50
```

The following example has been taken from [Ban88].

```
S1: X = (Y + 1)
L1: DO I = 2, 30, 1
  S2: C( <I> ) = (X + B( <I> ))
  S3: A( <I> ) = (C( <I-1> ) + Z)
  S4: C( <I+1> ) = (B( <I> ) * A( <I> ))
  L2: DO J = 2, 50, 1
    S5: F( <I> , <J> ) = (F( <I> , <J-1> ) + X)
  ENDDO
ENDDO
S6: Z = (Y + 3)
```

The following dependences are assumed, where the dependences marked with a $(*)$ were given in [Ban88]. The other dependences are between non-assignment statements ($L_2 \delta_< L_2$), or are input dependences.

```
Dependence                 Variable  Number

S1 d-flow        S2           X       29             *
S1 d-flow        S5           X       1421           *
S1 d-input       S6           Y       1
S2 d-input <     S2           X       ?
S2 d-flow  <     S3           C       28             *
S4 d-outp  <     S2           C       28             *
S2 d-input =     S4           B       29
S2 d-input =     S5           X       ?
S5 d-input <     S2           X       ?
S3 d-input <     S3           Z       ?
S4 d-flow  <     S3           C       27             *
S3 d-flow  =     S4           A       29             *
S3 d-anti        S6           Z       29             *
L2 d-outp  <     L2           J       28
S5 d-flow  =<    S5           F       1392           *
S5 d-input =<    S5           X       ?
S5 d-input <*    S5           X       ?

Number of static input  dependences : 8
Number of static output dependences : 2
Number of static flow   dependences : 6
Number of static anti   dependences : 1

Total number of static  dependences : 17
```

The following program computes the so-called Laplace operator:

```
      L1: DO I = 2, 99, 1
        L2: DO J = 2, 99, 1
          S1: A( <I>, <J>) = ((((A( <I-1>, <J>) + A( <I+1>, <J>
     + )) + A( <I>, <J-1>)) + A( <I>, <J+1>)) / 4)
        ENDDO
      ENDDO
```

As can be seen from the resulting static data dependences, cross iteration dependences exists for both loops (the input dependences are not shown).

```
Dependence                  Variable  Number

L2 d-outp  <     L2            J       97
S1 d-anti  =<    S1            A       9506
S1 d-flow  =<    S1            A       9506
S1 d-anti  <=    S1            A       9506
S1 d-flow  <=    S1            A       9506

Number of static input  dependences : 12
```

```
Number of static output dependences : 1
Number of static flow   dependences : 2
Number of static anti   dependences : 2

Total number of static  dependences : 17
```

The innermost DO-loop however can be skewed with a factor 1, which results in the following program.

```
      L1: DO I = 2, 99, 1
        L2: DO J = (I + 2), (I + 99), 1
          S1: A( <I>, <-1*I+J>) = ((((A( <I-1>, <-1*I+J>) + A( <I
     + +1>, <-1*I+J>)) + A( <I>, <-1*I+J-1>)) + A( <I>, <-1*I+J+1>
     + )) / 4)
        ENDDO
      ENDDO
```

The following dependences (shown only for flow and anti dependences that are caused by references to the array `A`) are determined. Note that the number of underlying dependences cannot be determined because the bounds of the innermost DO-loop are too complicated for the method used.

```
Dependence                   Variable  Number


...
S1 d-anti  =<    S1             A       ?
S1 d-flow  =<    S1             A       ?
S1 d-anti  <<    S1             A       ?
S1 d-flow  <<    S1             A       ?
...

Number of static input  dependences : 16
Number of static output dependences : 1
Number of static flow   dependences : 4
Number of static anti   dependences : 2

Total number of static  dependences : 23
```

Note that the second direction of the last two dependences shown are determined with the ad-hoc solution for the starting '<' and '>' directions. This method also works in the following fragment in which a negative stride is used.

```
      L1: DO I = 1, 100, 1
        L2: DO J = 100, 1, -1
          S1: A( <I+J> ) = 10.000000
          S2: C( <I> , <J> ) = A( <I+J+1> )
        ENDDO
      ENDDO
```

```
Dependence                    Variable  Number

L2 d-outp  <     L2              J       99
S1 d-outp  <<    S1              A       666700
S1 d-flow  <<    S2              A       666600
S2 d-anti  <[    S1              A       666600
S1 d-flow  =<    S2              A       666600
S2 d-input <<    S2              A       666700
S2 d-input <<    S2              A       666700

Number of static input  dependences : 2
Number of static output dependences : 2
Number of static flow   dependences : 2
Number of static anti   dependences : 1

Total number of static  dependences : 7
```

All dependences found in the following fragment that uses a so-called wrap around variable in the subscript expression of array `B`, are listed below (without the final statistics).

```
S1: II = 100
L1: DO I = 1, 100, 1
  S2: A( <I> ) = B(II)
  S3: II = I
ENDDO
```

```
Dependence                    Variable  Number

S1 d-flow        S2              II      100
S1 d-outp        S3              II      1
S2 d-input <     S2              B       ?
S2 d-input <     S2              B       ?
S2 d-input <     S2              II      ?
S2 d-anti  =     S3              II      100
S3 d-flow  <     S2              II      99
S3 d-outp  <     S3              II      99
```

The following example shows the fact that the number of dependences is computed without extensive analysis of the flow of control in the program. For example, it is assumed that the dependence $S_1 \delta^O S_1$ occurs 99 times, although the evaluation of the condition is unknown at compile time. Again the statistics are not shown.

```
L1: DO I = 1, 100, 1
  C1: IF (...) S1: R = (R + A( <I> ))
ENDDO
```

```
Dependence                  Variable  Number

S1 d-outp  <    S1              R       99
S1 d-anti  =    S1              R       100
S1 d-flow  <    S1              R       99
S1 d-anti  <    S1              R       99
S1 d-input <    S1              R       ?
```

The following example shows that the dependences $L_1 \delta L_2$, $L_2 \delta^i_< L_2$ and $L_2 \overline{\delta} L_3$ result from the decision to keep loop-control variables in the IN set during the determination of dependences between DO-loops. Note that even the correct number of underlying dependences can be determined if one keeps in mind that the variables used in bounds and stride are only read before all iterations are executed. Remember that `I` and `J` are no elements of the IN sets of the assignment statements S_1 and S_2.

```
      L1: DO I = 1, 100, 1
        L2: DO J = I, 100, 1
          S1: A( <I> , <J> ) = 100.000000
        ENDDO
      ENDDO
      L3: DO I = 1, 100, 1
        S2: B( <I> ) = 100.000000
      ENDDO

Dependence     Variable  Number

L1 d-flow        L2            I       100
L1 d-outp        L3            I       1
L2 d-outp  <     L2            J       99
L2 d-input <     L2            I       ?
L2 d-anti        L3            I       100
```

The last example shows all static dependences generated for a fragment with three nested DO-loops (without statistics).

```
      L1: DO I = 1, 100, 1
        L2: DO J = 1, 100, 1
          L3: DO K = 1, 100, 1
            S1: A( <I+2*J+2*K> ) = 10
            S2: E( <I> , <J> , <K> ) = A( <I+2*J+2*K+1> )
          ENDDO
        ENDDO
      ENDDO

Dependence                  Variable  Number

L2 d-outp  <     L2             J      99
```

```
L3 d-outp  =<    L3              K       9900
L3 d-outp  <*    L3              K       99
S1 d-outp  <**   S1              A       ?
S1 d-outp  =<*   S1              A       ?
S1 d-flow  <**   S2              A       ?
S2 d-anti  <**   S1              A       ?
S2 d-input <**   S2              A       ?
S2 d-input <**   S2              A       ?
S2 d-input =<*   S2              A       ?
S2 d-input =<*   S2              A       ?
```

Note that the self output dependence of S_1 is generated once, while the self input dependences appear twice. This is a result of the fact that the entry of (In,In) in the lookup table allows self dependences.

6.9 Concluding Remarks

The methods to determine static data dependences used in this prototype compiler, determine from a subscript expression the resulting dependences in three steps (is a dependence possible?; which direction does it have in the iteration space?; and which dependences result?). Another approach can be found in [Ban88], in which it is determined for all possible directions in the iteration space, if corresponding subscript expressions of two array variables can be equal. This is done, by using the general solution to the equations defined in terms of parameters in the resulting constraints to determine if a solution exists. This can be done with so-called 'linear programming' algorithms. If a solution can be found, the corresponding dependence is concluded. Since most of the computational effort to determine the general solution can be done during the evaluation of the GCD-test, the 'linear programming' step will require most computational time.

The following fragment, found in [Ban88], illustrates that the effectiveness of the technique proposed by Banerjee is not greater than the technique used in this prototype compiler in case of subscript expressions in which only one loop-control variable is used.

```
L1: DO I = L, U, 1
  S1: <variable>   = A( 3*I + 1 )
  S2: A( 2*I + 7 ) = <expression>
ENDDO
```

A static flow dependence between S_2 and S_1 exists, if the following equation can be satisfied under the given constraints (i and j correspond to the iteration of S_1 and S_2 respectively).

$$\exists_{(i,j)} : 3 * i + 1 = 2 * j + 7 \wedge j < i \wedge L \leq i \leq U \wedge L \leq j \leq U$$

The general solution of the linear diophantine equation, is $(i, j) = (2+2*t, 3*t)$. Using this solution, the problem can be rewritten into the following form.

$$\exists_t : 3 * t < 2 + 2 * t \wedge L \leq 3 * t \leq U \wedge L \leq 2 + 2 * t \leq U$$

The dependence holds, if such t can be found. Analog constraints result if it is determined that a cross iteration anti dependence holds ($i < j$) or an anti dependence in the same iteration holds ($i = j$). That the same dependences result with the techniques of the prototype compiler, can be seen in the following pictures. Figure 15 illustrates the conclusion that can be drawn in terms of parameter t, while the figure 16 illustrates that the same conclusion can directly be derived, using i and j.

In case of several loop-control variables, the technique of [Ban88] performs better in general, since every possible direction at every possible place is examined in turn, and used in the constraints, while the technique of the prototype compiler only accounts for the resulting constraints for a prefix of '=' directions and single '<' and '>' directions.

However, a future extension of the dependence computation is to use the technique of [Ban88] in the dependence generation algorithm of section 6.5 for every unknown direction (i.e. '$*$' and '+') before a dependence is added. In this case the generation must not stop after the first '<', since it is possible that not all the following '$*$' directions really hold. By doing so, the efficient algorithm acts as a filter, while expensive 'linear programming' is only used for the cases that are difficult to analyze.

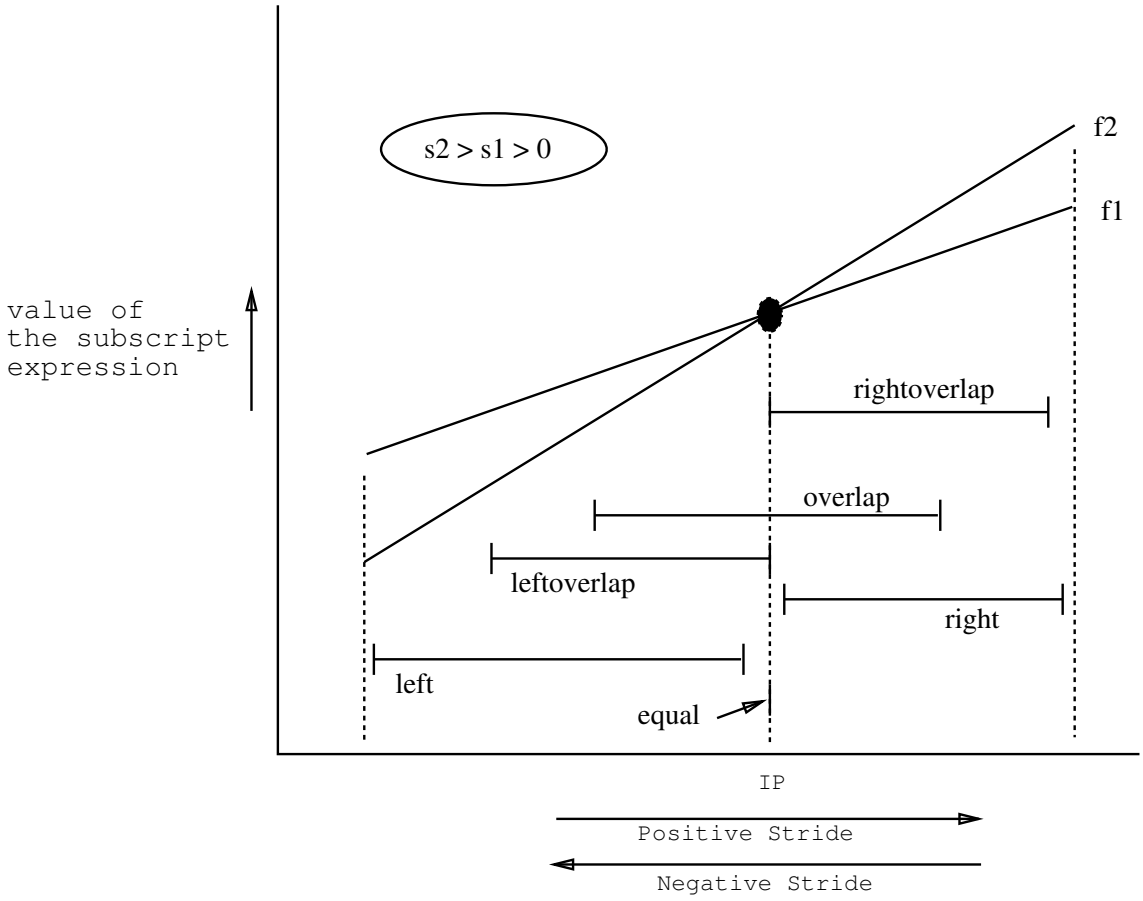


Figure 12: Dependence Direction

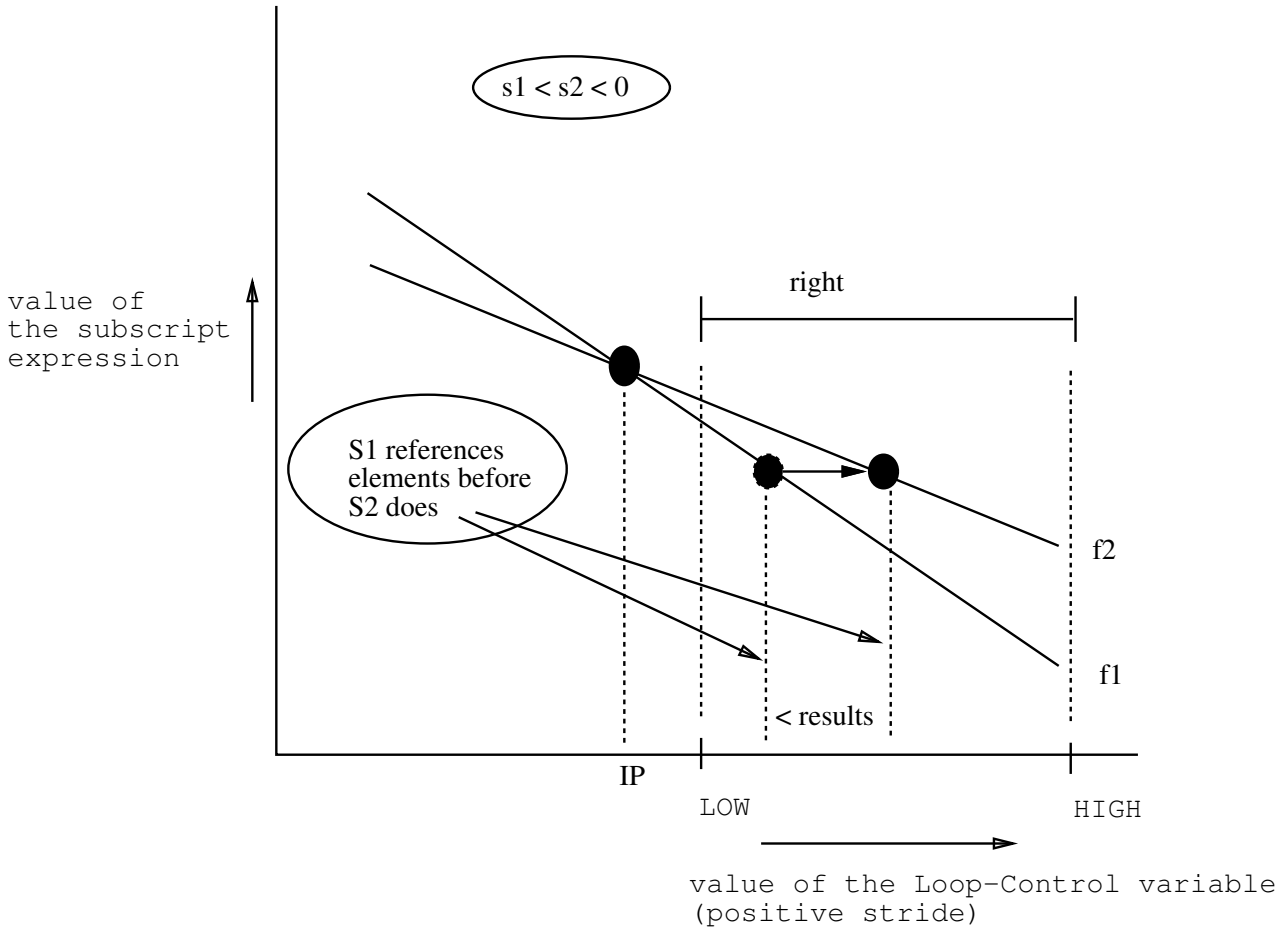


Figure 13: Dependence Direction

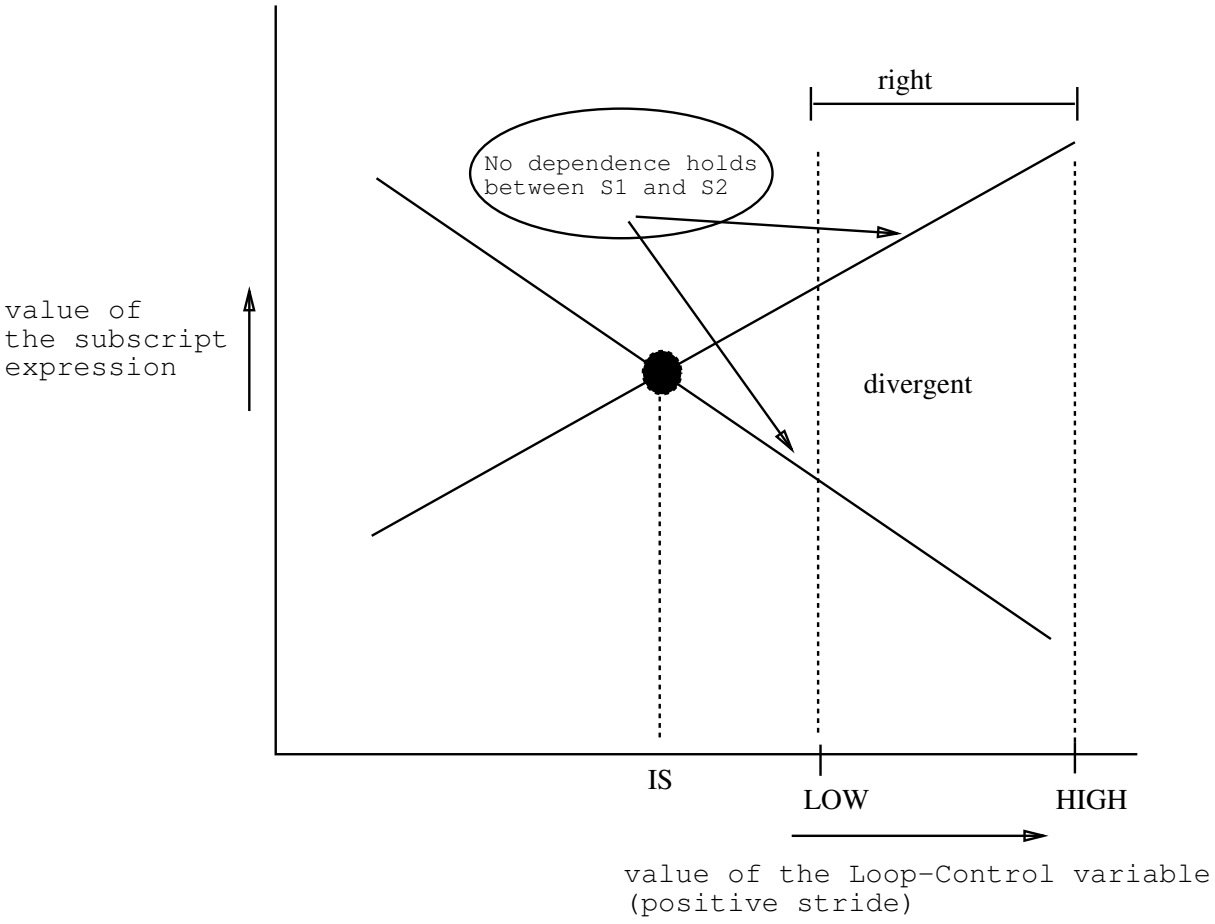


Figure 14: No Dependence

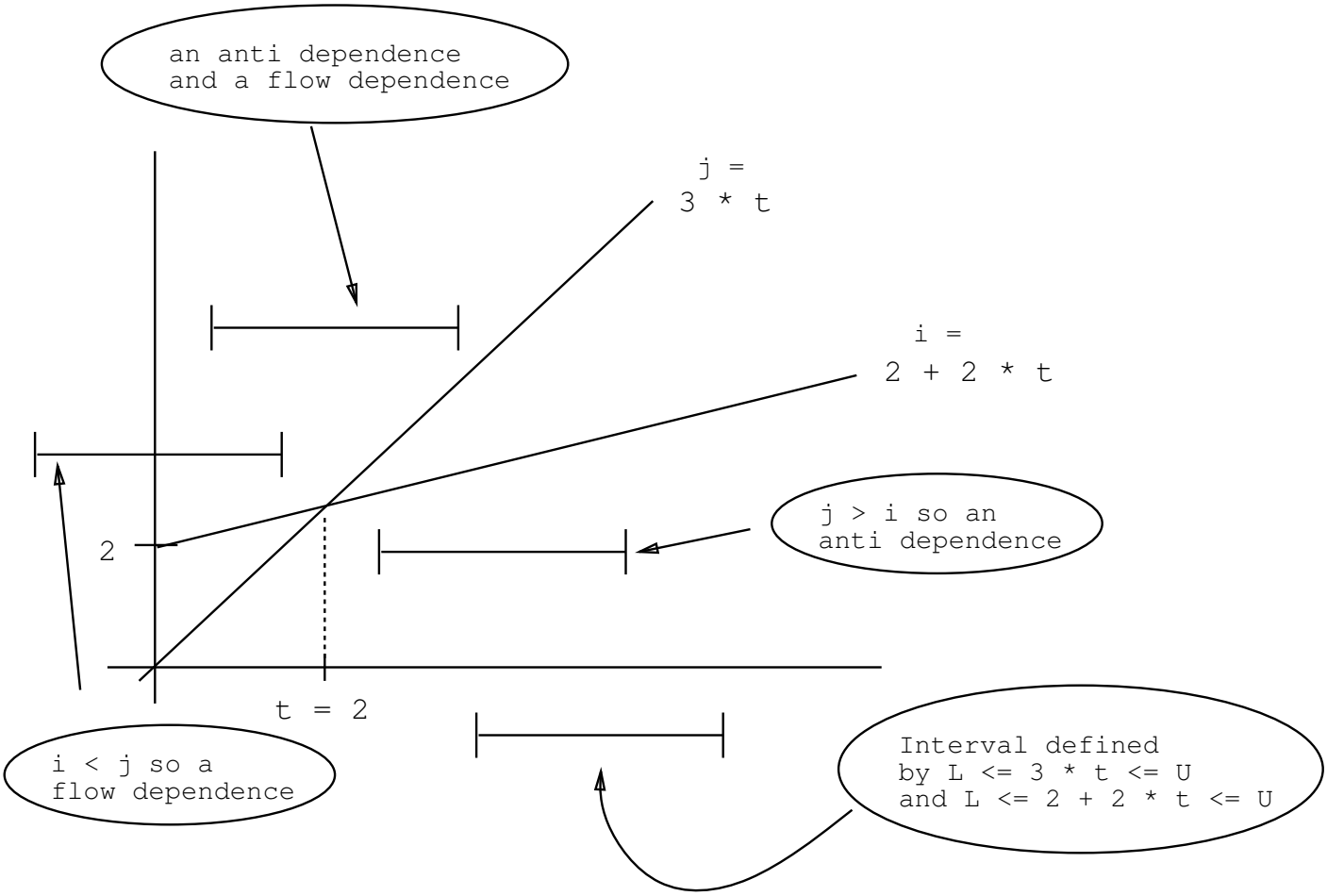

an anti dependence
and a flow dependence
j =
3 * t
i =
2 + 2 * t
j > i so an
anti dependence
2
t = 2
i < j so a
flow dependence
Interval defined
by L <= 3 * t <= U
and L <= 2 + 2 * t <= U

Figure 15:

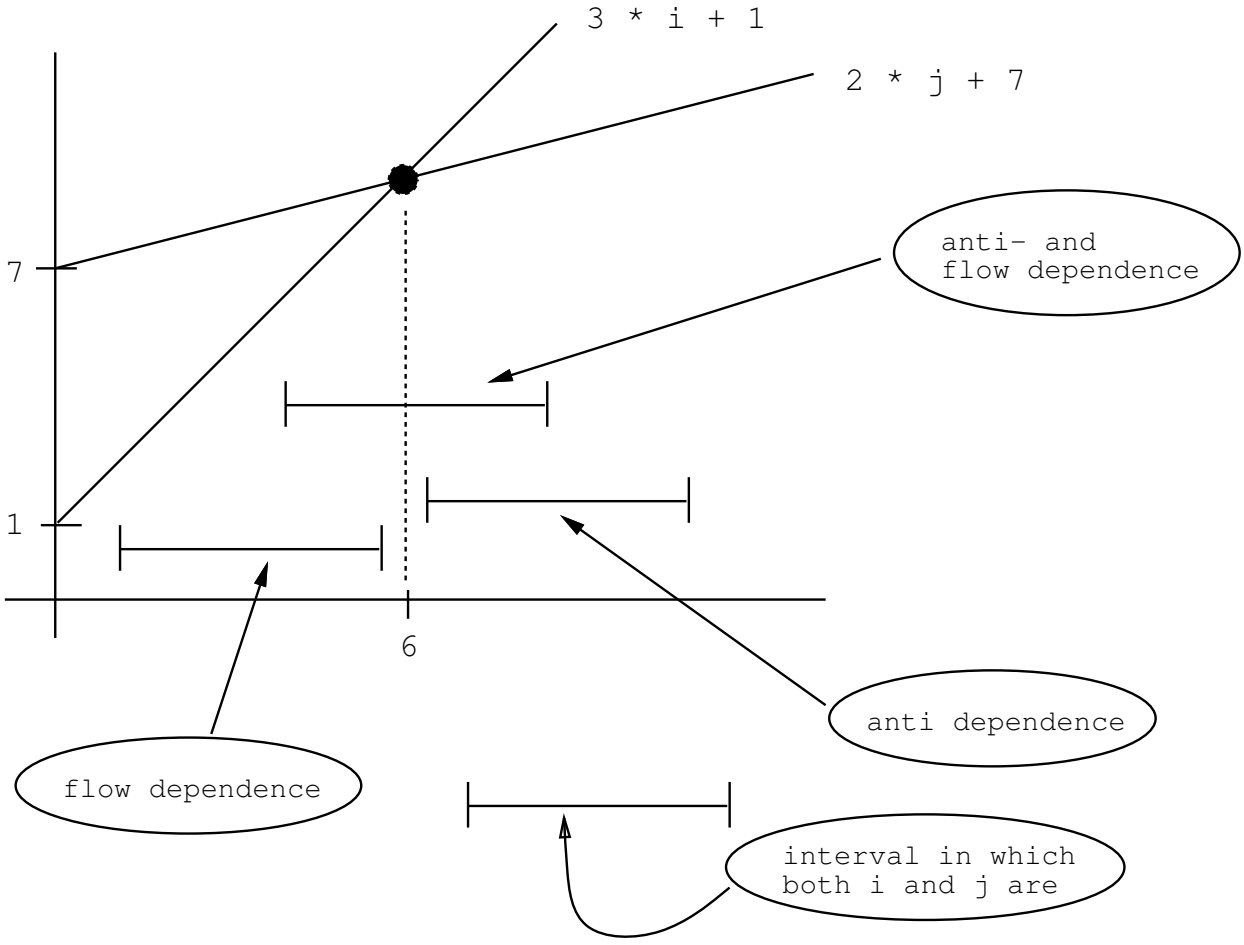

3 * i + 1
2 * j + 7
anti- and
flow dependence
7
1
6
anti dependence
flow dependence
interval in which
both i and j are

Figure 16:

7 The transformation language

The prototype compiler enables the user to define transformations which the compiler will use during the transformation phase. This is done by means of a simple transformation language to describe transformations on program patterns with associated conditions under which a transformation can be applied. The only conditions available in the first version of the transformation language, are conditions on data dependences, and are necessary to determine if a particular transformation is valid, i.e. applying the transformation does not change the semantics of the program. An extension of this language is to enable the user to define for example conditions on the values of variables or the number of underlying dependences. This is not necessary for determining if the transformation is valid, but to determine if the transformation is advantageous.

The general form of a single transformation with its conditions is shown in the following picture.

transform
  < *pattern*1 >
into
  < *pattern*2 >
condition
  < *conditionlist* >
;

Whenever < *pattern*1 > (which will be referred to as the left-hand-side pattern) matches on a fragment in the program, and all the conditions in < *conditionlist* > hold, the fragment in the program will be replaced with < *pattern*2 > (the right-hand-side pattern) if this transformation is applied. The patterns are created from simple templates, specifying statement lists, DO-loops, IF-statements and assignment statements, and statement-pointer variables, which will be bound to program fragments during the transformation phase. Patterns for expressions inside these statements can be specified using constants and operators as in FORTRAN 77 and expression-pointer variables. It is the responsibility of the user to specify the transformation and its conditions, in such a way that the semantics of the original code are preserved whenever this transformation is applied.

Because the goal of this prototype compiler is to explore the required nature of a transformation language and its effect on the behavior of the compiler, certain demands are placed on the first version of the transformation language and its implementation in the prototype compiler.

- It must be powerful enough to define most known program transformations and their conditions.
- It must have as few semantic constraints as possible, because constraints prohibit the flexibility of the language.
- Its implementation must be easy to adapt, because it must be possible to experiment with different language constructs.

Therefore it is decided to use the LEX and YACC tools to generate the lexical and syntax analyzer, since the underlying source language using these tools can be easily adapted. Only the front end phases are necessary to read in the transformations, since only the transformations have to be stored in a data structure. This is illustrated in the following picture.

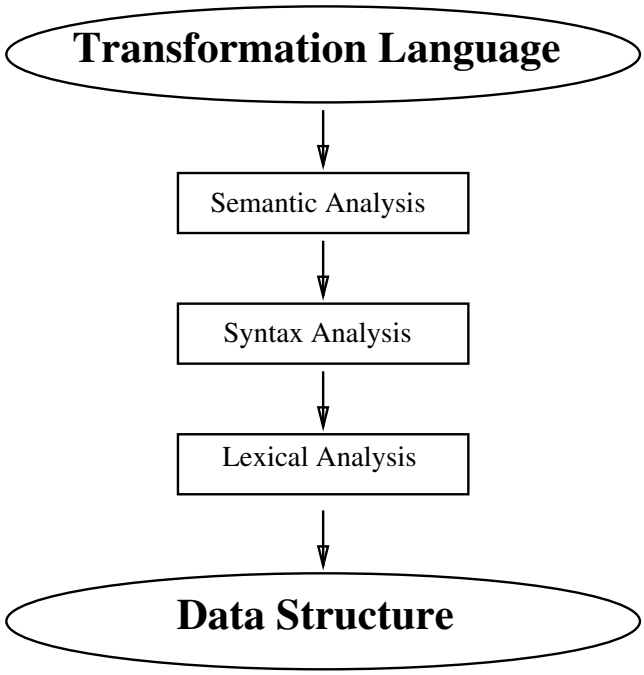


Figure 17: Reading in the Transformation Language

The following sections present the lexical, syntax and semantic analysis of the transformation language, and how transformations are stored. The use of the language to define transformations is discussed in section 7.5.

7.1 Lexical Analysis

All characters in the transformation language must be in lower case. The following strings are reserved keywords, used for specifying patterns and conditions: `and`, `assign`, `condition`, `do`, `doall`, `end`, `if`, `ifbody`, `into`, `list`, `merge` `next`, `nil`, `nodep`, `transform`, `true`, `flow`, `anti`, `output`, `input`, and `vectorize`. Integer, real and logical constants and operators are defined as in FORTRAN 77 (text in constants must also be in lower case). The value of an integer or the representation of a logical constant is stored in the attribute of the corresponding token, while the value of a real constant is saved in a temporary variable. Strings that match on one of the following regular expressions `"!e"{digit}{digit}?` and `"!s"{digit}{digit}?` are passed to the parser as `EXPRVAR` (an expression-pointer variable) or `STMTVAR` (a statement-pointer variable) respectively. The attribute of this token contains the number identifier of the variable.

Spaces, tabs and newlines between the lexemes are ignored. All text after a '#' symbol is ignored up to and including the first newline, to provide the

possibility to include comments in the transformation file. All strings that do not match keywords are assumed to be the name of variables or intrinsic functions [34], and are inserted in the symbol table. The entry in the symbol table is passed to the parser in the attribute of the token.

The LEX definitions for the transformation file can be found in module *scanner.l*, which is listed in appendix O.

7.2 Syntax Analysis

The transformation language is parsed according to the following context-free grammar.

$$
\begin{array}{lcl}
\text{trafos} & \rightarrow & \text{trafo trafos} \\
 & | & \text{END} \\
 & ; &
\end{array}
$$

[34] Which are supported in the resulting code, as is explained in section 7.5.

```
trafo      →  TRANSFORM pattern
              INTO pattern
              CONDITION conditions
           ;
pattern    →  LIST '(' stmt ',' pattern ')'
           |  STMTVAR
           |  MERGE '(' pattern ',' pattern ')'
           |  NIL
           ;
stmt       →  DO '(' exp ',' exp ',' exp ',' exp ',' pattern ')'
           |  DOALL '(' exp ',' exp ',' exp ',' exp ',' pattern ')'
           |  ASSIGN '(' exp ',' exp ')'
           |  IF '(' exp ',' pattern ')'
           ;
exp        →  EXPRVAR
           |  exp '+' exp
           ⋮  ⋮
           |  NOT exp
           |  '-' exp
           |  '(' exp ')'
           |  INTCONST
           |  REALCONST
           |  BOOLCONST
           |  VECTORIZE '(' EXPRVAR '|' EXPRVAR ',' exp ':' exp ':' exp ')'
           |  FUNC '(' exp ',' exp ')'
           |  FUNC '(' exp ')'
           |  FUNC
           ;
conditions →  condition AND conditions
           |  condition
           ;
condition  →  TRUE
           |  NODEP DEPKIND dirvec '(' s_indic ',' s_indic ')' onclause
           ;
onclause   →  '>' EXPRVAR
           |  ϵ
dirvec     →  dir dirvec
           |  ϵ
           ;
dir        →  '='
           |  '<'
           |  '>'
           |  '*'
           ;
```

```
s_indic  →  '$' attribs
         |
         ;
attribs  →  '.' NEXT
         |  '.' DOBODY
         |  '.' IFBODY
         |  '.' HEAD
         |  ϵ
         ;
```

Program patterns are described with the templates `do(...)` , `doall(...)`, `assign(...)`, and `if(...)` [35] inside a statement list template `list(...)`. This last template describes a statement list as a single statement followed by a statement list (again specified by a pattern). An empty statement list if specified with the template `nil`. The body of a DO-loop or an IF statement is another pattern. Every pattern ends with the description of an empty statement list `nil`, a statement pointer variable (e.g. `!s12`), which will be bound during the application of this transformation to the program fragment appearing at that place, or a **merge** operator, which will merge two statement lists during the transformation phase (this operator can only be used in the right-hand-side pattern).

Expressions are specified with the operators and constants of FORTRAN 77 or expression-pointer variables. The last three rules of *exp* are necessary to introduce variables (scalar or array variables of dimension 1 or 2) or intrinsic functions with one or two arguments. This construction can therefore only be used in right-hand-side patterns. Another construction that can only be used on right-hand-side patterns is the **vectorize** operator, which transforms expressions into vectorized form.

Every transformation has an associated condition list, which consists of conditions separated by the keyword `and`. A condition is either `true`, which always holds, or an expression of the following form: `nodep` $<$ *kind* $>$ $<$ *datadirectionvector* $>$ ($<$ *structure*1 $>$, $<$ *structure*2 $>$). The structures are used to select statement lists inside the patterns.

The YACC definitions can be found in module *parser.y*, which is listed in appendix P. If syntax errors appear in the transformation definitions, the first one is reported to the user, after which the parsing phase terminates.

7.3 Semantics of the Transformation Language

Although the semantic checking of the language is limited, some checks are performed to prevent serious problems during the transformation application phase.

Statement- and expression-pointer variables used inside right-hand-side patterns or the associated condition must also appear in the corresponding left-hand-side pattern, because otherwise they will be unbound during the application phase. If such unbound variables are detected, an error is generated. If a

[35] This templates matches on general-IF statements as well as on logical-IF statements.

statement-pointer variable is used more than once in a left-hand-side pattern an error is generated. This is not done for expression-pointer variables that are used several times, because this is allowed to indicate that the bindings of these variable must be consistent in a matching program fragment.

If the **merge** or **vectorize** operator or the construction for introducing variables or intrinsic functions occurs inside a left-hand-side pattern, an error is generated. If a **vectorize** operator is used inside another **vectorize** operator, an error is generated too. It is also tested whether the structures inside the conditions match on the left-hand-side pattern of that transformation and the number of associated directions correspond with the *common nest* of two statement lists, defined by these structures. See section 7.5 for more explanation.

7.3.1 Example of semantic checking

The following errors are generated, when the following transformation file is presented to the prototype compiler (the transformation specified is completely useless, but is used to illustrate the generation of errors).

```
    # Example of an incorrect transformation file

    transform
      list( do(!e1, !e2, !e3, !e4, merge(!s1,!s1)), !s1 )
    into
      list( doall(!e1, !e2, !e3, !e5, !s1), !s2 )
    condition
      nodep flow << ($.next,$.ifbody)
    ;

    end

=> readtrf wrongtrafo

- Error: Variable s1 is used more than once (line 4)
- Error: Merge cannot be used at left hand side (line 4)
- Error: Variable e5 is not defined before use (line 6)
- Error: Variable s2 is not defined before use (line 6)
- Error: Incorrect number of directions (line 9)
- Error: Second structure in condition does not match pattern (line 9)

1 transformation

Terminated
```

7.4 Storing Transformations

Every template, expression and condition is stored in a node, with additional fields if necessary. The first field in every node indicates what is stored in that

node. The second field is a pointer to the following node, so that templates and conditions can be linked together into a list, as was done to store the FORTRAN 77 program. Three dynamic [36] arrays are used as pointers to the lists of the left- and right-hand-side patterns and the condition list respectively. Every entry in these arrays corresponds to a single transformation. A stack of nodes has been implemented to support the process of creating the data structure for transformations. All type definitions required can be found in file *trafo.h*, listed in appendix N, which must be included in all modules working with this data structure. The actions to build the data structure are executed again during the parsing phase, so some of the functions can be found in module *parser.y*, while others are implemented in modules *trafo.c* and *trafoapp.c*, listed in appendix R and S respectively.

The different templates are stored using the following fields per node (since the `list` template has been used to link patterns, it is stored implicitly in 'next' fields).

- **DO-loop pattern** Four extra fields are pointers to the representation of the variable, lower bound, upper bound and stride expressions. Another field contains a pointer to the pattern that describes the DO-loop body.
- **Assignment pattern** Two fields contain pointers to the representation of the left- and right-hand-side expressions of an assignment statement.
- **If pattern** The first field is a pointer to the representation of the condition, while the second one points to the representation of the IF-body.
- **Statement-pointer Variable** One extra field is required, which contains the number identifier of the variable.
- **Nil pattern** No additional fields are required.

The data structure of the DO-loop, assignment and IF pattern templates is illustrated in the following picture.

Expressions inside the patterns are either FORTRAN 77 expressions or expression pointer variables. Therefore, most expressions can be stored similar to the way they were stored for FORTRAN 77 programs, while the data structure for an expression-pointer variable contains the number identifier of the variable in an extra field.

The data structures for the remaining constructions are described below.

- **Merge Operator** Two fields are used to store pointers to the representation of the patterns that must be merged during the application phase.
- **Vectorize Operator** One field contains the number identifier of the expression variable that must be vectorized, while the other fields are pointers to the variable that must be vectorized, the lower bound, upper bound, and stride of that variable, respectively.

[36] A dynamic array is used in order to be able to store any number of transformations.

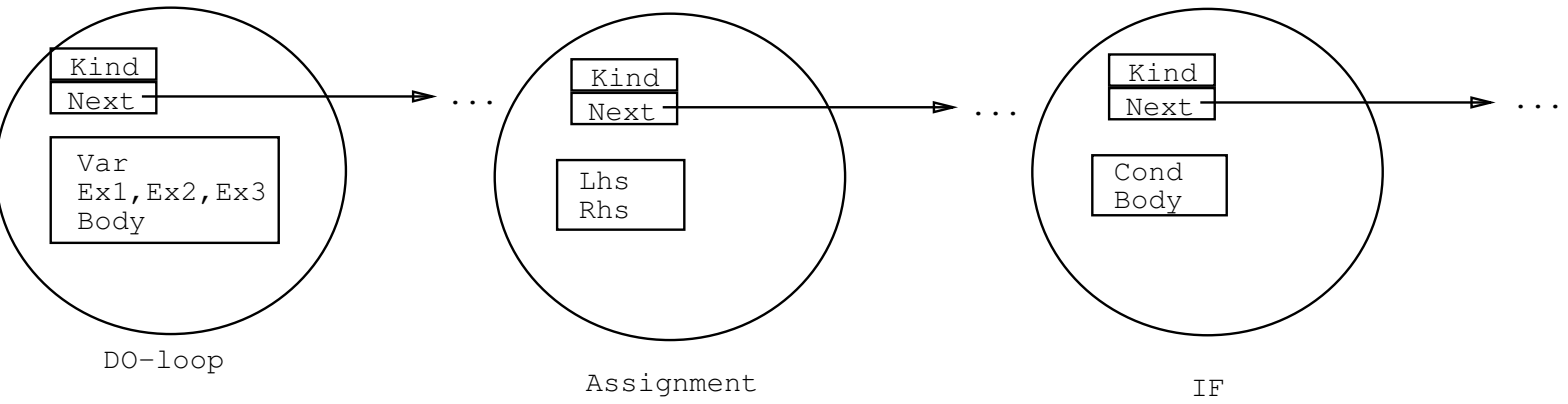


Figure 18: Template Data Structure

- **Introduction Construction** One field contains an entry in a dynamic array of characters, while a second one contains the length of the identifier stored at that place. Two fields can be used to store pointers to the representation of the expressions that will be generated as subscript expressions or actual arguments, depending on the fact if an array variable, or intrinsic function is introduced.

- **Conditions** The data structure of a **nodep** condition is as follows. One field is used as entry into a dynamic array of characters of the direction vector, while two others are used as entries in the same array to point to the representation of the structures in an associated condition. The kind of dependence is also stored in an attribute. The structures are stored as strings, where **.next** is stored as a 'n', **.dobody** as a 'd', **.ifbody** as an 'i', and **.head** as an 'h' (so an empty string represents '**$**', since all structures start with a '**$**'). The data structure for **true** has no extra fields.

7.5 Transformation Definitions

The patterns are described using the templates `do` for a serial DO-loop, `doall` for a DOALL-loop, `assign` for an assignment statement and `if` for a general or logical-IF statement. The templates are linked together using the templates `list`, which is a single statement followed by a statement list (pattern). The template `nil` can be used to specify an empty statement list, so it can be used to match the end of a body or the end of the program. Whenever a whole statement list must be matched without knowledge of the exact form that it will have during the application phase, a statement-pointer variable can be used at all places where a pattern is allowed. That variable will be bound to the statement list at that place during the application phase, and can be used in right-hand-side patterns again to indicate the insertion of that statement list there. Because the environment of statements inside that list may change, and duplicating rules are allowed (i.e. a statement-pointer variable appears more

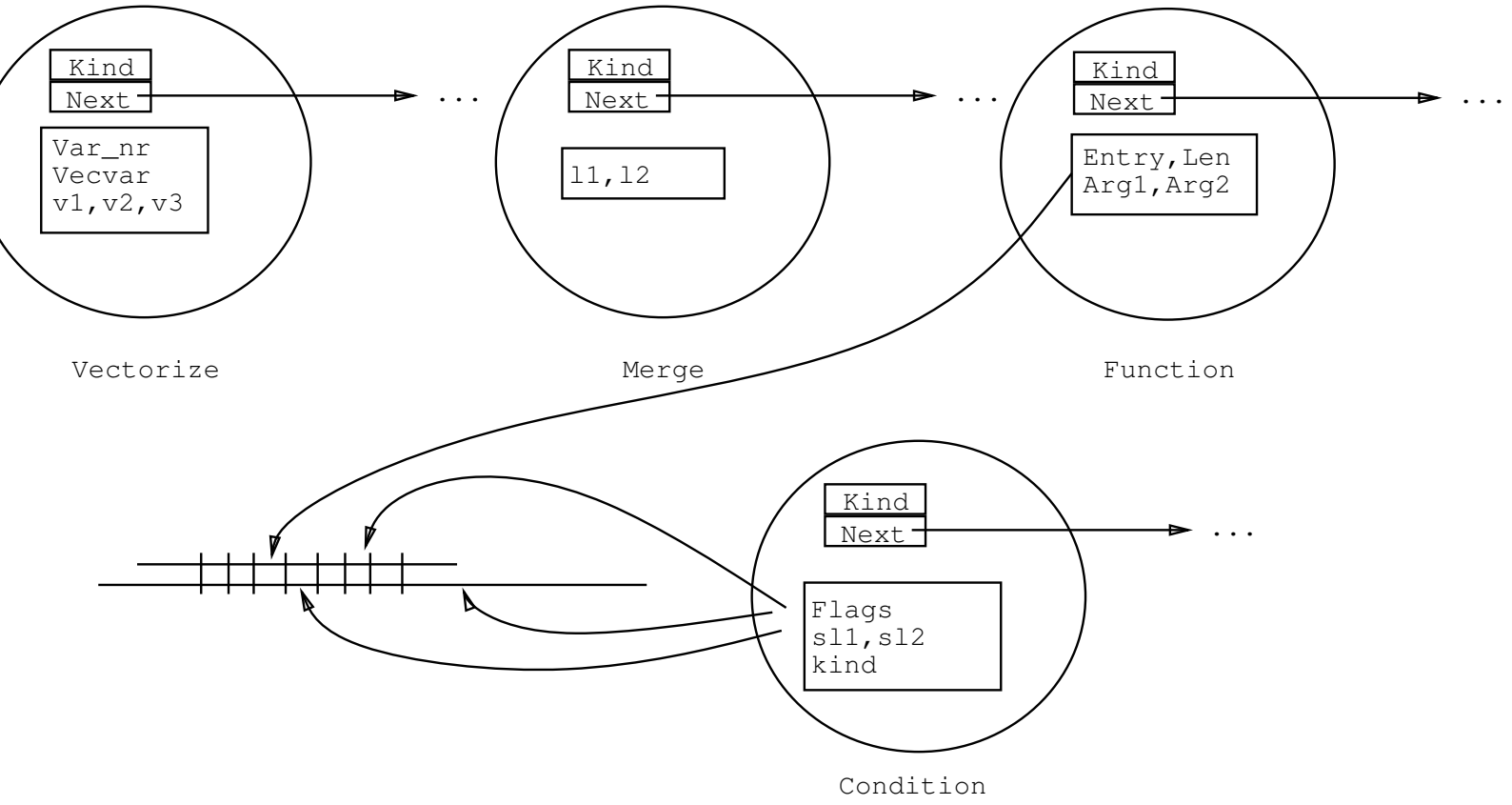


Figure 19: Remaining Data Structures

than once in a right-hand-side pattern), those lists will be copied whenever the transformation is applied.

Expressions are described using the operators and constants of FORTRAN 77 or expression-pointer variables at the places where a binding must be created during the application phase. If an expression-pointer variable is used more than once in a left-hand-side pattern, the expressions used at all following places where the variable occurs must be consistent with the binding created earlier. Otherwise the pattern does not match. Note that this technique cannot be used on statement-pointer variables. Multiple occurrences of expression-pointer variables in right-hand-side patterns will result in the duplication of the expression bound to that variable. The routines for **constant folding** are not applied on the resulting expression, to reflect the real transformation applied.

As an example, the following transformation defines the elimination of useless assignments from a program, since only assignment statements with the same expression at both sides will match. The variable $!s1$ is only necessary to save the rest of the program, since transformations on single statements cannot be defined with this transformation language. This variable will be referred to as the 'rest of program' pointer, and as will be explained later on, its behavior concerning copying is somewhat different from other variables.

```
transform
  list( assign( !e1, !e1 ), !s1 )
into
  !s1
condition
  true
;
```

Note that since expression-pointer variables can be bound to any kind of expression, the user must ensure that no non-variable expressions are introduced at places where they are not allowed (i.e. in loop-control variable definitions, and in left-hand-side expressions of assignment statements). To keep the language flexible, no semantic checks are performed to prevent this from happening. However, some checks are performed during the application phase (see section 8.5).

The structures in conditions point to specific statement lists in the left-hand-side pattern. This is done starting at the begin of the matching program fragment (indicated by a '**$**') and using the constructions **.next**, **.dobody**, and **.ifbody** to access the next statements or body of a DO-loop or IF statement respectively. The following example illustrates how to use structures to specify specific statement lists, in a program fragment that matches on the following pattern.

```
list( do(!e1, !e2, !e3, !e4, list( if(!e5, !s1), !s2 )), !s3 )
```

It also illustrates the binding of the statement-pointer variables. The variable `!s3` is the 'rest of program' pointer. Since a body consists of a single statement list, **$.body.ifbody** and `!s1` for example, are only bound to the '`...`' statements inside the IF body. All following statements are not visible.

```
            $ →  DO I = 1, 100
      $.dobody →    IF (L2) THEN
$.dobody.ifbody →      ...          ← !s1
                   ENDIF
  $.dobody.next →  ...              ← !s2
                   ENDDO
         $.next →  ...              ← !s3
```

Note the difference between a pattern, that specifies the form of a program, and a structure that is used to 'enter' the matching fragment during the transformation application phase. The structures discussed so far can only specify statement lists. Therefore, the construction **.head** has been implemented which always occur at the end of a structure and indicates that only the first statement of the statement list indicated by the structure preceding the **.head** must be considered. Note that if this single statement is a DO-loop or an IF statement, the statements inside the body are also considered (since they belong to that single statement). So in the given example **$.head** specifies the DO-loop with its body (consisting of the IF statement with its body and all statements in the list bound to `!s2`), while **$** specifies this DO-loop with all statements

in its body and the following statement list (bound to `!s3` and specified with **$.next**).

The first structure associated with a **nodep** construction specifies the statement list from which the source statements must be taken, while the second one specifies the list from which the sink statements must be taken. A direction vector must also be given, with a direction ('<', '>', or '*') for every loop-control variable in common in the pattern (during the application phase the surrounding loops not present in the pattern will be taken care of; see section 8.2).

A specification of variables on which the checking of dependences must be done is optional and can be specified with `> !e<num>`. All variable names that occur in the expression bound to the expression-pointer variable during transformation application phase are extracted and used to examine only the dependences caused by these variables. Note that this does not mean that the dependence must also have the statement in which `> !e<num>` occurs as sink or source statement, since only the names of the variables are used. This is done because expression-pointer variables can occur in several statements (disabling the uniqueness of the statement) and the specification of the sink/source statement can be done with the **.head** construction.

As an illustration of the use of structures, the transformation 'loop concurrentization' is specified below.

```
transform
  list( do   ( !e1, !e2, !e3, !e4, !s1 ), !s2 )
into
  list( doall( !e1, !e2, !e3, !e4, !s1 ), !s2 )
condition
   nodep flow   < ($.dobody,$.dobody) and
   nodep anti   < ($.dobody,$.dobody) and
   nodep output < ($.dobody,$.dobody)
;
```

The three conditions state that, if no cross-iteration flow, anti or output dependences from which the source and the sink statements are in the DO-loop body hold, concurrentization of the loop is valid. Cross-iteration input dependences are allowed, since they can be violated without affecting the semantics of the program.

The **merge** operator can be used to merge two patterns into one which is illustrated below in a transformation called 'loop fusion'. In this transformation the expression variables are used to check if the loop-control variables and its bounds and stride are identical. The conditions express that no flow, anti or output dependences may hold between statements in the first DO-loop body and statements in the body of the second DO-loop.

```
transform
   list( do(!e1, !e2, !e3, !e4, !s1),
   list( do(!e1, !e2, !e3, !e4, !s2), !s3 ) )
into
```

```
      list( do(!e1, !e2, !e3, !e4, merge(!s1, !s2)), !s3 )
   condition
      nodep flow   ($.dobody,$.next.dobody) and
      nodep anti   ($.dobody,$.next.dobody) and
      nodep output ($.dobody,$.next.dobody)
    ;
```

Since this transformation has as goal to fuse consecutive DOALL-loops, to minimize the startup and synchronization time needed, the patterns can better be defined using `doall` templates, because otherwise the user is also asked in some cases if two consecutive DO-loops must be merged from which one cannot even be converted into a DOALL-loop.

The **vectorize** construction is used to convert expressions that are bound to certain expression-pointer variables (the first argument of this construction) into vector format, i.e. using subscript triplets. Occurrences in subscript expressions of the variable, bound to the expression-pointer variable that is given as second argument are converted into the triplet specified as last argument during the application phase. If that expression-pointer variable is *not* bound to a scalar variable during the application phase an error is reported and the transformation cannot be performed. The transformation 'vectorization' is defined below. Note that the **.head** is not really necessary in this case, because the left-hand-side pattern enforces the form of the DO-loop body to be a single assignment. Formally this transformation is not correct, because the scalar variable can have another value after the vector instruction in the resulting program as a result from the fact that the DO-loop is eliminated, and this value might be used by following statements. However, this detail is usually ignored in vectorizing compilers since this value is seldomly reused. An extra assignment after the vector instruction can solve this problem [37]. To keep the examples compact in this and the following examples this is not included.

```
transform
  list( do(!e1, !e2, !e3, !e4, list( assign(!e5, !e6), nil )), !s1 )
into
  list( assign(vectorize(!e5 | !e1, !e2:!e3:!e4),
               vectorize(!e6 | !e1, !e2:!e3:!e4) )
condition
  nodep flow < ($.dobody.head,$.dobody.head)
;
```

The reason that no cross-iteration flow dependence is allowed results from the hardware used to implement vector instructions, in which values recently written may still be in the data stream where they are inaccessible.

The introduction of variable or intrinsic functions with arguments is illustrated in the following definition of 'strip mining'.

[37] A **nodep** clause on flow dependence on the loop-control variable between the DO-loop and the 'rest of program' is insufficient, because it is not always the case that all following statements are available with that variable. For example, in a match on a DO-loop in the body of a surrounding DO-loop the 'rest of program' is only bound to all following statements in that body.

```
transform
  list( do(!e1, !e2, !e3, 1, !s1), !s2 )
into
  list( do(itemp, !e2, !e3, 32,
  list( do(!e1, itemp, min2(itemp + 31, !e3), 1 , !s1) , nil ) ), !s2 )
condition
  true
;
```

Since `itemp` is not a keyword and it has no arguments, it is seen as a scalar variable. Its type is determined according to the implicit data typing rules (INTEGER in this case) and it is inserted in the symbol-table. The same is done for `min2`, which is assumed to have dimension 2. If the variable is already present in the symbol-table, it is determined if this type and dimension are consistent. If this is not the case, the transformation cannot be applied. It is therefore important to use identifiers that are not commonly used in programs. It is also important to introduce consistent names, i.e. if the name `itemp` is used in more than one transformation, it must always be used as a scalar variable, because otherwise the second transformation cannot be applied, due to a dimension conflict. This introducing technique can be used to introduce array variables or intrinsic functions. In this case the intrinsic function *min*2 is introduced. Array variables or intrinsic functions are stored in the symbol-table with the dimension but without any bounds. In the generated program the declaration `INTEGER min2()` will appear to reflect the fact that `min2` is a function or array of type INTEGER. In this way intrinsic functions are simulated for the resulting code (while in fact they are always stored as array variables).

The transformation 'loop distribution' is described below, and illustrates the need for the **.head** construction. It is specified for the case that the DO-loop body starts with an assignment statement.

```
transform
  list( do(!e1, !e2, !e3, !e4, list( assign(!e5, !e6), !s1 )), !s2 )
into
  list( do(!e1, !e2, !e3, !e4, list( assign(!e5, !e6), nil) ),
  list( do(!e1, !e2, !e3, !e4, !s1), !s2) )
condition
  nodep flow   < ($.dobody.next,$.dobody.head) and
  nodep anti   < ($.dobody.next,$.dobody.head) and
  nodep output < ($.dobody.next,$.dobody.head)
;
```

If the **.head** was not present, the following DO-loop could not be distributed due to the dependence $S_2 \overline{\delta}_< S_2$ (remember that **$.dobody** alone indicates that the sink statement must be in that statement list, which is { S_1, S_2 } in this case). With the **.head** construction, the only allowed sink statement is S_1, while **$.dobody.next** enforces the source statement to be S_2.

```
      L1: DO I = 1, 100, 1
```

```
      S1: B(I) = 10
      S2: A(I) = A(I + 1)
    ENDDO
```

The following transformation is used to convert a reduction operation into a call to an efficient intrinsic function, and illustrates the use of the optional specification of the variables used in considering the dependences.

```
transform
  list( assign(!e1, 0.0),
  list( do(!e2, !e3, !e4, !e5,
           list( assign(!e1, !e1 + !e6 * !e7), nil )), !s1 ) )
into
  list( assign(!e1, dot(vectorize(!e6 | !e2, !e3:!e4:!e5),
                        vectorize(!e7 | !e2, !e3:!e4:!e5))), !s1 )
condition
  nodep flow < ($.next.dobody.head,$.next.dobody.head) > !e6
  nodep flow < ($.next.dobody.head,$.next.dobody.head) > !e7
;
```

The application of this transformation is illustrated below. Since 'vectorization' is applied, a test must be performed whether no cross iteration flow dependences hold. However, without the specification on which variables no dependence may hold, the transformation would be useless, since a cross iteration flow dependence on `ASUM` always holds.

```
ASUM = 0.0
DO I = 1, 100
  ASUM = ASUM + A(I) * B(I)  >>>  ASUM = dot(A(1:100:1),B(1:100:1))
ENDDO
```

Note the use of the FORTRAN constant 0.0 of type REAL in the transformation definition. The transformation definition can be made more general, by only considering the reduction, so it will match on reduction with any start value, as is illustrated below for a sum reduction.

```
transform
  list( do(!e2, !e3, !e4, !e5,
  list( assign(!e1, !e1 + !e6), nil )), !s1 )
into
  list( assign(!e1, !e1 +
       sum(vectorize(!e6 | !e2, !e3:!e4:!e5))), !s1 )
condition
  nodep flow < ($.dobody.head,$.dobody.head) > !e6
;
```

Another reduction operation is the determination of the maximum value in an array. The corresponding transformation is shown below, with an application.

```
  transform
    list( do(!e1, !e2, !e3, !e4, list( if(!e5.gt.!e6,
        list( assign(!e6, !e5), nil) ), nil )), !s1 )
  into
    list( assign(!e6, max2(!e6,
     maxvec(vectorize(!e5 | !e1, !e2:!e3:!e4)))), !s1 )
  condition
    nodep flow < ($.dobody.head,$.dobody.head) > !e5
  ;
```

```
MAX = A(1)
DO I = 1, 100, 1                    MAX = A(1)
  IF (A(I).GT.MAX) MAX = A(I) >>> MAX = max2(MAX,maxvec(A(2:100:1))
ENDDO
```

Consider the following DO-loops.

```
L1: DO I = 1, 100. 1
  L2: DO J = 1, 100, 1
     S1: A(I,J) = 100
  ENDDO
ENDDO
```

It is well known that both DO-loops can be executed in parallel. However, with the definition of 'loop concurrentization' given earlier, only the innermost loop is converted into a DOALL-loop. This is a result from the fact that the dependence $L_2\delta^O_< L_2$ is computed prohibiting the concurrentization of the outermost DO-loop. In fact this conclusion is correct, since complete concurrentization of the outermost DO-loop requires 100 different local `J` variables. The following transformation definition reflects this need by insertion of the ad hoc notation `J = local(J)` and concurrentization of the outermost DO-loop.

```
  transform
    list( do(!e1, !e2, !e3, !e4,
     list( do(!e5, !e6, !e7, !e8, !s1), nil )), !s2 )
  into
    list( doall(!e1, !e2, !e3, !e4,
     list( assign(!e5, local(!e5)),
      list( do(!e5, !e6, !e7, !e8, !s1), nil ) )), !s2 )
  condition
    nodep flow   <* ($.dobody.dobody,$.dobody.dobody) and
    nodep anti   <* ($.dobody.dobody,$.dobody.dobody) and
    nodep output <* ($.dobody.dobody,$.dobody.dobody)
  ;
```

A transformation called 'loop interchanging' changes the order of nested DO-loops with as goal to have the stride-1 loop-control variable in the innermost DO-loop, with vectorization of that DO-loop in mind. The definition for 'loop interchanging' with vectorization is given below.

```
transform
  list( do(!e1, !e2, !e3, !e4, list( do(!e5, !e6, !e7, !e8,
    list( assign(!e9, !e10), nil )), nil )), !s1 )
into
  list( do(!e5, !e6, !e7, !e8,
   list( assign( vectorize(!e9  | !e1, !e2:!e3:!e4),
                 vectorize(!e10 | !e1, !e2:!e3:!e4)), nil )), !s1 )
condition
  nodep flow   <> ($.dobody.dobody.head,$.dobody.dobody.head) and
  nodep anti   <> ($.dobody.dobody.head,$.dobody.dobody.head) and
  nodep output <> ($.dobody.dobody.head,$.dobody.dobody.head) and
  nodep flow <= ($.dobody.dobody.head,$.dobody.dobody.head)   and
  nodep flow ($.head,$.dobody.head) > !e1 # second loop is independent
;
```

The first three conditions are necessary to check if the two DO-loops can be interchanged. The fourth test determines if vectorization of the resulting innermost loop is allowed (no cross-iteration flow dependence in one iteration of the resulting outermost DO-loop). The last condition checks if the value of the loop-control variable is not used in the bounds or stride of the innermost DO-loop. Because it is also explicitly stated that the DO-loops are perfectly nested, other changes to the value of the stride or the bounds of the innermost (and outermost) DO-loop during all iterations are not accounted for. If the outermost loop-control variable is used in the bounds of the innermost DO-loop, a so-called triangular loop can result. The transformation 'loop interchanging' on a particular triangular loop is shown below.

```
transform
  list( do(!e1, 1, !e2, 1, list( do(!e3, 1, !e1, 1, !s1), nil )), !s2 )
into
  list( do(!e3, 1, !e2, 1, list( do(!e1, !e3, !e2, 1, !s1), nil )), !s2 )
condition
  nodep flow   <> ($.dobody.dobody,$.dobody.dobody) and
  nodep anti   <> ($.dobody.dobody,$.dobody.dobody) and
  nodep output <> ($.dobody.dobody,$.dobody.dobody)
;
```

The last transformation in this section describes a transformation called 'loop collapsing'. This transformation can be applied on two nested loops with lower bound and stride equal to 1.

```
transform
  list( do(!e1, 1, !e2, 1,
     list( do(!e3, 1, !e4, 1, !s1), nil )), !s2 )
into
  list( do(itemp, 0, !e2 * !e4 - 1, 1,
    list( assign(!e1, div(itemp, !e4) + 1),
          list( assign(!e3, mod(itemp, !e4) + 1), !s1 ) )), !s2 )
```

```
condition
  true
;
```

Since this transformation is required if the target machine only allows simple DOALL-loops, it is better to express the `do` templates in terms of `doall` templates. This is because if the original transformation is applied first, the subscript expressions inside the body can no longer be converted into *normal form*, since they are not expressed in terms of a loop-control variable, possibly resulting in the assumption of too many data dependences and therefore prohibiting the conversion of the outermost DO-loop into a DOALL-loop. Note that in this case the introduction of `J = local(J)` must not be performed, since otherwise the patterns still do not match.

8 Transformation Application Phase

The transformation application phase is initiated with the command **start** from within the interactive environment, after which the following algorithm shown in a simplified version, will be executed.

```
forall transformations t do
  cs = give_program();
  while (cs ≠ NULL) do
    if match(t, cs) and cond_holds(t, cs) then
      show_matching_fragment(cs);
      new = compute_new_fragment(t);
      show_new_fragment(new);
      if accept_it() then
        replace(new, cs);
        cs = next_unmodifiedstmt(cs);
      else
        cs = next_stmt();
      end if
  end do
  if #transformations applied ≥ 1 then
    recompute_data_dependences();
  end if
end do
```

If this algorithm has been executed for all transformations defined and some of them have been applied the algorithm is restarted. The transformation application phase terminates otherwise.

The function *give_program*() returns a pointer to the first statement of the current program. This pointer is saved in the variable *cs* (current statement) which is shifted over the consecutive statements with the function *next_stmt*() until a matching fragment has been found. This shifting is done in such a way that all statements are pointed to once in lexical order, as is illustrated with the following program.

```
S1: R = 10
L1: DO I = 1, N, 1
  C1: IF (L2) S2: A(I) = R
  C2: IF (L3) THEN
    S3: A(I) = A(I) + 1
  ELSE
    S4: R = R + 1
  ENDIF
ENDDO
S5: R = R + 1
```

The pointer will point to S_1, L_1, C_1, S_2, C_2, S_3, S_4, and S_5 respectively.

The following sections discuss the implementation of the other functions. This implementation can be found in module *trafoapp.c*, listed in appendix S.

8.1 Determination of Next Matching Fragment

It is determined if the statement to which *cs* points is of the same kind as the first statement described in the pattern of t. If that is the case, the expressions and bodies of the statement in the program are matched against the corresponding patterns of t. If a pattern consists of a description of a statement list, consecutive statements in the program must match before a whole match can be concluded. Whenever a statement pointer variable occurs in a pattern, it is bound to the statement list currently matched against that pattern. Remember that this is only done once for every separate statement pointer variable, since it can only occur one time in the left-hand-side pattern of one transformation. The representation of the last statement that is used in the match is stored in a pointer *endmarker*. This pointer can be NULL [38] or point to a LINKUP data structure, if the last 'statement' used in the match is the end of a statement list (which matches on pattern `nil` or a statement-pointer variable). The 'rest of program' pointer is set to NULL in those two cases.

The expressions in the program are matched against the patterns that describe the expression in a template, by checking if the same operators and operands are used. Whenever an expression pointer variable occurs in a pattern, it is bound to the expression currently matched when it is the first occurrence of that variable. Otherwise a check is performed whether the expression currently matched is the same as the expression already bound to that variable.

The whole process is illustrated with the following example, in which the matching fragment with the given pattern is determined.

```
L1: DO I = 1 + Z, 100 + Z, 1
  S1: A(I) = 200
ENDDO
S3: R = 300
```

```
list( do(1 + !e1, !e2 + !e1, !e3, list( assign(!e4, !e5), nil) ), !s1 )
```

Suppose that *cs* points to L_1. Since a DO-loop matches on the kind of statement described first in the pattern, matching the expressions is initiated. Pattern `1 + !e1` matches on `1 + Z` with `!e1` bound to `Z`. After that `!e2` is bound to 100 and the binding of `!e1` is checked against `Z`. Finally, `!e3` is bound to 1. Subsequently the body of the DO-loop is matched on the corresponding pattern. This results in a match with `!e4` bound to `A(I)` and `!e5` to 200. Since here the DO-loop body ends, and the description is `nil`, the match still holds. Finally `!s1` is matched on S_3, after which the binding `!s1` (which is the 'rest of program') points to S_3. The algorithm concludes that a match has been found, and returns the bindings together with *endmarker*, which also points to S_3.

8.2 Evaluation of Conditions

All conditions in a condition list must hold, before a transformation may be applied. The separate conditions are evaluated as follows during the application

[38] This can only happen at the end of the program.

phase. Note that the **true** condition always succeeds. In case of the **nodep** condition two pointers are set on the statements in the matching fragment, according to the specified structures, by starting at *cs*, following the appropriate pointers (`n` → next field, `d` → DO-loop body field, and `i` → IF-body field). After that a function is called that determines if dependences of specified kind and with given data direction vector can be found in the dependence table with source statements in the statement list belonging to the first structure and sink statements in the one specified by the second structure. In doing so, care must be taken of the prefix of the data direction vector which is non-empty if the matching fragment appears inside a DO-loop and has as length the nesting depth of *cs* at that moment as is illustrated below. Assume that during the application phase, *cs* points to L_2, and the transformation 'loop concurrentization' is considered.

```
L1: DO I = 1, 100
  L2: DO J = 1, 100
    S1: A(I,J) = A(I + 1,J + 1)
   ENDDO
ENDDO
```

The DO-loop body of L_1 matches on the pattern (*cs* points to L_2), but self dependences of S_1 are described with two direction flags, since $minnest = 2$ during dependence analysis. The transformation for vectorization cannot anticipate all possible surrounding nestings. This is, fortunately, even unnecessary, since as long as the first DO-loop is executed serially cross iteration dependences cannot be violated by a legal transformation on its DO-loop body as can easily be seen. The only dependence in this loop: $S_1\delta_{<,<}S_1$ would prohibit vectorization if all dependences would have been taken into consideration, although vectorization of the `J` DO-loop is legal, since the flow dependence will not be violated due to data pipelining as long as the `I` loop is executed serially.

So only dependences with a prefix of nest depth of *cs* '=' directions must be considered in the condition evaluation. The comparison of the direction vector specified in the transformation is done starting at position nest depth of *cs* + 1.

This evaluation is demonstrated for the condition `nodep flow < (struct1,struct2)` in which the statement lists defined by `struct1` and `struct2` are { L_3 , S_4 } and { S_1 } respectively, and for which the nesting depth of *cs* is 2.

Dependences stored	Action	Comment
$S_4\delta_{<,=,<}S_1$	Ignore	'<' direction at first position
$S_4\delta_{=,=,<}S_3$	Ignore	$S_3 \notin \{ S_1 \}$
$L_1\overline{\delta}_{=,=,<}S_1$	Ignore	Not a flow dependence
$S_4\delta_{=,=,<}S_1$	Test Fails	Search terminates
.....		

If the optional specification of variables on which the dependences considered is given, all variable names that occur inside the expression bound to the given variable are extracted and used as a filter on the process described above

(i.e. the action Ignore is performed if the variable stored with the dependence is not an element of this list).

Only if the whole table can be scanned, without encountering a dependence that satisfies the given specification in the `nodep` (so action Ignore is performed on every entry), the associated transformation can be applied. Since this table is scanned in a linear fashion, condition evaluation can be quite expensive in case of many assumed dependences. Therefore, an optimization in evaluation time can be achieved by using a more sophisticated data structure to store data dependences.

Now it is clear, why extra functions were implemented to determine the directions for trailing '=' directions, since these dependences are crucial for the success or failure of the condition evaluation inside the innermost loops. Less effort is required for dependences with a trailing '<' direction, since they are only important for the outermost loop.

8.3 Presentation of Matching Fragment

If a matching fragment is found and all conditions hold, the matching fragment is presented to the user by listing all statements between *cs* and *endmarker* after the header `** MATCH ON`. The statements in the list pointed to by the statement pointer variable bound to the 'rest of program', are not listed. The string '`.....`' is shown instead if this list is non-empty to prevent large statement lists, not really involved in the match, from being listed. The functions implemented in module *show.c* can be used for this purpose (see section 9).

8.4 Computation of Resulting Fragment

The resulting fragment is specified in the right-hand-side of the current transformation. The computation of this fragment is done using the statement stack (discussed earlier) on which pointers to the data structures of new statements are pushed. After this these data structures can be linked together. Expressions that must appear in the new statements are also specified in the structures and after a new expression has been computed, a pointer to its data structure is pushed on the expression stack. In this fashion the expressions can be passed to the statement building routines. Since this method is the same as the one used during the initial creation of the program data structure, all the routines of module *struct.c* can be used.

Because new *normal form*s must be computed as explained below, the environment of the *cs* must be available before the computation of the new fragment start, and must be maintained during all different actions presented below. The actions that must be taken for every different element inside a right-hand-pattern are given in the following sections.

8.4.1 Templates

The statement templates directly specify which kind of statement must be created, so the only thing that must be done, is the creation of the expressions and a possible body specified in the template itself. After this the appropriate

routines can be used. Since the template `list` is only implicitly present in the data structure of templates (the templates have been linked together), it can be implemented by linking the resulting statements on the statement stack together. The pattern `if` always results in the creation of a general-IF statement, because the generation of a logical-IF statement would require examination of the body (this must be a single statement). Because inside a DO-loop body the environment changes, it must also be maintained during the creation of a DO-loop (this requires a check on the expression appearing as loop-control variable, see section 8.5). Remember that the creation of a DO-loop or IF statement automatically inserts a LINKUP data structure.

8.4.2 Statement Pointer Variables

Routines are called that copy all statements with their expressions found in the list pointed to. A pointer to the first statement in that list is pushed on the stack. This copying is done, in contrast with just pushing the pointer on the statement stack, because statement pointer variables can be used to duplicate statements. This requires different copies to be made, since later transformations can be applied differently on both version. Besides that, the nesting depth of statements inside that list may change, so the *normal form* of all subscript expressions must be recomputed using the current environment, requiring a pass over all statements and expressions anyway. Another reason is the fact that garbage collection can be done more easily since one of the two fragments (old one or resulting one) can be eliminated from memory, without the danger of overlapping data structures. All statements that result from this copying receive their own number, to distinguish them from the original statements.

An exception to this copying is whenever the 'rest of program' pointer appears at the end of a resulting pattern, which is the last pattern in the outermost `list` template or the second argument of the `merge` construction at that place. In this case it is only specified that the resulting fragment precedes the 'rest of program' fragment, which is a result of the fact that the transformation language operates on statement lists instead of single statements. Copying the 'rest of program' in that case would result in expensive and unnecessary duplication of usually long fragments. Therefore, it is decided not to copy that fragment but simply to push the 'rest of program' variable on the stack. All occurrences of the 'rest of program' pointer at other places, however, are handled as normal variables, i.e. copying is applied.

Since new DO-loops and IF statements are created, the LINKUPs inside these bodies are correctly added. It is, however, possible that a single LINKUP data structure appears at the end of the list that must be copied. This LINKUP is naturally not copied, but converted into a NULL pointer. How this LINKUP is recovered is explained in section 8.7.

Logical-IF statements are not converted into General-IF statements, because it is known that the body will consist of a single statement.

The data structure of the resulting fragment will be of the following form (in which 'rest of program' can also be NULL).

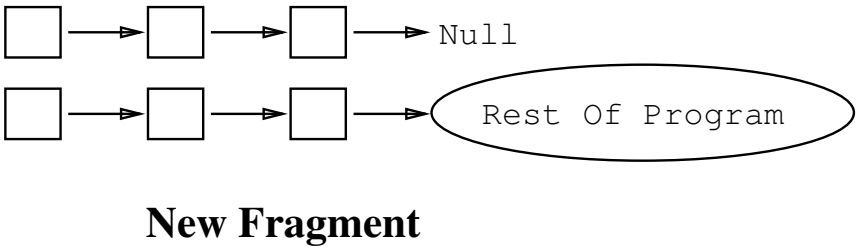


Figure 20: Resulting Fragment

8.4.3 Expressions

Constants and operators specified in a pattern can be created and pushed on the expression stack after evaluation of the operands in the last case. Every occurrence of an expression pointer variables requires the copying of the whole expression because the *normal form* of all subscript expressions of array variables must be recomputed, and simple garbage collection demands non-overlapping data structures.

8.4.4 Merge Operator

The statement lists specified in the arguments of this operator are created first, after which the tail of the first one is linked to the head of the second, and a pointer to the head of the first statement list is pushed on the stack. If the first argument specifies an empty statement list, only the second statement list is pushed on the statement stack. Remember that if the second argument of the **merge** operator is the 'rest of program' pointer, the copying mode must be disabled.

8.4.5 Vectorizing Operator

All occurrences of the scalar variable bound to the expression-pointer variable (which will be referred to as the vector variable) given as second argument of the **vectorize** operator, in subscript expression in the expression specified by the first argument, are replaced with vector triplets. All other expressions are unchanged. This may result in strange fragments as in the following example.

```
DO I = 1, 100
  A(I) = A(I) + I     >>>   A(1:100:1) = A(1:100:1) + I
ENDDO
```

This is, however, not considered to be a problem, since there is no syntactical convention to describe the vectorization of this loop, and the user can detect the incorrect syntax.

The lower bound, upper bound and stride that must be used in the resulting vector triplets are determined and evaluated. After that the values are type converted to INTEGER if necessary. Note that the lower bound and upper bound can be determined by evaluating the subscript expressions in which all

occurrences of the vector variable are replaced by the bounds found in the vector triplet specified in the operator. The determination of the stride requires multiplication of the sum of all coefficients of the vector variable inside the subscript expressions with the stride given in the triplet. The following example illustrates the **vectorize** operator: `vectorize( !e1 | !e2, 1:100:2 )`, with `!e1` bound to `A(2*I+I+1)` and `!e2` bound to `I`, results in `A(4:301:6)`. Some exceptions can occur during the evaluation of this operator. These are given in section 8.5.

8.4.6 Introduction Construction

The type of the identifier used is determined with the implicit data typing rules of FORTRAN 77. After that the symbol-table is checked if the identifier has already been defined. If this is the case the transformation can only be applied if the type and dimension found there are consistent with the type and dimension in this transformation. If the identifier has not been set, the symbol-table is updated with the new identifier. In case of an array variable, no bounds are stored to indicate the fact that it is unknown if a variable or intrinsic function is meant. Note that, if the transformation is not applied, all identifiers inserted in the symbol-table by this transformation must be deleted.

8.5 Checks Performed during Transformation Application Phase

Although few restrictions on the transformation language are desirable, some run time exceptions are necessary to prevent incorrect use of the transformation language. This means that the transformation is shown (as far as possible under the exception), but cannot be applied. A message is given to the user, to explain the cause of the exception. The transformation application phase continues after the exception, as if the application of this transformation was denied by the user.

Since no checks are performed on expression pointer variables, it is possible to introduce all kinds of expressions on the right-hand-side of assignment statements and DO-loops. This can result in incorrect programs since it is possible to create the assignment `3 = R`. All routines for data dependence analysis are implemented in such a way that this will not cause any problems (the *OUT* set is considered to be empty). Introducing a non-scalar variable as loop-control variable, however, cannot be tolerated, since this causes problems in the environment maintenance. Therefore, an exception is generated if this is done. During the evaluation of the **vectorize** operation, an exception is generated in one of the following cases.

1. The vector variable is a non-scalar variable.
2. The vector variable occurs inside a vector triplet.
3. The stride cannot be determined because the subscript expression is too complicated [39].

[39] In this case, the stride is set to 0.

4. The vector variable occurs in more than one subscript.

The second case prohibits vectorization because it is not always possible to vectorize inside a vector as is explained with the following nested DO-loop. In this case the `I` DO-loop cannot be vectorized, since the stride `S` is unknown.

```
DO I = 1, 100, S
  DO J = 1, 100                    DO I = 1, 100, S
    A(J+I) = <expr>       >>>        A(1+I:100+I:1) = <vecexpr>
  ENDDO                            ENDDO
ENDDO
```

In the last case vectorization would result in incorrect code as is illustrated below. The resulting code has quite different semantics than the original DO-loop.

```
DO I = 1, 100
  A(I,I) = 100          >>>      A(1:100:1,1:100:1) = 100
ENDDO
```

As stated before, an exception is also generated whenever a variable or function is introduced with inconsistent type or dimension.

8.6 Presentation of Resulting Fragment

The resulting fragment can be presented to the user in the same way as the matching fragment was shown, but now with as header '`** TRANSFORM INTO`' Again, a non-empty trailing 'rest of program' is shown as `.....`, although occurrences inside the resulting fragment are listed as a whole, since all statements in the statement list pointed to by the 'rest of program' variable have been copied and inserted in that case.

8.7 User Response Processing

After the matching and resulting fragment have been shown, the user is asked if this transformation must be applied. The following responses are possible (upper case characters are also allowed, all other inputs are seen as a 'n'-response).

- 'y' : Apply this transformation.
- 'n' : Do not apply this transformation and continue.
- 'q' : Do not apply this transformation and continue with the next transformation.
- 'e' : Do not apply this transformation and exit the transformation application phase.

Note that the processing of some of these responses are not reflected in the algorithm listed in section 8 to keep the presentation simple. If the transformation must not be applied, the recent computed fragment is deleted from memory,

and the transformation application phase continues as specified above. Since this fragment always ends with a NULL pointer or with the 'rest of program', the statements that must be deleted can be easily determined. The counters that administrate the current statement number of any kind are recovered and all symbols that have been inserted in the symbol-table by this transformation are deleted.

If the transformation must be applied, one of the following four insertions must be performed. For these insertions a pointer *prev* is required, which keeps track of the statement before *cs*. In case that *cs* is the first statement of a DO-loop or an IF body, *prev* points to the data structure of that DO-loop or IF statement. The *prev* pointer is maintained during the algorithm of section 8.

- **Program Insertion** If $prev = NULL$, the transformation is applied on the first statement of the program, so the pointer that keeps track of this statement must be set to the new fragment.
- **Normal Insertion** The new fragment must be inserted after *prev*, so the next field of the data structure pointed to by *prev* is set on that fragment. If the new fragment continues in the 'rest of program', the possible following LINKUP will be presented in the resulting program. So no LINKUP recovery is necessary. This kind of insertion is shown in the following picture.

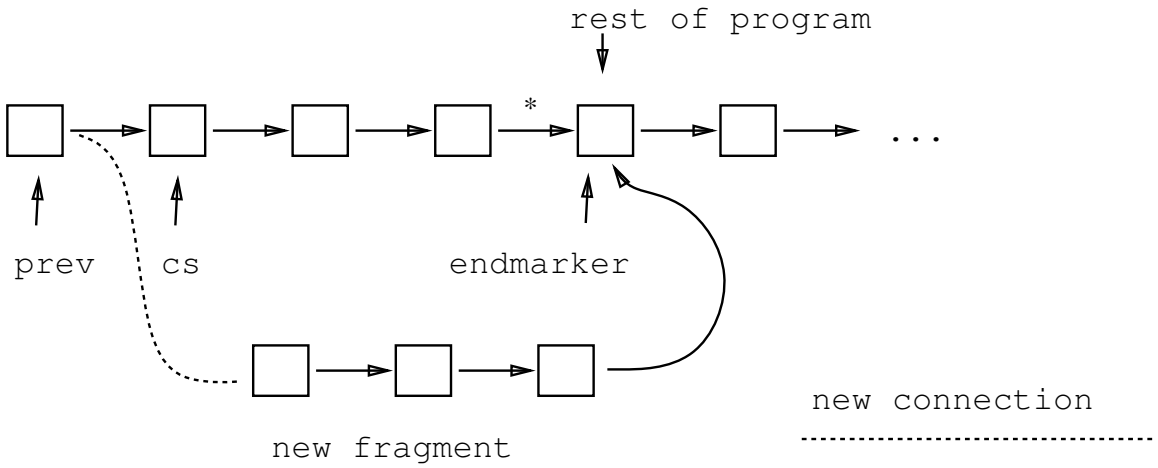


Figure 21: Program Insertion with LINKUP Recovery

If, however, the new fragment does not use 'rest of program' or this pointer is NULL because *endmarker* points to a LINKUP, a possibly following LINKUP will be detached from its corresponding DO-loop or IF statement. Therefore, the *old* fragment is scanned, starting at *endmarker*, un-

til a LINKUP data structure is encountered (or 'rest of program'/NULL [40] which indicates that a 'normal insertion' without recovery can be done). The body of the statement pointed to by this LINKUP will contain the new fragment. So a LINKUP data structure can be added to recover the total data structure. This process is illustrated in the next picture.

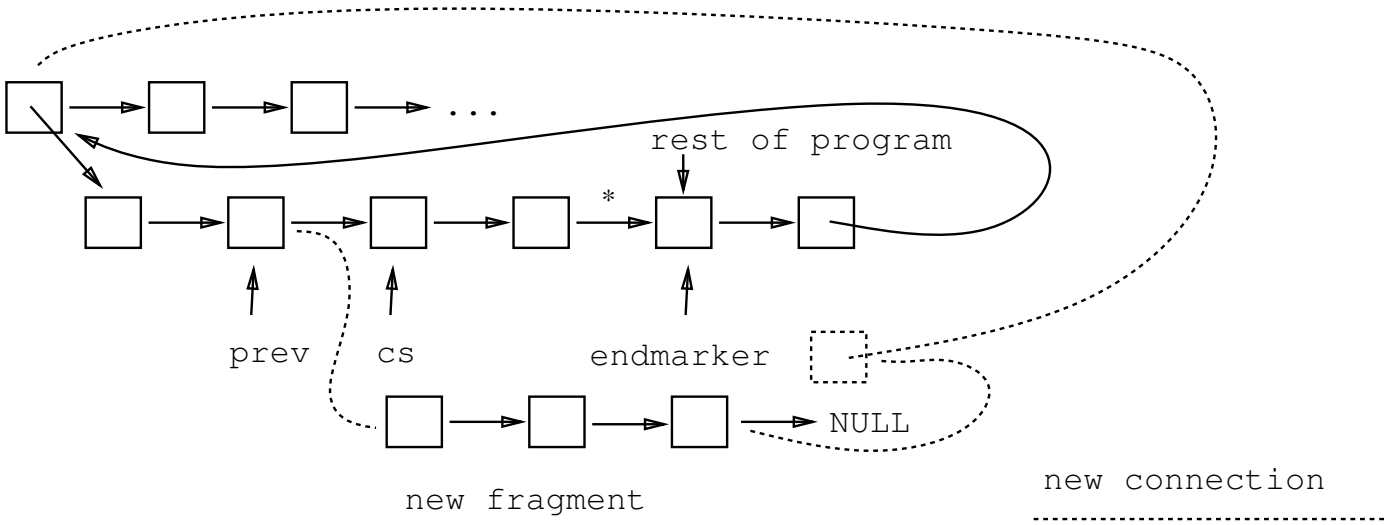


Figure 22: Program Insertion

- **DO-loop body Insertion** The DO-loop body field of the data structure pointed to by *prev* is set to the new fragment. Since again a LINKUP data structure can be detached if the new fragment does not use a non-empty 'rest of program', the same actions as given above are performed. This kind of insertion is illustrated in the following picture. Note that since *endmarker* points to a LINKUP, the 'rest of program' is NULL. This is because from within the body no following statements are seen. Although this 'rest of program' is used in the new fragment, it would result in the loss of a LINKUP data structure without the actions given.

- **IF-body Insertion** The IF body field of the data structure pointed to by *prev* is set to the new fragment. The old fragment is also considered starting at *endmarker* to recover from a possibly lost LINKUP. The case that this recovery is not necessary is shown below. If the insertion is done in a Logical-IF statement, it is converted into a General-IF statement, because the number of statements is the transformed body may change.

After the insertion has been done, *cs* is set to the first statement that has not been considered yet, and for which the old data dependences hold (remember that new data dependences are only computed after every pass). This statement

[40] This can happen at the end of the program.

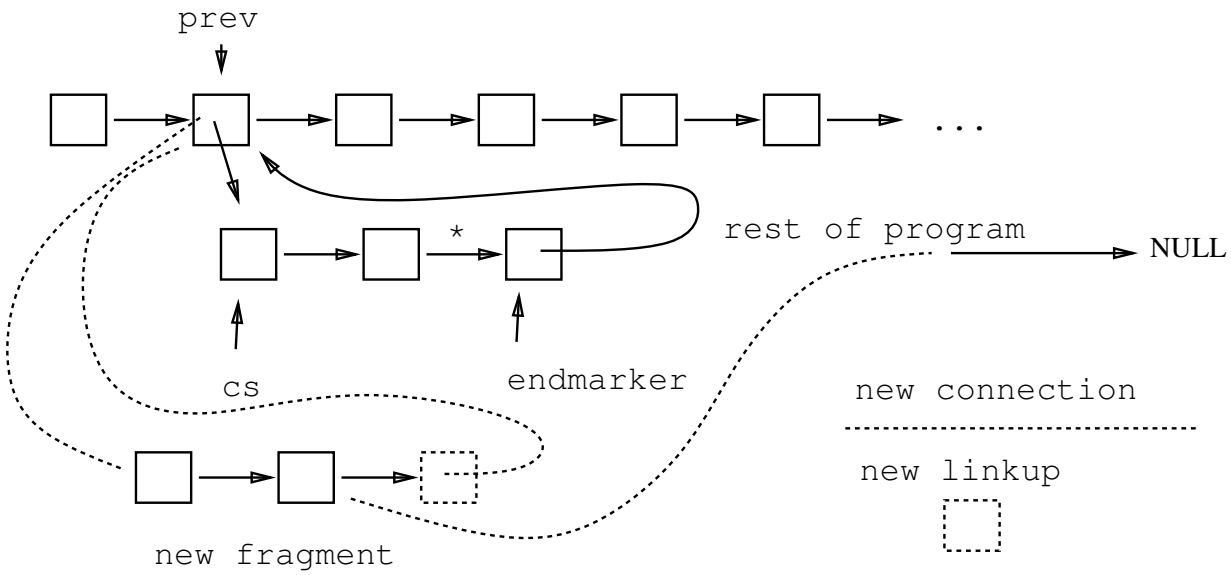


Figure 23: DO-loop body Insertion

can be found using the process described above, that starts scanning from the *endmarker* pointer. If this scanning ends in a LINKUP data structure, *prev* is set to the DO-loop or IF statement pointed to by this LINKUP and *cs* is set to the following statement, since the exploration of the body has finished. If this scanning ends in a NULL pointer, the end of the program has been reached. Otherwise the scanning ends in a non-empty 'rest of program', after which *prev* is scanned forwards until the following statement is 'rest of program' in case *cs* was at the same level as *prev*. After that, *cs* is set to 'rest of program'. If *prev* points to a DO-loop or IF-statement from which the body contained *cs*, it is determined if 'rest of program' is at the beginning of the body. If this is the case *prev* is unaltered. Otherwise the body is scanned to find the statement before 'rest of program', to which *prev* is set. In both cases, *cs* is set on 'rest of program'. If *prev* was NULL, the first statement of the program has been changed, and the statement before 'rest of program' must be searched. If the program begins with 'rest of program', *prev* remains NULL.

Finally the statements in the old fragment must be deleted from memory. This can be done by removing the pointer that points to *endmarker* (marked with a '$*$' in the pictures) and deleting the old statement list. The algorithm continues as explained above after these adjustments.

As a last remark it must be stated that if the left-hand-side pattern of a transformation only consists of the template `nil`, this will match on all empty statement lists in bodies of DO-loops and IF statements, except the empty list at the end of the program, since the algorithm terminates whenever $cs = NULL$. Because all transformations of this form are very artificial, this is not considered

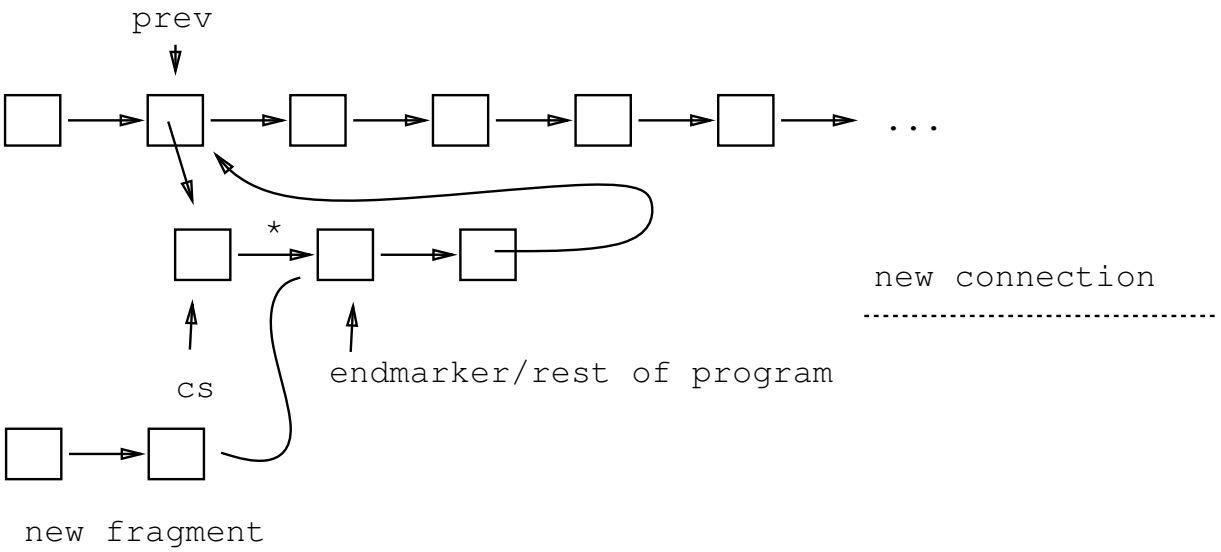


Figure 24: IF-body Insertion

to be a problem.

8.8 Adaptations to other Modules

The routines that compute data dependences must be adapted, since the resulting code can contain DOALL-loops, vector instructions, and new variables/functions. The DOALL-loops are handled as normal serial DO-loops. i.e. the cross-iteration dependences are generated as if the loop was executed serially. The bounds and stride of vector triplets that are used as subscript expressions in array variables are not used in the dependence computation. So it is assumed that all elements are used, as is illustrated below.

```
L1: DO I = 1, 100
  S1: A(I) = 100                S3: A(1:100:1) = 100
ENDDO                   >>>
L2:  DO I = 101, 200
  S2: A(I) = 200                S4: A(101:200:1) = 200
ENDDO
```

Before the two DO-loops are converted into vector instructions no dependence is assumed, but in the resulting fragment the dependence $S_3 \delta^O S_4$ is assumed. Therefore, it is recommended to apply vectorizing transformations after all other transformations.

The functions used to generate the symbol-table and the program listing must be adapted to handle intrinsic functions or array variables of which the bounds are unknown. Note that if transformations are applied, new files for the

symbol-table and program listing must be generated to keep these consistent with the current program in memory.

8.9 Example of Interactive Restructuring

This section shows some simple sessions to illustrate the interactive way in which restructuring is applied. Consider the following program with four loops and the definitions for the transformations for 'concurrentization' and 'vectorization', given in the previous section.

```
      L1: DO I = 1, 100, 1
        S1: A( <I> ) = (A( <I> ) + 1)
      ENDDO
      L2: DO I = 2, 100, 1
        S2: A( <I> ) = (A( <I-1> ) + 1)
      ENDDO
      L3: DO I = 1, 99, 1
        S3: A( <I> ) = (A( <I+1> ) + 1)
      ENDDO
      L4: DO I = 1, 50, 1
        S4: A( <I> ) = ((A( <2*I> ) + A( <2*I> )) + 1)
      ENDDO
```

In the following session all vectorizing transformations are chosen. Since the dependence $S_1 \delta_< S_1$ holds in the second DO-loop, this loop cannot be vectorized. Only the first DO-loop matches the transformation into a DOALL loop, because the only dependence is $S_1 \overline{\delta}_= S_2$.

```
=> start
--------- Transformation 1 -----------
** MATCH ON
      L1: DO I = 1, 100, 1
        S1: A(I) = (A(I) + 1)
      ENDDO
      .....
** TRANSFORM INTO
      L5: DOALL I = 1, 100, 1
        S5: A(I) = (A(I) + 1)
      ENDDOALL
      ......
** ACCEPT (y/n/q/e) ===> n
```

The transformation phase starts with transformation 1, which is 'loop concurrentization'. The first loop (L_1) can be converted into a DOALL-loop. A new DOALL-loop is created (L_5) and the body of the old loop is copied (resulting in assignment S_5, identical to S_1). The 'rest of program' variable points to L_2, but it is shown as dots. The transformation is not applied, since 'n' is given as response.

```
--------- Transformation 2 -----------
** MATCH ON
      L1: DO I = 1, 100, 1
        S1: A(I) = (A(I) + 1)
      ENDDO
      .....
** TRANSFORM INTO
      S5: A(1:100:1) = (A(1:100:1) + 1)
      ......
** ACCEPT (y/n/q/e) ===> y
```

No other loops can be converted into DOALL-loops, so the transformation phase continues with the following transformation 'loop vectorization' and starts at the begin of the program again. Since S_5 of the previous transformation has been deleted from memory, its number can be used again.

```
** MATCH ON
      L3: DO I = 1, 99, 1
        S3: A(I) = (A((I + 1)) + 1)
      ENDDO
      .....
** TRANSFORM INTO
      S6: A(1:99:1) = (A(2:100:1) + 1)
      ......
** ACCEPT (y/n/q/e) ===> y
** MATCH ON
      L4: DO I = 1, 50, 1
        S4: A(I) = ((A((I + I)) + A((2 * I))) + 1)
      ENDDO
      .....
** TRANSFORM INTO
      S7: A(1:50:1) = ((A(2:100:2) + A(2:100:2)) + 1)
      ......
** ACCEPT (y/n/q/e) ===> y

Computing Data Dependences

******** New Pass ********

--------- Transformation 1 -----------
--------- Transformation 2 -----------

Number of transformations applied: 3
```

After all loops have been vectorized, new data dependences are computed and the algorithm starts again with transformation 1. Since there are no more matching fragments, the algorithm terminates and reports the number of transformations applied.

In the following session, 'loop collapsing' and 'strip mining' are illustrated on a nested DO-loop.

```
--------- Transformation 1 -----------
** MATCH ON
      L1: DO I = 1, 100, 1
        L2: DO J = 1, 100, 1
          S1: A(I,J) = 10.000000
        ENDDO
      ENDDO
      .....
** TRANSFORM INTO
      L3: DO itemp = 0, ((100 * 100) - 1), 1
        S2: I = (div(itemp,100) + 1)
        S3: J = (mod(itemp,100) + 1)
        S4: A(I,J) = 10.000000
      ENDDO
      ......
** ACCEPT (y/n/q/e) ===> n
--------- Transformation 2 -----------
** MATCH ON
      L1: DO I = 1, 100, 1
        L2: DO J = 1, 100, 1
          S1: A(I,J) = 10.000000
        ENDDO
      ENDDO
      .....
** TRANSFORM INTO
      L5: DO itemp = 1, 100, 32
        L4: DO I = itemp, min2((itemp + 31),100), 1
          L3: DO J = 1, 100, 1
            S2: A(I,J) = 10.000000
          ENDDO
        ENDDO
      ENDDO
      ......
** ACCEPT (y/n/q/e) ===> e
```

As a last example, a session is shown in which a reduction operator is generated from a DO-loop. Parts of the symbol table and the resulting program are also listed to demonstrate the way in which the intrinsic function is represented.

```
=> start
--------- Transformation 1 -----------
** MATCH ON
      S1: PT = 0.000000
      L1: DO I = 1, 100, 1
        S2: PT = (PT + (A(I) * B(I)))
```

```
      ENDDO
      .....
** TRANSFORM INTO
      S3: PT = dot(A(1:100:1),B(1:100:1))
      ......
** ACCEPT (y/n/q/e) ===> y

...
      REAL    A(1:100)
      REAL    B(1:100)
      REAL    PT
      REAL    dot()

      S3: PT = dot(A(1:100:1),B(1:100:1))
...

Id      Dim             Type            Value          Bounds

...
A       1               REAL            -               ( 1:100 )
B       1               REAL            -               ( 1:100 )
PT      Scalar          REAL            -
dot     2               REAL            -               ( )
```

8.10 Some Results on FORTRAN 77 Programs

The prototype compiler is tested on two FORTRAN 77 programs, using the transformations given in the previous sections.

For a program that performs block LU factorization, found in [Bik91], the compiler can be compared with commercial compilers such as KAP and VAST. All DOTPRODUCT operations are recognized and the same DO-loops are converted into DOALL-loops or vector instructions. Some additional transformations were applied by KAP and VAST using the `WHERE` construction not yet accounted in the transformations specified. The collapsing of nested DOALL-loops which can be done by the prototype compiler, however, was not performed by these compilers. The prototype compiler applied 'loop interchanging' in more cases than VAST did, and performed comparable with KAP.

Of course, these simple programs cannot be considered as serious benchmarks. However, some of the features of the different compilers can be compared. A more elaborate study is planned to investigate the effectiveness of the different approaches.

9 Unparsing

This section discusses the unparsing phase, implemented in module *show.c*, which is listed in appendix Q. This module can be used, whenever the user wants to examine the program stored in memory. So after the program has been read or the transformation phase is terminated, the routines of this module are used to enable the user to examine the results so far. The user can write the program to a file in a format that can be presented to a compiler (so without the statement labels and subscript expressions in *normal form*) using the command **save**. The routines in this module are also used to show the matching and resulting program fragments on the terminal, during the transformation phase.

Since most of the resulting code is FORTRAN 90, the syntax of this language is used. But some FORTRAN 77 constructions are used (e.g. continuations), which is possible, since FORTRAN 90 is downwards compatible. The indentation of all statements is initially 6 spaces, and is increased inside bodies to improve readability. If text appears after the 64^{th} column, the text is cut off, and a continuation is used, to prevent the text from exceeding the 72^{th} column.

9.1 Program Header

First the keyword `PROGRAM` followed by the identifier found in the original program is listed, followed by a newline, and a blank line.

9.2 Declaration Statements

The information found in the symbol table of every variable is converted into a declaration statement and listed on separate lines. Array declarations are presented using subscript ranges with explicit lower bound. So, the declaration of an array `REAL A(100)` is converted into `REAL A(1:100)`. Parameter variables are not listed in the resulting program, because **constant folding** has eliminated all the occurrences of these variables. This affects the flexibility of the resulting FORTRAN 90 code, so adaptations to the program can be more easily done in the original program. The declaration statements are separated from the rest of the statements by a blank line.

9.3 Statements and expressions

The unparsing rules for the different statements and expressions are discussed in the following sections.

9.3.1 DO-loop/DOALL-loop statements

These statements are listed using the keywords `DO(ALL)` and `ENDDO(ALL)`, eliminating the need for labels. The statements inside the body of every loop are indented two spaces. Depending on whether a listing for the user or for the compiler is generated, the loop label can be listed.

9.3.2 Assignment statements

Assignment statements are presented on a single line possibly preceded by the assignment label.

9.3.3 Logical-IF statements

A logical-IF statement is listed on a single line possibly with its condition label, followed by the statement in its body. If this statement is an assignment statement, the label of this statement is also listed if necessary, as is shown below.

```
C1: IF (L1) S1: R = R + 3
```

9.3.4 General-IF statements

General-IF statements start with the keywords `IF (<condition>) THEN` possibly preceded by the condition label, in which the condition is listed using the unparsing rules of section 9.3.6. The statements inside the branches of a general-IF statements are indented two spaces. All the following `ELSE IF`s and the last `ELSE` are listed in the same column as the opening `IF` keyword, with their associated condition labels if required. Finally the keyword `ENDIF` is listed.

9.3.5 STOP statement

This statement is simply listed on a single line.

9.3.6 Expressions

Expressions are listed according to the following rules.

- Operators: Arithmetic operators are listed with one space at each side to improve readability of arithmetic expressions. All other operators are listed without spaces, since the two dots separates the operator from its operands (as in `3.NE.2`). All expressions with operators are annotated with brackets to reflect the priority. So, the following expression `R = R + 10 * R` is listed as `R = (R + (10 * R))`. Even the single operand of a unary operator is surrounded with brackets.

- Constants: Constants can simply be listed. Logical values are shown as .TRUE. or .FALSE..

- Variables: The name of the variable is listed. The subscript expressions of array variables are shown in their *normal form*, with enclosing $< ... >$ notation, if a listing for the user is generated. The original subscript expressions are used otherwise.

- Vector Triplets: These are listed by showing the lower bound, upper bound, and stride expressions, separated by a ':'.

9.3.7 End of Program

At the end of a program a blank line is generated, followed by the keyword `END`.

9.4 Example of unparsing

The following program is entered into the restructuring compiler, and shown below once it is read. Note that the prototype compiler acts as a 'pretty printer', since the layout of the original program has been discarded.

```
      PROGRAM SHOW
      INTEGER I, N
      PARAMETER (N = 100)
      LOGICAL L
      REAL R, A(N)
      R = 5.0
      L = (3 .NE.    (6 + 2))
      DO 10 I = 1, N, 1
      A(I) = 10.0 + R * 5.0
10    CONTINUE
      IF (L) R = R + 2
      IF (L) THEN
      R =  2.0
      ELSE IF (L .NEQV.
     +     .TRUE. ) THEN
      R = 3.0
      ENDIF
      STOP
      END
```

The array declaration of `A` has been changed. The statements are annotated with their labels, the expressions with brackets, and the subscript expression inside DO-loop 10 is shown in *normal form*. Parameter `N` has been eliminated.

```
      PROGRAM SHOW

      INTEGER I
      LOGICAL L
      REAL    R
      REAL    A(1:100)

      S1: R = 5.000000
      S2: L = .TRUE.
      L1: DO I = 1, 100
        S3: A( < I > ) = (10.000000 + (R * 5.000000))
      ENDDO
      C1: IF (L) S4: R = (R + 2)
      C2: IF (L) THEN
        S5: R = 2.000000
```

```
C3: ELSE IF ((L.NEQV..TRUE.)) THEN
  S6: R = 3.000000
END IF
STOP

END
```

10 Future Research

During the implementation of this prototype compiler some topics were left unexamined, and should be included in future research. This section enumerates some of those topics.

- The user-defined transformations are examined sequentially. It is still an open question in which order the transformations must be applied to achieve the best possible code. If such an order can be found, another question is how this must be specified in the transformation language. One possible way could be the use of a finite state machine.

- The transformation language has some inaccessible macros (**merge** and **vectorize**). The language can be made more powerful by enabling the user to define its own macros. One possible way to do this could be the extension of the language with user defined functions.

- The expression-pointer variables can only be bound to whole expressions. In some case it could be convenient if also access to the subscript expressions would be possible.

- The prototype compiler can only be used on a subset of FORTRAN 77. An extension to whole FORTRAN 77 would be a nice improvement, but requires more powerful techniques such as interprocedural analysis.

- The transformation language can only define program transformations, with a few side-effects on the data structure (introduction of new variables). Research on data structure transformation will result in the need for a more powerful transformation language.

- The conditions of every transformation can only enforce that certain dependences do not exist. Conditions on the value of variables, tests on variables if they are constants or variables, tests on the number of underlying dependences or global conditions are necessary to make the language more powerful.

- Better techniques for data dependence analysis can result in better exploitation of potential parallelism.

- More interaction with the user can result in better code. An idea is to use a new kind of variables for expressions (e.g. `?e11`) which ask the user to enter a certain expression [41]. The user can decide to enter the best value for this expression, eliminating the need to anticipate all possible values in the transformation definitions.

[41] In strip mining, for example, the best vector length can depend on different factors not accounted for in the transformation.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Ban88] Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer, Boston, 1988.

[BGA90] W.S. Brainerd, Ch.H. Goldberg, and J.C. Adams. *FORTRAN 90*. Academic Service, 1990.

[Bik91] Aart J.C. Bik. Program restructuring. INF/VER-91-19, 1991.

[ETW92] C. Eisenbeis, O. Temam, and H. Wijshoff. On efficiently characterizing solutions of linear diophantine equations and its application to data dependence analysis. In *Proceedings of the Seventh International Symposium on Computer and Information Sciences*, 1992.

[KF90] Koffman and Friedman. *Problem Solving and Structured Programming in FORTRAN 77*. Temple University, 1990.

[Pol88] Constantine D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer, Boston, 1988.

[Wol89] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman, London, 1989.

[ZC90] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, New York, 1990.

A The Interactive Environment

This appendix contains all commands that can be given from within the interactive environment, with their meaning and an explanation of all possible system responses. When the environment is entered from UNIX with the command **f2f**, the following message is generated before the prompt '`=>`' is shown.

```
*****************************************************
*** FORTRAN 77 to FORTRAN 90 Restructuring Compiler ***
*** Written by Aart J.C. Bik                       ***
*** under supervision of dr. H.A.G. Wijshoff       ***
*** UNIVERSITY OF UTRECHT 1991/1992                ***
*****************************************************
```

- **dep** This command switches between the different modes of dependence generation. The selected mode is listed.
- **exit** This command quits the interactive environment. A bye message is listed.
- **help** This command generates a listing of all possible commands.
- **nowarnings** This command disables the generation of warning messages during semantic checking of FORTRAN 77 programs. The message `Warnings disabled` is listed.
- **readprg** $< filename >$ This command reads the FORTRAN 77 file found in the argument given. If an incorrect argument is given, the message `*** Bad file name` *filename* is listed. If no program is read due to an error, the message `Terminated` is generated. All other error and warning messages generated during the different phases are listed in the appendices B and C.
- **readtrf** $< filename >$ This command reads the transformation file found in the argument given. If an incorrect argument is given, the message `*** Bad file name` *filename* is listed. If no transformations are read due to an error, the message `Terminated` is generated. Other errors generated during the semantic checking of a transformation file are listed in appendix D.
- **save** $< filename >$ This command writes the current [42] program to a file with as name the argument given. If no program is present, the message `No program to save` is listed.
- **showdep** This commands shows all static dependences found in the current program and saved in the file *program.dep* using the UNIX command `less`. The message `No Dependences` is listed if no program is present, or the current program is dependence free.

[42] When a program is correctly stored in memory, it is said to be the current program.

- **showprg** This program shows the current program which can be found in the file *program.txt* using the UNIX command `less`. The message `No Program in memory` is listed otherwise.
- **start** This command starts the transformation application phase. If no current program is present, the message `No program present to transform` and if no transformations are present, the message `No transformations in memory` is listed.
- **symtb** This program lists the symbol table of the current program using the UNIX command `less`, which can be found in the file *program.sym* if one is generated. The message `Empty` is generated otherwise.
- **vi** < *filename* > This command calls the `vi` editor of UNIX with the argument given.
- **warnings** This command enables the generation of warning messages during semantic checking of FORTRAN 77 programs. The message `Warnings enabled` is listed.

B Error Messages (F77)

In this appendix all the possible error messages of the FORTRAN 77 dialect parser are listed, together with a short explanation of their cause. Only syntax errors generated by the YACC-generated program will result in immediate termination of the parsing phase. All others errors are reported but parsing resumes afterwards, so if more errors occur in the program, they can be reported in the same run. The number of the line in which the error is detected is listed and for some errors the name of the variable involved is also given (listed below as <varname>). Naturally, the generation of errors cannot be disabled. Errors always result in the eventual removal of the program from memory.

- **Array variable used as Parameter → <varname>** An identifier of an array variable is used as scalar variable in a parameter definition. Note that the context free grammar of the FORTRAN dialect prohibits the use of subscripted parameter variables in parameter statements.
- **Assignment to Loop Variable → <varname>** A variable that is currently used as loop-control variable is being assigned.
- **Assignment to Parameter → <varname>** A variable that has been used in a parameter definition is being assigned.
- **CONTINUE unexpected** A CONTINUE statement is used outside a loop body (note that CONTINUE statements have a different function in this dialect).
- **Condition is not of type LOGICAL** An expression that is not of type LOGICAL is used as a conditional expression.

- **Division by zero** The right operand of a division operator can be evaluated at compile-time and is zero.
- **ELSE(IF) after ELSE** An ELSE or ELSE IF statement is used after an ELSE statement.
- **ELSE(IF) unexpected** The keywords `ELSE` or `ELSE IF` are used outside the body of a general IF statement.
- **Incorrect CONTINUE label** The label associated with a CONTINUE is not the same as the label in the DO-loop header.
- **Label has already been set** The label in the label field has been used before.
- **Lowerbound is greater than upperbound** The lower bound expression of a subscript range in an array declaration is greater than the upper bound.
- **More than seven dimensions → <varname>** The dimension of the declared array variable exceeds 7.
- **Parameter as loop variable → <varname>** A variable defined in a parameter statement is used as loop-control variable.
- **REAL operand in logical operator** An expression of type REAL is used as operand of a logical operator.
- **Same loop variable in DO-loop → <varname>** A variable currently used as loop-control variable is used again as loop-control variable.
- **Variable redeclared → <varname>** A variable is redeclared with a different type or dimension.
- **Wrong Dimension → <varname>** A variable is used with a different dimension than given in its declaration.

C Warning Messages (F77)

This appendix shows all possible warning messages with their cause. The generation of warnings can be disabled using the command **nowarnings**, and enabled with the command **warnings**. The number of the line in which the error is detected is also given, sometimes accompanied with the name of a variable involved (listed below as <varname>). Naturally, warnings do not prevent the program from correctly being stored in memory.

- **Different types in DO-loop** The types of the loop-control variable and expressions in a DO-loop are not identical.
- **DO-loop will not be executed** The bounds and stride of a DO-loop can be determined at compile-time, and have such values, that the loop will never be executed at run-time.

- **Duplicate declaration → <varname>** A variable is declared again but because the declaration is compatible (i.e. type and dimension are the same) no error is generated. In case of an array declaration, the subscript ranges may change, however.
- **IF (.TRUE.) ignored** The conditional expression can be evaluated at compile-time and has the value `.TRUE.`.
- **INTEGER operand in logical operator** An expression of type INTEGER is used as operand of a logical operator.
- **LOGICAL operand in arithmetic operator** An expression of type LOGICAL is used as operand of an arithmetic operator.
- **LOGICAL operand in relational operator** An expression of type LOGICAL is used as operand of a relational operator.
- **Lowerbound cannot be computed** The value of a lowerbound expression of a subscript range used in an array declaration cannot be determined at compile-time.
- **Other type in assignment to → <varname>** The right hand side expression of an assignment statement does not have the same type as the left hand side variable.
- **Other type in value of Parameter → <varname>** The expression in a parameter statement does not have the same type as the parameter variable.
- **Parameter not declared → <varname>** The parameter variable in a parameter statement has not been declared. Implicit data typing is used to determine its type.
- **Parameter redefined → <varname>** A parameter variable is defined again in a parameter statement. The value in this parameter statement will be associated with the parameter statement afterwards.
- **Parameter value cannot be computed** The value of an expression in a parameter statement cannot be determined at compile-time.
- **Statement after STOP cannot be reached** A statement appears after a STOP statement. If this is done inside a DO-loop or general-IF body, the line number in the warning message is the last line of the body.
- **Subscripts are not of type INTEGER** A subscript expression used in an array reference is not of type INTEGER.
- **Subscripts are not of type INTEGER in declaration** A subscript expression used in an array declaration is not of type INTEGER.
- **Subscript out of bound** The value of a subscript expression used in an array reference is not within the allowed range.

- **Upper bound cannot be computed** The value of an upper bound expression of a subscript range used in an array declaration cannot be determined at compile-time.
- **Variable not declared → <varname>** A variable has not been declared. Implicit data typing is used to determine its type.

D Error Messages (Transformation Language)

This appendix lists all error messages that can occur during the semantic checking of the transformation language, and during the transformation application phase. The number of the line in which the error is detected is listed. The name of a variable involved in an error is also given.

D.1 Semantic Errors

The following errors can be generated during the semantic checking of a transformation file.

- **First structure in condition does not match pattern** The first structure in the condition points to a kind of statement that is not (necessarily) present during the transformation application phase.
- **Incorrect number of directions** The data dependence vector that is given in a condition has an incorrect number of directions. The number must be equal to the *common nest* of the statement list indicated by the given structures.
- **Intrinsic function/array variable cannot be used at left hand side** The introduction construction is used in the left-hand-side pattern of a transformation.
- **Merge cannot be used at left hand side** The **merge** operator is used in the left-hand-side pattern of a transformation.
- **Scalar variable cannot be used at left hand side** The introduction construction of a scalar variable is used in the left-hand-side pattern of a transformation.
- **Second structure in condition does not match pattern** The second structure in the condition points to a kind of statement that is not (necessarily) present during the transformation application.
- **Variable e*num* is not defined before use** Expression-pointer variable `!e`*num* is used in the right-hand-side pattern of a transformation, but is does not occur in the corresponding left-hand-side pattern.
- **Variable s*num* is not defined before use** Statement-pointer variable `!s`*num* is used in the right-hand-side pattern of a transformation, but is does not occur in the corresponding left-hand-side pattern.

- **Variable s*num* is used more than once** Statement-pointer variable `!s`*num* occurs more than once in a left-hand-side pattern. Since the enforcing of equivalence of statement lists cannot be done with these variables, an error is generated.
- **Vectorize cannot be used at left hand side** The **vectorize** operator is used in a left-hand-side pattern of a transformation.
- **Vectorize cannot be used inside another vectorize** The **vectorize** operator is used inside another vectorize (this constraint results from an implementation issue and can be avoided by multiple application of this operator).

D.2 Application Errors

The following errors will generate an exception.

- **Introducing non scalar as loop-control variable** A DO-loop must be created with a non-scalar as loop-control variable.
- **Stride cannot be determined** The stride in the vector triplet cannot be determined, because a subscript expression is too complicated.
- **Unconsistent use of variable** A variable is introduced that is already present in the symbol-table with inconsistent type of dimension.
- **Vector variable is a non scalar** The expression-pointer variable used as second argument of the **vectorize** operator is not bound to a scalar variable.
- **Vector variable occurs in other vector** The vector variable occurs in another vector triplet, prohibiting correct vectorization of that expression.
- **Vector variable occurs more than once** A vector variable occurs in more than one subscript expression of one variable prohibiting correct vectorization.

E File Organization

The C program of the prototype compiler is defined in separate files in order to be able to apply the technique of separate compilation. Therefore, it is necessary to define the dependences between the different modules to avoid redundant compilation and to keep the resulting executable up to date without placing that burden on the user. The dependences and commands required to compile the programs are defined in a so-called $makefile$. This appendix shows the $makefile$ used, together with a graphical representation of this makefile. Note that some of the dependences and commands are already known to the system (e.g. $file.c \rightarrow file.o$ by **cc -O -c**), so they do not appear in the $makefile$. The command **make** will initiate the compilation process.

The substitution of all yy and YY in f_yy and F_YY, respectively, in the generation of the FORTRAN scanner and parser with the LEX and YACC tool is necessary to prevent duplicate names in the resulting sources, since these tools are also used to generate the scanner and parser for the transformation language.

```
YFLAGS = -d
CFILES = parser.o scanner.o f77parser.o fsup.o symbol.o \
 mem.o show.o struct.o dep.o deptb.o env.o \
 trafo.o trafoapp.o
LINTS = f77parser.ln fsup.ln symbol.ln mem.ln \
show.ln struct.ln dep.ln deptb.ln env.ln trafo.ln trafoapp.ln

f2f:        $(CFILES)
            cc $(CFILES) -o f2f -lm

scanner.o:   y.tab.h

f77parser.o: f77parser.c inter.c prgtype.h
             cc -O -c f77parser.c

f77parser.c: fparser.y
             yacc fparser.y
             sed 's/yy/f_yy/g' <y.tab.c|sed 's/YY/FYY/g' \
             |sed 's/stacks/yystacks/g' >f77parser.c
             rm y.tab.c

inter.c:     fscanner.l
             lex fscanner.l
             sed 's/yy/f_yy/g' <lex.yy.c|sed 's/YY/FYY/g' >inter.c
             rm lex.yy.c

fsup.c:      prgtype.h
mem.o:       trafo.h prgtype.h
show.o:      prgtype.h
struct.o:    prgtype.h
symbol.o:    prgtype.h
dep.o:       prgtype.h
deptb.o:     prgtype.h
parser.o:    trafo.h
trafoapp.o:  trafo.h prgtype.h
trafo.o:     trafo.h prgtype.h
```

The following command sequence is executed whenever all the files are updated.

```
yacc -d parser.y
mv y.tab.c parser.c
cc -O -c parser.c
```

```
lex  scanner.l
cc -O -c lex.yy.c
rm lex.yy.c
mv lex.yy.o scanner.o
yacc fparser.y
sed 's/yy/f_yy/g' <y.tab.c|sed 's/YY/FYY/g' \
    |sed 's/stacks/yystacks/g' >f77parser.c
rm y.tab.c
lex fscanner.l
sed 's/yy/f_yy/g' <lex.yy.c|sed 's/YY/FYY/g' >inter.c
rm lex.yy.c
cc -O -c f77parser.c
cc -O -c fsup.c
cc -O -c symbol.c
cc -O -c mem.c
cc -O -c show.c
cc -O -c struct.c
cc -O -c dep.c
cc -O -c deptb.c
cc -O -c env.c
cc -O -c trafo.c
cc -O -c trafoapp.c
cc parser.o scanner.o f77parser.o fsup.o symbol.o  mem.o show.o
   struct.o dep.o deptb.o env.o  trafo.o trafoapp.o   -o f2f -lm
```

The following picture illustrates the dependences defined in the *makefile*.

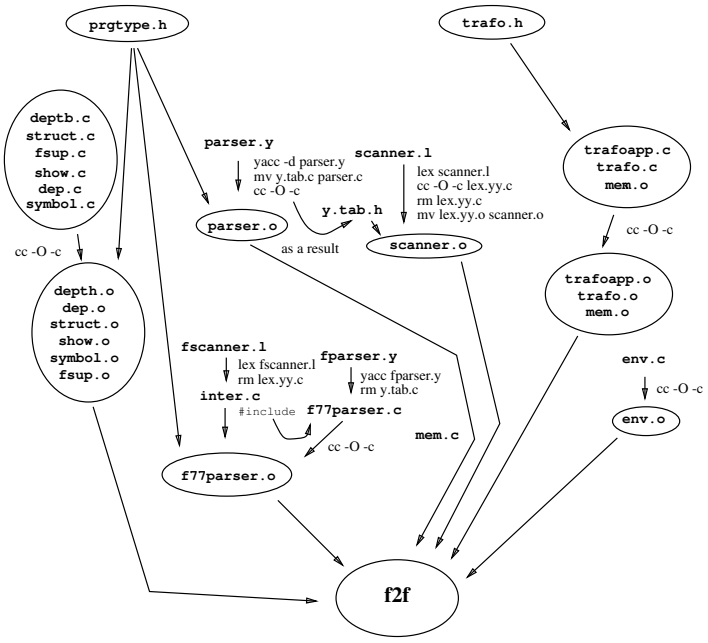


Figure 25: File Organzation

F Program Type Information

```
/* Source to Source Compiler
   by Aart J.C. Bik
   Information File    */

#define NESTSIZE  25

/* Program Data Structure Information */

#define K_DO       'a'
#define K_ASSIGN   'b'
#define K_LOGICIF  'c'
#define K_GENIF    'd'
#define K_ELSE     'e'
#define K_ELSEIF   'f'
#define K_WHILE    'g'
#define K_STOP     'h'
#define K_LINKUP   'i'
#define K_LINKIFUP 'j'

#define E_VAR      'a'
#define E_CONST    'b'
#define E_MUL      'c'
#define E_DIV      'd'
#define E_EXP      'e'
#define E_EQ       'f'
#define E_NE       'g'
#define E_GE       'h'
#define E_GT       'i'
#define E_LE       'j'
#define E_LT       'k'
#define E_EQV      'l'
#define E_NEQV     'm'
#define E_AND      'n'
#define E_OR       'o'
#define E_NOT      'p'
#define E_UMIN     'q'
#define E_ADD      'r'
#define E_MIN      's'
#define E_VEC      't'

#define NORM       'n'
#define ALL        'a'

union valuerec
     { int i;
```

```
    double f;
    int b;
      } ;

struct stmt_node
      { char kind;
    struct stmt_node *next;
    union{ struct{ struct expr_node *index,*expr1,*expr2,*expr3;
                struct stmt_node *body;
                int    loopno;
                char   ext;
                     } do_loop;
                struct{ struct expr_node *lhs,*rhs;
                int    stmtno;
                     } assign;
                struct{ struct expr_node *condition;
                struct stmt_node *body;
                int condno;
                     } s_if;
           } u ;
      } ;

struct expr_node
      { char kind;
    union{ struct{ struct expr_node *arg1,*arg2;
             } operands;
           struct{ int entry;
                struct sub_node *dim_list;
                     } var;
           struct{ int type;
                union valuerec val;
                     } expr;
                struct{ struct expr_node *e1,*e2,*e3;
             } vec;
             } u;
       } ;

struct sub_node
      { struct expr_node *head;
    int    *normexpr;
    struct sub_node  *tail; } ;

struct dim_node
      { int high,low;
    struct dim_node *next;
      } ;
```

```
typedef struct stmt_node *stmt_ptr;
typedef struct expr_node *expr_ptr;
typedef struct sub_node  *sub_ptr;
typedef struct dim_node  *dim_ptr;

expr_ptr e_pop();
stmt_ptr s_pop(),give_program();
union valuerec   cast_expr();

/* Symbol Table Information */

/* Type Identifications */

#define UNDEF     0
#define INTTYPE   1
#define REALTYPE  2
#define LOGICTYPE 3

/* Symboltable Functions */

char    *st_pos();
union   valuerec st_returnval();
dim_ptr st_returndimlist();

/* Dependence Table Information */

#define Cj     'c'
#define Lj     'l'
#define Sj     's'
#define IN     'i'
#define OUT    'o'
#define FLOW   'f'
#define ANTI   'a'
#define OUTPUT 'p'
#define INPUT  'n'

/* Memory Management Information */

char *alloc_mem(),*new_mem();
```

G LEX Definitions for the FORTRAN 77 Dialect

```
%{

/* Source to Source Compiler
```

```
   by Aart J.C. Bik
   LEX-definitions for FORTRAN */
/* #define YYLMAX 2000 */

%}

/* Regular Expressions */

letter      [A-Za-z]
digit       [0-9]
special     "+"|"*"|"/"|"="|"-"|"."|"$"
blank       [ \t]

comment       ("C"|"c"|"*").*\n
emptyline     {blank}*\n
continuation  "     "({letter}|{digit}|{special})

label1        ({digit}{1,5})
label2        " "({digit}{1,4})
label3        "  "({digit}{1,3})
label4        "   "({digit}{1,2})
label5        "    "{digit}
label         ({label5}|{label4}|{label3}|{label2}|{label1})

identifier    {letter}(({digit}|{letter})*)

int_num       {digit}+
decpoint      {digit}+\.{digit}*
real_num      {decpoint}(("E"|"e")("+"|"-")?{digit}+)?

%%

\n/({comment}|{emptyline})?{continuation} { f_line++; }

^{label}          { sscanf(yytext,"%d",&yylval); return(LABEL); }
^{comment}        { f_line++; }
^{emptyline}      { f_line++; }
^{continuation}   { ; }

{blank}+          { ; }
\n                { return('\n'); /* parser must increase f_line */     }

^{label}{blank}*"CONTINUE" { sscanf(yytext,"%d",&yylval); return(LCONTINUE);
}
^{label}{blank}*"continue" { sscanf(yytext,"%d",&yylval); return(LCONTINUE);
}
```

```
"DO"          { return(DO); }
"do"          { return(DO); }
"ELSE"        { return(ELSE); }
"else"        { return(ELSE); }
"END"         { return(END); }
"end"         { return(END); }
"ENDDO"       { return(ENDDO); }
"enddo"       { return(ENDDO); }
"END DO"      { return(ENDDO); }
"end do"      { return(ENDDO); }
"ENDIF"       { return(ENDIF); }
"endif"       { return(ENDIF); }
"END IF"      { return(ENDIF); }
"end if"      { return(ENDIF); }
"IF"          { return(IF); }
"if"          { return(IF); }
"INTEGER"     { return(INTEGER); }
"integer"     { return(INTEGER); }
"LOGICAL"     { return(LOGICAL); }
"logical"     { return(LOGICAL); }
"PARAMETER"   { return(PARAMETER); }
"parameter"   { return(PARAMETER); }
"PROGRAM"     { return(PROGRAM); }
"program"     { return(PROGRAM); }
"REAL"        { return(REAL); }
"real"        { return(REAL); }
"STOP"        { return(STOP); }
"stop"        { return(STOP); }
"THEN"        { return(THEN); }
"then"        { return(THEN); }
"WHILE"       { return(WHILE); }
"while"       { return(WHILE); }

"*"           { return('*'); }
"/"           { return('/'); }
"+"           { return('+'); }
"-"           { return('-'); }
"**"          { return(EXP); }
"("           { return('('); }
")"           { return(')'); }
"="           { return('='); }
":"           { return(':'); }
","           { return(','); }
".EQ."        { return(EQ); }
".eq."        { return(EQ); }
```

```
".NE."          { return(NE); }
".ne."          { return(NE); }
".GE."          { return(GE); }
".ge."          { return(GE); }
".GT."          { return(GT); }
".gt."          { return(GT); }
".LE."          { return(LE); }
".le."          { return(LE); }
".LT."          { return(LT); }
".lt."          { return(LT); }
".EQV."         { return(EQV); }
".eqv."         { return(EQV); }
".NEQV."        { return(NEQV); }
".neqv."        { return(NEQV); }
".NOT."         { return(NOT); }
".not."         { return(NOT); }
".AND."         { return(AND); }
".and."         { return(AND); }
".OR."          { return(OR); }
".or."          { return(OR); }
".TRUE."        { currentval.b = 1; yylval = LOGICTYPE; return(NUM_BOOL);
}
".true."        { currentval.b = 1; yylval = LOGICTYPE; return(NUM_BOOL);
}
".FALSE."       { currentval.b = 0; yylval = LOGICTYPE; return(NUM_BOOL);
}
".false."       { currentval.b = 0; yylval = LOGICTYPE; return(NUM_BOOL);
}

{identifier}    { yylval = st_insertid(yyleng,yytext); return(ID); }
{real_num}      { sscanf(yytext,"%lf",&currentval.f);
        yylval = REALTYPE; return(NUM_REAL); }
{int_num}       { sscanf(yytext,"%d",&currentval.i);
        yylval = INTTYPE; return(NUM_INT); }

.               { return(yytext[0]); /* Parser generates the error */ }

%%

/* Function needed by LEX */

yywrap() { return 1; }
```

H YACC Definitions for the FORTRAN 77 Dialect

```
%{
```

```
/* Source to Source Compiler
   by Aart J.C. Bik
   YACC-definitions for FORTRAN  */

#include <stdio.h>
#include "prgtype.h"
#define  __RUNTIME_YYMAXDEPTH /* Runtime memory allocation in YACC
*/

/* External variables */

extern int     error,f_line;

/* Administration Variables */

int            curtp,tp,assignno,loopnr,condno;
union valuerec currentval;

/* Attributes:  expr and simple_expr hold the type of the expression
        expr_list holds the merged type
        var holds the symboltable entry
        internlab holds the label value
              NUM_INT NUM_REAL and NUM_BOOL hold the type of the constant
        LCONTINUE and LABEL hold the label value */
%}

%token LABEL EQ NE GE GT LE LT EQV NEQV OR AND NOT UMIN
%token LCONTINUE DO ELSE END ENDDO ENDIF EXP
%token GOTO ID IF INTEGER LOGICAL
%token PARAMETER PROGRAM REAL STOP THEN WHILE
%token NUM_INT NUM_REAL NUM_BOOL

%nonassoc  EQV NEQV
%left      OR
%left      AND
%nonassoc  NOT
%nonassoc  EQ NE GE GT LE LT
%left      '+' '-'
%left      '*' '/'
%right     EXP
%right     UMIN

%start     program

%%
```

```
program       : PROGRAM ID newline
        { printf("\nPROGRAM %s\n",st_pos($2));
          assignno = loopnr = condno = 0; }
         program_list
       END newline   { create_program(); }
           ;

program_list  : spec newline program_list    { ; }
        | stmt_list                  { ; }
        ;

spec          : PARAMETER '(' par_list ')'               { ; }
        | INTEGER { curtp = INTTYPE; }   defvar_list  { ; }
        | REAL    { curtp = REALTYPE; }  defvar_list  { ; }
        | LOGICAL { curtp = LOGICTYPE; } defvar_list  { ; }
        ;

par_list      : par ',' par_list     { ; }
        | par                { ; }
        ;

par           : ID '=' expr          { test_par($1,$3); }
        ;

defvar_list   : defvar ',' defvar_list  { ; }
        | defvar                 { ; }
        ;

defvar        : ID      { reset_dim(); give_type($1,curtp,(int) 0); }
        | ID '('              { intpush((int) 0); reset_dim(); }
          subscriptlist ')'  { give_type($1,curtp,intpop()); }
          ;

subscriptlist : subscript ',' subscriptlist  { intpush( (intpop()+1) ); }
        | subscript                  { intpush( (intpop()+1) ); }
        ;

subscript     : expr             { test_subscript($1,INTTYPE);
                       make_sub(1); }
        | expr ':' expr    { test_subscript($1,$3);
                make_sub(2); }
        ;

stmt_list     : LABEL                    { is_unique($1); }
      stmt newline stmt_list   { s_link(); }
```

```
        | stmt newline stmt_list   { s_link(); }
        |                         { s_push(NULL); }
        ;

stmt        : DO internlab scalarvar '=' expr ',' expr
            { test_loop($3,$5,$7,INTTYPE);
              stack($2,$3); intpush(++loopnr); }
            newline stmt_list
            LCONTINUE  { is_unique($11); unstack($11);
                 make_loop(1,intpop(),NORM); }
            | DO internlab scalarvar '='  expr ',' expr ',' expr
            { test_loop($3,$5,$7,$9); stack($2,$3); intpush(++loopnr); }
            newline stmt_list
            LCONTINUE  { is_unique($13); unstack($13);
                 make_loop((int) 0,intpop(),NORM); }
            | IF '(' expr ')' singlestmt
            { test_if($3); make_logicalif(++condno); }
            | IF '(' expr ')' THEN
            { test_if($3); add_genif();
             intpush(++condno); intpush(1); }
            newline stmt_list
                 ENDIF { intpop(); make_generalif(intpop()); }
            | ELSE IF '(' expr ')' THEN
           { test_else(1); test_if($4); make_else(1,++condno); }
            | ELSE { test_else((int) 0); make_else((int) 0,(int) 0); }
            | DO WHILE '(' expr ')'
            { add_do(); test_if($4); intpush(++condno); }
            newline stmt_list
            ENDDO   { make_while(intpop()); }
            | singlestmt
            ;

singlestmt    : STOP            { make_stop(); }
            | var '=' expr    { test_assign($1,$3); make_assign(++assignno); }
            ;

internlab     : NUM_INT       { $$ = currentval.i; }

var           : scalarvar     { $$ = $1; }
            | arrayvar      { $$ = $1; }
            ;

scalarvar     : ID            { $$ = $1; /* entry */
                test_type($1,(int) 0,INTTYPE);
                make_var($1,(int) 0); }
              ;
```

```
arrayvar      : ID '('        { intpush((int) 0); }
       expr_list ')' { $$ = $1; /* entry */ curtp = intpop();
              test_type($1,curtp,$4); make_var($1,curtp); }
        ;

expr_list     : expr ',' expr_list
        { intpush( (intpop()+1) ); $$ = mergetype($1,$3); }
        | expr
        { intpush( (intpop()+1) ); $$ = $1; }
        ;

expr          : expr '+' expr
       { tp = arith($1,$3); make_expr(E_ADD,tp); $$ = tp; }
        | expr '-' expr
       { tp = arith($1,$3); make_expr(E_MIN,tp); $$ = tp; }
        | expr '*' expr
       { tp = arith($1,$3); make_expr(E_MUL,tp); $$ = tp; }
        | expr '/' expr
       { tp = arith($1,$3); make_expr(E_DIV,tp); $$ = tp; }
        | expr EXP expr
       { tp = arith($1,$3); make_expr(E_EXP,tp); $$ = tp; }
        | expr EQ expr
       { tp = relat($1,$3); make_expr(E_EQ,tp); $$ = tp; }
           | expr NE expr
       { tp = relat($1,$3); make_expr(E_NE,tp); $$ = tp; }
           | expr GE expr
       { tp = relat($1,$3); make_expr(E_GE,tp); $$ = tp; }
           | expr GT expr
       { tp = relat($1,$3); make_expr(E_GT,tp); $$ = tp; }
           | expr LE expr
       { tp = relat($1,$3); make_expr(E_LE,tp); $$ = tp; }
           | expr LT expr
       { tp = relat($1,$3); make_expr(E_LT,tp); $$ = tp; }
        | expr EQV expr
       { tp = logic($1,$3); make_expr(E_EQV,tp); $$ = tp; }
        | expr NEQV expr
       { tp = logic($1,$3); make_expr(E_NEQV,tp); $$ = tp; }
        | expr AND expr
       { tp = logic($1,$3); make_expr(E_AND,tp); $$ = tp; }
        | expr OR expr
       { tp = logic($1,$3); make_expr(E_OR,tp); $$ = tp; }
        | NOT expr
       { tp = ulogic($2);   make_expr(E_NOT,tp); $$ = tp; }
        | '(' expr ')'              { $$ = $2; }
        | '-' expr     %prec UMIN  { make_expr(E_UMIN,$2); $$ = $2; }
```

```
        | var           { $$ = st_returntype($1);
              /* var has made expression node */ }
        | constant      { $$ = $1; make_const($1,currentval); }
        ;

constant    : NUM_REAL { $$ = REALTYPE;  /* currentval.f is set */ }
        | NUM_INT  { $$ = INTTYPE;   /* currentval.i is set */ }
        | NUM_BOOL { $$ = LOGICTYPE; /* currentval.b is set */ }
        ;

newline     : '\n'  { f_line++; }
        ;

%%

#include "inter.c"

/* Reports an error message in case of a syntax error in the source code */

yyerror(s) char *s;
  { printf("\n*** %s in FORTRAN file: line <%d>\n",s,f_line);
    error = 1;          /* sets the error flag */
    while (yylook() > 0); /* lex - hack */
  }
```

I Supporting YACC (F77) Routines

```
/* Source to Source Compiler
   by Aart J.C. Bik
   Supporting YACC routines for FORTRAN  */

#include <stdio.h>
#include "prgtype.h"

/* External Variables */

extern dim_ptr  lastdim,firstdim;
extern FILE     *f_yyin;

/* Administration Variables */

int warnings,error,f_line,ifnest,whilenest;

/* Warning and Error Procedures */

sayerror(str) char *str;
```

```
    { printf("- Error: %s (line %d)\n",str,f_line);
      error = 1; /* set the error flag */ }

warning(str) char *str;
    { if (warnings) printf("- Warning: %s (line %d)\n",str,f_line); }

chain_var(str,i,err) char *str; int i,err;
   { char mess[55];

     strcpy(mess,str);
     strcat(mess,st_pos(i));
     if (err) sayerror(mess);
      else warning(mess);
   }

/* Assigns a type and a dimension to a variable in the symboltable
   and attaches a subscriptlist
   If a type is already assigned, a warning is generated if the types
   and dimensions are compatible, an error is generated otherwise
   If the dimension exceeds 7 an error is generated              */

give_type(i,tp,dim) int i,tp,dim;
   { if (st_returntype(i) == UNDEF)
       st_givetype(i,tp);
       else { if ((st_returntype(i) == tp) && (st_returndim(i) == dim))
         chain_var("Duplicate declaration -> ",i,0);
        else chain_var("Variable redeclared -> ",i,1);
          del_sublist(st_returndimlist(i));
        }
     if (dim > 7)
       chain_var("More than seven dimensions -> ",i,1);
     st_givedim(i,dim); st_givedimlist(i,firstdim);
   }

/* Determines the type of a variable according to the FORTRAN 77 rules
   (IMPLICIT DATA TYPING)  I - N     INTEGERs
              remaining REALs      */

int dettype(i) int i;
  { if ( ((st_pos(i)[0]) ≥ 'i') && ((st_pos(i)[0]) ≤ 'n')     ||
         ((st_pos(i)[0]) ≥ 'I') && ((st_pos(i)[0]) ≤ 'N')  )
      return INTTYPE;
      else return REALTYPE;
  }

int determine(i) int i;
```

```
  { switch(dettype(i))
    { case INTTYPE:
    if (warnings) printf("\t\t\t\t  type INTEGER assumed\n");
    return INTTYPE;
      case REALTYPE:
    if (warnings) printf("\t\t\t\t  type REAL assumed\n");
    return REALTYPE;
      default:
     printf("*** Corrupt\n"); exit(1);
    }
  }

/* Test if the use of a variable is correct
   If its type is not declared,
   a type is determined first using determine(i) */

test_type(i,dim,sbtp) int i,dim,sbtp;
  { int localdim;

    if (st_returntype(i) == UNDEF)
   { chain_var("Variable not declared -> ",i,0);
     st_givetype(i,determine(i));
   }
    /* Note that dimension is also used to indicate a PARAMETER */
    localdim = st_returndim(i); if (localdim < 0) localdim = 0;
    if (dim ≠ localdim) chain_var("Wrong Dimension -> ",i,1);
    if (sbtp ≠ INTTYPE) warning("Subscripts are not of type INTEGER");
  }

/* Checks if subscript expressions are of type INTEGER */

test_subscript(t1,t2) int t1,t2;
  { if ((t1 ≠ INTTYPE) || (t2 ≠ INTTYPE))
      warning("Subscripts are not of type INTEGER in declaration");
  }

/* Merges two types into INTTYPE or UNDEF  */

int mergetype(t1,t2) int t1,t2;
  { if ( (t1 == INTTYPE) && (t2 == INTTYPE) ) return INTTYPE;
      else return UNDEF;  }

/* Checks a parameter variable and sets its value in the symboltable  */

test_par(i,partype) int i,partype;
  {  int       bool;
```

```
    expr_ptr  ex;

    if (st_returndim(i) < 0)
   chain_var("Parameter redefined -> ",i,0);
    if (st_returntype(i) == UNDEF)
   { chain_var("Parameter not declared -> ",i,0);
    st_givetype(i,determine(i)); }
    if (st_returndim(i) > 0)
   chain_var("Array variable used as Parameter  -> ",i,1);
    if (st_returntype(i) ≠ partype)
   chain_var("Other type in value of Parameter -> ",i,0);
    /* Mark as parameter and cast the value to type of variable */
    st_givedim(i,−1);ex = e_pop();
    st_giveval(i,cast_expr(ex,st_returntype(i),&bool)); del_expr(ex);
    if (! (bool)) warning("Parameter value cannot be computed");
   }

/* Checks if the types of expression in an assignment on the left hand side
   and the right hand side are the same and if the left-hand-side variable
   is not a parameter or a loop-variable  */

test_assign(i,rhstype) int i,rhstype;
   { if (st_returndim(i) < 0)
       chain_var("Assignment to Parameter -> ",i,1);
     if (st_returntype(i) ≠ rhstype)
       chain_var("Other type in assignment to -> ",i,0);
     if (is_loopindex(i))
       chain_var("Assignment to Loop Variable -> ",i,1);
   }

/* This procedure checks if the types of the expressions in a DO-loop
   are the same as the type of the loop variable
   A warning is reported if different types are used
   An error is generated if the variable is an parameter */

test_loop(i,t1,t2,t3) int i,t1,t2,t3;
   { if (is_loopindex(i))
       chain_var("Same loop variable in DO-loop -> ",i,1);
     if ( (st_returntype(i) ≠ t1) || (st_returntype(i) ≠ t2)
      || (st_returntype(i) ≠ t3) )
       warning("Different types in DO-loop");
     if ( st_returndim(i) < 0 )
       chain_var("Parameter as loop variable -> ",i,1);
   }

/* Checks if the type in an IF-statement is LOGICAL */
```

```
test_if(tp) int tp;
  { if (tp ≠ LOGICTYPE)
      sayerror("Condition is not of type LOGICAL");
  }

add_genif() { ifnest++; }

add_do() { whilenest++; }

/* Checks if a ELSE of ELSE IF is expected */

test_else(arg) int arg;
  { int per;

     if (ifnest ≤ 0) sayerror("ELSE(IF) unexpected");
       else { per = intpop();  /* Something is on the stack */
              if (per == 0) sayerror("ELSE(IF) after ELSE");
        else if (arg == 0) per = 0;
              intpush(per); }
    }

/* The functions Arith,Relat, Logic and ULogic test the types of the operands
   and reports an error or a warning if necessary
   It also computes the resulting type of the expression */

int arith(i1,i2) int i1,i2;
   { if ((i1 == LOGICTYPE) || (i2 == LOGICTYPE))
       warning("LOGICAL operand in arithmetic operator");
     if ((i1 == REALTYPE) || (i2 == REALTYPE)) return REALTYPE;
       else return INTTYPE;
   }

int relat(i1,i2) int i1,i2;
   { if ((i1 == LOGICTYPE) || (i2 == LOGICTYPE))
       warning("LOGICAL operand in relational operator");
     return LOGICTYPE;
   }

int logic(i1,i2) int i1,i2;
   {  if ((i1 == REALTYPE) || (i2 == REALTYPE))
       { sayerror("REAL operand in logical operator"); return LOGICTYPE;
}
       /* Real operands are not allowed, so an error is reported */
       else if ((i1 == INTTYPE) || (i2  == INTTYPE))
```

```
        { warning("INTEGER operand in logical operator"); return INTTYPE;
}
        else return LOGICTYPE;
   }

int ulogic(i1) int i1;
   { if (i1 == REALTYPE)
       { sayerror("REAL operand in logical operator"); return LOGICTYPE;
}
       else if (i1 == INTTYPE)
        { warning("INTEGER operand in logical operator"); return INTTYPE;
}
          else return LOGICTYPE;
     }

/* Initializes the parser */

f_parserinit()
  { fgarbage_collect(); fstacks_init();
    st_init(); struct_init();
    f_line = 1; whilenest = ifnest = 0; }

/* Calls the parser if the filename exists
   The following tasks are performed:
     + lexical scanning
     + parsing
     + semantic checking
     + internal data structure creation with constant folding
     + data dependence analysis                                  */

f_parser(fname,wrn,shwdep) char *fname; int wrn,shwdep;
  { f_parserinit();
    error = 0; warnings = wrn;
    f_yyin = fopen(fname,"r");
    if (f_yyin == NULL)
      printf("\n*** Bad filename: %s\n",fname);
      else { f_yyparse(); fclose(f_yyin);
         if (error)
           f_parserinit();
           else { datadep_analysis(give_program());
              fortran_symboltable(); dep_dump(shwdep);
              dump_program((int) 1,"program.txt"); }
           }
    return error;
  }
```

J Symbol Table Routines

```
/* Source to Source Compiler
   by Aart J.C. Bik
   Symbol Table Operations   */

#include <stdio.h>
#include "prgtype.h"

#define F_MAXSYM    30
  /* Initial Max. number of symbols in symbol table */
#define F_MAXSTORE  210
  /* Initial Max. number of characters stored       */

/* The symboltable data structure */

int      *f_symbol,*f_dim,*f_type;          /* [F_MAXSYM]   */
dim_ptr  *f_dimlist;                        /* [F_MAXSYM]   */
union    valuerec *f_val;                   /* [F_MAXSYM]   */
char     *f_symstore;                       /* [F_MAXSTORE] */
int      f_storept,fsympt;

/* Returns begin address of string at position i in symbol table */

char *st_pos(i) int i;
  { return &(f_symstore[ (f_symbol[i]) ]); }

/* Returns number of symbols in the symbol table */

int st_number() { return fsympt; }

/* Returns type of symbol at position i */

int st_returntype(i) int i;
   { return f_type[i]; }

/* Returns value of symbol at position i */

union valuerec st_returnval(i) int i;
   { return f_val[i]; }

/* Returns dimension of symbol at position i */

int st_returndim(i) int i;
   { return f_dim[i]; }

/* Returns dimensionlist of symbol at position i */
```

```
dim_ptr st_returndimlist(i) int i;
  { return f_dimlist[i]; }

/* Assigns a type to a symbol at position i */

st_givetype(i,tp) int i,tp;
  { f_type[i] = tp; }

/* Assigns a value to a symbol at position i */

st_giveval(i,val) int i; union valuerec val;
  { f_val[i] = val; }

/* Assigns a dimension to a symbol at position i */

st_givedim(i,dim) int i,dim;
  { f_dim[i] = dim; }

/* Assigns a dimensionlist to a symbol at position i */

st_givedimlist(i,dl) int i; dim_ptr dl;
  { f_dimlist[i] = dl; }

/* Inserts a lexeme in the symboltable if it is new and
   returns the position in the symboltable                */

int st_insertid(length,str) int length; char *str;
  { static   int SYM_CUR   = F_MAXSYM;
    static   int STORE_CUR = F_MAXSTORE;
    register int   i;
    int      place = 0;
    char     *new_mem();

    if (length > 6) { length = 6; str[6] = '\0'; }
    while (place < fsympt)
    if (strcmp(st_pos(place),str) == 0) break; else place++;
    if (place == fsympt)
      { /* A new symbol has been found */
        if (fsympt ≥ SYM_CUR)
      { SYM_CUR += 20;
        f_symbol = (int *) new_mem(f_symbol,(SYM_CUR * sizeof(int)));
            f_dim = (int *) new_mem(f_dim, SYM_CUR * sizeof(int));
            f_type = (int *) new_mem(f_type,(SYM_CUR * sizeof(int)));
        f_dimlist = (dim_ptr *) new_mem(f_dimlist,
             (SYM_CUR * sizeof(dim_ptr)));
```

```
            f_val  = (union valuerec *) new_mem(f_val,
             (SYM_CUR * sizeof(union valuerec)));
           }
        f_type[fsympt] = UNDEF; f_dim[fsympt] = 0;
    f_dimlist[fsympt] = NULL; f_symbol[fsympt++] = f_storept;
        for (i = 0; i ≤ length; i++)
          { if (f_storept ≥ STORE_CUR)
             { STORE_CUR +=  140;
            f_symstore = (char *) new_mem(f_symstore,
              (sizeof(char) * STORE_CUR));
             }
            f_symstore[f_storept++] = str[i];
       }
      }
   return place;
 }

/* Writes the symbols used into the file program.sym */

fortran_symboltable()
  {  FILE     *symfile;
     dim_ptr  diml;
     register int  j;

     symfile = fopen("program.sym","w");
     if (symfile == NULL)
       printf("*** Error: can't open program.sym");
       else
    { fprintf(symfile,"Id\tDim\t\tType\t\tValue\t\tBounds\n\n");
      for (j=0; j<fsympt; j++)
         { fprintf(symfile,"%s\t",st_pos(j));
              if (f_type[j] == UNDEF)
           fprintf(symfile,"-\t\tUndefined\t-\n");
         else
          { if (f_dim[j] > 0)
                fprintf(symfile,"%d\t\t",f_dim[j]);
                else if (f_dim[j] == 0) fprintf(symfile,"Scalar\t\t");
                   else fprintf(symfile,"Parameter\t");
             switch (f_type[j])
            { case INTTYPE:
                 if (f_dim[j] < 0)
               fprintf(symfile,"INTEGER\t\t%d",f_val[j].i);
               else fprintf(symfile,"INTEGER\t\t-");
             break;
                 case REALTYPE:
             if (f_dim[j] < 0)
```

```
                fprintf(symfile,"REAL\t\t%f",f_val[j].f);
                else fprintf(symfile,"REAL\t\t-");
                     break;
                  case LOGICTYPE:
              if (f_dim[j] < 0)
                { if (f_val[j].b == 0)
                    fprintf(symfile,"LOGICAL\t\t.FALSE.");
                    else fprintf(symfile,"LOGICAL\t\t.TRUE.");
                       }
                else fprintf(symfile,"LOGICAL\t\t-");
                     break;
            default:
              printf("*** Corrupt\n"); exit(1); }
       if (f_dim[j] > 0)
         { fprintf(symfile,"\t\t( ");
           diml = f_dimlist[j];
           while (diml ≠ NULL)
             { fprintf(symfile,"%i:%i ",diml→low,diml→high);
               diml = diml → next; }
                    fprintf(symfile,")\n");
                  } else fprintf(symfile,"\n");
         }
          }
        fclose(symfile);
      }
   }

/* Symbol table memory management */

st_memory()
{ f_symbol   = (int      *)      alloc_mem(F_MAXSYM    * sizeof(int));
  f_dim      = (int      *)      alloc_mem(F_MAXSYM    * sizeof(int));
  f_type     = (int      *)      alloc_mem(F_MAXSYM    * sizeof(int));
  f_dimlist  = (dim_ptr  *)      alloc_mem(F_MAXSYM    * sizeof(dim_ptr));
  f_val      = (union valuerec *) alloc_mem(F_MAXSYM  * sizeof(union valuerec));
  f_symstore = (char     *)      alloc_mem(F_MAXSTORE  * sizeof(char));
  fsympt = 0;    /* symbol table empty */
 }

st_garbage_collect()
   { register int i;  for (i=0; i<fsympt; i++)  del_sublist( f_dimlist[i] ); }

st_init()
  { f_storept = fsympt = 0; }
```

K Memory Management Routines

```
/* Source to Source Compiler
   by Aart J.C. Bik
   Procedures for memory management */

#include <stdio.h>
#include "prgtype.h"
#include "trafo.h"

/* External variables */

extern dim_ptr   firstdim;
extern int       trafpt;
extern trafo_ptr *trafoin,*trafoout,*trafocond;

/* Stack declarations */

#define E_STACKSIZE 30
   /* Initial Max. number of expressions stacked */
#define S_STACKSIZE 100
   /* Initial Max. number of statements stacked  */
#define T_STACKSIZE 100
   /* Initial Max. number of transformations stacked */
#define INTSTACK    20
   /* Initial Max. number of integers stacked    */
#define NUMLABELS   80
   /* Initial Max. numbers of labels stored */

int      tsp=0,ssp=0,esp=0,intp=0,stackpt=0,labelpt=0;
int      *intstack;                    /* [INTSTACK]    */
stmt_ptr *s_stack;                     /* [S_STACKSIZE] */
expr_ptr *e_stack;                     /* [E_STACKSIZE] */
trafo_ptr *t_stack;                    /* [T_STACKSIZE] */
int      *l_store;                     /* [NUMLABELS    */
int      stck[NESTSIZE],lpname[NESTSIZE];

/* Memory management routines */

/* Allocates memory and tests if memory is really allocated */

char *alloc_mem(size) int size;
   { char *malloc(),*p;

     p = malloc(size);
     if (p == NULL) out_of_mem();
     return p;
```

```
  }

char *new_mem(p,size) char *p; int size;
  { char *realloc();

    p = realloc(p,size);
    if (p == NULL) out_of_mem();
    return p;
  }

/* Out of memory-handler */

out_of_mem()
 { printf("*** Out of memory\n");
   exit(1);
 }

/* This procedure allocates dynamic memory for:
   + The statement stack
   + The expression stack
   + The trafo stack
   + The integer stack
   + The DO-loop stack  <- static in prototype  */

parsers_memory()
{ s_stack    = (stmt_ptr  *)    alloc_mem(S_STACKSIZE * sizeof(stmt_ptr));
  e_stack    = (expr_ptr  *)    alloc_mem(E_STACKSIZE * sizeof(expr_ptr));
  t_stack    = (trafo_ptr *)    alloc_mem(T_STACKSIZE * sizeof(trafo_ptr));
  intstack   = (int       *)    alloc_mem(INTSTACK    * sizeof(int));
  l_store    = (int       *)    alloc_mem(NUMLABELS   * sizeof(int));
 }

/* Garbage collecting is performed when an error has occured
   during the parsing of a FORTRAN 77 program or when a new
   program is read in
   Dimension nodes, and statement nodes and expression nodes pointed
   to on the stack are deleted, the symbol-table is cleared
   and the program is deleted */

fgarbage_collect()
  { register int i;

    /* Use loops instead of unstack/emtpy routines */
    for (i=0; i<ssp; i++)     del_stmtlist(s_stack[i]);
    for (i=0; i<esp; i++)     del_expr(e_stack[i]);
    st_garbage_collect();
```

```
    del_sublist(firstdim);    del_stmtlist(give_program());
  }

/* Garbage collecting is performed when an error has occured
   during the parsing of the transformation file or when
   a new file is read in
   The trafo nodes on te stack are deleted          */

trafo_garbage_collect()
  { register int i;

    /* Use a loop instead of unstack/empty routines */
    for (i = 0; i<tsp; i++) del_trafolist(t_stack[i]);
    for (i = 0; i<trafpt; i++)
   { del_trafolist(trafoin[i]); del_trafolist(trafoout[i]);
     del_trafolist(trafocond[i]); }
  }

/* Clears a do-trafo-node */

del_trafdo(td) trafo_ptr td;
 { del_trafolist(td → u.do_pattern.ex1);
   del_trafolist(td → u.do_pattern.ex2);
   del_trafolist(td → u.do_pattern.ex3);
   del_trafolist(td → u.do_pattern.body);
 }

/* Clears a list of transformation nodes */

del_trafolist(tl) trafo_ptr tl;
   { if (tl ≠ NULL)
   { switch (tl → kind)
     { case T_DO     : del_trafdo(tl); break;
       case T_DOA    : del_trafdo(tl); break;
       case T_ASSIGN : del_trafolist(tl → u.assign_pattern.lhs);
               del_trafolist(tl → u.assign_pattern.rhs); break;
       case T_IF     : del_trafolist(tl → u.if_pattern.cond);
               del_trafolist(tl → u.if_pattern.body); break;
           case T_EXP    : del_trafolist(tl → u.exp.op1);
               del_trafolist(tl → u.exp.op2); break;
       case T_VECTOR : del_trafolist(tl → u.vec.v1);
               del_trafolist(tl → u.vec.v2);
               del_trafolist(tl → u.vec.v3); break;
       case T_MERGE  : del_trafolist(tl → u.merge.l1);
               del_trafolist(tl → u.merge.l2); break;
           case T_FUNC   : del_trafolist(tl → u.func.arg1);
```

```
                del_trafolist(tl → u.func.arg2); break;
        default       : break; /* No pointer attributes */
          }
      del_trafolist(tl → next); free(tl);
       }
   }

/* Clears the nodes used by subscripts */

del_sublist(dp) dim_ptr dp;
  { if (dp ≠ NULL)
      { del_sublist(dp → next);
    free(dp); }
  }

/* Deletes the statementlist (with it expressions) pointed to by s */

del_stmtlist(s) stmt_ptr s;
   { if ( s ≠ NULL)
    { if ((s → kind ≠ K_LINKUP) && (s → kind ≠ K_LINKIFUP))
        { switch (s → kind)
          { case K_DO:        del_expr(s → u.do_loop.index);
                              del_expr(s → u.do_loop.expr1);
                              del_expr(s → u.do_loop.expr2);
                              del_expr(s → u.do_loop.expr3);
                    del_stmtlist(s → u.do_loop.body); break;
            case K_ASSIGN:    del_expr(s → u.assign.lhs);
                        del_expr(s → u.assign.rhs); break;
                  case K_LOGICIF:   del_expr(s → u.s_if.condition);
                    del_stmtlist( s → u.s_if.body); break;
        case K_GENIF:     del_expr(s → u.s_if.condition);
                    del_stmtlist( s → u.s_if.body); break;
/* WHILE */      case K_WHILE:     del_expr(s → u.s_if.condition);
                    del_stmtlist( s → u.s_if.body); break;
        case K_ELSEIF:    del_expr(s → u.s_if.condition); break;
                  case K_ELSE:      break;
        case K_STOP:      break;
                  default:          printf("*** Corrupt\n"); exit(1);
           }
            del_stmtlist(s → next);
           }
      free(s);
        }
   }

del_expr(e) expr_ptr e;
```

```
  { if ( e ≠ NULL)
   { switch( e → kind )
       { case E_VAR:   del_dimlist( e → u.var.dim_list ); break;
         case E_CONST: break;
         case E_VEC:   del_expr(e → u.vec.e1);
                       del_expr(e → u.vec.e2);
                       del_expr(e → u.vec.e3); break;
         default:      del_expr(e → u.operands.arg1 );
                       del_expr(e → u.operands.arg2 ); break;
           }
     free(e);
      }
  }

del_dimlist(d) sub_ptr d;
  { if ( d ≠ NULL)
     { del_expr( d → head );
    del_dimlist( d → tail );
    if ((d → normexpr) ≠ NULL) free(d → normexpr);
    free(d);
     }
   }

/* Stack Procedures for integer */

intpush(i) int i;
  { static int INT_CUR = INTSTACK;

    intstack[intp++] = i;
    if (intp ≥ INT_CUR)
     { INT_CUR += 20; printf("More int\n");
    intstack = (int *) new_mem(intstack,(INT_CUR * sizeof(int)));
     }
   }

int intpop()
  { if (intp > 0) return (intstack[--intp]);
     else { printf("*** Int Stack underflow\n"); return 0; }
  }

/* Stack Procedures for expr pointers */

e_push(p) expr_ptr p;
 { static int E_CURSIZE = E_STACKSIZE;

  e_stack[esp++] = p;
```

```
   if (esp ≥ E_CURSIZE)
     { E_CURSIZE += 30;
       e_stack = (expr_ptr *) new_mem(e_stack,(E_CURSIZE * sizeof(expr_ptr)));
     }
 }

expr_ptr e_pop()
      { if (esp > 0) return (e_stack[--esp]);
      else { printf("*** Expr Stack underflow\n");return NULL; }
      }

/* Stack Procedures for stmt pointers */

s_push(p) stmt_ptr p;
  { static int S_CURSIZE = S_STACKSIZE;

    s_stack[ssp++] = p;
    if (ssp ≥ S_CURSIZE)
      { S_CURSIZE += 200;
        s_stack = (stmt_ptr *) new_mem(s_stack,(S_CURSIZE * sizeof(stmt_ptr)));
      }
  }

stmt_ptr s_pop()
      { if (ssp > 0) return (s_stack[--ssp]);
      else { printf("*** Stmt Stack underflow\n"); return NULL; }
      }

/* Generates an error if a label has already been defined  */

is_unique(lb) int lb;
  { register int i; int ok = 1;
    static int L_CURSIZE = NUMLABELS;

    for (i = 0; i < labelpt; i++)
       if (l_store[i] == lb)
     { sayerror("Label has already been set"); ok = 0; break; }
     if (ok) l_store[labelpt++] = lb;
    if (labelpt ≥ L_CURSIZE)
      { L_CURSIZE += 80;
        l_store = (int *) new_mem(l_store,(L_CURSIZE * sizeof(int))); }
   }

/* Stack Procedures for trafo pointers */

t_push(p) trafo_ptr p;
```

```
{ static int T_CURSIZE = T_STACKSIZE;

  t_stack[tsp++] = p;
  if (tsp ≥ T_CURSIZE)
    { T_CURSIZE += 20;
      t_stack = (trafo_ptr *) new_mem(t_stack,(T_CURSIZE * sizeof(trafo_ptr)));
    }
}

trafo_ptr t_pop()
  { if (tsp > 0) return (t_stack[--tsp]);
      else { printf("*** Trafo Stack underflow\n"); return NULL; }
  }

/* Returns true if variable is currently used as loop index
   (value returned is index in lpname + 1)                  */

int is_loopindex(vr) int vr;
 { register int i;

   for (i=0; i<stackpt; i++)
     if (vr == lpname[i]) return (i + 1);
   return 0;
 }

/* Returns the number of items on the stack */

stacksize() { return stackpt; }

/* Stack procedures to check DO & CONTINUE pairs
   The symboltable entry of the loop variable is stored in lpname */

stack(lb,vr) int lb,vr;
   { lpname[stackpt] = vr; stck[stackpt++] = lb;
     if (stackpt ≥ NESTSIZE)
   { printf("*** Nesting of DO-loops too deep\n"); exit(1); }
   }

unstack(expl) int expl;
   { if (stackpt ≤ 0)
   sayerror("CONTINUE unexpected");
   else if (stck[--stackpt] ≠ expl)
         sayerror("Incorrect CONTINUE label");
   }

/* Initializes the stack pointers for the FORTRAN 77 parser */
```

```
fstacks_init() { stackpt = ssp = esp = intp = labelpt = 0; }

/* Initializes the stack pointer for the trafo parser */

tstack_init() { tsp = 0; }
```

L Data Dependence Storage Routines

```
/* Source to Source Compiler
   by Aart J.C. Bik
   Procedures for dependences storage */

#include "prgtype.h"
#include <stdio.h>

#define MAXDEP  4000
  /* Initial Max. number of dependences in the dependence table */
#define MAXFLAGS 4000
  /* Initial Max. number of direction flags */

/* The dependence table data structure */

int      *no1,*no2;                  /* [MAXDEP] */
char     *kd1,*kd2,*kinddep;         /* [MAXDEP] */
int      *depvar,*depvarpos;         /* [MAXDEP] */
long int *nod;                       /* [MAXDEP] */

char   *dflags;                     /* [MAXFLAGS] */

/* Administration Variables */

int      deppt,dflag;
long int dep_number;
FILE     *depfile;

/* External Variables */

extern char     *charco,myflags[];
extern int      minnest,stackpt;
extern expr_ptr var1,e_v[];

/* Allocates memory for the dependence table */

dep_memory()
  { no1     = (int  *) alloc_mem(MAXDEP * sizeof(int));
```

```
    no2     = (int  *) alloc_mem(MAXDEP * sizeof(int));
    kd1     = (char *) alloc_mem(MAXDEP * sizeof(char));
    kd2     = (char *) alloc_mem(MAXDEP * sizeof(char));
    kinddep = (char *) alloc_mem(MAXDEP * sizeof(char));
    depvar  = (int  *) alloc_mem(MAXDEP * sizeof(int));
    nod     = (long int *) alloc_mem(MAXDEP * sizeof(long int));
    depvarpos  = (int  *) alloc_mem(MAXDEP * sizeof(int));
    dflags     = (char *) alloc_mem(MAXFLAGS * sizeof(char));
    deppt      = 0; /* dependence table empty */
 }

/* Initializes the dependence table pointers  */

dep_init() { dflag = deppt = 0; }

/* Writes the dependence table to the file program.dep */

dep_dump(lv) int lv;
   { register int i; int noi,noo,nof,noa;

     noi = noo = nof = noa = 0;
     depfile = fopen("program.dep","w");
     if (depfile == NULL)
       printf("*** Error: can't open program.dep");
       else { fprintf(depfile,"Dependence\t\t   Variable  Number\n\n");
          for (i=0; i < deppt; i++)
              { switch(kinddep[i])
           { case FLOW:   nof++; break;
             case ANTI:   noa++; break;
             case OUTPUT: noo++; break;
             case INPUT:  noi++; break;
             default:     printf("*** Corrupt\n"); exit(1); }
                if ((((lv ≠ 1) && (lv ≠3 )) || (kinddep[i] ≠ INPUT)) &&
                (((lv ≠ 2) && (lv ≠ 3)) ||
             ((kd1[i] == Sj) && kd2[i] == Sj) ))
           { switch(kd1[i])
                     { case Sj: fprintf(depfile,"S"); break;
                       case Cj: fprintf(depfile,"C"); break;
            case Lj: fprintf(depfile,"L"); break;
            default:     printf("*** Corrupt\n"); exit(1); }
                    fprintf(depfile,"%i",no1[i]);
                   switch(kinddep[i])
              { case FLOW:   fprintf(depfile," d-flow  "); break;
                case ANTI:   fprintf(depfile," d-anti  "); break;
                case OUTPUT: fprintf(depfile," d-outp  "); break;
                case INPUT:  fprintf(depfile," d-input "); break;
```

```
            default:     printf("*** Corrupt\n"); exit(1); }
                  fprintf(depfile,"%s\t",&dflags[depvarpos[i]]);
                  switch(kd2[i])
                     { case Sj: fprintf(depfile," S"); break;
                       case Cj: fprintf(depfile," C"); break;
            case Lj: fprintf(depfile," L"); break;
            default: printf("*** Corrupt\n"); exit(1); }
                  fprintf(depfile,"%i\t\t%s\t",no2[i],st_pos(depvar[i]));
           if (nod[i] < 0)
             fprintf(depfile,"?\n");
             else fprintf(depfile,"%ld\n",nod[i]);
         }
             }
      fprintf(depfile,"\nNumber of static input  dependences : %i",noi);
      fprintf(depfile,"\nNumber of static output dependences : %i",noo);
      fprintf(depfile,"\nNumber of static flow  dependences : %i",nof);
      fprintf(depfile,"\nNumber of static anti  dependences : %i",noa);
      fprintf(depfile,"\n\nTotal number of static  dependences : %i\n",deppt);
      fclose(depfile);
     }
   }

/* Returns true if there are dependences */

int dep_defined() { return (deppt > 0); }

dep_add(kind,ki1,ki2,n1,n2) char kind,ki1,ki2; int n1,n2;
 {   register int i;
     static   int DEPSYM = MAXDEP;
     static   int FLAG   = MAXFLAGS;

/* Both DEPSYM and FLAG grow at the same rate (emp.) */

   if (dep_number ≠ 0)
    { if (deppt ≥ DEPSYM)
       { /* Allocates more Memory */
     DEPSYM += 8000;
        no1     = (int  *) new_mem(no1,DEPSYM * sizeof(int));
        no2     = (int  *) new_mem(no2,DEPSYM * sizeof(int));
        kd1     = (char *) new_mem(kd1,DEPSYM * sizeof(char));
        kd2     = (char *) new_mem(kd2,DEPSYM * sizeof(char));
        kinddep = (char *) new_mem(kinddep,DEPSYM * sizeof(char));
        depvar  = (int  *) new_mem(depvar,DEPSYM * sizeof(int));
     depvarpos  = (int  *) new_mem(depvarpos,DEPSYM * sizeof(int));
        nod       = (long int *) new_mem(nod,DEPSYM * sizeof(long int));
       }
```

```
    if ((dflag+2+minnest) ≥ FLAG)
      { /* Allocates more memory */
    FLAG  += 8000;
    dflags = (char *) new_mem(dflags,FLAG * sizeof(char)); }
     no1[deppt] = n1;  kd1[deppt] = ki1;
     no2[deppt] = n2;  kd2[deppt] = ki2;
     kinddep[deppt] = kind; depvar[deppt] = (var1 → u.var.entry);
     nod[deppt]     = dep_number; depvarpos[deppt++] = dflag;
     for (i=1; i≤minnest; i++) dflags[dflag++] = myflags[i];
     dflags[dflag++] = '\0';
   }
  }

int flags_equal(i,f) int i,f;
  { register int r;
    int bool = 1;

    for (r = 0; r < stackpt; r++)
       switch( dflags[i++] )
      { case '<' : bool = 0; break;
        case '+' : bool = 0; break;
        case '\0': bool = 0; break; /* Nesting not correct */
      }
     while ((bool) && (charco[f] ≠ '\0'))
        { switch( dflags[i++] )
       { case '<' :
           bool = ((charco[f] == '<') || (charco[f] == '*')); break;
         case '>' :
       bool = ((charco[f] == '>') || (charco[f] == '*')); break;
         case '[' :
       bool = (charco[f] ≠ '>'); break;
         case ']' :
       bool = (charco[f] ≠ '<'); break;
         case '=' :
       bool = ((charco[f] == '=') || (charco[f] == '*')); break;
         case '+' : bool = (charco[f] ≠ '='); break;
         case '*' : break;
         case '\0': bool = 0; break; /* Nesting not correct */
         default  : printf("*** Corrupt\n"); exit(1);
       }
           f++;
     }
    return bool;
  }

/* Checks the particular statement kinds */
```

```
int in_if(s,n,k) stmt_ptr s; int n; char k;
 { if ((k == Cj) && (s → u.s_if.condno == n))
     return 1;
     else return in_slist(s → u.s_if.body,n,1,k); }

/* Tests if a statement appears in the statement list */

int in_slist(s,n,hd,k) stmt_ptr s; int n,hd; char k;
  { int bool = 0;

    while ((s ≠ NULL) && (bool == 0))
      { switch(s → kind)
      { case K_DO      : if ((k == Lj) && (s → u.do_loop.loopno == n))
                bool = 1;
                         else bool = in_slist(s → u.do_loop.body,n,1,k);
                        break;
        case K_ASSIGN  : if ((k == Sj) && (s → u.assign.stmtno == n))
                bool = 1;
                        break;
        case K_LOGICIF : bool = in_if(s,n,k);     break;
        case K_GENIF   : bool = in_if(s,n,k);     break;
        case K_ELSEIF  : bool = in_if(s,n,k);     break;
/* WHILE */  case K_WHILE   : bool = in_if(s,n,k);     break;
        case K_LINKIFUP: hd = 1; break;
        case K_LINKUP  : hd = 1; break;
        default        : break;
          }
        if (hd) s = NULL;
       else s = s → next;
       }
    return bool;
  }

int in_expr(ex,vr) expr_ptr ex; int vr;
  { if (ex == NULL) return 0;
    else switch(ex → kind)
      { case E_VAR:   return (ex → u.var.entry == vr);
        case E_CONST: return 0;
        case E_VEC:   return ( (in_expr(ex → u.vec.e1,vr)) ||
                (in_expr(ex → u.vec.e2,vr)) ||
                (in_expr(ex → u.vec.e3,vr)) );
        default:      return ( (in_expr(ex → u.operands.arg1,vr)) ||
                (in_expr(ex → u.operands.arg2,vr)) );
          }
   }
```

```
int on_vars(on,vr) int on,vr;
  { if (on == −1) return 1;
      else return (in_expr(e_v[on],depvar[vr]));
  }

/* Determines if there are no dependences of given kind with
   given directions in the statementlists given, on the variables
   in the expression specified */

int nodep(s1,s2,h1,h2,f,on,kind) stmt_ptr s1,s2; int h1,h2,f,on; char kind;
  { register int i;
    int      bool=1;

    for (i = 0; i < deppt; i++)
      if ( (kinddep[i] == kind) && (flags_equal(depvarpos[i],f)) &&
         (on_vars(on,i)) && in_slist(s1,no1[i],h1,kd1[i]) &&
       in_slist(s2,no2[i],h2,kd2[i]) )
          { bool = 0; break; }
    return bool;
  }
```

M Data Dependence Analysis Routines

```
/* Source to Source Compiler
   by Aart J.C. Bik
   Procedures for Data-Dependence Analysis */

#include <stdio.h>
#include "prgtype.h"

#define OutOut 10
#define InOut  11
#define OutIn  12
#define InIn   13

/* Administration variables for Data Dependence Analysis */

int   mynest,hisnest,minnest,*n1,*n2,hisnumber,mynumber,flowpos,ifnest;
char  depkind,mykind,hiskind;
int   env1[NESTSIZE],my1known[NESTSIZE],my2known[NESTSIZE];
int   lower[NESTSIZE],upper[NESTSIZE],stride[NESTSIZE];
int   env2[NESTSIZE],his1known[NESTSIZE],his2known[NESTSIZE],rolldep[2];
int   lower2[NESTSIZE],upper2[NESTSIZE],stride2[NESTSIZE],invdep[NESTSIZE];
char  myflags[NESTSIZE+1],invchar[NESTSIZE][NESTSIZE+1];
char  rollchar[2][NESTSIZE+1];
```

```
expr_ptr dumexpr1,dumexpr2,crtout,crtin,otherset,var1,var2,set2;

/* Administration variables for Determination of underlying number */

int coll1[NESTSIZE],coll2[NESTSIZE],a1,a2,a3,a4,u1,u2,d12,d34;
int arec[6],mrec[6],nrec[6],str[6];

long int arraynum(),scalarnum();

/* External Variable */

extern long int dep_number;

/* Computes the Data-Dependences in the program */

datadep_analysis(prg) stmt_ptr prg;
 { static make_node = 1;
   printf("\nComputing Data Dependences\n");
   if (make_node)
     { make_dummies(); make_node = 0; }
   dep_init(); mynest = 0; dep_it(prg);
 }

make_dummies()
 { /* Make dummy add-nodes for multiple insets in DO-loops */
  dumexpr1 = (expr_ptr) alloc_mem(sizeof(struct expr_node));
  dumexpr1 → kind = E_ADD;
  dumexpr1 → u.operands.arg2 = (expr_ptr) alloc_mem(sizeof(struct expr_node));
  dumexpr1 → u.operands.arg2 → kind = E_ADD;
  dumexpr2 = (expr_ptr) alloc_mem(sizeof(struct expr_node));
  dumexpr2 → kind = E_ADD;
  dumexpr2 → u.operands.arg2 = (expr_ptr) alloc_mem(sizeof(struct expr_node));
  dumexpr2 → u.operands.arg2 → kind = E_ADD;
 }

/* Handles a single statement */

dep_it(s) stmt_ptr s;
  { register int i;

    if ((s ≠ NULL) && (s → kind ≠ K_LINKUP) && (s → kind ≠ K_LINKIFUP))
       { /* Recovers 'his' environment */
      minnest = mynest; hisnest = mynest; ifnest = 0; flowpos = 1;
      for (i = 0; i < minnest; i++)
         { lower2[i] = lower[i]; upper2[i] = upper[i];
           stride2[i] = stride[i]; env2[i] = env1[i];
```

```
        his1known[i] = my1known[i]; his2known[i] = my2known[i]; }
    switch(s → kind)
      { case K_DO:      dep_do(s); mynest++; dep_it(s → u.do_loop.body);
                mynest−−;  break;
        case K_ASSIGN:  dep_assign(s); break;
        case K_LOGICIF: dep_genif(s); dep_it(s → u.s_if.body); break;
            case K_GENIF:   dep_genif(s); dep_it(s → u.s_if.body); break;
            case K_ELSEIF:  dep_elseif(s); break; /* Body in next */
        case K_ELSE:    break;
        case K_STOP:    break;
        case K_WHILE:   break; /* WHILE */
        default:        printf("*** Corrupt\n"); exit(1);
          }
        dep_it(s → next);
      }
  }

/* Computes the Data Dependences in which a do loop is involved */

dep_do(s) stmt_ptr s;
   { int    bl;
     double d;

     mynumber = s → u.do_loop.loopno; mykind = Lj;
  /* Create one INSET */
     (dumexpr1 → u.operands.arg1) = (s → u.do_loop.expr1);
     (dumexpr1 → u.operands.arg2 → u.operands.arg1) = (s → u.do_loop.expr2);
     (dumexpr1 → u.operands.arg2 → u.operands.arg2) = (s → u.do_loop.expr3);
     crtin = dumexpr1; crtout = s → u.do_loop.index;
  /* Test Self Dependence First */
     hiskind = mykind; hisnumber = mynumber;
     /* Do not filter on mynest > 0 (self antidependences too !) */
     otherset = crtout; dep_compare(OUT);
     if (mynest > 0) { otherset = crtin; dep_compare(IN); }
  /* Set Environment and Loop over the Following Statements */
     d = cast_expr(s → u.do_loop.expr1,REALTYPE,&bl).f;
     if ( (bl) && ( ( d − ( (int) d) ) == 0 ) )
       { lower2[hisnest] = lower[mynest] = (int) d;
         his1known[hisnest] = my1known[mynest] = 1; }
       else { my1known[mynest] = his1known[hisnest] = 0; }
     d = cast_expr(s → u.do_loop.expr2,REALTYPE,&bl).f;
     if ( (bl) && ( ( d − ( (int) d) ) == 0 ) )
       { upper2[hisnest] = upper[mynest] = (int) d;
         his2known[hisnest] = my2known[mynest] = 1; }
       else { my2known[mynest] = his2known[hisnest] = 0; }
     if (s → u.do_loop.expr3 ≠ NULL)
```

```
      { d = cast_expr(s → u.do_loop.expr3,REALTYPE,&bl).f;
        if ( (bl) && ( ( d − ( (int) d) ) == 0 ) )
          stride2[hisnest] = stride[mynest] = (int) d;
        else stride2[hisnest] = stride[mynest] = 0; }
      else stride2[hisnest] = stride[mynest] = 1;
    env2[hisnest++] = env1[mynest] = ((s → u.do_loop.index) → u.var.entry);

    dep_dofollowing(s → u.do_loop.body); /* Uses linkup */
  }

/* Determines if variable at index is not a loop-control variable */

int is_free(index) int index;
  { int free=1; register int i;

    for (i = 0; i < mynest; i++)
      if ( env1[i] == index) { free = 0; break; }
    if (free) for (i = 0; i < hisnest; i++)
        if ( env2[i] == index) { free = 0; break; }
    return free;
  }

/* Computes the Data-Dependences in which an assignment is involved */

dep_assign(s) stmt_ptr s;
  { mykind   = Sj;  mynumber = s → u.assign.stmtno;
    crtout   = s → u.assign.lhs; crtin = s → u.assign.rhs;
    /* Test Self Dependence First */
    hiskind = mykind; hisnumber = mynumber;
    /* Do not filter on mynest > 0 (self antidependences too !) */
    otherset = crtout; dep_compare(OUT);
    if (mynest > 0) { otherset = crtin; dep_compare(IN); }
    dep_dofollowing(s → next);
  }

/* Computes the Data-Dependences in which an if statement is involved */

dep_genif(s) stmt_ptr s;
  { if (s → u.s_if.condition → kind ≠ E_CONST)
      { /* Do only examine rest for non-constants ! */
    mykind   = Cj; mynumber = s → u.s_if.condno;
    crtout   = NULL; crtin  = s → u.s_if.condition;
       /* Test Self Dependence First */
    if (mynest > 0)
      { hiskind = mykind; hisnumber = mynumber;
          otherset = crtin; dep_compare(IN); }
```

```
    ifnest++; dep_dofollowing(s → u.s_if.body); /* uses linkup */
      }
  }

/* Computes the Data-Dependences in which an elseif statement is involved */

dep_elseif(s) stmt_ptr s;
  { if (s → u.s_if.condition → kind ≠ E_CONST)
      { /* Do only examine rest for non-constants ! */
    mykind   = Cj; mynumber = s → u.s_if.condno;
    crtout   = NULL; crtin  = s → u.s_if.condition;
       /* Test Self Dependence First */
    if (mynest > 0)
      { hiskind = mykind; hisnumber = mynumber;
          otherset = crtin; dep_compare(IN); }
    ifnest++; dep_dofollowing(s → next); /* Body in next */
      }
  }

/* Loops over every following statement */

dep_dofollowing(s) stmt_ptr s;
 { if (s ≠ NULL)
    { switch(s → kind)
      { case K_ASSIGN:    dep_followassign(s); break;
        case K_LOGICIF:   dep_followif(s); break;
        case K_GENIF:     dep_followif(s); break;
     case K_ELSEIF:    dep_followelse(s); break;
     case K_ELSE:      dep_followelse(s); break;
        case K_DO:        dep_followdo(s); break;
        case K_LINKUP:    if ((−−hisnest) < minnest) minnest = hisnest;
                  dep_dofollowing(s → next → next); break;
        case K_LINKIFUP:  if ((−−ifnest) < 0) ifnest = 0; flowpos = 1;
              dep_dofollowing(s → next → next); break;
        case K_STOP:      dep_dofollowing(s → next); break;
     case K_WHILE:     break; /* WHILE */
     default:          printf("*** Corrupt\n"); exit(1);
      }
    }
 }

/* Handles a following assignment */

dep_followassign(s) stmt_ptr s;
  { hiskind   = Sj; hisnumber = s → u.assign.stmtno;
    otherset  = s → u.assign.lhs;
```

```
    dep_compare(OUT);       /* Compare i and out Sj */
    otherset  = s → u.assign.rhs;
    dep_compare(IN);        /* Compare i and in  Sj */
    dep_dofollowing(s → next);
  }

/* Handles a following if statement */

dep_followif(s) stmt_ptr s;
  { hiskind  = Cj; hisnumber = s → u.s_if.condno;
    otherset = s → u.s_if.condition;
    dep_compare(IN);        /* Compare i and in Cj */
    ifnest++; dep_dofollowing(s → u.s_if.body); /* Uses linkup */
  }

/* Handles a following else(if) statement */

dep_followelse(s) stmt_ptr s;
  { if (ifnest == 0) flowpos = 0;
    if (s → kind == K_ELSEIF)
      { otherset = s → u.s_if.condition;
        hiskind  = Cj; hisnumber = s → u.s_if.condno;
        dep_compare(IN); /* Compare i and in Cj */ }
     dep_dofollowing(s → next); /* Body in next */
  }

/* Handles a following do loop */

dep_followdo(s) stmt_ptr s;
  { double d; int bl;

    hiskind   = Lj; hisnumber = s → u.do_loop.loopno;
    otherset  = s → u.do_loop.index;
    dep_compare(OUT);       /* Compare i and out Lj */
 /* Creates one INSET */
    (dumexpr2 → u.operands.arg1) = (s → u.do_loop.expr1);
    (dumexpr2 → u.operands.arg2 → u.operands.arg1) = (s → u.do_loop.expr2);
    (dumexpr2 → u.operands.arg2 → u.operands.arg2) = (s → u.do_loop.expr3);
    otherset = dumexpr2; dep_compare(IN);      /* Compare i and in Lj */
 /* Changes 'his' environment */
    env2[hisnest] = s → u.do_loop.index → u.var.entry;
    d = cast_expr(s → u.do_loop.expr1,REALTYPE,&bl).f;
    if ( (bl) && ( ( d − ( (int) d) ) == 0 ) )
     { lower2[hisnest] = (int) d; his1known[hisnest] = 1; }
     else his1known[mynest] = 0;
    d = cast_expr(s → u.do_loop.expr2,REALTYPE,&bl).f;
```

```
    if ( (bl) && ( ( d − ( (int) d) ) == 0 ) )
      { upper2[hisnest] = (int) d; his2known[hisnest] = 1; }
     else his2known[hisnest] = 0;
     if (s → u.do_loop.expr3 ≠ NULL)
       { d = cast_expr(s → u.do_loop.expr3,REALTYPE,&bl).f;
         if ( (bl) && ( ( d − ( (int) d) ) == 0 ) )
           stride2[hisnest] = (int) d;
       else stride2[hisnest] = 0;  }
       else stride2[hisnest] = 1;          hisnest++;
    dep_dofollowing(s → u.do_loop.body); /* Uses linkup */
  }

/* Actual Data Dependence test between the currentsets and otherset */

dep_compare(dir) int dir;
  { if (dir == OUT)
      { set2 = otherset; depkind = OutOut; dep_gen1(crtout);
    set2 = otherset; depkind = InOut;  dep_gen1(crtin); }
      else { /* dir == IN */
        set2 = otherset; depkind = OutIn; dep_gen1(crtout);
        set2 = otherset; depkind = InIn;  dep_gen1(crtin); }
  }

/* Generates all tests between arbitrary sets */

dep_gen1(aux1) expr_ptr aux1;
  { sub_ptr sublist;

    if (aux1 ≠ NULL)
      switch(aux1 → kind)
        { case E_VAR:   var1 = aux1; dep_gen2(set2);
            /* Handles hidden INSET */
            intpush(depkind);
            switch(depkind)
              { case OutOut:  depkind = InOut; break;
                case OutIn:   depkind = InIn;  break;
                        }
            sublist = aux1 → u.var.dim_list;
            while (sublist ≠ NULL)
                { dep_gen1(sublist → head);
                  sublist = sublist → tail; }
                      depkind = intpop(); break;
       case E_NOT:   dep_gen1(aux1 → u.operands.arg1); break;
       case E_UMIN:  dep_gen1(aux1 → u.operands.arg1); break;
       case E_CONST: break;
       case E_VEC:   dep_gen1(aux1 → u.vec.e1);
```

```
            dep_gen1(aux1 → u.vec.e2);
            dep_gen1(aux1 → u.vec.e3); break;
        default:      dep_gen1(aux1 → u.operands.arg2);
            dep_gen1(aux1 → u.operands.arg1); break;
        }
  }

/* Generates all tests between var1 and a set */

dep_gen2(aux2) expr_ptr aux2;
  { sub_ptr sublist;

    if (aux2 ≠ NULL)
      switch(aux2 → kind)
        { case E_VAR:  /* Only consider identical 'free' variables */
          if ( (var1 → u.var.entry == aux2 → u.var.entry)
                 &&
               ( ((mykind == Lj) && (hiskind == Lj))
                || (is_free(var1 → u.var.entry))) )
           { var2 = aux2; dep_vars(); }
          /* Handles hidden INSET */
          intpush(depkind);
          switch(depkind)
            { case OutOut:  depkind = OutIn; break;
              case InOut:   depkind = InIn;  break;
                      }
          sublist = aux2 → u.var.dim_list;
          while (sublist ≠ NULL)
              { dep_gen2(sublist → head);
                sublist = sublist → tail; }
                    depkind = intpop(); break;
        case E_NOT:   dep_gen2(aux2 → u.operands.arg1); break;
        case E_UMIN:  dep_gen2(aux2 → u.operands.arg1); break;
        case E_CONST: break;
        case E_VEC:   dep_gen2(aux2 → u.vec.e1);
            dep_gen2(aux2 → u.vec.e2);
            dep_gen2(aux2 → u.vec.e3); break;
        default:      dep_gen2(aux2 → u.operands.arg1);
            dep_gen2(aux2 → u.operands.arg2); break;
        }
  }


/* Returns the matching direction of two directions
  '!' returned if no such direction can be found    */
```

```
char dep_matchdir(c1,c2) char c1,c2;
  { switch(c1)
     { case '*': return c2;
    case '=': if ((c2 == '+') || (c2 == '<') || (c2 == '>'))
           return '!';
           else return '=';
    case '<': if ((c2 == '>') || (c2 == '=') || (c2 == ']'))
            return '!';
            else return '<';
       case '>': if ((c2 == '=') || (c2 == '<') || (c2 == '['))
           return '!';
           else return '>';
       case '[':  switch(c2)
            { case '*': return '['; case '=': return '=';
          case '<': return '<'; case '>': return '!';
          case '[': return '['; case ']': return '=';
             case '+': return '<';
                    default:  printf("*** Corrupt\n"); exit(1);
                  }
       case ']':  switch(c2)
            { case '*': return ']'; case '=': return '=';
          case '<': return '!'; case '>': return '>';
          case '[': return '='; case ']': return ']';
             case '+': return '>';
                    default:  printf("*** Corrupt\n"); exit(1);
                  }
       case '+':  switch(c2)
            { case '*': return '+'; case '=': return '!';
          case '<': return '<'; case '>': return '>';
          case '[': return '<'; case ']': return '>';
             case '+': return '+';
                    default:  printf("*** Corrupt\n"); exit(1);
                  }
       default:  printf("*** Corrupt\n"); exit(1);
     }
  }

/* Turns the direction vector around */

turn()
  { register int i;

    for (i=1; i≤minnest; i++)
     { switch (myflags[i])
        { case '>':  myflags[i] = '<'; break;
       case '<':  myflags[i] = '>'; break;
```

```
        case ']':  myflags[i] = '['; break;
        case '[':  myflags[i] = ']'; break;
             }
         }
  }

/* Determines the dependence using the direction vector */

dep_vector(vecdir) char vecdir;
  { int   self;
    char  dep;

    if (vecdir == '>') turn();
    self = ((mynumber ≠ hisnumber) || (mykind ≠ hiskind));
    switch(depkind)
      { case InIn:
            switch(vecdir)
          { case '<': self = 1; dep = INPUT; break;
            case '>': self = 1; dep = INPUT; break;
            case '=': dep = INPUT; break; } break;
        case InOut:
         switch(vecdir)
           { case '<': self = 1; dep = ANTI; break;
             case '>': self = 1; dep = FLOW; break;
             case '=': self = 1; dep = ANTI; break; } break;
    case OutIn:
         switch(vecdir)
           { case '<': dep = FLOW; break; /* do not allow self here */
             case '>': dep = ANTI; break; /* it has been handled by */
             case '=': dep = FLOW; break; } break; /* InOut !       */
    case OutOut:
          switch(vecdir)
            { case '<': self = 1; dep = OUTPUT; break;
              case '>': dep = OUTPUT; break; /* no self again! */
              case '=': dep = OUTPUT; break; } break;
        default: printf("*** Corrupt\n"); exit(1);
         }
  if (self)
    { if (vecdir == '>') dep_add(dep,hiskind,mykind,hisnumber,mynumber);
      else dep_add(dep,mykind,hiskind,mynumber,hisnumber); }
  if (vecdir == '>') turn();
 }

/* Handles < and > directions */

do_ne(i,c) int i; char c;
```

```
 { register int r;
   int index;

   index = (c == '<') ? 0 : 1; myflags[i] = c;
   if ((i == 1) && (rolldep[index]) && (abs(stride[0]) ≥ 1))
     for (r = 1; r ≤ minnest; r++) myflags[r] = rollchar[index][r];
   if ((rolldep[index]) || (i > 1)) dep_vector(c);
 }


/* Determines the dependences caused by two variables with the same name */

dep_vars()
 { register int i,r;
   int          lc,depassum=1,coll,ct=0;
   sub_ptr      sublist1,sublist2;
   dim_ptr      dimlist;

   /* Number determination preparation */
   dep_number = −1; lc = st_returndim(var1 → u.var.entry);
   coll = ((lc > 0) && ((mynest + hisnest) > 0) ) ? 1 : 0;
   if (coll)
     for (i=0; i < mynest;  i++)
    if ((my1known[i]) && (my2known[i]) && (stride[i])
        && (st_returntype(env1[i]) == INTTYPE)) coll1[i] = 0;
      else { coll = 0; break; }
   if (coll)
     for (i=0; i < hisnest; i++)
    if ((his1known[i]) && (his2known[i]) && (stride2[i])
        && (st_returntype(env2[i]) == INTTYPE)) coll2[i] = 0;
      else { coll = 0; break; }
   if (lc > 0)
    { sublist1 = var1 → u.var.dim_list;
      sublist2 = var2 → u.var.dim_list;
      dimlist  = st_returndimlist(var1 → u.var.entry);
      for (i=1; i ≤ minnest; i++) myflags[i] = '*';
      /* Prepare inv. optimization */
      for (r = 0; r < minnest; r++)
     { invdep[r] = 1;
       for (i = 1; i ≤ r + 1; i++)
          invchar[r][i] = '=';
       for (i = r + 2; i ≤ minnest; i++)
          invchar[r][i] = '*';
         }
      /* Prepare roll optimization */
      rolldep[0] = rolldep[1] = 1;
```

```
  rollchar[0][1] = '<'; rollchar[1][1] = '>';
  for (i = 2; i ≤ minnest; i++)
 rollchar[0][i] = rollchar[1][i] = '*';
  while ((sublist1 ≠ NULL) && (depassum))
   { if (((sublist1 → normexpr) ≠ NULL) &&
     ((sublist2 → normexpr) ≠ NULL))
       { n1 = sublist1 → normexpr; n2 = sublist2 → normexpr;
      depassum = dep_norm(&lc);
    }
    else coll = 0;
 sublist1 = sublist1 → tail;
 sublist2 = sublist2 → tail;

 /* number determination */
 if ((coll) && (depassum))
   { for (i=0; i < mynest; i++)
    coll1[i] += (n1[i] * coll);
         for (i=0; i < hisnest; i++)
    coll2[i] -= (n2[i] * coll);
         ct -= n1[mynest] * coll; ct += n2[hisnest] * coll;
     if (dimlist ≠ NULL)
       { coll *= ((dimlist → high) − (dimlist → low) + 1);
         if (coll < 0) coll = 0;
         dimlist  = dimlist  → next;
       }
       else coll = 0;
       }
   }
  if ((depassum) && (coll))
{ int nonz=0;
  for (i=0; i < minnest; i++) if (coll1[i]) nonz++;
  for (i=0; i < hisnest; i++) if (coll2[i]) nonz++;
  if (nonz ≤ 4)
    dep_number = arraynum(ct);
    }
}
if ((depassum) && (lc > 0) && (minnest > 0))
  {  /* Takes care of directed dependences */
 int busy=1;

 i = 1;
 while (busy)
  { switch(myflags[i])
      { case '<': do_ne(i,'<'); busy = 0; break;
        case '>': do_ne(i,'>'); busy = 0; break;
    case '+': do_ne(i,'<'); do_ne(i,'>'); busy = 0; break;
```

```
        case '=': busy = ((invdep[i−1]) && (i < minnest));
          for (r = 1; r ≤ minnest; r++)
            myflags[r] = invchar[i−1][r];
             if ((i == minnest) && (flowpos) && (invdep[i−1]))
           dep_vector('=');
                 else i++;
                   break;
    case '[': do_ne(i,'<'); myflags[i] = '='; break;
    case ']': do_ne(i,'>'); myflags[i] = '='; break;
    case '*': do_ne(i,'<'); do_ne(i,'>'); myflags[i] = '='; break;
          default:  printf("*** Corrupt\n"); exit(1);
         }
       }
  }
else if ((depassum) && ((lc == 0) || (minnest == 0)))
  { int  self;
    char dep1,dep2;

    /* Takes care of Scalar Like Variables */
    self = ( (mynumber ≠ hisnumber) || (mykind ≠ hiskind) );
    switch(depkind)
  { case InIn:   if (mynest == 0) dep_number = scalarnum(0,0,0,1);
           else if (hisnest == 0) dep_number = scalarnum(0,0,1,0);
           dep1 = INPUT;  dep2 = INPUT;  break;
    case InOut:  if (lc == 0) dep_number = scalarnum(minnest,0,1,0);
        dep1 = ANTI;   dep2 = FLOW; self = 1; break;
    case OutIn:  if (lc == 0) dep_number = scalarnum(minnest,0,0,1);
        dep1 = FLOW;   dep2 = ANTI; break;
       case OutOut: if (lc == 0) dep_number = scalarnum(minnest,0,0,0);
        dep1 = OUTPUT; dep2 = OUTPUT; break;
    default:     printf("*** Corrupt\n"); exit(1);
      }
    for (i=1; i≤minnest; i++) myflags[i] = '=';
    if ((self) && (flowpos))
   dep_add(dep1,mykind,hiskind,mynumber,hisnumber);
    i = minnest;  /* lc > 0 implies minnest == 0 */
    while (i > 0)
      { switch(dep2)
       { case INPUT:  dep_number = −1; break;
         case OUTPUT: dep_number = scalarnum(i−1,1,0,0); break;
         case FLOW:   dep_number = scalarnum(i−1,1,1,0); break;
         case ANTI:   dep_number = scalarnum(i−1,1,0,1); break; }
    myflags[i] = '<';  /* Self and Wrap Dependences */
    dep_add(dep2,hiskind,mykind,hisnumber,mynumber);
    myflags[i−−] = '*';
      }
```

```
    }
  }

/* Determines if a dependence is possible with some invariant part */

dep_invpart(diff) int diff;
  { register int i,r;
    int      saved1[NESTSIZE],saved2[NESTSIZE];
    char     saved3[NESTSIZE];

    for (r = 0; r < minnest; r++)
       { saved1[r] = n1[r]; saved2[r] = n2[r]; saved3[r+1] = myflags[r+1]; }
    for (r = 0; r < minnest; r++)
       if (invdep[r])
         { for (i = 1; i ≤ minnest; i++)
             if (dep_place(i−1,invchar[r][i]) == 0)
           { invdep[r] = 0; break; }
             if (invdep[r])
        { if (n2[r])
            { n1[r] = n1[r] − n2[r]; n2[r] = 0;
              invdep[r] = performtests(diff);
                   }
                  /* 'else ' ignore since coefficents were already zero
               but update and continue for rest */
                  if (invdep[r])
              for (i = 1; i ≤ minnest; i++)
                 invchar[r][i] = myflags[i];
                     else break;
                 }
            else break;
             }
        else break;
    for (r = 0; r < minnest; r++)
       { n1[r] = saved1[r]; n2[r] = saved2[r]; myflags[r+1] = saved3[r+1]; }
  }

rolldir(flag,diff) int flag,diff;
 { register int i;
   int      place=0;
   char     saved[NESTSIZE + 1],dir1='*',dir2='*';

   if ((flag) && (n1[0] ≠ 0) && (n1[0] == n2[0]))
     { flag = 0;
       for (i = 1; i < minnest; i++)
          if ((n1[i] ≠ 0) || (n2[i] ≠ 0))
        { if (n1[i] == n2[i]) { flag++; place = i; }
```

```
            else flag = 2; }
        if (flag == 1)
         { if ((n1[0] > 0) && (n1[place] > 0))
         { if (diff ≥ −n1[0]) dir1 = (diff == −n1[0]) ? ']' : '>';
           if (diff ≤ n1[0])  dir2 = (diff ==  n1[0]) ? '[' : '<'; }
           else if ((n1[0] > 0) && (n1[place] < 0))
            { if (diff ≥ −n1[0]) dir1 = (diff == −n1[0]) ? '[' : '<';
           if (diff ≤ n1[0])  dir2 = (diff ==  n1[0]) ? ']' : '>'; }
           else if ((n1[0] < 0) && (n1[place] > 0))
         { if (diff ≤ −n1[0]) dir1 = (diff ==  −n1[0]) ? '[' : '<';
           if (diff ≥ n1[0])  dir2 = (diff == n1[0]) ? ']' : '>'; }
           else
         { if (diff ≤ −n1[0]) dir1 = (diff == −n1[0]) ? ']' : '>';
           if (diff ≥ n1[0])  dir2 = (diff == n1[0]) ? '[' : '<'; }
           if (stride[0] < 0)
         { char tmp; tmp = dir1; dir1 = dir2; dir2 = tmp; }
         }
      }
    for (i = 1; i ≤ minnest; i++)
      saved[i] = myflags[i];
    /* < direction */
    for (i = 1; i ≤ minnest; i++)
      if (dep_place(i−1,rollchar[0][i]) == 0)
        { rolldep[0] = 0; break; }
    if (rolldep[0]) rolldep[0] = dep_place(place,dir1);
    for (i = 1; i ≤ minnest; i++)
      { rollchar[0][i] = myflags[i]; myflags[i] = saved[i]; }
    /* > direction */
    for (i = 1; i ≤ minnest; i++)
      if (dep_place(i−1,rollchar[1][i]) == 0)
        { rolldep[1] = 0; break; }
    if (rolldep[1]) rolldep[1] = dep_place(place,dir2);
    for (i = 1; i ≤ minnest; i++)
      { rollchar[1][i] = myflags[i]; myflags[i] = saved[i]; }
  }

/* Auxiliary functions for data dependence tests */

int down(a,l,u) int a,l,u;
  { if (a > 0) return (a*l);
      else return (a*u);
  }

int up(a,l,u) int a,l,u;
  { if (a > 0) return (a*u);
      else return (a*l);
```

```
  }

int comp_gcd(i1,i2) int i1,i2;
  { if (i2 == 0) return i1;
      else return comp_gcd(i2,(i1 % i2)); }

/* Determines if a dependence (+ direction) exists
   between two normalized expressions
   note: at mynest is the first constant
     at hisnest is the second constant
     minnest loop variables are in common   */

int dep_norm(l) int *l;
  { int      diff,dep=0;
    register int i;

    diff = n2[hisnest] - n1[mynest];
    /* CONSTANT INDEX TEST */
    for (i=0; i < mynest; i++)
       if (n1[i]) { dep = 1; break; }
    for (i=0; i < hisnest; i++)
       if (n2[i]) { dep = 1; break; }
     if (dep)
      dep = performtests(diff);
      else if (diff == 0)
             { dep = 1; (*l)--; }
             else dep = 0;
    return dep;
  }

/* Places a direction if still possible */

int dep_place(p,c) int p; char c;
  { char c2;

    if ((c == '<') || (c == '>'))
      { if (stride[p] == 0) c = '+';
    else if (stride[p] < 0)
          switch(c)
          { case '<': c = '>'; break;
            case '>': c = '<'; break; }
       }
    else if ((c == '[') || (c == ']'))
      { if (stride[p] == 0) c = '*';
    else if (stride[p] < 0)
          switch(c)
```

```
            { case '[': c = ']'; break;
              case ']': c = '['; break; }
      }
    c2 = dep_matchdir(c,myflags[p+1]);
    if (c2 == '!')
      return 0;
      else { myflags[p+1] = c2; return 1; }
  }

/* Performs some Data Dependence Analysis Tests on two
   normalized expressions possibly at different nestings depth
   - GCD test
   - Bounds test */

int performtests(diff) int diff;
  { register int i; int dep=1,c;

  { /* GCD TEST */
    int gcd=0; c = diff;
    for (i=0; i<mynest; i++)
      { if (st_returntype(env1[i]) ≠ INTTYPE) { gcd = 1; break; }
     if ((my1known[i]) && (stride[i]))
      { gcd = comp_gcd(gcd,abs(stride[i] * n1[i]));
        c = c - n1[i] * lower[i]; }
      else if ((i < minnest) && (stride[i]) && (n1[i] == n2[i]))
        gcd = comp_gcd(gcd,abs(stride[i] * n1[i]));
      else     gcd = comp_gcd(gcd,abs(n1[i]));

      }
    for (i=0; i<hisnest; i++)
      { if (st_returntype(env2[i]) ≠ INTTYPE) { gcd = 1; break; }
     if ((his1known[i]) && (stride2[i]))
      { gcd = comp_gcd(gcd,abs(stride2[i] * n2[i]));
        c = c + n2[i] * lower2[i]; }
      else if ((i < minnest) && (stride2[i]) && (n2[i] == n1[i]))
        gcd = comp_gcd(gcd,abs(stride[i] * n2[i]));
          else gcd = comp_gcd(gcd,abs(n2[i]));
      }
    if (gcd ≠ 0) dep = ((abs(c) % gcd) == 0);
  }
  { /* Bounds TEST */
    int comp=1,max=0,min=0; int l,u;
    if (dep)
      { for (i=0; i<mynest; i++)
          if ((my1known[i]) && (my2known[i]))
            { l = lower[i]; u = upper[i];
```

```
        if (l > u) { int temp; temp = l; l = u; u = temp; }
        min = min + down(n1[i],l,u);
            max = max + up(n1[i],l,u); }
          else if (n1[i] ≠ 0) { comp = 0; break; }
        /* ignore zero coefficients */
      if (comp)
    { for (i=0; i<hisnest; i++)
            if ((his1known[i]) && (his2known[i]))
              { l = lower2[i]; u = upper2[i];
            if (l > u) { int temp; temp = l; l = u; u = temp; }
                min = min + down((−n2[i]),l,u);
                max = max + up((−n2[i]),l,u); }
              else if (n2[i] ≠ 0) { comp = 0; break; }
          /* ignore zero coefficients */
          if (comp)
            dep = ((min ≤ diff) && (diff ≤ max));
        }
    }
}
{ /* Determine Direction */
  int place,flag=1; static int not_active = 1;

  if (dep)
    { /* DIFFERENT LOOP INDICES TEST */
      for (i=minnest; i < mynest; i++)
        if (n1[i]) { flag = 0; break; }
      for (i=minnest; i < hisnest; i++)
     if (n2[i]) { flag = 0; break; }
      if (flag)  /* Only indices from common nest */
        { int num=0;
          for (i = 0; i < minnest; i++)
                if ((n1[i] ≠ 0) || (n2[i] ≠ 0))
              { place = i; num++; }
           if (num == 1)
      /* ONE LOOP-CONTROL VARIABLE */
          dep = dep_intersect(n1[place],n2[place],diff,place);
          }
      if ((dep) && (not_active) && (minnest > 0))
        { not_active = 0; dep_invpart(diff); not_active = 1;
          if ((minnest > 1) && (abs(stride[0]) ≥ 1)) rolldir(flag,diff); }
    }
 }
 return dep;
}
```

/* Determines the direction vector using two linear subscript expressions */

```
int dep_intersect(c1,c2,diff,place) int c1,c2,diff,place;
 {  if (c1 == c2) /* Parallel Lines */
      { if (diff == 0) return(dep_place(place,'='));
      else { if ( ((diff > 0) && (c1 > 0)) ||
             ((diff < 0) && (c1 < 0))  )
            return(dep_place(place,'>'));
                else return(dep_place(place,'<'));  }
      }
      else
    { /* Intersecting Lines */
      double isp;

      isp = ( ((double) diff) / (c1 − c2) );
      switch( dep_detinterval(place,isp) )
     { case 0:
         return 1;
      case 1:
        if ((c1 > 0) && (c2 > 0))
          { if (c1 > c2) return(dep_place(place,'<'));
          else return(dep_place(place,'>')); }
              else if ((c1 < 0) && (c2 <0))
           { if (c2 < c1) return(dep_place(place,'>'));
            else return(dep_place(place,'<')); }
              else return 0;  /* divergent */
       case 2:
       if ((c1 > 0) && (c2 > 0))
         { if (c1 > c2) return(dep_place(place,'>'));
         else return(dep_place(place,'<')); }
              else if ((c1 < 0) && (c2 <0))
          { if (c2 < c1) return(dep_place(place,'<'));
            else return(dep_place(place,'>')); }
              else return 0;  /* divergent */
      case 3:
        if ((c1 > 0) && (c2 > 0))
          { if (c1 > c2) return(dep_place(place,'['));
          else return(dep_place(place,']')); }
              else if ((c1 < 0) && (c2 <0))
         { if (c2 < c1) return(dep_place(place,']'));
            else return(dep_place(place,'[')); }
              else return dep_place(place,'=');
       case 4:
        if ((c1 > 0) && (c2 > 0))
          { if (c1 > c2) return(dep_place(place,']'));
          else return(dep_place(place,'[')); }
              else if ((c1 < 0) && (c2 <0))
```

```
          { if (c2 < c1) return(dep_place(place,'['));
            else return(dep_place(place,']')); }
              else return dep_place(place,'=');
         case 5:
        return dep_place(place,'=');
         default:
        printf("*** Corrupt\n"); exit(1);
       }
  }
}

/* Determines the interval of a loop variable */

int dep_detinterval(p,x_i) int p; double x_i;
  {  if ((my1known[p]) && (my2known[p]) &&
     (lower[p] == x_i) && (upper[p] == x_i) ) return 5;
     else if (stride[p] > 0)
      { if ( (my1known[p]) && (lower[p] ≥ x_i) )
          return ((lower[p] > x_i) ? 1 : 3);
        else if ( (my2known[p]) && (upper[p] ≤ x_i) )
          return ((upper[p] < x_i) ? 2 : 4);
        else return 0;
      }
    else if (stride[p] < 0)
    { if ( (my2known[p]) && (upper[p] ≥ x_i) )
            return ((upper[p] > x_i) ? 1 : 3);
          else if ( (my1known[p]) && (lower[p] ≤ x_i) )
            return ((lower[p] < x_i) ? 2 : 4);
          else return 0;
        }
    else if ((my1known[p]) && (my2known[p]))
       { if ( (lower[p] ≤ x_i) && (upper[p] ≤ x_i) )
           return ((lower[p] < x_i) && (upper[p] < x_i)) ? 2 : 4;
             else if ( (lower[p] ≥ x_i) && (upper[p] ≥ x_i) )
           return ((lower[p] > x_i) && (upper[p] > x_i)) ? 1 : 3;
         else return 0;
           }
    else return 0;
  }

/* Determines the number of iteration using the loop-bounds */

long int no_myiterations(p) int p;
  { long int noi = −1;

    if ((my1known[p]) && (my2known[p]) && (stride[p]))
```

```
    { noi = (long int) ((upper[p] − lower[p]) / stride[p]);
   if (noi < 0) noi = 0;
    else noi = noi + 1; }
   return noi;
  }

long int no_hisiterations(p) int p;
  { long int noi = −1;

   if ((his1known[p]) && (his2known[p]) && (stride2[p]))
    { noi = (long int) ((upper2[p] − lower2[p]) / stride2[p]);
   if (noi < 0) noi = 0;
    else noi = noi + 1; }
   return noi;
  }

/* These procedures tries to determine the number of
   underlying dependences for scalar variables */

long int scalarnum(r1,cross,my,his) int r1,cross,my,his;
  { long int number=1,temp;
   register int i;

   for (i = 0; i ≤ (r1 − 1); i++)
     { number *= no_myiterations(i); if (number ≤ 0) break; }
   if ((number > 0 ) && (cross))
     { temp = no_myiterations(r1);
    if (temp == 0) number = 0;
      else if (temp > 0) number *= (temp − 1);
      else number = −1; }
   if ((number > 0 ) && (my))
    for (i = minnest; i < mynest; i++)
    { number *= no_myiterations(i); if (number ≤ 0) break; }
   if ((number > 0 ) && (his))
    for (i = minnest; i < hisnest; i++)
    { number *= no_hisiterations(i); if (number ≤ 0) break; }
   return number;
  }

/* Determines the number of underlying dependences for array variables */

long int delta2(c) int c;
  { double ceil(),floor();
   long int res;

   res = (1 + ((long int) (floor(((double)  c * u1)/((double) a2)))) −
```

```
            ((long int) ( ceil(((double) -c * u2)/((double) a1)))));
    if (res < 0) res = 0;
    return res;
  }

long int delta3(c) int c;
  { register int lab;
    int      j03,bound;
    long int sum=0;

    j03 = (c / a3) % d12; bound = (int) ((c - a3 * j03) / (a3 * d12));
    for (lab=0; lab <= bound; lab++)
       sum += delta2(c - a3 * j03 - lab * a3 * d12);
    return sum;
  }

long int delta4(c) int c;
  { register int lab;
    int      mu,bound;
    long int sum=0;

    mu = (c / d12) % d34; bound = (int) ((c - d12 * mu) / (d12 * d34));
    for (lab=0; lab <= bound; lab++)
    sum += delta2((c - mu * d12 - lab * d12 * d34)) *
       delta2((mu * d12 + lab * d12 * d34));
    return sum;
  }

long int arraynum(c) int c;
  { int fac=1,count=0,gcd=0;
    register int i;

    for (i=0; i < mynest; i++)
      if (coll1[i])
      { arec[count] = coll1[i]; str[count] = stride[i];
    mrec[count] = lower[i]; nrec[count++] = upper[i]; }
      else fac *= no_myiterations(i);
    for (i=0; i < hisnest; i++)
      if (coll2[i])
      { arec[count] = coll2[i]; str[count] = stride2[i];
    mrec[count] = lower2[i]; nrec[count++] = upper2[i]; }
      else fac *= no_hisiterations(i);
    /* normalize loops */
    for (i=0; i < count; i++)
       { c = c - mrec[i] * arec[i]; arec[i] = arec[i] * str[i];
         nrec[i] = (nrec[i] - mrec[i]) / str[i];
```

```
if (nrec[i] < 0) { fac = 0; break; } /* mrec[i] = 0; */ }
/* make all ai > 0 and recover mi = 0 constraint */
for (i=0; i < count; i++)
 if (arec[i] < 0)
   { arec[i] = −arec[i];  c = c + arec[i] ∗ nrec[i]; }
/* divide by gcd(ai) */
for (i=0; i < count; i++)
{ gcd = comp_gcd(gcd,arec[i]); nrec[i]++; }
if (gcd ≠ 0)
  { if ((c % gcd) ≠ 0) count = −1;
      else { c = c / gcd;
             for (i=0; i < count; i++) arec[i] = arec[i] / gcd;
           }
  }
if (fac == 0) return 0;
else switch(count)
  { case −1:  return 0;
case 0 :  return fac;
case 1 :  return fac; /* line and constant */
case 2 :
  a1 = arec[0]; a2 = arec[1]; set_us(a1,a2);
  return fac ∗ ( delta2(c) − delta2(c − a1∗nrec[0]) −
   delta2(c − a2∗nrec[1]) + delta2(c − a1∗nrec[0] − a2∗nrec[1]) );
    case 3 :
  a1 = arec[0]; a2 = arec[1]; a3 = arec[2]; set_us(a1,a2);
  return fac ∗ ( delta3(c) − delta3(c − a1∗nrec[0]) −
              delta3(c − a2∗nrec[1]) − delta3(c − a3∗nrec[2]) +
              delta3(c − a1∗nrec[0] − a2∗nrec[1]) +
              delta3(c − a1∗nrec[0] − a3∗nrec[2]) +
      delta3(c − a2∗nrec[1] − a3∗nrec[2]) −
      delta3(c − a1∗nrec[0] − a2∗nrec[1] − a3∗nrec[2]) );
    case 4 :
  a1 = arec[0]; a2 = arec[1]; a3 = arec[2]; a4 = arec[3];
     set_us(a1,a2); d34 = comp_gcd(a3,a4);
     return fac ∗ (delta4(c) − delta4(c − a1∗nrec[0])
            − delta4(c − a2∗nrec[1]) − delta4(c − a3∗nrec[2])
        − delta4(c − a4∗nrec[3])
     + delta4(c − a1∗nrec[0] − a2∗nrec[1])
            + delta4(c − a1∗nrec[0] − a3∗nrec[2])
            + delta4(c − a1∗nrec[0] − a4∗nrec[3])
        + delta4(c − a2∗nrec[1] − a3∗nrec[2])
              + delta4(c − a2∗nrec[1] − a4∗nrec[3])
        + delta4(c − a3∗nrec[2] − a4∗nrec[3])
        − delta4(c − a1∗nrec[0] − a2∗nrec[1] − a3∗nrec[2])
        − delta4(c − a1∗nrec[0] − a2∗nrec[1] − a4∗nrec[3])
        − delta4(c − a1∗nrec[0] − a3∗nrec[2] − a4∗nrec[3])
```

```
            - delta4(c - a2*nrec[1] - a3*nrec[2] - a4*nrec[3])
        + delta4(c - a1*nrec[0] - a2*nrec[1] - a3*nrec[2] - a4*nrec[3]) );
      }
  }

set_us(n,m) int n,m;
  { int x1=1,x2=0,y1=0,y2=1;

    while ((n ≠ 0) && (m ≠ 0))
      if (n > m)
        { x1 -= (n / m) * x2; y1 -= (n / m) * y2; n = (n % m); }
      else
        { x2 -= (m / n) * x1; y2 -= (m / n) * y1; m = (m % n); }
    if (n == 0) { d12 = m; u1 = x2; u2 = y2; }
     else { d12 = n; u1 = x1; u2 = y1; }
  }
```

N Transformation Type Information

```
/* Source to Source Compiler
   by Aart J.C. Bik
   Trafo Storage Information File  */

/* Trafo Data Structure Information */

#define T_DO      'a'
#define T_DOA     'b'
#define T_ASSIGN  'c'
#define T_IF      'd'
#define T_NIL     'e'
#define T_EVAR    'f'
#define T_SVAR    'g'
#define T_EXP     'h'
#define T_ICONST  'i'
#define T_RCONST  'j'
#define T_LCONST  'k'
#define T_VECTOR  'l'
#define T_COND    'm'
#define T_TRUE    'n'
#define T_MERGE   'o'
#define T_FUNC    'p'

struct trafo_node
  { char   kind;
    struct trafo_node *next;
    union{ struct{ struct trafo_node *var,*ex1,*ex2,*ex3,*body;
```

```
        } do_pattern;
          struct{ struct trafo_node *lhs,*rhs;
                  } assign_pattern;
          struct{ struct trafo_node *cond,*body;
                } if_pattern;
        struct{ struct trafo_node *op1,*op2;
          char    expr_kind;
           } exp;
          struct{ int var_nr,vecvar;
          struct trafo_node *v1,*v2,*v3;
                } vec;
          struct{ struct trafo_node *l1,*l2;
        } merge;
          struct{ int  entry,len;
          struct trafo_node *arg1,*arg2;
        } func;
          int var_nr;
      int ival;
      double rval;
      struct{ int flags,sl1,sl2,on;
          char kind;
        } cond;
         } u;
  } ;

typedef struct trafo_node *trafo_ptr;

trafo_ptr t_pop();
```

O LEX Definitions for the transformation language

```
%{

/* Source to Source Compiler
   by Aart J.C. Bik
   LEX-definitions for Trafo-Definition File */

#include   "y.tab.h"

extern int    line,yylval,tcharpt;
extern double rval;

%}

scan       [ \t]
skip       {scan}+
```

```
nl          [\n]

letter      [a-z]
digit       [0-9]
number      {digit}{digit}?
intnum      {digit}+
decpoint    {digit}+\.{digit}*
realnum     {decpoint}(("E"|"e")("+"|"-")?{digit}+)?
identifier  {letter}(({letter}|{digit})*)

%%

{skip}            { /* Skip white space */ }
#.*\n             { line++; /* Skip comments */  }
{nl}              { line++; /* Skip return */ }

"and"             { return AND; }
"assign"          { return ASSIGN; }
"condition"       { return CONDITION; }
"do"              { return DO; }
"doall"           { return DOALL; }
"dobody"          { return DOBODY; }
"end"             { return END; }
"head"            { return HEAD; }
"if"              { return IF; }
"ifbody"          { return IFBODY; }
"into"            { return INTO; }
"list"            { return LIST; }
"merge"           { return MERGE; }
"next"            { return NEXT; }
"nil"             { return NIL; }
"nodep"           { return NODEP; }
"transform"       { return TRANSFORM; }
"true"            { return TRUE; }
"."               { return '.'; }
";"               { return ';'; }
"("               { return '('; }
")"               { return ')'; }
","               { return ','; }
"<"               { return '<'; }
">"               { return '>'; }
"="               { return '='; }
"+"               { return '+'; }
"-"               { return '-'; }
"*"               { return '*'; }
"/"               { return '/'; }
```

```
"**"            { return EXP; }
".eq."          { return EQ; }
".neq."         { return NE; }
".ge."          { return GE; }
".gt."          { return GT; }
".le."          { return LE; }
".lt."          { return LT; }
".eqv."         { return EQV; }
".neqv."        { return NEQV; }
".and."         { return LAND; }
".or."          { return OR; }
".not."         { return NOT; }

"!e"{number}      { sscanf(&yytext[2],"%d",&yylval); return EXPVAR; }
"!s"{number}      { sscanf(&yytext[2],"%d",&yylval); return STMTVAR; }

"flow"          { yylval = 1; return DEPKIND; }
"anti"          { yylval = 2; return DEPKIND; }
"output"        { yylval = 3; return DEPKIND; }
"input"         { yylval = 4; return DEPKIND; }
".true."        { yylval = 1; return BOOLCONST; }
".false."       { yylval = 0; return BOOLCONST; }
"vectorize"     { return VECTOR; }

{intnum}        { sscanf(yytext,"%d",&yylval); return INTCONST; }
{realnum}       { sscanf(yytext,"%lf",&rval);  return REALCONST; }
{identifier}    { register int i;
          yylval = tcharpt;
          if (yyleng > 6) yyleng = 6;
          for (i = 0; i < yyleng; i++) char_insert(yytext[i]);
          char_insert('\0'); return FUNC; }

.               { return yytext[0];  /* Parser generates the error */ }

%%

/* Function needed by lex */

yywrap() { return 1; }
```

P YACC Definitions for the transformation language

```
%{

/* Source to Source Compiler
```

```
   by Aart J.C. Bik
   YACC-definitions for Trafo-Definition File  */

#include      <stdio.h>
#include      "trafo.h"
#include      "prgtype.h"
#define       _RUNTIME_YYMAXDEPTH /* Runtime memory allocation in YACC
*/

double rval;
int    pos1,pos2,pos3;

/* External variables */

extern int       err,line,vectmode,tcharpt;
extern trafo_ptr inpattern;

/* Attributes:
    dirvec holds the number of directions
    EXPVAR and STMTVAR hold the variable number
    FUNC holds the character entry of the identifier
    DEPKIND holds a representation for the kind of dependence
    INTCONST BOOLCONST hold the representation of the constant value */
%}

%token AND ASSIGN BOOLCONST CONDITION CURRENT DEPKIND
%token DIGIT DO DOALL DOBODY FUNC END EXPVAR HEAD IF
%token IFBODY INTCONST LIST INTO MERGE NEXT
%token NIL NODEP REALCONST STMTVAR TRANSFORM TRUE VECTOR
%token EXP EQ NE GE GT LE LT EQV NEQV LAND OR NOT UMIN

%nonassoc EQV NEQV
%left     OR
%left     LAND
%nonassoc NOT
%nonassoc EQ NE GE GT LE LT
%nonassoc ':'
%left     '+' '-'
%left     '*' '/'
%right    EXP
%right    UMIN

%start trafos

%%
```

```
trafos     : trafo trafos   { ; }
           | END            { printf("\n%i transformation",trafo_num());
                if (trafo_num() == 1) printf("\n");
                  else printf("s\n"); }
           ;

trafo      : TRANSFORM     { var_init(); }
             pattern   { /* TOP */ inpattern = t_pop(); t_push(inpattern); }
           INTO          { def(); }
             pattern
               CONDITION conditions ';'   { trafo_insert(); }
           ;

pattern    : LIST '(' stmt ',' pattern ')'  { t_link(); }
           | STMTVAR { mark_stmtvar($1); set_stmtvar($1); }
         | MERGE '(' pattern ',' pattern ')' { set_merge(); }
         | NIL    { set_nil(); }
         ;

stmt       : DO '(' exp ',' exp ',' exp ',' exp ',' pattern ')'
           { set_do(T_DO); }
           | DOALL '('  exp ',' exp ',' exp ',' exp ','  pattern ')'
           { set_do(T_DOA); }
           | ASSIGN '('  exp ',' exp ')'
           { set_assign(); }
           | IF '(' exp ',' pattern ')'
           { set_if(); }
         ;

exp        : EXPVAR              { mark_exprvar($1); set_exprvar($1); }
         | exp '+' exp          { set_exprkind(E_ADD); }
         | exp '-' exp          { set_exprkind(E_MIN); }
         | exp '*' exp          { set_exprkind(E_MUL); }
         | exp '/' exp          { set_exprkind(E_DIV); }
         | exp EXP exp           { set_exprkind(E_EXP); }
         | exp EQ exp           { set_exprkind(E_EQ); }
         | exp NE exp           { set_exprkind(E_NE); }
         | exp GE exp           { set_exprkind(E_GE); }
         | exp GT exp           { set_exprkind(E_GT); }
         | exp LE exp           { set_exprkind(E_LE); }
         | exp LT exp           { set_exprkind(E_LT); }
         | exp EQV exp           { set_exprkind(E_EQV); }
         | exp NEQV exp          { set_exprkind(E_NEQV); }
         | exp LAND exp          { set_exprkind(E_AND); }
         | exp OR exp           { set_exprkind(E_OR); }
         | NOT exp              { set_exprkind(E_NOT); }
```

```
      | '(' exp ')'
      | '-' exp %prec UMIN   { set_exprkind(E_UMIN); }
      | INTCONST             { set_const($1,0); }
      | REALCONST             { set_rconst(rval); }
      | BOOLCONST             { set_const($1,1); }
      | { vectmode++; }
        VECTOR '(' EXPVAR '|' EXPVAR ',' exp ':' exp ':' exp ')'
       { set_vector($4,$6); }
        | FUNC '(' exp ',' exp ')'  { set_func($1,2); }
      | FUNC '(' exp ')'          { set_func($1,1); }
      | FUNC                     { set_func($1,0); }
      ;

conditions : condition AND conditions { t_link(); }
      | condition               { ; }
      ;

condition  : TRUE  { set_true(); }
      | NODEP DEPKIND
         { pos1 = tcharpt; } dirvec
         { pos2 = tcharpt; } '(' s_indic ','
         { pos3 = tcharpt; } s_indic ')' onclause
         { test_nest($4,pos2,pos3); set_cond(pos1,pos2,pos3,$2,$12); }
      ;

onclause   : ">" EXPVAR { $$ = $2; }
      |           { $$ = -1; }
         ;
dirvec     : dir dirvec { $$ = $2 + 1; }
      |           { $$ = 0; char_insert('\0'); }
      ;

dir        : '='  { char_insert('='); }
      | '<'  { char_insert('<'); }
      | '>'  { char_insert('>'); }
      | '*'  { char_insert('*'); }
      ;

s_indic    : '$' attribs { ; }
      ;

attribs    : '.' NEXT    { char_insert('n');  } attribs { ; }
      | '.' DOBODY  { char_insert('d');  } attribs { ; }
      | '.' IFBODY  { char_insert('i');  } attribs { ; }
      | '.' HEAD    { char_insert('h');  char_insert('\0'); }
      |             { char_insert('\0'); }
```

```
      ;

%%

/* Reports an error message in case of a syntax error */

yyerror(s) char *s;
  { printf("\n*** %s in Transformation file: line <%d>\n\n",s,line);
    err = 1;              /* sets the error flag */
    while (yylook() > 0); /* lex - hack */
  }
```

Q Unparsing Routines

```
/* Source to Source Compiler
   by Aart J.C. Bik
   Procedures for showing the program */

#include <stdio.h>
#include "prgtype.h"

/* Administration variables */

FILE     *filen;
int      norm_on,show_on,nest,level,column;
int      loopname[NESTSIZE];
char     string[80];

/* This procedure writes the string to the output file
   and cuts of at the 72th column using continuation     */

dump72(s) char *s;
  { register int i;

    if ((column > 64) && (s[0] ≠ '\n'))
      { fprintf(filen,"\n     + "); column = 0; }
    for (i = 0; i < strlen(s); i++)
       { fprintf(filen,"%c",s[i]);
     if (s[i] == '\n') column = 0;
       else column++;
       }
  }

/* These procedures show particular expressions */

show_norm(nrm) int *nrm;
```

```
  { register int i;
    int      set=0;

    dump72(" <");
    for (i=0; i<nest; i++)
       if (nrm[i])
         { if ((set) && (nrm[i] > 0)) dump72("+");
       /* minus is printed automatically */
       sprintf(string,"%i*",nrm[i]);
       if (nrm[i] ≠ 1) dump72(string);
       dump72(st_pos(loopname[i])); set=1; /* Always a scalar */
         }
    if (nrm[nest])
      { if ((set) && (nrm[nest] > 0)) dump72("+");
    sprintf(string,"%i",nrm[nest]); dump72(string); set = 1;
      }
    if (set == 0) dump72("0");
    dump72("> ");
   }

show_var(e) expr_ptr e;
  { sub_ptr now;

    dump72(st_pos(e → u.var.entry));
    now = e → u.var.dim_list;
    if (now ≠ NULL)
      { dump72("(");
    while (now ≠ NULL)
        { if ((norm_on) && (now → normexpr ≠ NULL))
        show_norm(now → normexpr);
           else show_expr( now → head );
          now = now → tail;
          if (now ≠ NULL) dump72(",");
        }
       dump72(")");
      }
  }

show_const(e) expr_ptr e;
  { switch (e → u.expr.type )
     { case INTTYPE:   sprintf(string,"%i",e → u.expr.val.i);
              dump72(string); break;
       case REALTYPE:  sprintf(string,"%lf",e → u.expr.val.f);
              dump72(string); break;
       case LOGICTYPE: if (e → u.expr.val.b) dump72(".TRUE.");
                 else dump72(".FALSE."); break;
```

```
          default:        printf("*** Corrupt\n"); exit(1);
        }
  }

show_uoper(e) expr_ptr e;
   { switch (e → kind)
    { case E_UMIN: dump72("(-"); break;
         case E_NOT:  dump72(".NOT.("); break;
      default:     printf("*** Corrupt\n"); exit(1);
       }
     show_expr( e → u.operands.arg1 );
     dump72(")");
   }

show_oper(e) expr_ptr e;
  { dump72("(");
    show_expr( e → u.operands.arg1 );
    switch ( e → kind)
     { case E_MUL:  dump72(" * "); break;
       case E_DIV:  dump72(" / "); break;
       case E_EXP:  dump72(" ** "); break;
       case E_EQ:   dump72(".EQ."); break;
       case E_NE:   dump72(".NE."); break;
       case E_GE:   dump72(".GE."); break;
       case E_GT:   dump72(".GT."); break;
       case E_LE:   dump72(".LE."); break;
       case E_LT:   dump72(".LT."); break;
       case E_EQV:  dump72(".EQV."); break;
       case E_NEQV: dump72(".NEQV."); break;
       case E_AND:  dump72(".AND."); break;
       case E_OR:   dump72(".OR."); break;
       case E_ADD:  dump72(" + "); break;
       case E_MIN:  dump72(" - "); break;
       default:     printf("*** Corrupt\n"); exit(1);
      }
    show_expr( e → u.operands.arg2 );
    dump72(")");
  }

show_triplet(e) expr_ptr e;
 { show_expr(e → u.vec.e1); dump72(":");
   show_expr(e → u.vec.e2); dump72(":");
   show_expr(e → u.vec.e3);
 }
```

/∗ This procedure shows an expression ∗/

```
show_expr(e) expr_ptr e;
   { if (e == NULL) dump72("<Empty>");
     else switch (e → kind)
      { case E_VAR:   show_var(e); break;
        case E_CONST: show_const(e); break;
        case E_UMIN:  show_uoper(e); break;
        case E_NOT:   show_uoper(e); break;
        case E_VEC:   show_triplet(e); break;
        default:      show_oper(e); break;
          }
     }

/* This procedure handles the indentation */

level_it()
  { register int i;
    int      max;

    max = level;
    if (max > 20) max = 20;
    for (i=1; i≤max; i++) dump72(" ");
  }

/* These procedures show one particular statement */

show_body(s) stmt_ptr s;
  {  show_expr(s → u.do_loop.index);
     dump72(" = "); show_expr(s → u.do_loop.expr1);
     dump72(", ");  show_expr(s → u.do_loop.expr2);
     dump72(", "); show_expr(s → u.do_loop.expr3);
     loopname[nest++] = (s → u.do_loop.index) → u.var.entry;
     dump72("\n"); level += 2; show_stmtlist(s → u.do_loop.body); level −= 2;
     nest−−; level_it();
  }

show_doloop(s) stmt_ptr s;
   { level_it(); if (show_on) fprintf(filen,"L%i: ",s → u.do_loop.loopno);
     switch (s → u.do_loop.ext)
    { case NORM : dump72("DO "); show_body(s);
              dump72("ENDDO\n"); break;
         case ALL  : dump72("DOALL "); show_body(s);
              dump72("ENDDOALL\n"); break;
      default   : printf("*** Corrupt\n"); exit(1);
        }
   }
```

```
show_assign(s) stmt_ptr s;
  { level_it(); if (show_on) fprintf(filen,"S%i: ",s → u.assign.stmtno);
    show_expr(s → u.assign.lhs); dump72(" = ");
    show_expr(s → u.assign.rhs); dump72("\n");
  }

show_logicalif(s) stmt_ptr s;
     { int oldlevel;

       level_it(); if (show_on) fprintf(filen,"C%i: ",s → u.s_if.condno);
       dump72("IF (");
       show_expr(s → u.s_if.condition);
       dump72(") ");
       oldlevel = level; level = 0;
       show_stmt(s → u.s_if.body); level = oldlevel;
     }

show_genif(s) stmt_ptr s;
   { level_it(); if (show_on) fprintf(filen,"C%i: ",s → u.s_if.condno);
     dump72("IF ("); show_expr(s → u.s_if.condition); dump72(") THEN\n");
     level += 2; show_stmtlist(s → u.s_if.body); level −= 2;
     level_it(); dump72("END IF\n");
   }

show_elseif(s) stmt_ptr s;
  { level −=2; level_it(); level += 2;
    if (show_on) { sprintf(string,"C%i: ",s → u.s_if.condno); dump72(string); }
    dump72("ELSE IF ("); show_expr(s → u.s_if.condition); dump72(") THEN\n");
  }

show_while(s) stmt_ptr s;
  { level_it();
    if (show_on) { sprintf(string,"C%i: ",s → u.s_if.condno); dump72(string); }
    dump72("DO WHILE ("); show_expr(s → u.s_if.condition); dump72(")\n");
    level += 2; show_stmtlist(s → u.s_if.body); level −= 2;
    level_it(); dump72("ENDDO\n");
  }

show_else()
  { level −=2; level_it(); level += 2; dump72("ELSE\n"); }

show_stop()
  { level_it(); dump72("STOP\n"); }

/* This procedure shows one particular statement */
```

```
show_stmt(s) stmt_ptr s;
    { switch (s → kind)
   { case K_DO:        show_doloop(s); break;
        case K_ASSIGN:    show_assign(s); break;
     case K_LOGICIF:   show_logicalif(s); break;
     case K_GENIF:     show_genif(s); break;
     case K_ELSEIF:    show_elseif(s); break;
     case K_ELSE:      show_else(); break;
     case K_WHILE:     show_while(s); break;
     case K_STOP:      show_stop(); break;
     default:          printf("*** Corrupt\n"); exit(1);
       }
    }

/* This procedure sends the statementlist pointed to by s
   in a readable fashion to the appropriate filen        */

show_stmtlist(s) stmt_ptr s;
   { if ((s ≠ NULL) && (s → kind ≠ K_LINKUP) && (s → kind ≠ K_LINKIFUP))
   { show_stmt(s); show_stmtlist(s → next); }
   }

/* Shows a subscriptlist */

show_subscriptlist(dp) dim_ptr dp;
  { dump72("(");
    while (dp ≠ NULL)
   { sprintf(string,"%i:%i",dp → low,dp → high);
     dump72(string);
     dp = dp → next;
     if (dp ≠ NULL) dump72(",");
       }
   dump72(")");
  }

/* Writes the symboltable into a readable form */

dump_symbols()
  { register int i;

    for (i = 0; i < st_number(); i++)
   if ((st_returndim(i) ≥ 0) && (st_returntype(i) ≠ UNDEF))
      { switch(st_returntype(i))
         { case INTTYPE:   dump72("      INTEGER "); break;
           case REALTYPE:  dump72("      REAL    "); break;
```

```
            case LOGICTYPE: dump72("      LOGICAL "); break;
              }
          dump72(st_pos(i));
            if (st_returndim(i) > 0)
           show_subscriptlist(st_returndimlist(i));
            dump72("\n");
          }
      dump72("\n");
  }

/* Sends the program in memory to the filen program.out
   show_on controls the numbering of statements
   norm_on controls the presentation of the normal forms */

dump_program(on,s) int on; char *s;
  {  norm_on = show_on = on;
     filen = fopen(s,"w");
     if (filen == NULL)
    printf("\n*** Error: can't open %s\n",s);
    else { column = 0;
          dump72("      PROGRAM "); dump72(st_pos(0)); dump72("\n");
             dump_symbols(); nest = 0; level = 6;
          show_stmtlist(give_program());
             dump72("\n      END\n");
          fclose(filen);
            }
    }

/* Opens the standard output for displaying code */

open_program()
  { show_on = 1; filen = stdout;
    norm_on = column = nest = 0; level = 6; }
```

R Transformation Storage Routines

```
/* Source to Source Compiler
   by Aart J.C. Bik
   Transformation storage procedures */

#include <stdio.h>
#include "trafo.h"
#include "prgtype.h"

/* Administration Variables */
```

```
int          vectmode,line,err,setting;
int          s_used[100],e_used[100];
trafo_ptr    inpattern;
char         em[80];

extern FILE *yyin;
extern char *charco;

/* Error routine */

set_error(s) char *s;
  { printf("- Error: %s (line %d)\n",s,line);
    err = 1; /* Set the error flag */
  }

/* Variable processing procedures */

var_init()
  { register int i;

    setting = 1; /* Set the 'setting-mode' flag */
    for (i = 0; i < 100; i++)
       s_used[i] = e_used[i] = 0;
   }

def() { setting = 0; }

/* Checks that only lhs variables appear at the rhs and that
   stmt-var-s are only matched once        */

mark_stmtvar(i) int i;
  { if (setting)
      { s_used[i]++;
    if (s_used[i] == 2)
      { sprintf(em,"Variable s%i is used more than once",i);
        set_error(em); }
      }
      else if (s_used[i] == 0)
         { sprintf(em,"Variable s%i is not defined before use",i);
           set_error(em); }
  }

mark_exprvar(i) int i;
  { if (setting)
      e_used[i] = 1;
      else if (e_used[i] == 0)
```

```
        { sprintf(em,"Variable e%i is not defined before use",i);
            set_error(em); }
  }

/* Tests the nesting of a condition */

test_nest(nod,n1,n2) int nod,n1,n2;
  { int count=0;

    while ((charco[n1] == charco[n2]) && (charco[n1] ≠ '\0'))
      { if (charco[n1] == 'd') count++;
     n1++; n2++;
      }
    if (nod ≠ count)
      set_error("Incorrect number of directions");
  }

/* Determines if the structure in a condition is in conflict
   with the in pattern of a transformation */

int cond_conflict(i,t) int i; trafo_ptr t;
  { if ( (t == NULL) || (t → kind == T_NIL))
      return (charco[i] ≠ '\0');
      else switch(charco[i])
      { case '\0' : return 0;
        case 'd'  : return (((t → kind ≠ T_DO) && (t → kind ≠ T_DOA))
                || (cond_conflict(i+1,t → u.do_pattern.body)) );
        case 'i'  : return ( (t → kind ≠ T_IF) ||
                      (cond_conflict(i+1,t → u.if_pattern.body)) );
        case 'n'  : return cond_conflict(i+1,t → next);
        case 'h'  : return 0;
           default   : printf("*** Corrupt\n"); exit(1);
         }
  }

/* Links to trafo nodes in sequence */

t_link()
  { trafo_ptr t1,t2;

    t2 = t_pop(); t1 = t_pop();
    if (t1 == NULL)
      { printf("Cannot link"); exit(1); }
      else t1 → next = t2;
     t_push(t1);
  }
```

```
/* Procedures to create nodes for particular transformations */

set_true()
  { trafo_ptr newtrafo;

    newtrafo = (trafo_ptr) alloc_mem(sizeof(struct trafo_node));
    newtrafo → kind = T_TRUE;
    newtrafo → next = NULL;
    t_push(newtrafo);
  }

set_cond(p1,p2,p3,depkind,onvar) int p1,p2,p3,depkind,onvar;
  { trafo_ptr newtrafo;

    newtrafo = (trafo_ptr) alloc_mem(sizeof(struct trafo_node));
    newtrafo → kind = T_COND;
    newtrafo → next = NULL;
    newtrafo → u.cond.flags = p1;
    newtrafo → u.cond.sl1 = p2;
    newtrafo → u.cond.sl2 = p3;
    newtrafo → u.cond.on  = onvar;
    switch(depkind)
      { case 1: newtrafo → u.cond.kind = FLOW;   break;
    case 2: newtrafo → u.cond.kind = ANTI;   break;
    case 3: newtrafo → u.cond.kind = OUTPUT; break;
    case 4: newtrafo → u.cond.kind = INPUT;  break;
    default: printf("*** Corrupt\n"); exit(1);
      }
    /* Test structures of both conditions */
    if (cond_conflict(p2,inpattern))
      set_error("First structure in condition does not match pattern");
    if (cond_conflict(p3,inpattern))
      set_error("Second structure in condition does not match pattern");
    t_push(newtrafo);
  }

set_do(kd) char kd;
  { trafo_ptr newtrafo;

    newtrafo = (trafo_ptr) alloc_mem(sizeof(struct trafo_node));
    newtrafo → kind = kd;
    newtrafo → next = NULL;
    newtrafo → u.do_pattern.body = t_pop();
    newtrafo → u.do_pattern.ex3 = t_pop();
    newtrafo → u.do_pattern.ex2 = t_pop();
```

```
    newtrafo → u.do_pattern.ex1 = t_pop();
    newtrafo → u.do_pattern.var = t_pop();
    t_push(newtrafo);
  }

set_if()
  { trafo_ptr newtrafo;

    newtrafo = (trafo_ptr) alloc_mem(sizeof(struct trafo_node));
    newtrafo → kind = T_IF;
    newtrafo → next = NULL;
    newtrafo → u.if_pattern.body = t_pop();
    newtrafo → u.if_pattern.cond = t_pop();
    t_push(newtrafo);
  }

set_assign()
  { trafo_ptr newtrafo;

    newtrafo = (trafo_ptr) alloc_mem(sizeof(struct trafo_node));
    newtrafo → kind = T_ASSIGN;
    newtrafo → next = NULL;
    newtrafo → u.assign_pattern.rhs = t_pop();
    newtrafo → u.assign_pattern.lhs = t_pop();
    t_push(newtrafo);
  }

set_vector(vnr,lpv) int vnr,lpv;
 { trafo_ptr newtrafo;

    if (setting)
      set_error("Vectorize cannot be used at left hand side");
    if ((--vectmode) > 0)
      set_error("Vectorize cannot be used inside another vectorize");
    newtrafo = (trafo_ptr) alloc_mem(sizeof(struct trafo_node));
    newtrafo → kind = T_VECTOR;
    newtrafo → next = NULL;
    newtrafo → u.vec.vecvar = lpv;
    newtrafo → u.vec.var_nr = vnr;
    newtrafo → u.vec.v3 = t_pop();
    newtrafo → u.vec.v2 = t_pop();
    newtrafo → u.vec.v1 = t_pop();
    t_push(newtrafo);
  }

set_merge()
```

```
{ trafo_ptr newtrafo;

   if (setting)
     set_error("Merge cannot be used at left hand side");
   newtrafo = (trafo_ptr) alloc_mem(sizeof(struct trafo_node));
   newtrafo → kind = T_MERGE;
   newtrafo → next = NULL;
   newtrafo → u.merge.l2 = t_pop();
   newtrafo → u.merge.l1 = t_pop();
   t_push(newtrafo);
 }

set_func(entry,arg) int entry,arg;
 { trafo_ptr newtrafo;

   if (setting)
     { if (arg == 0)
         set_error("Scalar variable cannot be used at left hand side");
     else set_error("Intrinsic function cannot be used at left hand side");
     }
   newtrafo = (trafo_ptr) alloc_mem(sizeof(struct trafo_node));
   newtrafo → kind = T_FUNC;
   newtrafo → next = NULL;
   newtrafo → u.func.entry = entry;
   newtrafo → u.func.len = (strlen(&charco[entry]));
   if (arg == 2)
     newtrafo → u.func.arg2 = t_pop();
     else newtrafo → u.func.arg2 = NULL;
   if (arg == 0)
     newtrafo → u.func.arg1 = NULL;
     else newtrafo → u.func.arg1 = t_pop();
   t_push(newtrafo);
 }

set_exprvar(vnr) int vnr;
  { trafo_ptr newtrafo;

   newtrafo = (trafo_ptr) alloc_mem(sizeof(struct trafo_node));
   newtrafo → kind = T_EVAR;
   newtrafo → next = NULL;
   newtrafo → u.var_nr = vnr;
   t_push(newtrafo);
  }

set_const(i,logic) int i,logic;
  { trafo_ptr newtrafo;
```

```
    newtrafo = (trafo_ptr) alloc_mem(sizeof(struct trafo_node));
    if (logic) newtrafo → kind = T_LCONST;
      else newtrafo → kind = T_ICONST;
    newtrafo → next = NULL;
    newtrafo → u.ival = i;
    t_push(newtrafo);
  }

set_rconst(f) double f;
  { trafo_ptr newtrafo;

    newtrafo = (trafo_ptr) alloc_mem(sizeof(struct trafo_node));
    newtrafo → kind = T_RCONST;
    newtrafo → next = NULL;
    newtrafo → u.rval = f;
    t_push(newtrafo);
  }

set_exprkind(ek) char ek;
  { trafo_ptr newtrafo;

    newtrafo = (trafo_ptr) alloc_mem(sizeof(struct trafo_node));
    newtrafo → kind = T_EXP;
    newtrafo → next = NULL;
    newtrafo → u.exp.expr_kind = ek;
    if ((ek == E_UMIN) || (ek == E_NOT))
      newtrafo → u.exp.op2 = NULL;
      else newtrafo → u.exp.op2 = t_pop();
    newtrafo → u.exp.op1 = t_pop();
    t_push(newtrafo);
  }

set_stmtvar(vnr) int vnr;
  { trafo_ptr newtrafo;

    newtrafo = (trafo_ptr) alloc_mem(sizeof(struct trafo_node));
    newtrafo → kind = T_SVAR;
    newtrafo → next = NULL;
    newtrafo → u.var_nr = vnr;
    t_push(newtrafo);
  }

set_nil()
  { trafo_ptr newtrafo;
```

```
    newtrafo = (trafo_ptr) alloc_mem(sizeof(struct trafo_node));
    newtrafo → kind = T_NIL;
    newtrafo → next = NULL;
    t_push(newtrafo);
  }

/* Initializes the parser */

parserinit()
  { trafo_garbage_collect(); tstack_init(); trafo_init();
  }

/* Calls the parser if the filename exists */

parser(name) char *name;
  { line = 1; vectmode = err = 0; parserinit();
    yyin = fopen(name,"r");
    if (yyin == NULL)
      printf("\n*** Bad filename: %s\n",name);
      else { yyparse(); fclose(yyin);
        if (err) parserinit(); }
    return err;
  }
```

S Transformation Application Routines

```
/* Source to Source Compiler
   by Aart J.C. Bik
   Procedures for storing and applying transformations */

#include <stdio.h>
#include "prgtype.h"
#include "trafo.h"

#define TRAFOSIZE 50
#define CHARSIZE  150

#define BODY   'a'
#define DOBODY 'b'
#define IFBODY 'c'

/* Administration Variables */

stmt_ptr  s_v[100],previous,currentstmt,endmarker,restofprg;
trafo_ptr *trafoin,*trafoout,*trafocond;   /* [TRAFOSIZE] */
int       change,tcharpt,trafpt=0,e_def[100];
```

```
int       search=0,vector=0,compstride,vecvar,exception;
expr_ptr  e_v[100],vec1,vec2,vec3;
char      lnorm,*charco; /* [CHARSIZE] */

/* External Variables */

extern int  fsympt,f_storept,assignno,loopnr,condno;
extern char command[];

/* This procedure allocates dynamic memory for the transformations */

trafo_memory()
  { trafoin   = (trafo_ptr *) alloc_mem(TRAFOSIZE * sizeof(trafo_ptr));
    trafoout  = (trafo_ptr *) alloc_mem(TRAFOSIZE * sizeof(trafo_ptr));
    trafocond = (trafo_ptr *) alloc_mem(TRAFOSIZE * sizeof(trafo_ptr));
    charco    = (char      *) alloc_mem(CHARSIZE  * sizeof(char));
  }

/* Resets the number of transformation stored */

trafo_init()
 { tcharpt = trafpt = 0; }

/* Returns the number of transformations stored */

int trafo_num()
  { return trafpt; }

/* Stores a character */

char_insert(c) char c;
  { static int CHARCUR = CHARSIZE;

    if (tcharpt ≥ CHARCUR)
      { CHARCUR += 300;
    charco = (char *) new_mem(charco,CHARCUR * sizeof(char));
      }
    charco[tcharpt++] = c;
  }

/* Stores an in and out pattern */

trafo_insert()
 { static int TRAFCUR = TRAFOSIZE;

   if (trafpt ≥ TRAFCUR)
```

```
    { TRAFCUR += 100;
      trafoin =   (trafo_ptr *) new_mem(trafoin,  TRAFCUR * sizeof(trafo_ptr));
      trafoout =  (trafo_ptr *) new_mem(trafoout, TRAFCUR * sizeof(trafo_ptr));
      trafocond = (trafo_ptr *) new_mem(trafocond,TRAFCUR * sizeof(trafo_ptr));
    }
  trafocond[trafpt] = t_pop();
  trafoout[trafpt]  = t_pop();
  trafoin[trafpt++] = t_pop();
 }

/* Tests if two subscript-lists are equal */

int dimlist_equal(d1,d2) sub_ptr d1,d2;
 { int bool=1;

   while ((bool) && (d1 ≠ NULL))
    { if (d2 == NULL) bool = 0;
    else if (expr_equal(d1 → head,d2 → head))
       { d1 = d1 → tail; d2 = d2 → tail; }
       else bool = 0;
    }
   return ((bool) && (d2 == NULL));
 }

/* Tests if two expressions are equal */

int expr_equal(e1,e2) expr_ptr e1,e2;
 { if (e1 == NULL) return (e2 == NULL);
   else if (e2 == NULL) return (e1 == NULL);
   else switch(e1 → kind)
    { case E_VAR :
       return ((E_VAR == e2 → kind) &&
          (e1 → u.var.entry == e2 → u.var.entry) &&
          (dimlist_equal(e1 → u.var.dim_list,
                e2 → u.var.dim_list)));
      case E_CONST:
       if ( (E_CONST == e2 → kind) &&
        (e1 → u.expr.type == e2 → u.expr.type) )
      switch(e1 → u.expr.type)
         { case INTTYPE:
           return (e1 → u.expr.val.i == e2 → u.expr.val.i);
                 case REALTYPE:
           return (e1 → u.expr.val.f == e2 → u.expr.val.f);
                 case LOGICTYPE:
           return ( (e1 → u.expr.val.b == 0) ==
            ( e2 → u.expr.val.b == 0) );
```

```
                default:
          printf("*** Corrupt\n"); exit(1);
                }
             else return 0;
      case E_UMIN:
        return ((E_UMIN == e2 → kind) &&
           (expr_equal(e1 → u.operands.arg1,
              e2 → u.operands.arg1)) ) ;
      case E_NOT:
        return ((E_NOT == e2 → kind) &&
           (expr_equal(e1 → u.operands.arg1,
              e2 → u.operands.arg1)) ) ;
      default:
        return ((e1 → kind == e2 → kind) &&
           (expr_equal(e1 → u.operands.arg1,
              e2 → u.operands.arg1)) &&
              (expr_equal(e1 → u.operands.arg2,
              e2 → u.operands.arg2)) ) ;
      }
 }

/* The following procedures set variable values if no binding
   has been set and checks is the same binding is used otherwise */

int match_exprvar(e,i) expr_ptr e; int i;
 { if (e_def[i])
    return (expr_equal(e,e_v[i]));
    else { e_v[i] = e; e_def[i] = 1; return 1; }
 }

match_stmtvar(s,i) stmt_ptr s; int i;
 { s_v[i] = s;  /* s_v is only set once */ }

/* This procedure determines if a match with an expression exists */

int match_expr(p1,p2) expr_ptr p1; trafo_ptr p2;
 { if (p2 → kind == T_EVAR) return match_exprvar(p1,p2 → u.var_nr);
   else if (p1 == NULL) return 0;
   else if (p2 → kind == T_ICONST)
      return ( (p1 → kind == E_CONST) && (p1 → u.expr.type == INTTYPE)
&&
         (p1 → u.expr.val.i == p2 → u.ival) );
   else if (p2 → kind == T_RCONST)
      return ( (p1 → kind == E_CONST) && (p1 → u.expr.type == REALTYPE)
&&
         (p1 → u.expr.val.i == p2 → u.rval) );
```

```
    else if (p2 → kind == T_LCONST)
        return ( (p1 → kind == E_CONST) && (p1 → u.expr.type == LOGICTYPE)
                && (  (p1 → u.expr.val.i == 0 ) ==
            (p2 → u.ival == 0) ) );
    else switch(p2 → u.exp.expr_kind)
     { case E_UMIN:
         return ( (p1 → kind == E_UMIN) &&
                  (match_expr(p1 → u.operands.arg1,p2 → u.exp.op1)) );
       case E_NOT:
         return ( (p1 → kind == E_NOT) &&
                  (match_expr(p1 → u.operands.arg1,p2 → u.exp.op1)) );
       default:
         return ( (p1 → kind == p2 → u.exp.expr_kind) &&
              (match_expr(p1 → u.operands.arg1,p2 → u.exp.op1)) &&
              (match_expr(p1 → u.operands.arg2,p2 → u.exp.op2)) );
        }
  }

/* This procedure determines if a match with a kind of do-loop exists */

int match_do(p1,p2,kind) stmt_ptr p1; trafo_ptr p2; char kind;
  { if ( (p1 ≠ NULL) &&  (p1 → kind == K_DO) &&
     (p1 → u.do_loop.ext == kind) &&
     (match_expr(p1 → u.do_loop.index,p2 → u.do_pattern.var)) &&
     (match_expr(p1 → u.do_loop.expr1,p2 → u.do_pattern.ex1)) &&
     (match_expr(p1 → u.do_loop.expr2,p2 → u.do_pattern.ex2)) &&
        (match_expr(p1 → u.do_loop.expr3,p2 → u.do_pattern.ex3)) &&
     (t_match(p1 → u.do_loop.body,p2 → u.do_pattern.body)))
       return 1;
      else return 0;
  }

/* This procedure determines if a match with an assignment exists */

int match_assign(p1,p2) stmt_ptr p1; trafo_ptr p2;
  { if ( (p1 ≠ NULL) && (p1 → kind == K_ASSIGN) &&
        (match_expr(p1 → u.assign.lhs,p2 → u.assign_pattern.lhs)) &&
        (match_expr(p1 → u.assign.rhs,p2 → u.assign_pattern.rhs)) )
      return 1;
      else return 0;
  }

/* This procedure determines if a match with a logical/general if exists */

int match_if(p1,p2) stmt_ptr p1; trafo_ptr p2;
  { if ( (p1 ≠ NULL) &&
```

```
    ((p1 → kind == K_GENIF) || ( p1 → kind == K_LOGICIF)) &&
       (match_expr(p1 → u.s_if.condition,p2 → u.if_pattern.cond)) &&
       (t_match(p1 → u.s_if.body,p2 → u.if_pattern.body)) )
     return 1;
     else return 0;
  }

/* This procedure determines if a patterns matches with a fragment
   from the source program */

int t_match(p1,p2) stmt_ptr p1; trafo_ptr p2;
  { static int active = 0; int busy = 1;

     active++;
     while ((p2 ≠ NULL) && (busy))
       { switch(p2 → kind)
       { case T_DO:     busy = match_do(p1,p2,NORM);
              if (busy) p1 = p1 → next; break;
         case T_DOA:    busy = match_do(p1,p2,ALL);
              if (busy) p1 = p1 → next; break;
         case T_ASSIGN: busy = match_assign(p1,p2);
              if (busy) p1 = p1 → next; break;
            case T_IF:     busy = match_if(p1,p2);
              if (busy) p1 = p1 → next; break;
         case T_SVAR:   if (active == 1)
                 { endmarker = p1;
              if ((p1 == NULL) || (p1 → kind == K_LINKUP)
                 || (p1 → kind == K_LINKIFUP))
                   restofprg = NULL;
                   else restofprg = p1;
                              match_stmtvar(restofprg,p2 → u.var_nr);
                            } else match_stmtvar(p1,p2 → u.var_nr);
                         break;
         case T_NIL:    busy = (int)
                ( (p1 == NULL) || (p1 → kind == K_LINKUP)
                   || (p1 → kind == K_LINKIFUP) );
                         if ((busy) && (active == 1))
                 { endmarker = p1; restofprg = NULL; }
              break;
         default:       printf("*** Corrupt\n"); exit(1);
           }
         p2 = p2 → next;
       } active−−; return busy;
  }

/* Determines binding of condition structure */
```

```
stmt_ptr cond_slist(s,i,hd) stmt_ptr s; int i, *hd;
  { while (charco[i] ≠ '\0')
      switch (charco[i++])
          { case 'n' : s = s → next; break;
            case 'd' : s = s → u.do_loop.body; break;
            case 'i' : s = s → u.s_if.body; break;
        case 'h' : (*hd) = 1; break;
            default  : printf("Corrupt\n"); exit(1);
          }
    return s;
  }

/* Determines if all the conditions hold */

int t_cond(s,cond) stmt_ptr s; trafo_ptr cond;
  { stmt_ptr s1,s2; int h1=0,h2=0;

    if (cond == NULL)
      return 1;
      else switch(cond → kind)
        { case T_TRUE : return t_cond(s,cond → next);
          case T_COND :
            s1 = cond_slist(s,cond → u.cond.sl1,&h1);
            s2 = cond_slist(s,cond → u.cond.sl2,&h2);
            return
         ((nodep(s1,s2,h1,h2,cond → u.cond.flags,
             cond → u.cond.on,cond → u.cond.kind)) &&
            (t_cond(s,cond → next)));
          default     : printf("*** Corrupt\n"); exit(1);
            }
  }

/* This procedure shows the matching bock */

show_matchingblock(s) stmt_ptr s;
  { printf("** MATCH ON\n");
    while (s ≠ endmarker)
      { show_stmt(s); s = s → next; /* 1 level */ }
    if ((s ≠ NULL) && (s → kind ≠ K_LINKIFUP) && (s → kind ≠ K_LINKUP))
      printf("      .....\n");
  }

/* Determines the stride of a vector instruction
   if the stride cannot be determined it is set to 0 */
```

```
int det_stride(e) expr_ptr e;
  { int a1,a2;

    if (e ≠ NULL)
      switch(e → kind)
    { case E_VAR:   if (vecvar == e → u.var.entry)
                { copyexpr(vec3); return 1; }
              else { copyexpr(e); return 0; }
      case E_CONST:  make_const(e → u.expr.type,e → u.expr.val);
            return 0;
      case E_MUL:    a1 = det_stride(e → u.operands.arg1);
            a2 = det_stride(e → u.operands.arg2);
            if ((a1 + a2) == 2) compstride = 1;
            make_expr(E_MUL,INTTYPE);
            return (a1 + a2 > 0);
      case E_ADD:    a1 = det_stride(e → u.operands.arg1);
            a2 = det_stride(e → u.operands.arg2);
            if (((a1 + a2) == 2) || (a1 + a2 == 0))
              make_expr(E_ADD,INTTYPE);
            else if (a1) del_expr(e_pop());
            else { expr_ptr save;
              save = e_pop(); del_expr(e_pop());
              e_push(save); }
                    return ( (a1 + a2) > 0);
      case E_MIN:    a1 = det_stride(e → u.operands.arg1);
            a2 = det_stride(e → u.operands.arg2);
            if (((a1 + a2) == 2) || ((a1 + a2) == 0))
              make_expr(E_MIN,INTTYPE);
            else if (a1) del_expr(e_pop());
            else { expr_ptr save;
              save = e_pop(); del_expr(e_pop());
              e_push(save); make_expr(E_UMIN,INTTYPE); }
                    return ( (a1 + a2) > 0);
        case E_UMIN:  a1 = det_stride(e → u.operands.arg1);
          make_expr(E_UMIN,INTTYPE);
          return a1;
        default:      compstride = 1; e_push(NULL); return 0;
      }
     else { e_push(NULL); return 0; }
  }

/* Set vectorizing information */

canvec(t) trafo_ptr t;
  { if ( (e_v[t → u.vec.vecvar] ≠ NULL) &&
     (e_v[t → u.vec.vecvar] → kind == E_VAR) &&
```

```
    (st_returndim(e_v[t → u.vec.vecvar] → u.var.entry) == 0) )
     { vecvar = e_v[t → u.vec.vecvar] → u.var.entry;
   make_newexpr(t → u.vec.v1); vec1 = e_pop();
   make_newexpr(t → u.vec.v2); vec2 = e_pop();
   make_newexpr(t → u.vec.v3); vec3 = e_pop(); vector = 1; }
      else { printf("=> Vector variable is a non scalar\n"); exception = 1; }
    copyexpr(e_v[t → u.vec.var_nr]); vector = 0;
  }

/* Determines if a vector variable is present in an expression */

int present(e) expr_ptr e;
  { if (e ≠ NULL)
      switch(e → kind)
    { case E_VAR:   return (e → u.var.entry == vecvar);
      case E_CONST: return 0;
          case E_VEC:
        if ( (present(e → u.vec.e1)) || (present(e → u.vec.e2)) ||
             (present(e → u.vec.e3)) )
          { exception = 1;
        printf("=> Vector variable occurs in other vector\n"); }
            return 0;
         default: return ( (present( e → u.operands.arg1)) ||
                  (present( e → u.operands.arg2)) );
       }
      else return 0;
   }

/* Copies an expression, vectorizes it if required
   and determines the new normal form */

copyexpr(e) expr_ptr e;
  { sub_ptr  subpt; int sub=0;

   if (e == NULL) e_push(NULL);
   else switch(e → kind)
      { case E_VAR:
      subpt = e → u.var.dim_list;
          while (subpt ≠ NULL)
             { if ((vector) && (search == 0) && (present(subpt → head)))
               { /* Vectorize mode */
                 if (sub == 1)
                   { printf("=> Vector variable occurs more than once\n");
                     exception = 1; }
                 search = 1; copyexpr(subpt → head);
         search = 2; copyexpr(subpt → head);
```

```
                search = compstride = 0; det_stride(subpt → head);
                if (compstride)
                  { union valuerec val;

                    printf("=> Stride cannot be determined\n");
                    exception = 1; del_expr(e_pop());
                    val.i = 0; make_const(INTTYPE,val);
                  }
                make_vector(); sub++;
              }
             else copyexpr(subpt → head);
          subpt = subpt → tail;
            }
          if ((search > 0) && (e → u.var.entry == vecvar))
            if (search == 1)
          { search = vector = 0; copyexpr(vec1); search = vector = 1; }
          else { search = vector = 0; copyexpr(vec2);
             search = 2; vector = 1; }
            else make_var(e → u.var.entry,st_returndim(e → u.var.entry));
          break;
    case E_CONST: make_const(e → u.expr.type,e → u.expr.val); break;
    case E_VEC:   copyexpr(e → u.vec.e1); copyexpr(e → u.vec.e2);
            copyexpr(e → u.vec.e3); make_vector(); break;
    default:      copyexpr(e → u.operands.arg1);
                 if ((e → kind ≠ E_UMIN) && (e → kind ≠ E_NOT))
           copyexpr(e → u.operands.arg2);
            if (search)
          make_expr(e → kind,INTTYPE);
             else make_expr(e → kind,0);
               break;
      }
  }

/* Creates an expression */

make_newexpr(t) trafo_ptr t;
  { union valuerec val;

    if (t == NULL) e_push(NULL);
      else switch(t → kind)
      {  case T_EVAR:
       copyexpr(e_v[t → u.var_nr]); break;
         case T_VECTOR:
       canvec(t); break;
         case T_ICONST:
        val.i = t → u.ival; make_const(INTTYPE,val); break;
```

```
        case T_RCONST:
       val.f = t → u.rval; make_const(REALTYPE,val); break;
        case T_LCONST:
       val.b = t → u.ival; make_const(LOGICTYPE,val); break;
        case T_FUNC:
      { int i,arg=0;
        i = st_insertid(t → u.func.len,&charco[t → u.func.entry]);
        if (t → u.func.arg1 ≠ NULL)
          { arg++; make_newexpr(t → u.func.arg1); }
        if (t → u.func.arg2 ≠ NULL)
          { arg++; make_newexpr(t → u.func.arg2); }
           if ((st_returntype(i) ≠ UNDEF) && (st_returndim(i) ≠ arg))
          {  printf("=> Unconsistent use of variable\n"); exception = 1; }
             else { st_givedim(i,arg); st_givetype(i,dettype(i)); }
         make_var(i,arg);
           break;
      }
        default:
      make_newexpr(t → u.exp.op1);
      if ( (t → u.exp.expr_kind ≠ E_UMIN) &&
       (t → u.exp.expr_kind ≠ E_NOT) )
            make_newexpr(t → u.exp.op2);
         make_expr(t → u.exp.expr_kind,0);
      break;
      }
  }

/* These procedures copy certain kind of statemens */

copy_do(s) stmt_ptr s;
  { copyexpr(s → u.do_loop.index);
    copyexpr(s → u.do_loop.expr1);
    copyexpr(s → u.do_loop.expr2);
    copyexpr(s → u.do_loop.expr3);
    stack(0,s → u.do_loop.index → u.var.entry); /* environment */
    make_newstmt(s → u.do_loop.body);
    unstack(0); /* environment */
    make_loop(0,++loopnr,s → u.do_loop.ext);
  }

copy_assign(s) stmt_ptr s;
  { copyexpr(s → u.assign.lhs);
    copyexpr(s → u.assign.rhs);
    make_assign(++assignno);
  }
```

```
copy_if(k,s) char k; stmt_ptr s;
  { if (k ≠ K_ELSE) copyexpr(s → u.s_if.condition);
    make_newstmt(s → u.s_if.body);
    switch(k)
      {    case K_WHILE  : make_while(++condno); break;  /* WHILE */
    case K_GENIF  : make_generalif(++condno); break;
    case K_LOGICIF: make_logicalif(++condno); break;
    case K_ELSEIF : make_else(1,++condno); break;
    case K_ELSE   : make_else(0,0); break;
    default       : printf("*** Corrupt\n"); exit(1);
      }
  }

/* Creates a new statement list by copying */

make_newstmt(s) stmt_ptr s;
 { stmt_ptr head=NULL,old=NULL;

   while (s ≠ NULL)
      { switch( s → kind )
      { case K_DO:       copy_do(s); s = s → next; break;
        case K_ASSIGN:   copy_assign(s); s = s → next; break;
        case K_LOGICIF:  copy_if(K_LOGICIF,s); s = s → next; break;
        case K_GENIF:    copy_if(K_GENIF,s); s = s → next; break;
        case K_ELSE:     copy_if(K_ELSE,s); s = s → next; break;
        case K_ELSEIF:   copy_if(K_ELSEIF,s); s = s → next;  break;
/* WHILE */ case K_WHILE:    copy_if(K_WHILE,s); s = s → next; break;
        case K_STOP:     make_stop(); s = s → next; break;
        case K_LINKUP:   s_push(NULL); s = NULL; break;
        case K_LINKIFUP: s_push(NULL); s = NULL; break;
        default:         printf("*** Corrupt\n"); exit(1);
      }
      if (head == NULL) old = head = s_pop();
        else old = old → next = s_pop();
       }
    s_push(head);
  }

/* Creates a new DO-loop */

make_gendo(t,ext) trafo_ptr t; char ext;
  { expr_ptr control;

    make_newexpr(t → u.do_pattern.var);
    control = e_pop(); e_push(control); /* TOP computation */
    make_newexpr(t → u.do_pattern.ex1);
```

```
    make_newexpr(t → u.do_pattern.ex2);
    make_newexpr(t → u.do_pattern.ex3);
    /* Check if only scalars are used */
    if ((control ≠ NULL) && (control → kind == E_VAR)
     && (st_returndim(control → u.var.entry) == 0) )
      stack(0,control → u.var.entry); /* environment */
      else { printf("=> Introducing non scalar as loop-control variable\n");
         stack(0,0); exception = 1; }
    comp_newblock(t → u.do_pattern.body);
    if (exception == 0) unstack(0); /* environment */
    make_loop(0,(++loopnr),ext);
  }

/* Computes the resulting fragment */

comp_newblock(t) trafo_ptr t;
  { stmt_ptr head=NULL,old=NULL;
    static   int lv;

    lv++;
    while (t ≠ NULL)
     { switch(t → kind)
     { case T_DO    : make_gendo(t,NORM);  break;
       case T_DOA   : make_gendo(t,ALL); break;
       case T_ASSIGN: make_newexpr(t → u.assign_pattern.lhs);
            make_newexpr(t → u.assign_pattern.rhs);
            make_assign(++assignno); break;
         case T_IF    : make_newexpr(t → u.if_pattern.cond);
            comp_newblock(t → u.if_pattern.body);
            /* Note: always a general IF */
            make_generalif(++condno); break;
         case T_MERGE : comp_newblock(t → u.merge.l1);
            lv--; comp_newblock(t → u.merge.l2); lv++;
            { stmt_ptr s1,s2,scan,prev=NULL;
              s2 = s_pop(); scan = s1 = s_pop();
              while (scan ≠ NULL)
              { prev = scan; scan = scan → next; }
                        if (prev == NULL)
               s_push(s2);
               else { prev → next = s2; s_push(s1); }
                      } break;
         case T_NIL   : s_push(NULL); break;
       case T_SVAR  : if ((s_v[t → u.var_nr] == restofprg) && (lv == 1))
            s_push(restofprg);
            else make_newstmt(s_v[t → u.var_nr]);
            break;
```

```
      default      : printf("*** Corrupt\n"); exit(1);
        }
       if (head == NULL) old = head = s_pop();
     else old = (old → next = s_pop());
       t = t → next;
    }
    lv−−; s_push(head);
  }

/* Shifts the statement pointer one statement ahead */

next_stmt()
  { /* currentstmt != NULL holds */
    switch(currentstmt → kind)
     { case K_LINKUP:   unstack(0);  /* environment */
            previous = currentstmt → next; lnorm = BODY;
            currentstmt = currentstmt → next → next; break;
       case K_LINKIFUP: previous = currentstmt → next; lnorm = BODY;
            currentstmt = currentstmt → next → next; break;
       case K_DO:       stack(0,currentstmt → u.do_loop.index → u.var.entry);
            /* environment */
            previous = currentstmt; lnorm = DOBODY;
            currentstmt = currentstmt → u.do_loop.body; break;
       case K_LOGICIF:  previous = currentstmt; lnorm = IFBODY;
            currentstmt = currentstmt → u.s_if.body; break;
       case K_GENIF:    previous = currentstmt; lnorm = IFBODY;
            currentstmt = currentstmt → u.s_if.body; break;
       case K_WHILE:    previous = currentstmt; lnorm = IFBODY; /* WHILE
*/
            currentstmt = currentstmt → u.s_if.body; break;
       default:         previous = currentstmt; lnorm = BODY;
            currentstmt = currentstmt → next; break;
     }
  }

/* Applies the transformation */

apply(s) stmt_ptr s;
  { stmt_ptr garbage=NULL;

    change++;
    if (previous == NULL)
      { s_push(s); create_program(); }
      else switch(lnorm)
             { case BODY:   previous → next = s; break;
               case DOBODY: previous → u.do_loop.body = s; break;
```

```
            case IFBODY: previous → u.s_if.body = s;
                    previous → kind = K_GENIF; break;
            default    : printf("*** Corrupt\n"); exit(1);
        }
    /* Reset the pointers */
    /* detach old fragment */
    if ((currentstmt ≠ NULL) && (currentstmt ≠ endmarker))
      { garbage = currentstmt;
        while (currentstmt → next ≠ endmarker)
            currentstmt = currentstmt → next;
        currentstmt → next = NULL; }
    /* find new settings */
    currentstmt = endmarker;
    while ((currentstmt ≠ NULL) && (currentstmt ≠ restofprg)
         && (currentstmt → kind ≠ K_LINKUP)
           && (currentstmt → kind ≠ K_LINKIFUP) )
       currentstmt = currentstmt → next;
    if ( (currentstmt ≠ NULL) && ((currentstmt → kind == K_LINKUP) ||
      (currentstmt → kind == K_LINKIFUP)) )
      { previous = currentstmt → next;
        link_up(previous,currentstmt → kind);
        /* environment */
    if (currentstmt → kind == K_LINKUP)
      unstack(0);
        currentstmt = previous → next; lnorm = BODY;
      }
      else { int scan = 1;

          if (previous == NULL)
       { if (give_program() ≠ currentstmt)
            previous = give_program();
            else scan = 0;  }
             else if (lnorm == IFBODY)
       { if (previous → u.s_if.body ≠ currentstmt)
           previous = previous → u.s_if.body;
           else scan = 0; }
             else if (lnorm == DOBODY)
       { if (previous → u.do_loop.body ≠ currentstmt)
           previous = previous → u.do_loop.body;
           else scan = 0; }
             if (scan)
              { while (previous → next ≠ currentstmt)
                      previous = previous → next;
                 lnorm = BODY; }
           }
/* Deletes old statementlist */
```

```
 del_stmtlist(garbage);
}

/* Deletes new statementlist */

reject(s) stmt_ptr s;
  { stmt_ptr garbage;

    if ( (s ≠ NULL) && (s ≠ restofprg))
      { garbage = s;
        while ((s → next ≠ restofprg) && (s → next ≠ NULL))
          s = s → next;
    s → next = NULL;
    del_stmtlist(garbage);
      }
  }

/* Shows the resulting fragment and gets the user response */

int accept_it()
  { stmt_ptr s,show;

    printf("** TRANSFORM INTO\n"); show = s = s_pop();
    while ((show ≠ NULL) && (show ≠ restofprg))
      { show_stmt(show); show = show → next;  /* level 1 */ }
    if (show ≠ NULL) printf("      ......\n");
    if (exception == 1)
      { reject(s); return 0; }
    else { printf("** ACCEPT (y/n/q/e) =="); readcommand();
           if ((strcmp(command,"y") == 0) || (strcmp(command,"Y") == 0))
             { apply(s); return 1; }
             else if ((strcmp(command,"q") == 0) || (strcmp(command,"Q") == 0))
               { reject(s); return 2; }
             else if ((strcmp(command,"e") == 0) || (strcmp(command,"E") == 0))
               { reject(s); return 3; }
             else
               { reject(s); return 0; }
           }
  }

/* Performs all the transformations on the current program */

start_app(dmp) int dmp;
  { register int i,j;  int res,save[5],tbusy=1,localchange;

 open_program(); change = 0;
```

```
while (tbusy)
  { for (i = 0; i < trafpt; i++)
    { printf("--------- Transformation %i -----------\n",i+1);
      previous = NULL; localchange = 0;
      currentstmt = give_program(); fstacks_init();
      while (currentstmt ≠ NULL)
        { for (j = 0; j < 100; j++) e_def[j] = 0;
      save[0] = loopnr; save[1] = assignno; save[2] = condno;
      save[3] = fsympt; save[4] = f_storept; exception = 0;
          if (t_match(currentstmt,trafoin[i]) &&
          t_cond(currentstmt,trafocond[i]))
            { show_matchingblock(currentstmt);
      comp_newblock(trafoout[i]); res = accept_it();
       switch (res)
        { case 0: next_stmt(); break;
                  case 1: localchange++; break;
                  case 2: currentstmt = NULL; break;
                  case 3: currentstmt = NULL; tbusy = 0; i = trafpt; break;
                 }
           if (res ≠ 1)
           { loopnr = save[0]; assignno = save[1]; condno = save[2];
          fsympt = save[3]; f_storept = save[4]; }
             }
            else next_stmt();
        }
         if (localchange > 0)
        datadep_analysis(give_program());
    }
    if (localchange > 0)
      printf("\n******** New Pass ********\n\n");
      else tbusy = 0;
   }

 printf("\nNumber of transformations applied: %i\n\n",change);
 if (change > 0)
   { dump_program(1,"program.txt"); dep_dump(dmp);
    fortran_symboltable(); }
}
```

T Environment

```
/* Source to Source Compiler
   by Aart J.C. Bik
   C-program — Interactive environment    */

#include <stdio.h>
```

```
/* The Interactive environment variables */

#define MAXSTRING 160
char    command[MAXSTRING]; /* Two lines in 80 columns */
char    *argument;
int     warn,showdep;

/* Initializes variables of the main program and calls
   the initializing functions of the parsers          */

init_program()
  { warn = 0;             /* Default: warnings off */
    showdep = 3;
    parsers_memory();     /* Allocates dyn.mem. for the two parsers */
    dep_memory();         /* Allocated dyn.mem. for dep.storage */
    st_memory();          /* Allocates dyn.mem. for symbol-table */
    trafo_memory();       /* Allocates dyn.mem. for trafo-storage */
  }

/* Reads the user defined transformations if the filename exists  */

read_trafos()
  { if (parser(argument)) printf("\nTerminated\n\n");
      else printf("\n");
  }

/* Reads in the FORTRAN program */

read_program()
  {  if (f_parser(argument,warn,showdep)) printf("\nTerminated\n\n");
      else printf("\n");
  }

/* Saves the FORTRAN program in memory */

save_prg()
  { if (prg_defined())
      { dump_program(0,argument); printf("\n"); }
      else printf("\nNo program to save\n\n");
  }

/* Compiles the source code using transormations read */

start()
  { if (prg_defined())
```

```
    { if (trafo_num())
      start_app(showdep);
      else printf("\nNo transformations in memory\n\n"); }
      else printf("\nNo program present to transform\n\n");
   }

/* Calls the editor vi  */

edit()
  { char call_vi[MAXSTRING];

    strcpy(call_vi,"vi "); strcat(call_vi,argument); system(call_vi);
  }

/* Changes the warning level */

warningon() { warn = 1; printf("\nWarnings enabled\n\n"); }

nowarnings() { warn = 0; printf("\nWarnings disabled\n\n"); }

/* Changes the mode of showing dependences */

change_dep()
  { showdep = showdep++;
    if (showdep > 3) showdep = 0;
    switch(showdep)
    { case 0: printf("\nAll dependences\n\n"); break;
      case 1: printf("\nOnly Flow, Anti and Output dependences\n\n"); break;
      case 2: printf("\nOnly dependences on Assignment statements\n\n"); break;
      case 3: printf(
"\nOnly Flow, Anti and Output dependences on Assignment statements\n\n");
              break;
    }
 }

/* USER-Communication Procedures OUTPUT  */

welcome()
  { printf("\n*********************************************************\n");
    printf("*** FORTRAN 77 to FORTRAN 90 Restructuring Compiler ***\n");
    printf("*** Written by Aart J.C. Bik                    ***\n");
    printf("*** under supervision of dr. H.A.G. Wijshoff      ***\n");
    printf("*** UNIVERSITY OF UTRECHT 1991/1992             ***\n");
    printf("*********************************************************\n\n");
    printf("'help' shows available commands\n\n");
  }
```

```
bye()
  { printf("\n************\n");
    printf("*** Bye! ***\n");
    printf("************\n\n");
  }

show_options()
    { printf("\nAvailable Commands are:\n\n");
      printf("  - exit\n");
      printf("  - help\n");
      printf("  - dep\n");
      printf("  - nowarnings\n");
      printf("  - readprg <filename.f>\n");
      printf("  - readtrf <filename>\n");
      printf("  - save <filename>\n");
      printf("  - showdep\n");
      printf("  - showprg\n");
      printf("  - symtb\n");
      printf("  - start\n");
      printf("  - vi <filename>\n");
      printf("  - warnings\n\n");
    }

show_st()
  { printf("\nSymbol table:\n\n");
    if (st_number() > 0)
      system("less program.sym");
      else printf("Empty\n");
    printf("\n");
  }

show_prg()
   { if (prg_defined())
       system("less program.txt");
       else printf("\nNo Program in memory\n\n");
   }

show_dep()
   { if (dep_defined())
       system("less program.dep");
       else printf("\nNo Dependences\n\n");
   }

/* USER-Communication Procedures INPUT */
```

```
readcommand()
    { register int  i,j;
      char      ch;

     printf("=> ");
     /* Skip space en tabs first */
     do  ch=getchar(); while ( ch == ' ' || ch == '\t' );
     ungetc(ch,stdin); i = 0;
     do { ch = getchar();
          if (ch == '\b')
            { if (i > 0) i--; }
            else if (i < MAXSTRING) command[i++] = ch;
        } while (ch ≠ '\n'); i--; command[i] = '\0';
     /* Find possible argument */
     argument = &command[i];
     for (j = 0;j < i;j++) if (command[j] == ' ')
        { command[j++] = '\0';
          while ((j < i) &&  ((command[j] == ' ') || (command[j] == '\t'))) j++;
          argument = &command[j]; break;
        }
 }

/* Main module (Interactive mode)
   Reads in and executes a user command   */

main()
   { welcome();
     init_program();
     while (1)
      { readcommand();
    if (strcmp(command,"help") == 0) show_options();
    else if (strcmp(command,"start") == 0) start();
    else if (strcmp(command,"readprg") ==0) read_program();
    else if (strcmp(command,"readtrf") == 0) read_trafos();
    else if (strcmp(command,"save") == 0) save_prg();
    else if (strcmp(command,"showprg") == 0) show_prg();
    else if (strcmp(command,"showdep") == 0) show_dep();
    else if (strcmp(command,"symtb") == 0) show_st();
    else if (strcmp(command,"vi") == 0) edit();
    else if (strcmp(command,"exit") == 0) { bye();exit(0); }
    else if (strcmp(command,"nowarnings") == 0) nowarnings();
    else if (strcmp(command,"dep") == 0) change_dep();
    else if (strcmp(command,"warnings") == 0) warningon();
    else if (strcmp(command,"") == 0) ;
    else printf("\n*** Unknown Command: %s\n\n",command);
      }
```

```
}
```