Proceedings of Conference on Pattern ... Inquiry ... Systems
SIGPLAN NOTICES
12(8):163-173 Aug 1977

PROGRAM MANIPULATION VIA AN EFFICIENT PRODUCTION SYSTEM*

D.F. Kibler, J.M. Neighbors and T.A. Standish
Department of Information and Computer Science
University of California at Irvine
Irvine, California 92717

Introduction

Systems for program transformation have been suggested by many authors [Knuth, Wegbreit, Loveman, Balzer, Standish2, Burstall & Darlington]. Several of these authors note that such transformation systems could be used to transform lucid, high-level, but possibly inefficient program descriptions into efficient but possibly less legible underlying concrete realizations. One problem, mentioned by Loveman [Loveman] is that of chaining together many low-level simple transformations to achieve high-level goals. For example, we may wish to chain together low-level transformations such as constant propagation, performing arithmetic at transformation time, dead variable elimination, empty statement removal and the like, to achieve the high-level goal of program simplification. This paper shows one technique for organizing sequences of low-level program transformations within an interactive programming medium to achieve nearly automatic global program improvements with low search times and minimal human intervention and guidance.

The work reported here deals with the development of a transformation system based on a production system (Burstall & Darlington and Loveman have implemented transformation systems but the detailed mechanisms are unpublished). To keep the system within a manageable size, we have not incorporated the techniques of global flow analysis and algebraic simplification although they would fit into the framework of a production based program manipulation system. The paper consists of two major sections and three appendices. The first section discusses the use of production systems and the second section shows how an interactive program manipulation system was implemented using a production system. The first appendix is an example worked out in detail, the second appendix contains a sample of the transformations in the system, and the third appendix contains example outputs from the system.

*This work was supported by the National Science Foundation under Grant DCR75-13875.

Production Systems

A pure production system [Davis1] consists of a data base and a collection of productions of the form LHS -> RHS where LHS and RHS are strings. If some LHS matches a portion of the data base then the matched segment is replaced by a substituted copy of RHS. Both LHS and RHS are allowed to contain pattern variables, i.e. variables which are bound by the matching of LHS and instantiated in RHS. Production systems typically involve two extensive searches - a search of the data base and a search of the space of productions. A method for avoiding these searches, distinct from the metarules of Davis [Davis2] and the filtering techniques of McDermott and Newell [McDermott], is a technique we call chaining. The basic idea behind chaining is that the successful firing of a production contains information about what to do next. We know where the production was applied and what the possible effects of the transformation were. The knowledge of where the last production was applied defines a limited scope or locality over which we may search. The knowledge of which production fired, and its context, provides an index into the space of productions. Both these forms of knowledge can be derived automatically by the system.

Let us assume, for the sake of concreteness, that the data base has the structure of a tree. Further assume that we have a production system where each pure production has been augmented by a list of directions. Each direction consists of a specification of a new locality in the tree followed by a list of productions that might be applicable in that locality. Since the data base is constantly changing we cannot give an

absolute designation of the new locality, rather, a relative direction must be given. For the tree structured data bases a direction might appear as (HERE P1 P2 P3) or (UP P2) where HERE means the current locality and UP means back up the tree one level to the enclosing locality. The successful firing of a production causes the first direction to be tried. The other items are held on an agenda. The agenda is a tree which expands and contracts as the productions are fired. At each possible locality in the data base a corresponding node in the agenda specifies a list of productions to be attempted in that locality. The effect of the agenda is to allow the most recently attached direction to be followed first. For the specific case of a tree structured data base the agenda might be a list of elements each of which represents a locality encountered in a depth first search of the tree. When the agenda is empty, i.e chaining is completed, control reverts back to that of a pure production system.

Automatic Derivation of Directions

The directions for a particular new production may be determined by comparing its syntactic structure with those productions already known to the system. The idea is that if the RHS of a production P1 matches a subtree of the LHS of production P2 with pattern variables taken into account then the direction (UP P2) is added to the direction list of P1. Similarly, if the LHS of a production P1 matches a subtree of the RHS of production P2 with pattern variables taken into account then the direction (HERE P1) is added to the direction list of P2. The following example of an augmented production system is abstracted from our program manipulation system (see footnote). Capital letters denote pattern variables and small letters denote constants.

depends on the depth one up may not be enough

	LHS	*RHS*
P1:	(A (a B C D) E)	(a B (A C E) (A D E))
P2:	(b c X)	(c)
P3:	(d c X)	(X)

Directions

P1:	(HERE P2 P3) (UP P1)
P2:	(UP P2 P3)
P3:	(UP P2 P3)

This example was abstracted from the following productions:
P1: (<Op> (if Bool X Y) Z)
-> (if Bool (<Op> X Z) (<Op> Y Z))
P2: (times 0 X) -> (0)
P3: (add 0 X) -> (X).

For a large set of productions the addition of a production may be costly, but the cost is only incurred once at the time of addition. Notice that the addition of a production only incrementally affects the directions so that an entire recomputation of the directions is unnecessary.

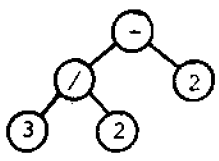
The augmentation of a production system does not interfere with the spirit of production systems. It has the effect of dynamically altering the order in which the rules are applied and represents an alternative mechanism for strategy, which, unlike metarules, does not require explicit declaration.

The Program Manipulation System

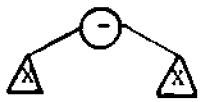
If a program manipulation system contains enough power to do the usual boolean identities, then the problem of transforming a program to its 'best' form is at least as hard as the problem of boolean satisfiability, which is NP-complete. Hence, our transformation system consists almost entirely of transformations which locally improve the code. Despite this restriction placed on our transformations, the system still achieves global improvements by chaining together local changes and is still powerful enough to simplify many programs (see appendix 3). The augmented production system discussed in the previous section is a very natural way to support certain basic program manipulation tasks . The productions are program transformations and the data base is the program to be transformed. In our LISP implementation of a production system driven program manipulation system the program is stored as a parse tree to avoid precedence, scoping and reparsing problems. For example, if one does purely syntactic matches on strings then the transformation

P4: X-X=> 0

yields incorrect results when applied to forms like 3/2-2. Our pattern variables match against subtrees of parse tree so that



would not match the LHS of P4 which is



Here the symbol  denotes the full subtree and not just a single node.

Program transformations are different from augmented productions in that some useful transformations may occur only if certain conditions hold on the pattern variables at the time of application. As an example consider the following transformation.

```
c*Y=c*Z => Y=Z
```

The transformation is certainly useful but it can only be performed if c is a nonzero constant. This condition is called an enabling condition [Standish1] and must be checked in addition to matching the LHS of the program transformation. Enabling conditions can also be used as a strategy mechanism in that some transformations can be characterized as sacrificing space for speed etc. and the enabling conditions can reference global parameters set by the user specifying the constraints under wnich his program is to be modified.

Some program transformations are not amenable to the LHS -> RHS style of presentation. Examples of such transformations are useless assignment elimination, constant propagation and empty statement elimination [Standish1]. These transformations are hand written in our system as procedures but they can still have enabling conditions, directions and sometimes a LHS. The following example shows a procedure oriented transformation with a LHS.

wrong say S is
I:=6
I:=I+1

LHS: FOR I:= c TO c DO S
RHS: a procedure which substitutes constant c for every "right hand" occurrence of I in S.

The directions for the procedure oriented transformations must be user specified except for those with an LHS for which some directions can be automatically produced.

Our initial program manipulation system incorporates six procedural transformations and forty-four syntactic transformations similar in scale and scope to those given in the Irvine Program Transformation Catalogue [Standish1]. The initial test problem was the simplification of a matrix multiplication program under different assumptions about the form of the matrices, e.g. symmetric, diagonal, triangular. The program to be manipulated is the following.

```
FOR I:=1 STEP 1 UNTIL N DO
    FOR J:=1 STEP 1 UNTIL N DO
        BEGIN
         C[I,J]:=0;
         FOR K:=1 STEP 1 UNTIL N DO
           C[I,J]:=C[I,J]+A[I,K]*B[K,J];
        END;
```

If matrix A is asserted to be a symmetric matrix (A[X,Y]=A[Y,X]) the system produces the following program in approximately 10 seconds.

```
FOR I:=1 STEP 1 UNTIL N DO
    FOR J:=1 STEP 1 UNTIL N DO
        BEGIN
         C[I,J]:=0;
         FOR K:=1 STEP 1 UNTIL I-1 DO
           C[I,J]:=C[I,J]+A[K,I]*B[K,J];
         FOR K:=I STEP 1 UNTIL N DO
           C[I,J]:=C[I,J]+A[I,K]*B[K,J];
        END;
```

Notice that the resulting code requires only half of matrix A to be stored. The above example is worked in detail in Appendix I. If matrix A is asserted to be the identity matrix the system produces the following program in approximately 30 seconds.

```
FOR I:=1 STEP 1 UNTIL N DO
    FOR J:=1 STEP 1 UNTIL N DO
        C[I,J]:=B[I,J];
```

Once one has proved that the transformations in the system preserve correctness, the transformations which produced the above program constitute a proof by program transformation that multiplying a matrix by the identity matrix simply results in a copy of the original matrix.

Currently the directions attached to each transformation are arranged in no particular order. The ordering is clearly important in that it specifies which transformation is to be tried first and which are to be tried later. Many approaches to ordering this list for more efficient searches of the space of transformations have been suggested [McDermott, Davis2]. As an example the list could be ordered by any of the following means:

1) by the expense of performing the transformation,
2) by the expense of trying the transformation,
3) by the probability that the transformation will fire,
4) by user intuition,
5) by a hybrid of the above methods.

Automatic methods of ordering the decision lists are desirable since it enables a naive user to enter a transformation

without worrying about the internals of the system. We envision the ordering process to be dynamic with the transformation system having the ability, in effect, to reject a suggested transformation by ordering it so far back in the direction lists in which it appears that it is seldom attempted. The system should modify the order in which to attempt transformations from experience with actual programs.

Conclusions

A production system based interactive program manipulation system was built and has proven to be efficient and flexible for our investigations. The results reported here have dealt only with transformations from one language into itself. Investigation of more powerful transformations which can synthesize a program from a high-level description of that program in a restricted problem domain is the next logical step in this work.

References

[Balzer]
Balzer, R., Goldman, N. and Wile, D.
On the Transformational Implementation Approach to Programming
2nd Intl. Software Engr. Conf.
Oct. 1976, San Francisco, pp. 337

[Burstall]
Burstall, R. and Darlington, J.
A Transformation System for Developing Recursive Programs
JACM Volume 24, Number 1
pp 44-67 January 1977

[Davis1]
Davis, R. and King, J.
An Overview of Production Systems
Technical Report STAN-CS-75-524
Stanford Computer Science Department
October 1975

[Davis2]
Davis, R.
Applications of Meta Level Knowledge to the Construction, Maintenance and Use of Large Knowledge Bases
Technical Report STAN-CS-76-552
Stanford Computer Science Department July 1976

[Knuth] Knuth, D.
Structured Programming with GOTO Statements
C. Surveys Volume 6, Number 4
pp261-301 December 1974

[Loveman]
Loveman, D.
Program Improvement by Source-to-Source Transformation
JACM Volume 24, Number 1
pp 121-145 January 1977

[McDermott]
McDermott, D., Newell, A. and Moore, J.
The Efficiency of Certain Production System Implementations
Technical Report Department of Computer Science
Carnegie-Mellon University September 1976

[Standish1]
Standish, T., Harriman, D., Kibler, D. and Neighbors, J.
The Irvine Program Transformation Catalogue
Computer Science Department
University of California at Irvine January 1976

[Standish2]
Standish, T., Kibler, D. and Neighbors, J.
Improving and Refining Programs by Program Manipulation
Proceedings of the 1976 ACM National Conference
pp 509-516 Houston, Texas October 1976

[Wegbreit]
Wegbreit, B.
Goal-directed Program Transformation
IEEE Transactions on Software Engineering
SE-2, 2 pp270-285 June 1976

Appendix I - Detailed Example

In this appendix we will show exactly how our system performs its manipulation of a program. For clarity the discussion in the paper was somewhat simplified. Each production consists of a four-tuple: (enabling conditions, LHS, RHS, and directions). Directions may be of the simple type discussed in the paper, but usually they consist of a call to a procedure LOOK-AT which examines a decision tree and returns a list of transformations which might be applicable in the current locality. The input to LOOK-AT is an operator or control construct keyword and its operands. For example the transformation

```
(if B then C else D)*E
=>(if B then C*E else D*E)
```

has the directions (HERE LOOK-AT(*,C,E))(HERE LOOK-AT(*,D,E)) and (UP LOOK-AT(HIOP,HIOPNS)) where HIOP stands for the operator which would be above the IF if the transformation were performed and HIOPNS stands for the operands. Recall that the basic flow of control is to try to apply the transformation where specified in the agenda. The successful firing of a transformation results in possible additions to the agenda by means of the directions attached to each transformation.

We will now step through the first example mentioned in the paper. To see the complete form of the transformations applied consult Appendix 2. We start with the initial conditions of the agenda being empty and the locality enclosing the entire matrix multiply program.

```
agenda= empty
locality= FOR I:=1 STEP 1 UNTIL N DO
           FOR J:=1 STEP 1 UNTIL N DO
             BEGIN
              C[I,J]:=0;
              FOR K:=1 STEP 1 UNTIL N DO
                 C[I,J]:=C[I,J]
                 +A[I,K]*B[K,J];
            END;
```

Since the agenda is empty, control is left with the user. He asserts that matrix A is symmetric. Consulting with a small base of stored facts the system constructs the transformation

```
a[X,Y] => if X≤Y then a[X,Y] else a[Y,X]
```

which it names substitution. Remember the convention that upper case letters in transformations represent pattern variables. The only direction on the substitution transformation is (UP LOOK-AT(HIOP,HIOPNS)). The substitution transformation is placed on the agenda as (HERE SUBSTITUTION). The recommendation on the agenda is tried and removed from the agenda. The locality focuses on the point in the program where the transformation succeeded. In the case of the above substitution the locality just before the transformation succeeded was A[I,K]. Afterwards it becomes IF I≤K THEN A[I,K]ELSE A[K,I]. In this locality HIOP=* and HIOPNS=(IF I≤K THEN A[I,K]ELSE A[K,I],B[K,J]). The call LOOK-AT(HIOP,HIOPNS) returns the name of a transformation DISTIF<OP> which reflects a decision to distribute the multiplication over the IF statement. The direction (UP DISTIF<OP>) is attached to the agenda. Thus at the end of the firing of the substitution transformation we have the following agenda and locality.

```
agenda= (UP DISTIF<OP>)
locality= IF I≤K THEN A[I,K]ELSE A[K,I]
```

SUB

Since the agenda is not empty the system continues to apply transformations.

The UP changes the locality to

```
(IF I≤K THEN A[I,K]ELSE A[K,I])*B[K,J]
```

and DISTIF<OP> successfully fires. The directions on DISTIF<OP> (see Appendix 2) cause LOOK-AT to be called three times.

```
(HERE LOOK-AT(*,A[I,K],B[K,J]))
(HERE LOOK-AT(*,A[K,I],B[K,J]))
(UP LOOK-AT(+,C[I,J],IF I≤K
THEN A[I,K]*B[K,J]ELSE A[K,J]*B[K,J]))
```

The first two calls to LOOK-AT do not find any useful transformations and result in nothing being added to the agenda. The third call results in (UP DIST<OP>IF) being added to the agenda reflecting the decision to distribute the + over the IF statement. Thus at the end of the firing of the DISTIF<OP>IF transformation we have the following agenda and locality.

```
agenda= (UP DIST<OP>IF)
locality= IF I≤K THEN A[I,K]*B[K,J]
                 ELSE A[K,I]*B[K,J]
```

The form and directions for each transformation are given in Appendix 2. From now on we will just show changes in the locality and the agenda.

```
locality= IF I≤K THEN A[I,K]*B[K,J]ELSE A[K,I]*B[K,J]        1 look ahead

                    ⇓ UP

locality= C[I,J]+(IF I≤K THEN A[I,K]*B[K,J]ELSE A[K,I]*B[K,J])

                    ⇓ DIST<OP>IF

locality= IF I≤K THEN C[I,J]+A[I,K]*B[K,J]ELSE C[I,J]+A[K,I]*B[K,J]   2 look ahead
agenda= (UP DIST<OP>IF)

                    ⇓ UP

locality= C[I,J]:=(IF I≤K THEN C[I,J]+A[I,K]*B[K,J]
                          ELSE C[I,J]+A[K,I]*B[K,J])


                    ⇓ DIST<OP>IF

locality= IF I≤K THEN C[I,J]:=C[I,J]+A[I,K]*B[K,J]                  3 look ahead
                  ELSE C[I,J]:=C[I,J]+A[K,I]*B[K,J]
agenda= (UP HILEQSPLIT)

                    ⇓ UP

locality= FOR K:= 1 step 1 UNTIL N DO
            IF I≤K THEN C[I,J]:=C[I,J]+A[I,K]*B[K,J]
                   ELSE C[I,J]:=C[I,J]+A[K,I]*B[K,J]

                    ⇓ HILEQSPLIT

locality= IF 1<I &I<N                                               4 LOOKAHEAD
             THEN BEGIN
                   FOR K:=1 STEP 1 UNTIL I-1 DO
                       C[I,J]:=C[I,J]+A[K,I]*B[K,J];
                   FOR K:= I STEP 1 UNTIL N DO
                       C[I,J]:=C[I,J]+A[I,K]*B[K,J];
                  END
             ELSE
               IF NOT I<N
                 THEN FOR K:=1 STEP 1 UNTIL N DO
                           C[I,J]:=C[I,J]+A[K,I]*B[K,J]
                 ELSE FOR K:=1 STEP 1 UNTIL N DO
                           C[I,J]:=C[I,J]+A[I,K]*B[K,J]

agenda= (TOP FORSMARTS1)(TOP FORSMARTS2)

                    ⇓ TOP

locality= FOR I:=1 STEP 1 UNTIL N DO
            FOR J:=1 STEP 1 UNTIL N DO
              BEGIN
                C[I,J]:=0;
                IF 1<I & I<N
                   THEN BEGIN
                        FOR K:=1 STEP 1 UNTIL I-1 DO
                            C[I,J]:=C[I,J]+A[K,I]*B[K,J];
                        FOR K:=I STEP 1 UNTIL N DO
                            C[I,J]:=C[I,J]+A[I,K]*B[K,J];
                        END
                  ELSE
                    IF NOT I<N
                       THEN FOR K:=1 STEP 1 UNTIL N DO
                                 C[I,J]:=C[I,J]+A[K,I]*B[K,J]
                       ELSE FOR K:=1 STEP 1 UNTIL N DO
                                 C[I,J]:=C[I,J]+A[I,K]*B[K,J]
               END;
```

The procedural transformation FORMARTS1 builds a transformation for each FOR it encounters

transformation asserts that in the body of the FOR statement the lower bound of the FOR statement is less than or equal to the variable of iteration.

In the current case only 1≤I=>TRUE does anything

⇓ 1≤I=>TRUE

5th transformation operates

```
locality= IF TRUE & I<N
            THEN BEGIN
                   FOR K:=1 STEP 1 UNTIL I-1 DO
                       C[I,J]:=C[1,J]+A[K,I]*B[K,J];
                   FOR K:=I STEP 1 UNTIL N DO
                       C[I,J]:=C[I,J]+A[I,K]*B[K,J];
                 END
            ELSE
              IF NOT I<N
                 THEN FOR K:=1 STEP 1 UNTIL N DO
                          C[I,J]:=C[I,J]+A[K,I]*B[K,J]
                 ELSE FOR K:=1 STEP 1 UNTIL N DO
                          C[I,J]:=C[I,J]+A[I,K]*B[K,J]
```

agenda= (HERE ANDTX)

Note that the (TOP FORSMARTS2) is still on the agenda for the previous locality. The most recent things added to the agenda are done first

⇓ ANDTX

```
locality= IF I<N
            THEN BEGIN
                   FOR K:=1 STEP 1 UNTIL I-1 DO
                     C[I,J]:=C[I,J]+A[K,I]*B[K,J];
                   FOR K:=I STEP 1 UNTIL N DO
                     C[I,J]:=C[I,J]+A[I,K]*B[K,J];
                 END
            ELSE
              IF NOT I<N
                 THEN FOR K:=1 STEP 1 UNTIL N DO
                          C[I,J]:=C[I,J]+A[K,I]*B[K,J]
                 ELSE FOR K:=1 STEP 1 UNTIL N DO
                          C[I,J]:=C[I,J]+A[J,K]*B[K,J]
```

the transformation adds nothing to the agenda for the current locality.

The agenda for the current locality is empty so it starts enlarging the locality until it finds a locality with something to do. If the agenda entry for each possible locality in the program is empty then control reverts back to the user. In the current case the only agenda entry is (TOP FORSMARTS2) which was placed on the agenda by HILEQSPLIT. FORSMARTS2 is similar to FORSMARTS1 except that it asserts that the variable of iteration is less than or equal to the upper loop bound.

agenda= (TOP FORSMARTS2)

⇓ TOP

```
locality= FOR I:=1 STEP 1 UNTIL N DO
            FOR J:=1 STEP 1 UNTIL N DO
              BEGIN
                C[I,J]:=0;
                IF I<N
                   THEN BEGIN
                          FOR K:=1 STEP 1 UNTIL I-1 DO
                              C[I,J]:= C[I,J]+A[K,I]*B[K,J];
                          FOR K:=I STEP 1 UNTIL N DO
                              C[I,J]:=C[I,J]+A[I,K]*B[K,J];
                        END
                   ELSE
                      IF NOT I<N
                         THEN FOR K:=1 STEP 1 UNTIL N DO
```

```
                    C[I,J]:=C[I,J]+A[K,I]*B[K,J]
                  ELSE FOR K:=1 STEP 1 UNTIL N DO
                    C[I,J]:=C[I,J]+A[I,K]*B[K,J]
          END;
```

⇓ FORSMARTS2
(only I≤N => TRUE does anything)
$\frac{1}{2}$ substitutions

```
locality= IF TRUE
          THEN BEGIN
                 FOR K:=1 STEP 1 UNTIL I-1 DO
                   C[I,J]:=C[I,J]+A[K,I]*B[K,J];
                 FOR K:=I STEP 1 UNTIL N DO
                   C[I,J]:=C[I,J]+A[I,K]*B[K,J];
               END
          ELSE
            IF NOT TRUE
              THEN FOR K:=1 STEP 1 UNTIL N DO
                     C[I,J]:=C[I,J]+A[K,I]*B[K,J]
              ELSE FOR K:=1 STEP 1 UNTIL N DO
                     C[I,J]:=C[I,J]+A[I,K]*B[K,J]

agenda= (HERE IFTRUEELSE)(HERE NOTT)
```

⇓ IFTRUEELSE

```
locality= BEGIN
            FOR K:=1 STEP 1 UNTIL I-1 DO
              C[I,J]:=C[I,J]+A[K,I]*B[K,J];
            FOR K:=I STEP 1 UNTIL N DO
              C[I,J]:=C[I,J]+A[I,K]*B[K,J];
          END
```

The (HERE NOTT) left on the agenda does not fire. All agenda entries are empty and control reverts to the user. The original program has been transformed to the one shown below

```
        FOR I:=1 STEP 1 UNTIL N DO
          FOR J:=1 STEP 1 UNTIL N DO
            BEGIN
              C[I,J]:=0;
              BEGIN
                FOR K:=1 STEP 1 UNTIL I-1 DO
                  C[I,J]:=C[I,J]+A[K,I]*B[K,J];
                FOR K:=I STEP 1 UNTIL N DO
                  C[I,J]:=C[I,J]+A[I,K]*B[K,J];
              END;
            END;
```

Notice that the resulting program only requires half of matrix A to be stored.

Appendix 2 - Transformations Used in Appendix 1

In the following list of transformations, capital letters denote pattern variables. The transformations are listed in the order in which they are applied in Appendix 1. Enabling conditions have not been included.

```
name: substitution
pattern: {determined by constructing routine}
directions: (UP LOOK-AT(HIOP,HIOPNS))

name: DISTIF<OP>
pattern: (if X then Y else Z)<op>W => if X then Y<OP>W else Z<OP>W
directions: (HERE LOOK-AT(<OP>,Y,W))(HERE LOOK-AT(<OP>,Z,W))(UP LOOK-AT(HIOP,HIOPNS))

name: DIST<OP>IF
pattern: W<OP>(if X then Y else Z)=> if X then W<OP>Y else W<OP>Z
directions: (HERE LOOK-AT(<OP>,W,Y))(HERE LOOK-AT(<OP>,W,Z))(UP LOOK-AT(HIOP,HIOPNS))

name: HILEQSPLIT
pattern: for W:= X step 1 until Y do(if Z≤W then S else R) =>
            if X≤Z & Z≤Y then begin
                                for W:=X step 1 until Z-1 do R;
                                for W:=Z step 1 until Y do S
                              end
                         else
                          if not Z≤Y then for W:=X step 1 until Y do R
                                     else for W:=X step 1 until Y do S
directions: (TOP FORSMARTS1)(TOP FORSMARTS2)(UP LOOK-AT(HIOP,HIOPNS))

name: FORSMARTS1
action: a procedure which asserts that the lower bound of a loop is less
        than or equal to  the variable of iteration for all loops
        in the locality.
directions: (UP LOOK-AT(HIOP,HIOPNS))

name: ANDTX
pattern: true & X=>X
directions: (UP LOOK-AT(HIOP,HIOPNS))

name: FORSMARTS2
action: a procedure which asserts that the variable of iteration is
        less than or equal to the upper bound for all loops in the locality
directions: (UP LOOK-AT(HIOP,HIOPNS))

name: IF TRUEELSE
pattern: if true then X else Y => X
directions: (UP LOOK-AT(HIOP,HIOPNS))

name: NOTT
pattern: not true => false
directions: (UP LOOK-AT(HIOP,HIOPNS))
```

Appendix 3 - Transformation System Examples

The following examples demonstrate the capability of the prototype transformation system. In all these examples the system begins with the standard program for multiplying matrices and modifies the code according to the user's specification. Each example is performed in about 30 sec running on a PDP-10 in time-sharing mode. Usually about twenty transformations are required and the user makes about one suggestion.

Original matrix multiplication program:

```
FOR I:=1 STEP 1 UNTIL N DO
    FOR J:=1 STEP 1 UNTIL N DO
    BEGIN
       C[I,J]:=0;
       FOR K:=1 STEP 1 UNTIL N DO
           C[I,J]:=C[I,J]+A[I,K]*B[K,J];
    END;
```

Example 1. Matrix A is diagonal.

resulting code:
```
FOR I:=1 STEP 1 UNTIL N DO
    FOR J:=1 STEP 1 UNTIL N DO
        C[I,J]:=A[I,I]*B[I,J];
```

remark: Notice that the innermost loop is removed.

Example 2. Matrix A and B are diagonal.

resulting code:
```
FOR I:=1 STEP 1 UNTIL N DO
BEGIN
  FOR J:=1 STEP 1 UNTIL I-1 DO C[I,J]:=0;
  C[I,I]:=A[I,I]*B[I,I];
  FOR J:=I+1 STEP 1 UNTIL N DO C[I,J]:=0;
END;
```

remark: Notice N multiplications and no additions are required in the resulting code.

Example 3. Matrix A is the identity.

resulting code:
```
FOR I:=1 STEP 1 UNTIL N DO
   FOR J:=1 STEP 1 UNTIL N DO
       C[I,J]:=B[I,J];
```

remark: In a sense we have a proof by program manipulation that multiplying a matrix by the identity constitutes just copying the original matrix.

Example 4. Matrix A is lower triangular.

resulting code:
```
FOR I:=1 STEP 1 UNTIL N DO
   FOR J:1 STEP 1 UNTIL N DO
   BEGIN
     C[I,J]:=0;
     FOR K:=I STEP 1 UNTIL N DO
         C[I,J]:=C[I,J]+A[I,K]*B[K,J];
   END;
```

remark: A lower triangular matrix is one with zero entries above the main diagonal.

Example 5. Matrix A and B are triangular.

resulting code:
```
FOR I:=1 STEP 1 UNTIL N DO
  FOR J:=1 STEP 1 UNTIL N DO
```

```
BEGIN
  C[I,J]:=0;
  FOR K:=I STEP 1 UNTIL J DO
        C[I,J]:=C[I,J]+A[I,J]*B[K,J]
END;
```

remark: Analysis shows [Standish2] that this program has six times the running speed of the original. Also, the accessing of A and B requires that only half of these matrices need be stored.