
# Generation of Minimal Cut Sets Using LISP

Y. Watanabe

November 1986

UWFDM-710

FUSION TECHNOLOGY INSTITUTE

UNIVERSITY OF WISCONSIN

MADISON WISCONSIN

# Generation of Minimal Cut Sets Using LISP

Y. Watanabe

Fusion Technology Institute
University of Wisconsin
1500 Engineering Drive
Madison, WI 53706

http://fti.neep.wisc.edu

November 1986

UWFDM-710

# Generation of Minimal Cut Sets Using LISP

Yoichi Watanabe

Fusion Technology Institute
Department of Nuclear Engineering
and Engineering Physics
University of Wisconsin-Madison
1500 Johnson Drive
Madison, WI 53706

November 1986

UWFDM-710

# Contents

# Abstract

A LISP program has been developed to find minimal cut sets of fault trees for reliability analysis. Nelson's algorithm with factoring is implemented in the program. The program is used with the symbolic manipulation program REDUCE to perform complete reliability evaluations.


# 1 Introduction

A minimal cut set (MCS) is defined as follows: "A cut set of a system is defined as a set of system events that, if they all occur, will cause system failure. $\cdots$ A minimal cut set of a system is a cut set consisting of system events that are not a subset of the events of any other cut set" [1].

Knowledge of minimal cut sets is useful because

(a) It simplifies the calculation of the probability of the top event. Suppose that a set of MCSs, $C$, is given by

$$C = \{C_i, i = 1, 2, ..., M \mid C_i \text{ is an MCS}\} \tag{1}$$

Then the probability of the top event occurrence $P(T)$ is approximately given by

$$P(T) = \sum_{i=1}^{M} P(C_i) \tag{2}$$

This equation is a good approximation if the probabilities of intersections of MCSs are small and it is usually true for risk assessment.

(b) It tells us which set of events is the most important from the reliability point of view; eliminate the MCSs with large probabilities to increase reliability.

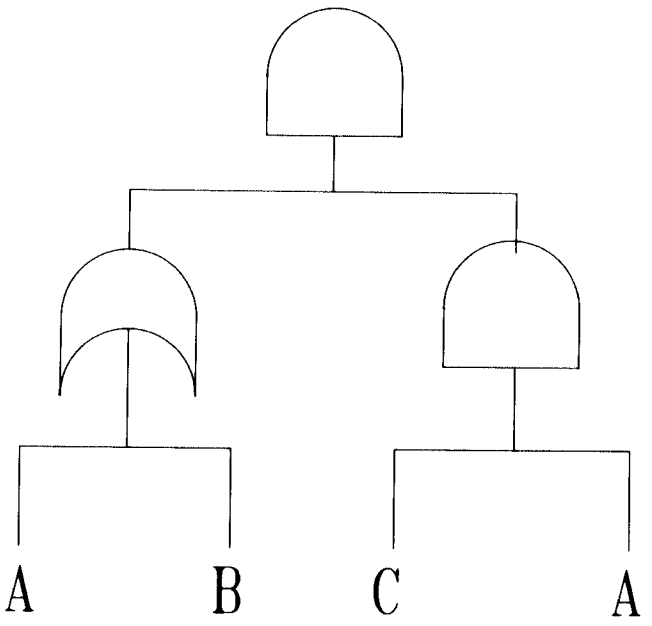


Figure 1: Sample fault tree

(c) The MCSs are subproducts of minimization of a Boolean expression, which is needed to eliminate identical events and complementary events so that the probability calculation can be performed. For example, let us consider the fault tree shown in Figure 1. The Boolean expression of the fault tree is given by

$$T = (A + B) * (C * A) \tag{3}$$

where $+$ and $*$ denote union and intersection, respectively. Applying appropriate Boolean algebraic rules, this expression can be rewritten as

$$T = A * C \tag{4}$$

where $\{A, C\}$ is an MCS of the fault tree. The probability of occurrence of $T$ is now given by

$$P(T) = P(A)P(C) \tag{5}$$

On the other hand, we must know the probabilities of occurrences of intersections of events A,B, and C to compute $P(T)$ from Eq. (3).

Many methods as well as computer programs have been developed to find MCSs. In reactor safety study PREP[2], MOCUS[3], WAM-CUT[4], ALLCUTS[5], TREE-MICSUP[6], PL-MOD[7], and SETS[8] programs are being used as shown in Table 10-1 of Ref. [1]. Computer scientists and electrical engineers along with mathematical logisticians have devised algorithms for minimization of Boolean expressions to design simpler electric circuits consisting of the minimum number of AND and OR (or NAND, NOT) gates. See Ref. [9] for a review.

PREP uses a combinatorial method; it chooses some number of events and sees if the occurrence of these events leads to the occurrence of the top event and searches for all possible combinations. It first finds cut sets, then selects only minimal cut sets from those. Suppose that there are N events. Then there are $2^N - N$ possible combinations. Thus for large fault trees this method is very inefficient.

MOCUS uses a top-down algorithm to construct MCSs and the algorithm works only for trees with AND, OR, and INHIBIT gates. WAM-CUT generalizes the MOCUS algorithm to include NOT gates.

SETS and most electrical engineers use a direct manipulation method of Boolean expressions. Mathematically the MCSs are equivalent to the prime impricants (PIs) of a Boolean expression. A sum of PIs is not always the minimum from the evaluation point of view. "A sum of prime implicants of p is called irredundant if it has the smallest possible number of product expressions amongst all those sums of prime implicants, which is equivalent to p"[10]. Finding the irredundant form is called minimization.

Electrical engineers are concerned about the minimization; meanwhile, reliability and safety scientists need a sum of PIs. The SETS code uses Nelson's algorithm with factoring to find PIs. For the minimization the most famous is the Quine-McCluskey algorithm. Since the direct manipulation of Boolean expressions, in particular for large fault trees is extremely time consuming on computers, much work has been devoted to find faster algorithms.

The present report discusses an implementation of Nelson's algorithm for finding MCSs. The program is written in the Portable Standard LISP

(PSL)[11] on Cray-1 and XMP computers at NMFECC. Although the algorithm can be written in FORTRAN as SETS, LISP was used. The reason is that the resulting expressions will be used in the REDUCE program, which is written in PSL, for further analysis and symbolic/numerical evaluation.

# 2 Nelson's Algorithm

Nelson's algorithm is explained by using a sample expression, which is also used in Ref. [12]:

$$F = A * B + \bar{B} * C * D + \bar{D} * C \tag{6}$$

where $\bar{B}$ is a complement (negation) of $B$, $+$ and $*$ denote union (or disjunction) and intersection (or conjunction), respectively. The algorithm is as follows:

**STEP 1:** Complement $F$, expand $\bar{F}$ into disjunctive normal form, drop zero product ($p * p = 0$), repeated literals ( $p * p = p$), and subsuming products ( $p + p * q = p$) and call the result $\bar{\Phi}$:

$$F = (\bar{A} + \bar{B}) * (B + \bar{C} + \bar{D}) * (D + \bar{C}) \tag{7}$$

$$\bar{\Phi} = \bar{A} * B * D + \bar{A} * \bar{C} + \bar{B} * \bar{C} \tag{8}$$

**STEP 2:** Complement $\bar{\Phi}$, expand $\Phi$ into a disjunctive normal form, drop zero products, repeated literals, and subsuming products, and call the result $\Sigma$:

$$\Phi = (A + \bar{B} + \bar{D}) * (A + C) * (B + C) \tag{9}$$

$$\Sigma = A * B + A * C + \bar{B} * C + \bar{D} * C \tag{10}$$

Thus the MCSs of the expression (6) are $\{A, B\}, \{A, C\}, \{\bar{B}, C\}$, and $\{\bar{D}, C\}$. Hulme and Worrell proposed to factor F and $\Phi$ before complementing in order to decrease the number of terms in the expanded expressions[12]. The timing comparison is given in the reference. The new algorithm significantly reduces the cost. However, it is noted that the CPU time on a CDC 6600 computer for the fault tree with 87 PIs (or MCSs) is about 7 minutes.

# 3 Nelson's Algorithm with Factoring in LISP

A LISP program has been developed by using Nelson's algorithm with factoring. This section describes the procedure and LISP functions defined. In the program a prefix form is used to express Boolean expressions. In the prefix form the + operator is denoted by "disj" and the $*$ operator is denoted by "conj". For example, $A + B * C$ in the infix form is converted to (disj $A$ (conj $B$ $C$)) in the prefix form.

**INF-TO-PRE:** converts an infix expression to a prefix expression.

**PRE-TO-INF:** converts a prefix expression to an infix expression.

**NEG:** complements an expression, defining $\bar{\text{p}}$ = (NEG p).

**OP:** expands a conjunctive form to obtain a disjunctive form.

**REARRANGE:** rearranges the disjunctive form so that the prefix expression does not contain consecutive disj operators.

**SIMPLIFY:** simplifies a Boolean expression.

**SIMPL1 :** drops repeated conjunctive literals: e.g. (conj p p) = p.

**SIMPL2 :** drops zero products and 0s; e.g. (conj p (NEG p)) = 0 and (conj p 0) = 0.

**SIMPL3 :** drops repeated disjunctive literals; e.g. (disj a a) = a.

**SIMPL4 :** drops a pair of complements; e.g. (disj a (NEG a)) = 1.

**SIMPL5 :** drops subsuming products; e.g. (disj a (conj a b)) = a.

**CONTRACT3:** drops zeros from a conjunctive form.

**CONTRACT4:** drops zeros from a disjunctive form.

**TEST1,2,3,4:** tests the type of the argument.

**FACTORING:** factors an expression.

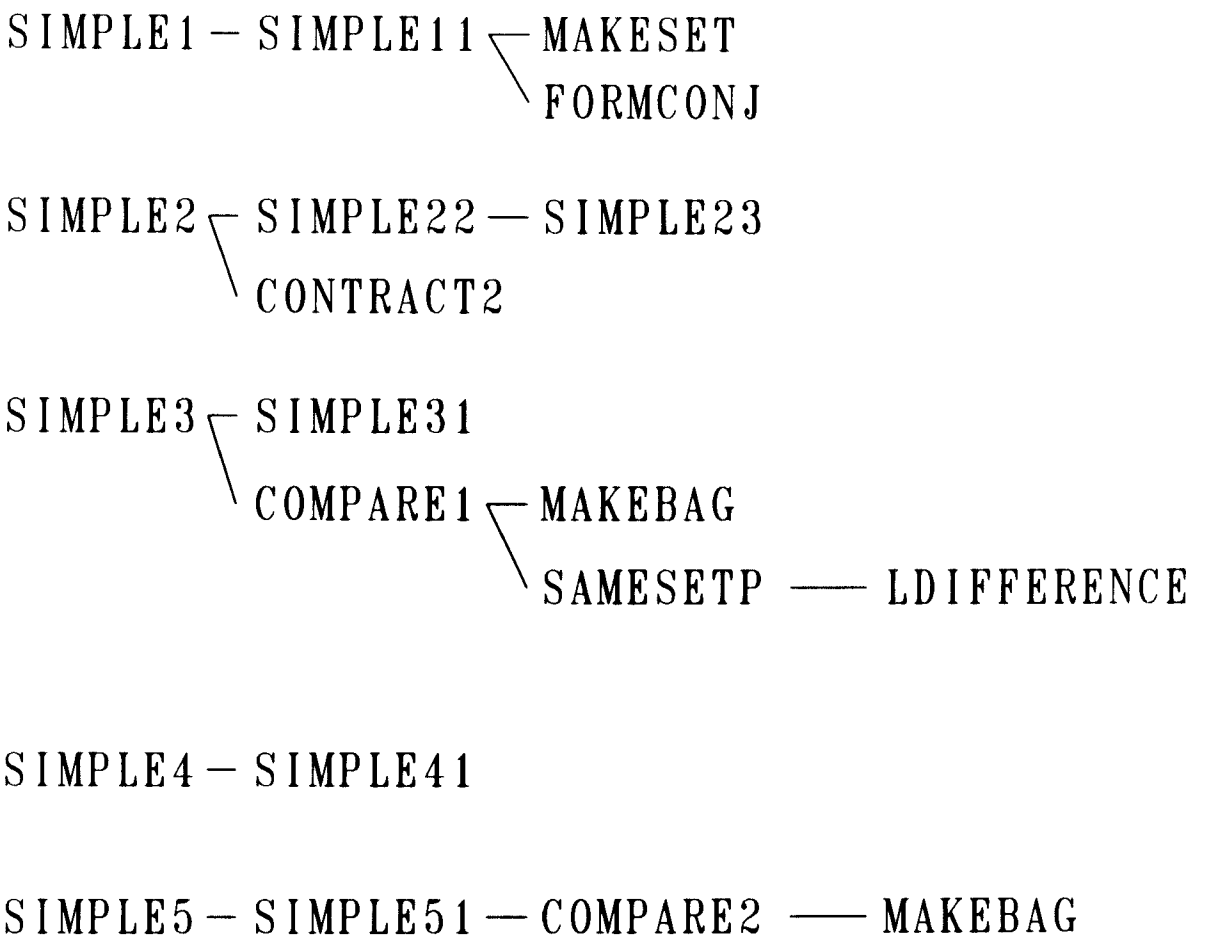


Figure 2: Structure of SIMPLIFY

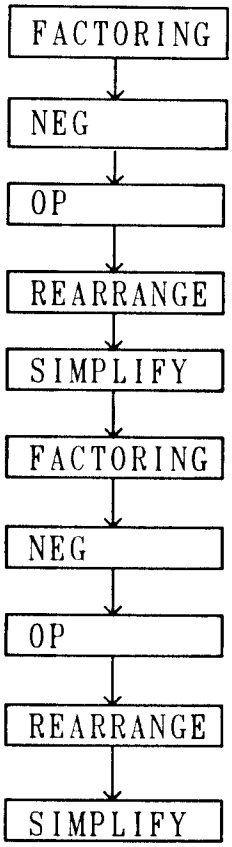


Figure 3: Computational structure for Nelson's method

Fig. 2 shows the sub-functions used by the functions SIMPL1 to 5. The procedure of manipulation using the functions described above is illustrated in Fig. 3.

# 4 Examples

The following problems have been solved by using the LISP program. Table 1 shows the central processing unit time (CPU) on a CRAY-1. The problems are rather small, but Example 4 already exhausted the five minute time limit and gave no result. Most of the time is consumed for the garbage collection to manage the computer memory. Unless a better garbage col-

Table 1: Computing time for examples

| Example No. | No. of MCSs | CPU time in seconds |
|---|---|---|
| 1 | 3 | 1.3 |
| 2 | 4 | 6.2 |
| 3 | 6 | 22.1 |
| 4 | 6 | did not finish |

(1) CRAY-1 CPU time.
(2) "did not finish" in 5 minutes.

lector is introduced in the LISP program, the current approach is not applicable to large fault trees which frequently contain hundreds of MCSs.

**Example 1: Ref.[10], p.53**

$$x\bar{y}z + \bar{x}y\bar{z} + xy\bar{z} + xyz \tag{11}$$

**Example 2**

$$abde + ab\bar{e} + a\bar{c}\bar{d} + \bar{a}\bar{d} + bc\bar{d} + \bar{b}\bar{c}\bar{d} \tag{12}$$

**Example 3: Ref.[13]**

$$a(b + h + c)(b + e + d(f + c + g)) \tag{13}$$

**Example 4: Ref.[10], p.53**

$$\bar{w}\bar{x}\bar{y}\bar{z}+\bar{w}\bar{x}y\bar{z}+\bar{w}x\bar{y}z+\bar{w}xy\bar{z}+\bar{w}xyz+w\bar{x}\bar{y}\bar{z}+w\bar{x}yz+wx\bar{y}z+wxyz+wxy\bar{z} \tag{14}$$

# 5 Conclusions

Nelson's algorithm has been used to write a LISP program for finding the minimal cut sets of fault trees. A disadvantage with this approach to generate MCSs is the computing time; for bigger problems the computing time increases significantly because of an increase of the number of disjunctive terms. (See Ref.[14] for a different approach.) It is necessary to find an algorithm suited to the LISP language. There are some advantages, however; for example, the data on interdependence of events can be stored in the LISP frame and used during the Boolean manipulation. The entire LISP program was incorporated in a REDUCE run sequence. The program for REDUCE, FINDMCS, as well as sample input files and a COSMOS batch run file are given in Appendix A.

# Acknowledgement

The work was supported by the Office of Magnetic Fusion in the U.S. Department of Energy.

# References


[1] N.J. McCormick, "Reliability and Risk Analysis", Academic Press, New York (1981).

[2] W.E. Vesely and R.E. Narum, "PREP and KITT: Computer Codes for the Automatic Evaluation of a Fault Tree," USAEC Rep. IN-1349. Idaho Nuclear Corporation (August 1970).

[3] J.B. Fussel, E.B. Henry, and N.H. Marshall, "MOCUS-A Computer Program to Obtain Minimal Sets from Fault Trees," US-AEC Rep. ANCR-1156, Aerojet Nuclear Company (August 1974).

[4] R.C. Erdmann, F.L. Leverenz, and H. Kirch, "WAMCUT, A Computer Code for Fault Tree Evaluation," EPRI-NP-803 (June 1978).

[5] W.J. Van Slyke and D.E. Griffing, "ALLCUTS-A Fast Comprehensive Fault Tree Analysis Code," ERDA Rep. ARH-ST-112, Atlantic Richfield Hanford Company (July 1978).

[6] P.K. Pande, M.E. Spector, and P. Chatterjee, "Computerized Fault Tree Analysis: TREEL and MICSUP," Rep. ORC-75-3(AD-A010 146), Operations Research Center, Univ. of California Berkeley California (April 1975).

[7] J. Olmos and L. Wolf, "A Modular Approach to Fault Tree and Reliability Analysis," Rep. MITNE-209, Department of Nuclear Engineering, Massachusetts Institite of Technology (August 1977).

[8] R.E. Worrell and D.W. Stack, "A SETS User's Manual for the Fault Tree Analysis," Rep. SAND-77-2051, Sandia National Laboratory (November 1978).

[9] U.T. Rhyne, P.S. Noe, and M.H. McKinney, "A New Technique for the Fast Minimization of Switching Functions," IEEE Trans. Computers, vol.C-26 (1978) 757-763.

[10] R. Lidl and G. Pilz, "Applied Abstract Algebra," Springer-Verlag, New York (1985).

[11] The Utah Symbolic Computation Group, "The Portable Standard LISP Users Mannual, Version 3.2," (April 1984) available as a computer document at NMFECC.

[12] B.L. Hulme and R.B. Worrell, "A Prime Implicant Algorithm with Factoring," IEEE Trans. Computers, vol.C-24 (1975) 1129-1131.

[13] D.M. Rasmuson and N.H. Marshall, "FATRAM - A Core Efficient Cut Set Algorithm," IEEE Trans. Reliability, R-27, 250-253 (1978).

[14] D.J. Finnicum and P.W. Pzasa, "LTREE-A-LISP Based Algorithm for Cutset Generation using Boolean Reduction," Int. ANS/ENS Topical Meeting on Probabilistic Safety Methods and Applications, San Francisco, Feb. 24-March 1, 1985.

## Appendix A.1 COSMOS input file to run FINDMCS

```
 1  %cosmos input for REDUCE runs with FINDMCS
 2  *select printlog=cosout
 3  *filem rds 1751 .availphi mcslib
 4  end
 5  *lib mcslib
 6  x all.
 7  next cutset
 8  x all.
 9  end
10  *reduce
11  in "findmcs"$
12  *tedi opt22
13  cfa1,*,33
14  wr;fort
15  end
16  *rcft i=fort,x=xfort
17  *xfort
18  %end of cosmos run
```

## Appendix A.2: Program FINDMCS

```
comment FINDMCS version 1.0
        This program is an input for REDUCE and used to
        find minimal cutsets and evaluate the top event
        probabilities with known event probabilities.
        A Fortran program "FNCT" is written;
comment needs two input files
        INP : includes necessary parameters,
        INP2: includes the original Boolean expression;
off echo;
operator x,y,r;
share zzf;
array inno(50),itp(50),vote(50),id(20,50);
array a(100),ind(100);
%perform Boolean minimization via Nelson's algorthm
symbolic;
 (lapin "inp2");
 (lapin "working");
 zzf:= (nelson2 ff);
 algebraic;
 y(1):=zzf;
%end of cutset operation
algebraic;
%read parameters necessary for the rest of manipulation
in "inp";
mm:=1;
%process results
%save structure function for later use
off nat;out funct;
write "off exp$";
write "operator x,y$";
for j:=1:mm do
 <<yy:=y(j);write " y(",j,"):=",yy>>;
write "end";
```

```
34  shut funct;on nat;
35  comment ****************************************;
36  comment  The following will create a FORTRAN program;
37  % requires event probabilities as inputs
38  cardno!*:=20;
39  fortwidth!*:=72;
40  on fort;
41  off echo;
42  out "fnct";
43  write "c--------------------------------";
44  write "      subroutine fnct(x,y)";
45  write "      real x(100),y(100)";
46  for n:=1 step 1 until mm do begin
47       ddd:=y(n);
48  off period;
49  write ddd;
50  write "      y(",n,")=ans";
51  on period;
52    end;
53  write "      return";
54  write "      end";
55  write "c--------------------------------";
56  write "      subroutine derv(x,dd)";
57  write "      dimension x(100),dd(100)";
58  for n:=1:nn do begin
59   ddd:=df(y(1),x(n));
60   off period;
61   write ddd;
62   write "       dd(",n,")=ans";
63   on period;
64  end;
65  write "      return";
66  write "      end";
67  shut "fnct";
68  off fort;on echo;out t;
69  % time used in milisecond
70  showtime;
71  bye;
```

## Appendix A.3: Input file for FINDMCS

```
1  %input for FINDMCS
2  off echo$
3  nn:=3$
4  aa:=x(1)$
5  ab:=x(2)$
6  ac:=x(3)$
7  end$
8  %end of inp
```

## Appendix A.4: Input expression for FINDMCS

```
%expression to be minimized
(setq ff '(disj (conj aa ab)(conj aa ac)))
```

## Appendix A.5: Fortran program added

```
c**************************************
c  STRCT: computes the system unavailability
c         by using the structure function
c--------------------------------------
c23456
      dimension x(100),y(100),cri(100),dd(100)
      namelist/par/ncomp
c
      call link("unit5=(i,open),unit6=(o,create,text)//")
c
      read(5,par)
c
      do 10 i=1,ncomp
       read(5,*) x(i)
 10   continue
c
      call fnct(x,y)
      call derv(x,dd )
      do 40 i=1,ncomp
 40   cri(i)=dd(i)*x(i)/y(1)
c
      write(6,901)
      do 50 i=1,ncomp
      write(6,902) i,x(i),dd(i),cri(i)
  50  continue
      write(6,905) y(1)
c
      call exit
901   format(10h comp. no.,2x,"unavailability",4x,
     "BIRNBAUM",14x,"Critical")
902   format(2x,i5,4x,3f12.6)
905   format(//" system unavailability = ",f10.4//)
      end
```

## Appendix B: LISP program

```
%Winston&Horn(1981) Chapter 11
%from infix form to prefix form
(de weight (operator)
    (cond ((equal operator 'dummy) -1)
          ((equal operator '=) 0)
          ((equal operator '+) 1)
          ((equal operator '-) 1)
          ((equal operator '*) 2)
          ((equal operator '//) 2)
          ((equal operator 'rm) 2)
          ((equal operator '**) 3)
      (t (print (list operator 'not 'an 'operator)) 'nop)))
(de opcode (operator)
    (cond ((equal operator 'dummy)(print 'hit-dummy) 'dummy)
          ((equal operator '=) 'setq)
          ((equal operator '+) 'disj)
          ((equal operator '-) 'difference)
          ((equal operator '*) 'conj)
          ((equal operator '//) 'quotient)
          ((eqaul operator 'rm) 'remainder)
          ((equal operator '**) 'expt)
      (t (print (list operator 'not 'an 'operator)) 'nop)))
(de inf-to-pre (ae)
    (prog (operands operators)
          (cond ((atom ae) (return ae)))
          (setq operators (list 'dummy))
           stuff
          (cond ((null ae)
                 (return 'unexpected-end)))
        (setq operands (cons (cond((atom (car ae))(car ae))
                                  (t (inf-to-pre (car ae))))
                             operands)
              ae (cdr ae))
           scan
          (cond ((and (null ae)
                      (equal (car operators) 'dummy))
                 (return (car operands))))
          (cond ((or (null ae)
                     (not (greaterp (weight (car ae))
                                    (weight (car operators)))))
                (setq operands
```

```
42                        (cons (list (opcode (car operators))
43                                    (cadr operands)
44                                    (car operands))
45                              (cddr operands))
46                        operators (cdr operators))
47                  (go scan))
48                 (t (setq operators (cons (car ae)
49                                          operators)
50                          ae (cdr ae))
51                  (go stuff)))))
52 %from prefix to infix (Answer of problem 12-2)
53 (de opsymbol (x)
54     (cond ((equal x 'setq) '=)
55           ((equal x 'disj) '+)
56           ((equal x 'difference) '-)
57           ((equal x 'conj) '*)
58           ((equal x 'quotient) '//)
59           ((equal x 'remainder) 'rm)
60           ((equal x 'expt) '^)
61           (t x)))
62  (de precedence (x)
63      (cond ((equal x 'setq) 0)
64            ((equal x 'disj) 1)
65            ((equal x 'difference) 1)
66            ((equal x 'conj) 2)
67            ((equal x 'quotient) 3)
68            ((equal x 'remainder) 3)
69            ((equal x 'expt) 4)
70            (t 5)))
71 (de pre-to-inf (l win)
72     (prog (wout)
73           (cond ((null l)(return l))
74                 ((atom l)(return (list l))))
75           (setq wout (precedence (car l)))
76           (return (cond ((lessp wout win)
77                          (list
78                       (append (pre-to-inf (cadr l) wout)
79                         (append (list (opsymbol (car l)))
80                            (pre-to-inf (caddr l) wout)))))
81                      (t (append (pre-to-inf (cadr l) wout)
82                           (append (list (opsymbol (car l)))
83                           (pre-to-inf (caddr l) wout))))))))
84 %cutset1
85 %define negation operator
```

```
(de neg (ae)
    (cond ((equal ae 1) 0)
          ((equal ae 0) 1)
          ((atom ae) (list 'neg ae))
          ((equal (car ae) 'neg) (cadr ae))
          ((equal (car ae) 'disj)
           (list 'conj (neg (cadr ae)) (neg (caddr ae))))
          ((equal (car ae) 'conj)
           (list 'disj (neg (cadr ae)) (neg (caddr ae))))
          ) )
%cutset2
%expanding an expression
(de op (ae)
  (prog (arg1 arg2)
   (cond ((test1 ae)(return ae)))
   (setq arg1 (cadr ae))
   (setq arg2 (caddr ae))
   (cond ((and (test1 arg1)(test1 arg2))
          (return ae)))
   (setq arg1 (op arg1))
   (setq arg2 (op arg2))
   (cond ((equal (car ae) 'conj)
   (cond ((and (test2 arg1)(not (test2 arg2)))
     (return (list 'disj (op (list 'conj (cadr arg1) arg2))
                         (op (list 'conj (caddr arg1) arg2)))))
         ((and (not (test2 arg1))(test2 arg2))
     (return (list 'disj (op (list 'conj arg1 (cadr arg2)))
                         (op (list 'conj arg1 (caddr arg2))))))
          ((and (test2 arg1)(test2 arg2))
(return (list 'disj (op (list 'conj (cadr arg1)(cadr arg2)))
       (list 'disj (op (list 'conj (caddr arg1)(cadr arg2)))
        (list 'disj (op (list 'conj (cadr arg1)(caddr arg2)))
                  (op (list 'conj (caddr arg1)(caddr arg2)))
                  )))))
         (t (return (list 'conj (op arg1)(op arg2))))))
    (t (return (list 'disj (op arg1) (op arg2)))))
 ))
%cutset3
%eliminate (conj a a) from a disjunctive form
(de simpl1 (ae)
  (prog (arg1 arg2)
    (cond ((test1 ae)(return ae)))
    (setq arg1 (cadr ae))
    (setq arg2 (caddr ae))
```

```
130      (cond ((test3 ae)
131             (return (simpl11 ae)))
132            ((test2 ae)
133             (return (list 'disj (simpl1 arg1)
134                                 (simpl1 arg2))))
135            (t (list 'error 'in 'simpl1)))
136    ))
137  %apply rule (conj a a) = a
138  (de simpl11 (ae)
139    (prog (set1)
140      (cond ((test1 ae) (return ae)))
141      (setq set1 (makebag ae))
142      (setq set1 (makeset set1))
143      (return (formconj set1))
144  ))
145  %create a set from a list containing idetical elements
146  (de makeset (ae)
147  (prog (result)
148   (cond ((test1 ae)(return (list ae))))
149   (cond ((or (test2 ae)(equal (length ae) 1))(return ae)))
150   (cond ((member (car ae)(cdr ae))
151          (return (makeset (cdr ae))))
152         (t (return (append (list (car ae))
153                            (makeset (cdr ae))))))))
154  %form a conjunctive form
155  (de formconj (ae)
156    (cond ((test1 ae) ae)
157          ((equal (length ae) 1)(car ae))
158           ((equal (length ae) 2)
159            (list 'conj (car ae)(cadr ae)))
160           (t (list 'conj (car ae) (formconj (cdr ae))))))
161  %cutset4
162  %set (conj a (neg a)) to (conj a 0)
163  (de simpl22 (ae)
164    (prog (element nelement arg2 result)
165     (cond ((test1 ae)(return ae))
166           ((test2 ae)
167            (return (list 'wrong 'form 'in 'simpl22))))
168     (setq element (cadr ae))
169     (setq arg2 (caddr ae))
170     (setq nelement (neg element))
171     (cond ((test4 arg2)
172            (setq arg2 (simpl23 element nelement arg2))
173            (return (list 'conj element (simpl22 arg2))))
```

```
174           (t (return (list 'conj element (simpl22 arg2)))))
175    ))
176  (de simpl23 (el nel ae)
177     (prog (arg1 arg2)
178      (cond ((test1 ae)
179             (cond ((equal nel ae)(return 0))
180                   (t (return ae))))
181            ((test3 ae)
182             (setq arg1 (cadr ae))
183             (setq arg2 (caddr ae))
184             (cond ((or (equal arg1 nel)(equal arg2 nel))
185                    (return 0))
186                (t (return (list 'conj (simpl23 el nel arg1)
187                                       (simpl23 el nel arg2))))))
188            (t (list 'error 'in 'simpl23)))
189   ))
190  % (conj a 0) to 0
191  (de contract2 (ae)
192     (prog  (arg1 arg2)
193        (cond ((or (atom ae)(equal (car ae) 'neg))
194               (return ae))
195               (t (setq arg1 (cadr ae))
196                  (setq arg2 (caddr ae))))
197        (cond ((equal arg1 0)(return 0))
198              ((equal arg2 0)(return 0))
199              ((equal (contract2 arg2) 0)(return 0))
200              (t (return ae)))
201    ))
202  %work out a disjunctive form
203  (de simpl2 (ae)
204    (prog (arg1 arg2)
205      (cond ((test1 ae)(return ae)))
206      (setq arg1 (cadr ae))
207      (setq arg2 (caddr ae))
208      (cond ((test3 ae)
209             (return (contract2 (simpl22 ae))))
210            ((test2 ae)
211             (return (list 'disj (simpl2 arg1)
212                                 (simpl2 arg2))))
213            (t (list 'error 'in 'simpl2)))
214   ))
215  %cutset5
216  %(disj a (disj a b)) to (disj a b)
217  (de simpl31 (element express)
```

```
218      (prog (final)
219        (cond ((test4 express)
220            (cond ((compare1 element express)(return 0))
221                  (t (return express))))
222              ((compare1 element (cadr express))
223               (setq final (simpl31 element (caddr express)))
224                (return final))
225               (t (setq final (list 'disj (cadr express)
226                         (simpl31 element (caddr express))))
227                  (return final)))
228    ))
229  %
230  (de simpl3 (ae)
231     (prog (arg1 arg2 result)
232        (cond ((test4 ae) (return ae)))
233        (setq arg1 (cadr ae))
234        (setq arg2 (caddr ae))
235        (cond ((test4 arg2)
236               (cond ((compare1 arg1 arg2)
237                      (return arg1))
238                     (t (setq result (list 'disj arg1 arg2))
239                        (return result))))
240              ((test2 arg2)
241               (setq result (simpl31 arg1 arg2))
242         (setq result (list 'disj arg1 (simpl3 result)))
243               (return result))
244              (t (list 'error 'in 'simpl3)))
245     ))
246  %create a set from an s-expression
247  (de makebag (ae)
248   (prog (result)
249    (cond ((test1 ae)(return (list ae))))
250    (cond ((test2 ae)
251           (return (list ae)))
252          ((test3 ae)
253           (return (append (makebag (cadr ae))
254                           (makebag (caddr ae)))))
255          (t (return (list 'error 'in 'makebag))))))
256  %compare two sets
257  %solutions for problem 4-9 in Winston@Horn(1981)
258  (de ldifference (in out)
259    (cond ((null in) nil)
260       ((member (car in) out)(ldifference (cdr in) out))
261           (t (cons (car in) (ldifference (cdr in) out)))))
```

```
(de samesetp (a b)
   (not (or (ldifference a b)
            (ldifference (cdr a) b))))
%compare two conjunctive forms
(de compare1 (ae1 ae2)
  (prog (set1 set2)
    (setq set1 (makebag ae1))
    (setq set2 (makebag ae2))
    (cond ((samesetp set1 set2)(return t))
          (t (return nil)))
 ))
%cutset6
%(disj a (disj (neg a) b)) to 1
(de simpl41 (negelmt express)
    (prog (arg1 arg2 final)
        (setq arg1 (cadr express))
        (setq arg2 (caddr express))
        (cond ((equal arg1 negelmt)
               (return 1))
              ((or (atom arg2)(equal (car arg2) 'neg)
                   (equal (car arg2) 'conj))
               (cond ((equal negelmt arg2)
                      (return 1))
                     (t (return (list 'disj arg1 arg2)))))
              (t (return (list 'disj arg1
                               (simpl41 negelmt arg2)))))
  ))
%
(de simpl4 (ae)
  (prog (element negelmt rhs final)
    (cond ((test4 ae)(return ae)))
    (setq element (cadr ae))
    (setq rhs (caddr ae))
    (setq negelmt (neg element))
    (cond ((test4 rhs)
           (cond ((equal rhs negelmt)(return 1))
                 (t (return (list 'disj element rhs)))))
          ((test2 rhs)
           (setq final (simpl41 negelmt rhs))
           (return (list 'disj element (simpl4 final))))
          (t (list 'error 'in 'simpl4)))
  ))
%cutset7
%(disj a (conj a b)) to a
```

```
%search other "objct"s in conj. of the entire expression
(de simpl51 (objct express)
  (prog (arg1 arg2 final)
     (cond ((test1 express)(return express)))
     (cond ((test3 express)(return express))
           (t (setq arg1 (cadr express))
              (setq arg2 (caddr express))
              (cond ((equal arg1 objct)
          (return (list 'disj arg1 (simpl51 objct arg2))))
                    ((test4 arg1)
                     (setq final (compare2 objct arg1))
         (return (list 'disj final (simpl51 objct arg2))))
                    (t (list 'error 'in 'simpl51)))))
 ))
%work out
(de simpl5 (ae)
   (prog (objct rhs final)
      (cond ((test4 ae)(return ae)))
      (setq rhs ae)
      (setq final ae)
      loop
      (cond ((test4 rhs)
             (setq objct rhs)
             (return (simpl51 objct final)))
            (t (setq objct (cadr rhs))
               (setq final (simpl51 objct final))
               (setq rhs (caddr rhs))
               (go loop)))
  ))
%compare two conjunctive forms and see if
%the first is a subexpression of the second.
(de compare2 (ae1 ae2)
  (prog (set1 set2)
    (setq set1 (makebag ae1))
    (setq set2 (makebag ae2))
    (cond ((samesetp set1 (intersection set1 set2))
           (return 0))
          (t (return ae2)))
 ))
%cutset8
%rearranging expressions
(de rearrange (ae)
  (prog (arg1 arg2)
    (cond ((test1 ae)(return ae)))
```

```
350      (setq arg1 (cadr ae))
351      (setq arg2 (caddr ae))
352      (cond ((test3 ae)
353             (return (list 'conj (rearrange arg1)
354                                 (rearrange arg2))))
355            ((test2 ae)
356             (cond ((and (test4 arg1)(test4 arg2))
357                    (return (list 'disj (rearrange arg1)
358                                        (rearrange arg2))))
359                   ((and (test2 arg1)(test4 arg2))
360                    (return (list 'disj (rearrange arg2)
361                                        (rearrange arg1))))
362                   ((and (test4 arg1)(test2 arg2))
363                    (return (list 'disj (rearrange arg1)
364                                        (rearrange arg2))))
365                   ((and (test2 arg1)(test2 arg2))
366                    (setq arg1 (rearrange arg1))
367                    (setq arg2 (rearrange arg2))
368     (setq arg2 (rearrange (list 'disj (caddr arg1) arg2)))
369                    (return (list 'disj (cadr arg1) arg2)))
370                   (t (list 'error 'in 'rearrange))))
371            (t (list 'error 'in 'rearrange)))
372    ))
373  %cutset9
374  %extracting 0s
375  (de contract3 (ae)
376      (prog (arg1 arg2)
377         (cond ((or (atom ae)
378                    (equal (car ae) 'neg))
379                (return ae))
380               ((equal (car ae) 'conj)
381                (return (contract2 ae))))
382         (setq arg1 (cadr ae))
383         (setq arg2 (caddr ae))
384         (cond ((or (atom arg1)
385                    (equal (car arg1) 'neg))
386            (return (list 'disj arg1 (contract3 arg2))))
387               ((equal (car arg1) 'conj)
388                (setq arg1 (contract2 arg1))
389            (return (list 'disj arg1 (contract3 arg2))))
390               (t (list 'error 'in 'contract3)))
391   ))
392  (de contract4 (ae)
393      (prog (arg1 arg2)
```

```
394         (cond ((or (atom ae)
395                    (equal (car ae) 'neg)
396                    (equal (car ae) 'conj))
397                (return ae)))
398         (setq arg1 (cadr ae))
399         (setq arg2 (caddr ae))
400         (cond ((equal arg1 0)
401                (return (contract4 arg2)))
402               ((equal (contract4 arg2) 0)
403                (return arg1))
404         (t (return (list 'disj arg1 (contract4 arg2)))))
405   ))
406 %factor an expression
407 (de factoring (ae)
408   (prog (set1 set2 set3 element final)
409     (cond ((test4 ae)(return ae)))
410     (cond ((equal (length ae) 1) (return (car ae))))
411     (setq set1 (makebag (cadr ae)))
412     (setq set3 set1)
413     (setq final ae)
414      loop
415     (setq element (car set1))
416     (setq set1 (cdr set1))
417     (cond ((test2 (caddr ae))
418            (setq set2 (makebag (cadr (caddr ae))))
419            (cond ((member element set2)
420            (setq set3 (delete element set3))
421            (setq set2 (delete element set2))
422            (setq final (list 'disj (list 'conj element
423                                          (factoring (list 'disj
424                                             (formconj set3)
425                                             (formconj set2))))
424                              (factoring (cddr (caddr ae) ae))))
427            (setq final (factoring final))
428            (return final))
429            (t (cond ((null set1)
430                      (setq final (list 'disj (cadr ae)
431                           (factoring (caddr ae))))
432                      (return final))
433                     (t (go loop))))))
434           (t (setq set2 (makebag (caddr ae)))
435              (cond ((member element set2)
436                     (setq set2 (delete element set2))
437                     (setq set3 (delete element set3))
```

```
438                     (setq final
439                       (list 'conj element
440                       (factoring (list 'disj (formconj set3)
441                                              (formconj set2)))))
442                     (return final))
443                    (t (cond ((null set1)(return final))
444                             (t (go loop)))))))
445  ))
446 %functions for testing arguments
447 (de test1 (ae)
448     (or (atom ae)
449         (equal (car ae) 'neg)))
450 (de test2 (ae)
451     (and (not (atom ae))(equal (car ae) 'disj)))
452 (de test3 (ae)
453     (and (not (atom ae))(equal (car ae) 'conj)))
454 (de test4 (ae)
455     (or (atom ae)
456         (equal (car ae) 'neg)
457         (equal (car ae) 'conj)))
458 %Applying Nelson's algorithm
459 %with factoring  (09/23/86)
460 (de nelson2 (ae)
461     (prog (express)
462       (setq express (factoring ae))
463       (setq express (neg express))
464       (setq express (op express))
465       (setq express (rearrange express))
466       (setq express (simplify express))
467       (setq express (factoring express))
468       (setq express (neg express))
469       (setq express (op express))
470       (setq express (rearrange express))
471       (setq express (simplify express))
472       (setq express (subst 'plus 'disj express))
473       (setq express (subst 'times 'conj express))
474       (return express)
475  ))
476 %simplification procedures
477 (de simplify (ae)
478     (prog (express)
479       (setq express (simpl1 ae))
480       (setq express (simpl2 express))
481       (setq express (simpl3 express))
```

```
482         (setq express (simpl4 express))
483         (setq express (simpl5 express))
484         (setq express (contract4 (contract3 express)))
485         (return express)
486    ))
487   %print routine name and resulted expression
488   (de message1 (express routine)
489      (cond (t (print routine)
490         (print express))))
491   %reading necessary files
492   (lapin "nelson2")
493   (lapin "cutset0")
494   (lapin "cutset1")
495   (lapin "cutset2")
496   (lapin "cutset3")
497   (lapin "cutset4")
498   (lapin "cutset5")
499   (lapin "cutset6")
500   (lapin "cutset7")
501   (lapin "cutset8")
502   (lapin "cutset9")
503   (lapin "test")
504   (lapin "factor")
505   %end of loading
```