stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 85/77 AUGUSTUS

L. AMMERAAL

FORMULA SIMPLIFICATION IN RELATION TO PROGRAM VERIFICATION

2e boerhaavestraat 49 amsterdam

BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS(MOS) subject classification scheme (1970): 68A15, 68A10

ACM-Computing Reviews-category 5.24, 5.7.

Formula simplification in relation to program verification

by

L. Ammeraal

ABSTRACT

Predicate transformers associated with assignment statements and conditional statements are straightforward and can easily be mechanized. Except for some simple cases, it is a non-trivial task to simplify the resulting predicates automatically. It appears that formula simplification is the heart of automatic aids for program verification. This paper shows how predicate transformations and formula simplifications can be expressed in ALGOL 68, a high-level programming language which has appropriate facilities for data structuring.

KEYWORDS & PHRASES: *simplification, formula manipulation, program verification.*

CONTENTS

1. GENERALIZED RATIONAL SIMPLIFICATION

In this paper, rational expressions are arithmetical expressions composed of integer constants, variables, parentheses and the arithmetical operators +, -, *, /, grouped together in the usual way. Thus

```
(y+x*2+3*y+0*x) * 1 * (y-(y-1))
```

is a rational expression and in this special case it is most likely that a simplification to the equivalent expression

```
2 * x + 4 * y
```

is desired. The study of efficient algorithms to simplify rational expressions is an interesting field of research [1], [2], [3]. It is, however, not our only subject. Along with rational expressions, there are relational expressions which we would immediately simplify by hand before presenting them. For example

```
(-x) * x + 3 * x > - 2 * x * x + 5
```

is preferably simplified to

```
x * x + 3 * x - 5 > 0.
```

We therefore add the relational operators >, <, $\geq$, $\leq$, =, $\neq$ to our language, and represent them by >, <, >=, <=, =, # , respectively. We also introduce the logical operators "and" and "or", which we represent by & and !, and the logical constants 't' (true) and 'f' (false). We can now replace obvious tantologies such as 0 = 0 and 5 > 0 by 't' and obvious contradictions such as 1 = 0 and 5 < 0 by 'f'.

The following simplifications are easily performed:

```
φ & 't'    is simplified to    φ,
φ ! 't'    is simplified to    't',
```

ϕ & 'f' is simplified to 'f', and
ϕ ! 'f' is simplified to ϕ.

We now introduce a less conventional extension of our language. If σ is a sequence of program statements (which we shall define syntactically in the next section) and ϕ is a boolean expression, we regard

$$\sigma \$ \phi$$

as a new logical formula. In Dijkstra's terminology [4], it is the weakest precondition that corresponds with σ and ϕ, or:

$$\sigma \$ \phi \stackrel{\text{def}}{=\!=} wp(\sigma,\phi).$$

In the backward direction, predicate ϕ is transformed by statement σ to $\sigma \$ \phi$. We may also define $\sigma \$ \phi$ as a necessary and sufficient condition imposed on all program variables before the execution of σ, to ensure that condition ϕ is satisfied after this execution. An example is

$$x := x + 1 \$ x > 5. \tag{1}$$

We have extended our formal language in such a way that (1) is a formula in this language, which happens to be equivalent to

$$x > 4. \tag{2}$$

We obtain (2) from $x + 1 > 5$, which is found by substituting $x + 1$ for x in $x > 5$. This substitution is usually referred to as Hoare's axiom for the assignment statement [5]. Thus notations as $S_f^x\, Q|$ (used by Church [6]), $S_f^x(Q)$ (used by Floyd [7]), $Q_x[f]$ (used by Schoenfield [8]), and $Q[f/x]$ (used by Apt & De Bakker [9]) are written as

$$x := f;\ Q \tag{3}$$

in our formal language.

It will now be clear that the notion of "rational simplification" can be generalized in such a way that it includes "predicate transformation" and "substitution" as special cases.

2. DESCRIPTION OF THE LANGUAGE *FORM*

Simplifications like those outlined in the previous section are actually performed by an automatic simplifier. This simplifier is an ALGOL 68 program which we shall discuss in more detail. First, however, the set of all formulas that are candidates for simplification needs to be defined. We denote this set by FORM. Thus FORM is the language of all input strings that are processed succesfully by our simplification program. The following is a context-free grammer for FORM in BNF; < formula > is the start symbol of this grammar:

```
< formula > ::= < boolean formula > | < arithmetic expression >
< boolean formula > ::= < boolean expression > |
                        < statement sequence > $ < boolean expression >
< statement sequence > ::= < statement > |
                           < statement sequence >; < statement >
< statement > ::= < assignment statement > |
                  < conditional statement >
< assignment statement > ::= < variable > := < arithmetic expression >
< conditional statement > ::= < boolean expression > • < alternation >
< alternation > ::= (< statement sequence > @ < statement sequence >)
< boolean expression > ::= < conjunction > |
                           < boolean expression > ! < conjunction >
< conjunction > ::= < boolean primary > |
                    < conjunction > & < boolean primary >
< boolean primary > ::= < 't' > |
                        < 'f' > |
```

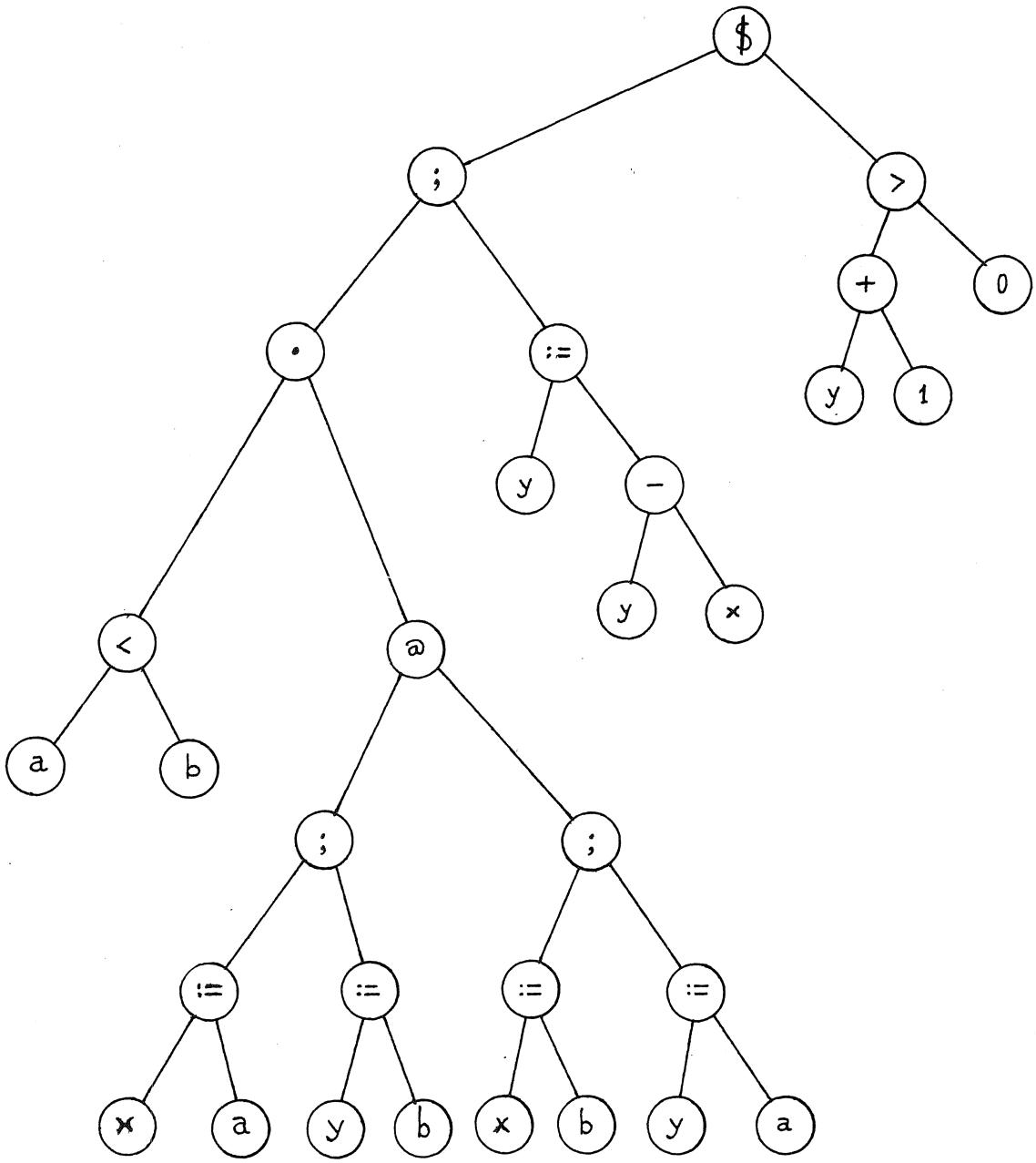
```
                    < arithmetic expression > < relation symbol >
                                                < arithmetic expression > |
                    (< boolean formula >)
< relation symbol > ::= = | # | > | <= | >= | <
< arithmetic expression > ::= < term >
                              < arithmetic expression > < adding symbol >
                                                        < term >
< adding symbol > ::= + | -
< term > ::= < factor > |
             < term > < multiplying symbol > < factor >
< multiplying symbol > ::= * | /
< factor > ::= < variable > |
               < constant > |
               (< arithmetic expression >)

< variable > ::= a | b | c | d | e | f | g | h | i | j | k | ℓ | m | n | o |
                 p | q | r | s | t | u | v | w | x | y | z
< constant > ::= < digit > |
                 < constant > < digit >
< digit >    ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

This syntax is such that a unique binary tree is associated with every sentence of the language. The unconventional syntax chosen for conditional statements is a consequence of this idea. Consider, for example, the formula

$$a < b \cdot (x:=a;\ y:=b \ @\ x:=b;\ y:=a);\ y := y - x \ \$\ y + 1 > 0.$$

(In more conventional terms, this formula denotes the weakest precondition that corresponds to the statement sequence if a < b then x := a; y := b else x := b; y := a fi; y := y - x and the postcondition y + 1 > 0).

According to our syntax, the following binary tree is associated with this formula:


$
;
>
+
0
y
1
•
:=
y
−
y
x
<
@
a
b
;
;
:=
:=
:=
:=
x
a
y
b
x
b
y
a

It will be clear from this example that parentheses do not occur explicitly in the tree; they may, however, influence the structure of the tree. In general, each leaf of the tree can be a variable, a constant, or a truth value ('t' or 'f'). The other nodes of the tree are operators. The following table lists all operators of FORM, in increasing order of precedence:

priority	operator
1	@
2	; $
3	.
4	:=
5	!
6	&
7	= # < > <= >=
8	+ -
9	* /

The choice of the operator representations was based on the availability of a conventional character set. Therefore # was taken instead of ≠, and ! rather than | or ∨. Notice that all operators in our language are dyadic and infix, i.e. they occur in the context:

left operand, operator, right operand.

This means that 0 must not be omitted in 0 - x, just like 1 must not be omitted in 1/x.

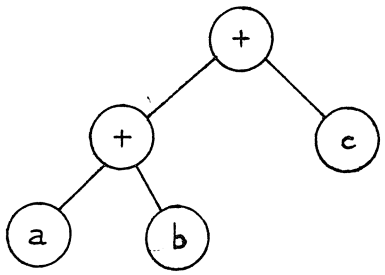
Another consequence of having only dyadic operators is the absence of a special operator for negation. We do not need it; for example, in FORM we express the negation of

```
(a<b & c#d) ! e = f
```

by

```
(a>=b ! c=d) & e # f.
```

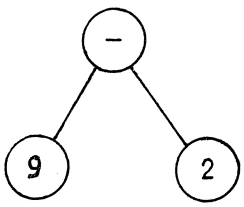
It goes without saying that a formula such as a + b + c is an abbreviated notation for (a+b) + c, and thus has the following associated binary tree



It might look strange that a formula in FORM may be of either arithmetic or boolean type. This somewhat liberal point of view was adopted for practical reasons. Our simplifier has to simplify both types of formulas anyhow, and we felt it convenient if not only boolean but also arithmetic formulas are accepted as input strings.

3. FORMULAS, BINARY TREES AND ALGOL 68

Algorithms are best expressed in high-level programming languages. We have chosen ALGOL 68, and, among the numerous facilities of this language, some concepts that we need will now be explained. Let us begin with a very simple example. With the formula 9-2 we associate the binary tree:



We could write the following mode declaration for this simple type of formulae:

```
mode simpform = struct (int left, char c, int right).
```

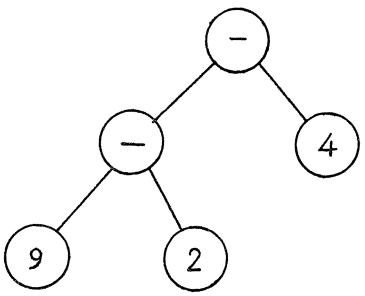
After the variable declaration

```
simpform f; ,
```

it would make sense to write the assignment statement

```
f:= (9,"-",2).
```

We now wish to implement formulas whose operands are in turn (non-atomic) formulas. An example is 9-2-4, which corresponds to the tree



Obviously, our first attempt is too restrictive, since the mode simpform allows only integers as operands. We want operands to be either integers or non-atomic formulas. Since all non-atomic formulas in FORM have the structure

operand 1, operator, operand 2,

we shall call them triples. We shall see that all atomic formulas in FORM can be represented by integers. Thus a formula is either an (atomic) integer or a (non-atomic) triple. In this terminology a triple has the structure

formula, operator, formula.

In ALGOL 68 we define the modes formula and triple by the mode declarations:

```
mode  formula  =  union (int, ref triple);
mode  triple   =  struct (formula left, char c, formula right);
```

If we now declare

```
formula f, g, h;
```

we can assign 3 to f, 3+4 to g, and 8-2-3 to h by

```
f := 3;   g:= heap triple := (3,"+",4);
h := heap triple :=
     (heap triple := (8,"-",2),
     "-",
     3);
```

Up to now we have used integer constants and no variables in the examples. Our language is such that these integer constants are non-negative. This offers the possibility to use negative integers to encode variables and truth values. In ALGOL 68 we have standard operators abs and repr which provides the desired one-to-one mapping, and its inverse, as follows:

letter	integer
ℓ	- abs ℓ
repr-i	i

Note that in FORM a variable consists of a single letter. As to the encoding of truth values, we declare

```
int true = - 1000, false = - 1001.
```

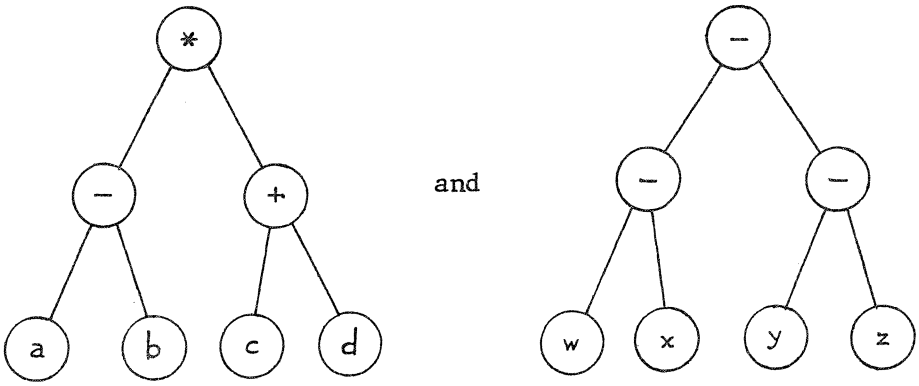
Assuming that variable f is declared as before, we can assign the formula a < 13 & 't' to f by the assignment statement

```
f := (heap triple := (- abs "a", "<", 13),
     "&",
     true);
```

Since a given formula f can be either an integer or a (reference to a) triple, we need a mechanism to find this out. In ALGOL 68 this is done as follows:

```
case f
in  (int i) = σ1,
    (ref triple t): σ2
esac
```

If formula f happens to be an integer, σ_1 is elaborated, in which we can use f through i. In the alternative case, σ_2 is elaborated, in which the current ref triple value of f is accessed through t. We illustrate this mechanism by a procedure to print a formula. A special provision will be needed to insert parentheses in cases like



which, if no simplifications were performed, must be printed as (a-b) * (c+d) and w - x -(y-z). Printing a (sub-) formula, we need information about the context to decide whether or not parentheses are to be inserted. Roughly speaking, they are to be inserted if a left(right) son operator binds looser (not tighter) than its father. The procedure "pr" below shows this more precisely. To avoid uninteresting complications, we ignore the fact that some operators (viz. :=, >= and <=) are composed of two characters. The standard procedure "whole" decomposes an integer in its decimal digits. The following procedure does the job of printing any formula f if it is called as pr(f,1):

```
proc pr = (formula f, int prio) void:
case f
in (int leaf):
   if leaf = true or leaf = false
```

```
        then print (if leaf = true then " 't' " else " 'f' ")
        elif leaf < 0
        then print (repr - leaf)
        else print (whole (leaf,0))
        fi ,
        (ref triple t):
        (char c = c of t; int p = priority (c);
              if p < prio then print ("(") fi;
              pr (left of t,p);
              print (c) ;
              pr (right of t, p+1);
              if p < prio then print (")") fi
        )
    esac;
```

Procedure "pr" shows how a tree representation of a formula is transformed to a string representation. We shall now deal with the inverse process, i.e. how to obtain the binary tree that is associated with a given formula. In other words, "pr" is for output, and the following procedures are for input. As before, we shall omit details that might distract our attention from essential points. We shall, in particular, not worry about diagnostic messages etc., but assume that only correct formulas, i.e. elements of FORM, are offered as input. (This attitude needs not necessarily be unrealistic, since one could imagine that the formulas have already passed some program for lexical and syntactic analyses, e.g. a compiler). As before, we employ procedure "priority", which yields the priority number for any operator, as given in the preceding section. To inspect the next character to be read, we have a one-character buffer, declared and initialized by:

```
char buf; read (buf);
```

and the boolean procedure

```
proc inp = (char x) bool:
if buf = x then read (buf); true else false fi.
```

The variable "buf" is inspected also by the following procedure "operator", which tests the next character for being an operator with a given priority number "prio"; if it is, it is passed by the output parameter "op":

```
proc operator = (int prio, ref char op) bool:
if priority(buf) = prio
then op:= buf; read(buf); true
else false
fi .
```

We shall process a complete formula of FORM, no matter how complex, or how simple, by the call

```
formproc (1).
```

Before dealing with this procedure "formproc", we introduce procedure "primary", which reads boolean primaries, variables, constants and expressions in parentheses; when "primary" is called, a correct formula must be present. The procedure yields a formula, just like a boolean procedure yields a boolean value. (Notice that "true" and "false" on the fourth and sixth line are not underlined: they are integer variables in ALGOL 68, but encoded truth values and hence formulas in FORM):

```
proc primary = formula:
if inp (" ' ")
then if inp ("t")
     then if ¬ inp (" ' ") then error fi; true
     elif inp ("f")
     then if ¬ inp (" ' ") then error fi; false
     else error
     fi
elif inp ("(")
then formula f := formproc (1);
     if ¬ (")") then error fi;
     f
```

```
else  char buf0 = buf;
      if buf ≥ "a" ∧ buf ≤ "z"
      then read(buf); - abs buf0
      else int i:= 0, d;
           while d:= abs buf - abs "0"; d ≥ 0 ∧ d ≤ 9
           do i:= 10 * i + d; read (buf)
           od;
           i
      fi
fi.
```

We shall now show the procedure "formproc" (It was called "formproc" rather than "formula" to avoid confusion with the mode formula that we have already). We could say that "formproc" does most of the work in reading and analyzing the input formula and in building the associated binary tree. The reader should not be misled by its compactness. It is in fact a generalization of the idea of having separate syntactic procedures for "expression", "term", "factor", etc., as is often used in recursive-descent parsing methods; so in spite of its compactness, it performs almost the whole task of syntactic analysis:

```
proc formproc = (int prio) formula:
if prio = 10
then primary
else formula f := formproc (prio+1);
     char op;
     while operator (prio,op)
     do f := heap triple := (f,op,formproc(prio+1))
     od;
     f
fi.
```

The reader will have noticed that the concept of recursion is often employed in these procedures. This is not surprising, since our parsing method is recursive by definition. Furthermore, we are manipulating binary

trees, which are most conveniently defined with the aid of recursion. For readers who are unfamiliar with either recursion, ALGOL 68, or manipulating trees, we proceed with showing three not too difficult procedures which are useful for our purposes.

The first of these is "identical". It determines whether or not two formulas f and g, stored in binary trees as usual, are identical:

```
proc identical = (formula f,g) bool:
case f
in (int fleaf):
   case g
   in (int gleaf): fleaf = gleaf,
      (ref triple gt): false
   esac,
   (ref triple ft):
   case g
   in (int gleaf): false,
      (ref triple gt):
      if c of ft = c of gt
      then  if identical (left of ft, left of gt)
            then identical (right of ft, right of gt)
            else false
            fi
      else  false
      fi
   esac
esac.
```

In contrast to "identical", whose only task is to construct an appropriate boolean value, the next procedure constructs a complete new tree, which is an exact copy of the one that is passed as a parameter. The newly generated tree does not share any nodes with the original one:

```
proc  copytree = (formula f) formula:
case  f
```

```
in  (int leaf): leaf,
    (ref triple t): heap triple :=
    (copytree (left of t),
     c  of  t,
     copytree (right of t)
    )
esac.
```

The next procedure contains a call of "copytree". It copies a given formula f but, in the meantime, it replaces (in the copy) all occurrences of a given variable v by a given formula g. Here too, the new tree is completely distinct from the original one:

```
proc  subst = (char v, formula g,f) formula:
case  f
in  (int leaf):
    if leaf = - abs v then copytree (g) else leaf fi,
    (ref triple t): heap triple :=
    (subst (v,g, left of t),
     c of t,
     subst (v,g, right of t)
    )
esac.
```

4. TIDYING UP FORMULAS

Formulas are usually written down in a more or less canonical form. A trivial example is the formula a + b which most of us would prefer to b + a. In automatic simplification it makes sense to strive after obtaining canonical forms, for various reasons.

One of them is that identical subformulae are easier detected this way. Suppose for example that we are given the formula

```
b * a * c + d - c * a * b.
```

Step-by-step simplification of this formula proceeds as follows:

Step 1 : Rearranging the factors of each term. This yields:

a * b * c + d - a * b * c

(In fact this is a rather big step which consists of several more elementary steps).

Step 2 : Rearranging the terms. This yields:

a * b * c - a * b * c + d

Step 3 : Canceling adjacent identical term with different signs. This yields:

0 + d

Step 4 : Deleting zero terms. This yields

d.

Rearranging factors and terms as in steps 1 and 2 above is similar to sorting a sequence of numbers. For numbers we have a total ordering ("<"), so it is natural to define a total ordering for formulas as well. Let us call this ordering "less". A slight complication, to be mentioned later on, will be ignored at this moment.

We express the ordering "less" on FORM by the following ALGOL 68 procedure:

```
proc  less = (formula f,g) bool:
case  f
in    (int f):
      case g
      in (int g):
         if f < 0 ∧ g < 0 then f > g else f < g fi,
         (ref triple g): false
      esac,
```

```
            (ref triple f):
            case g
            in  (int g): true
                (ref triple g):
                (int pf = priority (c of f), pg = priority (c of g);
                 if pf ≠ pg then pf > pg
                 elif  less (right of f, right of g) then true
                 elif  less (right of g, right of f) then false
                 elif  less (left of f, left of g) then true
                 elif  less (left of g, left of f) then false
                 else  c of f < c of g
                 fi
            esac
        esac.
```

Important properties of "less" are:

(i) If less (f,g) and less (g,h), then less (f,h) (Transitivity).
(ii) If less (f,g) then not less (g,f) (Asymmetry).
(iii) For all f ∈ FORM: not less (f,f) (Irreflexivity).
(iv) For all f, g ∈ FORM: either less (f,g),
or less (g,f),
or identical (f,g) (less is a total ordering).

These properties are important because we are aiming at obtaining canonical forms. [For this purpose, it is essential that the ordering is total. Suppose for instance that the ordering "<" for numbers were not defined for some pair of numbers, e.g. for 5 and 7. Then sorting the sequence 10, 5, 7, 3 could yield 3, 5, 7, 10, or alternatively, 3, 7, 5, 10; in other words the sorted sequence would not be uniquely determined].

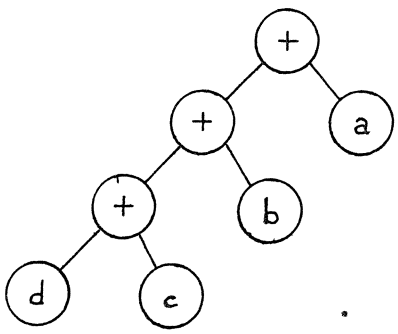
The complication mentioned previously has to do with a curious conventional distinction in ordering terms and ordering factors. Most of us probably prefer x + y + 2 to 2 + x + y as a canonical form, but at the same time we write 2 * x * y rather than x * y * 2. Thus in connection with "+" we appreciate the truth of "less (variable, constant)", whereas "less (constant,

variable)" ought to be true in a multiplicative context. We have solved this little problem by supplying the procedure "less" with a third parameter "prio" which tells which of both cases applies. We shall not dwell on this detail. We shall now present a procedure to rearrange terms on the basis of the procedure "less". To keep it readable we only deal with the commutative operators *, +, =, #, & and ! in this version. (In our implementation we did not ignore the complication of non-commutative operators, but at this stage we prefer readability to completeness. We hope that the simplified versions of some procedures shown here are easily understood; it will then be not too difficult to complement them with details which are not taken into consideration here. Moreover, the source listing of the complete program is given in Appendix A).

To exchange two branches of a binary tree, we have the following procedure:

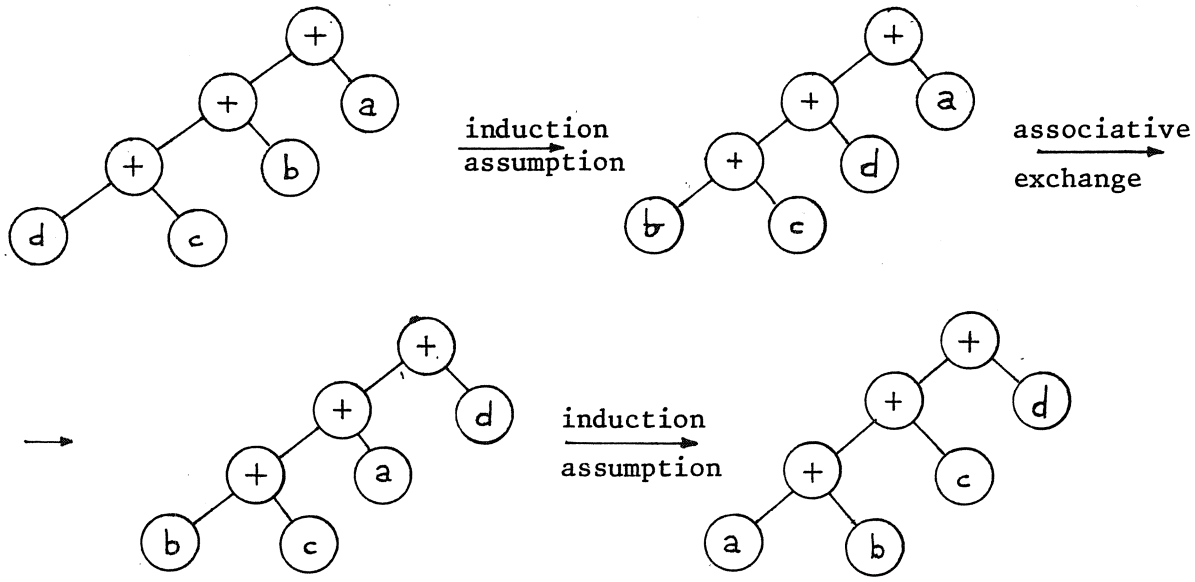
```
proc exchange = (ref formula f,g) void:
(formula h := f; f := g; g := h).
```

Now consider the formula d + c + b + a whose binary tree is



We want to transform this tree into a similar tree corresponding to the formula a + b + c + d, by exchanging branches. There are two kinds of exchanges to be made. At the lowest level the left branch d and the right branch c are to exchanged. We call it a commutative exchange, since it is justified by the commutative law. Another type of exchange is applied to, e.g., b and a. To justify this exchange we need not only the commutative law, but also the associative law. We therefore call it an associative exchange.

Reasoning inductively, we assume that we know how to rearrange the subtree for d + c + b, and we show how to rearrange the tree for d + c + b + a:

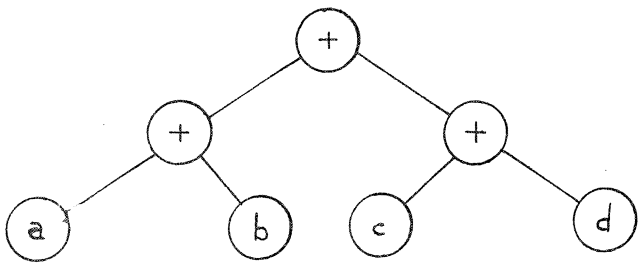


We express this in ALGOL 68 by

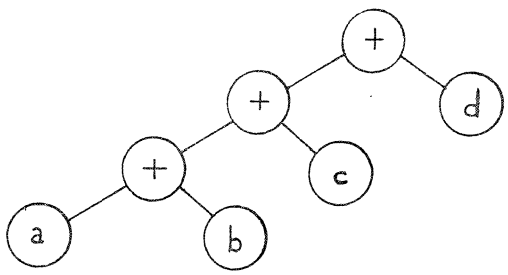
```
proc rearrange = (ref formula f) void:
case f
in (ref triple t):
   (ref formula ℓ = left of t,
                r = right of t;
    char op = c of t;
    if less (r,ℓ)
    then exchange (ℓ,r) #  commutative exchange #
    else case ℓ
         in (ref triple t left):
         (ref formula ℓ left = left of t left,
                      r left = right of t left;
          char opleft = c of tleft;
          if opleft = op
          then rearrange (ℓ) # induction assumption #;
               if less (r, rleft)
               then exchange (rleft,r)
```

```
                    # associative exchange #;
                    rearrange (ℓ)
                    # induction assumption #
                fi
            fi
           )
        fi
       )
    esac.
```

We call a formula like a + b + c + d (and its associated binary tree) left-associative, because it is an abbreviated notation of ((a+b)+c) + d. When a formula like a + b + (c+d) is offered as input to our program it is initially stored in the binary tree



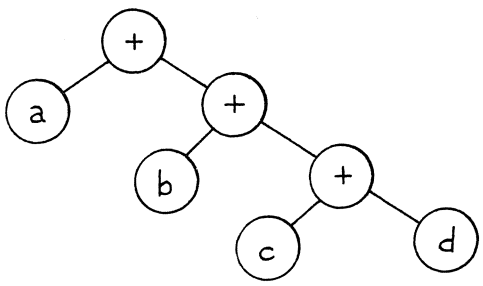
As we have seen, the parentheses enclosing c + d are re-inserted when this tree is offered to the procedure pr (given in section 3) for output. However, what we actually want is to transform the tree above to its left-associative equivalent:



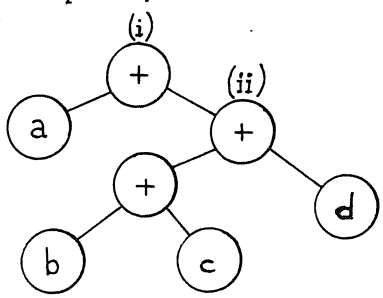
Restricting ourselves to commutative operators, we can perform this transformation by the following procedure:

```
proc leftassoc = (ref formula f) void:
case f
in (ref triple t):
   (leftassoc (left of t); leftassoc (right of t);
    case right of t
    in (ref triple tright):
       if c of tright = c of t
       then right of t := left of tright;
            left of tright := t;
            f := tright; leftassoc(f)
       fi
    esac
   )
esac.
```

Here again, we can verify the correctness of this procedure by induction on the complexity of the tree. We shall nct give a formal proof, but rather explain how it works by means of an example. Suppose that a + (b+(c+d)) is to be transformed to its leftassociative equivalent a + b + c + d. By the call "leftassoc (right of t)" on the fourth line of the procedure, the tree

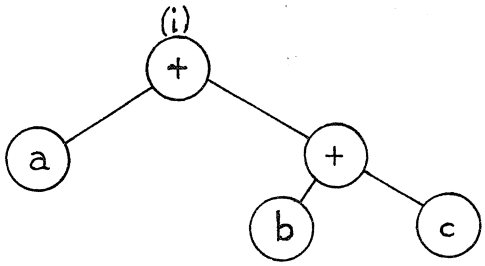


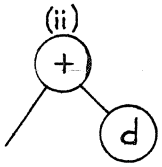
is (by our induction assumption) transformed to



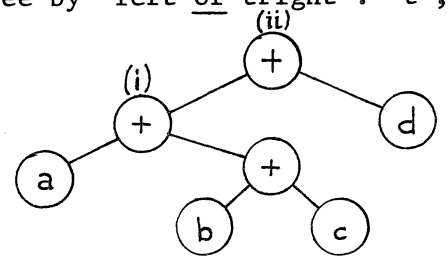
Nodes (i) and (ii) correspond with "c of t" and "c of tright", respectively. The assignment statement "right of t := left of tright" transforms this tree to:

BIBLIOTHEEK MATHEMATISCH CENTRUM
AMSTERDAM



The missing part 

is now attached at the right on top of this tree by "left of tright := t", yielding



The statement "f := tright" reflects the fact that a new node (ii) now becomes the root of the tree, instead of the old one (i). The new tree corresponds to the formula a + (b+c) + d which is less complex (with respect to left-associativity) than the original one. Therefore, by induction, the only step to complete the job is to apply "leftassoc" recursively to this tree.

5. SIMPLIFICATION AND PREDICATE TRANSFORMATION

Transforming b + a to a + b and other things that we did in the previous section might be considered too trivial to be called "simplification". We shall now mention some more substantial simplifications, and, to be concrete, mention a number of procedures of which our program is composed. It should be emphasized, however, that knowledge of these procedures is by no means needed for using our program. We are discussing techniques and we sometimes talk about procedures and about a program to stress that these techniques have been implemented and are not just loose ideas. The implementation is such that a formula of FORM followed by the closing symbol "\" are the only input data. The user does not have to specify special options or other control information depending on the nature of the input formula.

Let us start with some rather obvious simplications which have to do with occurrences of zeroes and ones. The following reductions are performed by the procedure "zerone":

$$
\begin{aligned}
0 + f &\rightarrow f \\
f \pm 0 &\rightarrow f \\
1 * f &\rightarrow f \\
f * 1 &\rightarrow f \\
0 * f &\rightarrow 0 \\
f * 0 &\rightarrow 0 \\
f / 1 &\rightarrow f \\
0 / f &\rightarrow 0 \\
f / 0 &\rightarrow ?
\end{aligned}
$$

The last two of these lines deserve an explanation. One could argue that the information $f \neq 0$ is lost by reducing $0/f$ to 0. Some might consider the possibility of defining $0/f$ equal to 1 if f happens to vanish. We prefer the point of view that any denominator must not vanish, even if the numerator does. On the other hand, whenever $0/f$ has a meaning, it is of arithmetic type and, for our purpose, it would be most inconvenient if it had to be transformed to something like

"0, provided $f \neq 0$"

which has arithmetic as well as boolean aspects. For the same reason, we shall simplify (x-2) / (x-2) to 1. Brown [2] does the same, and, as a justification, observes that the transformation conforms to the rules of the field Z(x) of rational expressions in x over the domain Z, of integers. The transformation "f/0 $\rightarrow$?" is easier to explain. Whenever an explicit zero occurs as a denominator, the question mark "?" is introduced. It should simply be interpreted as an error code.

The procedure "distr" applies the distributive law with respect to multiplication and addition (or subtraction). Thus

$$
\begin{aligned}
f * (g \pm h) &\rightarrow f * g \pm f * h \\
(f \pm g) * h &\rightarrow f * h \pm g * h.
\end{aligned}
$$

It is well-known that it depends on the context whether multiplying out or factoring leads to simpler formulas. Finding this out is too difficult a job for our simplifier. Therefore we have chosen for multiplying out, so (x-1) * (x+1) + 1 is nicely reduced to x * x. However, care has been taken to postpone the call of "distr" until reductions like

(x+2) * (x-2) / ((x+2) * (x-3)) → (x-2) / (x-3)

have been performed. For reductions like this, the procedures "rearrange" and "leftassoc" are vital. These procedures transform

(x+2) * (x-2) / ((x+2) * (x-3))

to

(x+2) / (x+2) * (x-2) / (x-3).

(Recall that "*" and "/" have the same priority number). Thus identical factors become neighbours, which are easily tested for identity (if they occur on either side of "/") by the procedure "identical", introduced in section 3. These tests and canceling of numerators and denominators are incorporated in the actual version of the procedure "rearrange" as listed in appendix A. Special attention is required for a case like

(x+2) * (x-2) / ((x+3) * (2-x)).

Since variable terms precede constant terms and since we have no monadic formulas, 2 - x is first transformed to 0 - x + 2. (We consider this as a special notation for - x + 2 and accept 0 as the only constant term that can precede a variable term. There is no danger that f + 0 - g will survive, because of our reduction f + 0 → f). Now 0 - x + 2 is hardly an improvement of 2 - x, and is far from being identical to x - 2. We therefore have a procedure called "negfac", which does away with "negative factors" like 0 - x + 2. In fact, the "nomadic minus", here written as "0-" is transported to an outer environment. For example

```
(0-x+3) * (0-y+z) / (0-p+1) / (0-q) * (0-r-1)
```

is transformed to

```
0 - ((x-3) * (y-z) / (p-1) / q * (r+1)).
```

Thus factors are brought into a canonical form, after which they can effectively be tested for identity. Our sample formula

```
(x+2) * (x-2) / ((x+3) * (2-x))
```

is indeed eventually reduced to

```
0 - (x+2) / (x + 3).
```

A substantial simplification is performed by the procedure "add". It reduces, e.g.

```
3 * a * (b+c) - 2 * a * (b+c) + 8 * a * (b+c)
```

to

```
9 * a * (b+c).
```

Here too, the procedure "identical" appears to be most useful. A complication is involved in terms whose coefficient are 1, since "1*" is usually omitted, or even removed by our own procedure "zerone". For addition, 1 * a * (b+c) + 3 * a * (b+c) is simpler than a * (b+c) + 3 * a * (b+c). We therefore have a procedure called "factone" which writes "1*" in front of products which begin with variables. Later on, superfluous occurrences of "1*" are removed by "zerone".

We are now going to deal with simplifications which are more often found in papers on program verification than in publications on formula manipulation. For the assignment statement and the conditional statement, sym-

bolically written as

x := E
if B then S else T fi,

it is well-known that with a given postcondition Q the corresponding weakest precondition can be expressed by

wp(x:=E,Q) = the result of substituting E for x in Q,

wp(if B then S else T fi, Q) = (B ∧ wp(S,Q)) ∨ (¬ B ∧ wp(T,Q)).

For the substitution result above, we can use our procedure "subst" listed at the end of section 3. As mentioned earlier, we write "&" for "∧", "!" for "∨", and "B•(S@T)" for "if B then S else T fi".

We also observed (in section 2) that we need no monadic not (¬) operator, because, e.g., ¬ (a<b) can be expressed by a > = b. Symbolically we shall indicate such an elimination of "¬" by writing $\bar{B}$ instead of ¬ B. In a terminology introduced by Dijkstra [4], wp(S,Q) is the result of transforming predicate Q by statement S. In [13] an explanation of predicate transformers is given in terms of elementary set theory. We have adopted the formula notation S $ Q for wp(S,Q). All formulas of the type S $ Q can be "simplified" according to the rules given above, which are in our notation:

x := E $ Q → subst (x,E,Q)

B•(S@T) $ Q → B & (S$Q) ! $\bar{B}$ & (T$Q).

By repeated application of these rules, all operators $ disappear, although, in the beginning, their number may increase. In our program these rules are implemented in the procedure "wp". This procedure is called by the procedure "reduce". The relation between both procedures is as follows. Let "wp" perform the redaction S_n $ Q → Q'; then "reduce" performs the reduction $S_1;...;S_{n-1}$; S_n $ Q → $S_1;...;S_{n-1}$ $ Q' if n > 1 and "reduce" simply calls "wp" if n = 1.

REFERENCES

[1] MOSES, J., *Algebraic Simplification: A Guide for the Perplexed*, Comm. ACM 14 (1971) 527-537.

[2] BROWN, W.S., *On Computing with Factored Rational Expressions*, SIGSAM Bull. 8 (1974) 27-34.

[3] HALL, A.D., *Factored Rational Expressions in ALTRAN*, SIGSAM Bull. 8 (1974) 35-45.

[4] DIJKSTRA, E.W., *A Simple Axiomatic Basis for Programming Language Constructs*, Proc. Kon. Ned. Akad., Ser. A, 77 (or Indagationes Math., 36), (1974) 1-15.

[5] HOARE, C.A.R., *An Axiomatic Basis for Computer Programming*, Comm. ACM 12 (1969) 576-580.

[6] CHURCH, A., *The Calculi of Lambda-conversion*, Princeton University Press (1941).

[7] FLOYD, R.W., *Assigning Meanings to Programs*, Proc. Symp. Appl. Math. 19, American Math. Soc. (1967) 19-32.

[8] SHOENFIELD, J.R., *Mathematical Logic*, Addison-Wesley Publishing Company (1967).

[9] APT, K.R. & J.W. DE BAKKER, *Exercises in Denotational Semantics*, in: Proc. 5th Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, Springer-Verlag, 45 (1976) 1-11.

[10] KING, J.C., *A Program Verifier*, Dept. of Computer Science, Carnegie-Mellon University (1969).

[11] GOOD, D.I., R.L. LONDON & W.W. BLEDSOE, *An Interactive Program Verification System*, Transactions on Software Engineering 1 (1975) 59-67.

[12] CAVINESS, B.F., *On Canonical Forms and Simplification*, Dept. of Computer Science, Carnegie-Mellon University (1968).

[13] AMMERAAL, L., *How Program Statements Transform Predicates*, in: Informatik-Fachberichte, Springer-Verlag, 5 (1976) 109-120.

APPENDIX A; THE SIMPLIFYING PROGRAM.

```
(# SIMPLIFICATION OF FORMULAS, L. AMMERAAL, 1 JULY 1977 #
 'INT' TRUE=-1000, FALSE=-1001; 'CHAR' CC;
 ON LINE END(STAND IN,
             ('REF' 'FILE' F)'BOOL';
               (NEWLINE(F); PRINT(NEWLINE); 'TRUE')
            );
 'MODE' 'FORMULA'='UNION'('INT','REF' 'TRIPLE');
 'MODE' 'TRIPLE'='STRUCT'('FORMULA' LEFT,'CHAR' C,'FORMULA' RIGHT);

 'PROC' INP=('CHAR' X)'BOOL':(BUF=X|READP(BUF);'TRUE'|'FALSE');

 'CHAR' BUF;

 'PROC' READP=('REF' 'CHAR' C)'VOID';
 ( 'WHILE' READ(C); PRINT(C); C = " " 'DO' 'SKIP' 'OD');

 'PROC' M=('BOOL' B)'VOID':('NOT' B|ERROR);

 'PROC' ERROR='VOID':(PRINT("ERROR"); STOP);

 'PROC' DIGIT=('CHAR' C)'BOOL': C 'GE' "0" 'AND' C 'LE' "9";

 'PROC' LETTER=('CHAR' C)'BOOL': C 'GE' "A" 'AND' C 'LE' "Z";

 'PROC' LESS=('FORMULA' X,Y, 'INT' PRIO)'BOOL':
 ( 'CHAR' CHX=NODE(X), CHY=NODE(Y);
   'BOOL' DX=DIGIT(CHX), DY=DIGIT(CHY), LX=LETTER(CHX), LY=LETTER(CHY);
   'BOOL' OX='NOT'(DX 'OR' LX), OY='NOT'(DY 'OR' LY), P9=(PRIO=9),
          L= CHX<CHY;
   ( OX 'AND' OY
   | 'INT' PX=PRIORITY(CHX), PY=PRIORITY(CHY);
     ( PX 'NE' PY | PX>PY
     | ( X
       | ( 'REF' 'TRIPLE' BX);
         ( Y
         | ( 'REF' 'TRIPLE' BY);
           ( 'REF' 'FORMULA' IX= LEFT 'OF' BX, IY= LEFT 'OF' BY,
                             JX=RIGHT 'OF' BX, JY=RIGHT 'OF' BY;
             ( LESS(JX,JY,PRIO) | 'TRUE' |: LESS(JY,JX,PRIO) | 'FALSE'
             |:LESS(IX,IY,PRIO) | 'TRUE' |: LESS(IY,IX,PRIO) | 'FALSE'
             | L
             )
           )
         )
       )
     )
   |: OX | 'NOT' P9 |: OY | P9 |: DX=DY | L | (P9|DX|DY)
   )
 );
```

```
'PROC' NODE=('FORMULA' E)'CHAR';
( E
| ( 'INT' C);
  ( C=0|"0"|;C=1|"1"|;C>1|"2"|;C=TRUE|"T"|;C=FALSE|"F"|'REPR'(=C)),
  ( 'REF' 'TRIPLE' B); C 'OF' B
);

'PROC' EXCHANGEC=('REF' 'CHAR' X,Y)'VOID';
('CHAR' Z=X; X:=Y; Y:=Z);

'PROC' EXCHANGEI=('REF' 'FORMULA' X,Y)'VOID';
('FORMULA' Z=X; X:=Y; Y:=Z);


'PROC' PRIMARY='FORMULA';
( INP("-") | 'HEAP' 'TRIPLE';=(0,"-",PRIMARY)
|:INP("+") | PRIMARY
|:INP("I") | (INP("T")|M(INP("I"));TRUE
             | M(INP("F"));M(INP("I"));FALSE
             )
|:INP("(")
| 'FORMULA' I:=FORMPROC(1); M(INP(")")); I
| 'CHAR' C=BUF;
  ( C 'GE' "A" 'AND' C 'LE' "Z" | READP(BUF); -'ABS' C
  | 'INT' I:=0, D; 'BOOL' ABSENT:='TRUE';
    'WHILE' D:='ABS' BUF-'ABS' "0"; D 'GE' 0 'AND' D 'LE' 9
    'DO' I:=10*I+D; READP(BUF); ABSENT:='FALSE'
    'OD';
    ( ABSENT | PRINT("LETTER OR DIGIT EXPEXTED"); STOP); I
  )
);

'PROC' OPERATOR=('INT' PRIO,'REF' 'CHAR' OP)'BOOL';
( PRIORITY(BUF)=PRIO
| OP:=BUF; READP(BUF);
  ( BUF="="
  | (OP="<"|OP:="["|;OP=">"|OP:="]"|M(OP=":="));
    READP(BUF)
  );
  'TRUE'
| 'FALSE'
);
```

```
'PROC' PRIORITY=('CHAR' C)'INT';
(C= "@"|1
|:C=";" |2
|:C="$" |2
|:C=","|3
|:C=":"|4
|:C="|"|5
|:C="&"|6
|:C="=" 'OR' C="#" 'OR' C=">" 'OR' C="≤"
        'OR' C="[" # 'LE' # 'OR' C="]" # 'GE' # |7
|:C="+" 'OR' C="-" |8
|:C="*" 'OR' C="/"|9
|0
);

'PROC' FORMPROC=('INT' PRIO)'FORMULA';
(PRIO=10|PRIMARY
|'FORMULA' E:=FORMPROC(PRIO+1); 'CHAR' OP;
 'WHILE' OPERATOR(PRIO,OP)
 'DO' E:='HEAP' 'TRIPLE':=(E,OP,FORMPROC(PRIO+1))
 'OD';
 E
);

'PROC' LEFTASSOC=('REF' 'FORMULA' E)'BOOL';
('BOOL' MODIF:='FALSE'; 'FORMULA' (E)
| ('REF' 'TRIPLE' B);
  ('CHAR' C=C 'OF' B; 'INT' P=PRIORITY(C);
    (RIGHT 'OF' B
    | ('REF' 'TRIPLE' BR);
      ('CHAR' CR = C 'OF' BR; 'INT' PR=PRIORITY(CR);
        (PR=P
        | RIGHT 'OF' B := 'FORMULA'(LEFT 'OF' BR);
          LEFT 'OF' BR := 'REF' 'TRIPLE'(B);
          E:=BR;
          (C="-"| C 'OF' BR := (CR="+"|"-"|"+")
          |:C="/"| C 'OF' BR:= (CR="*"|"/"|"*")
          );
          MODIF:='TRUE'
        )
      )
    );
    MODIF
  )
| 'FALSE'
);
```

```
'PROC' CONST=('FORMULA' E)'BOOL';
( 'FORMULA'(E)
| ( 'INT' I); I 'GE' 0,
  ( 'REF' 'TRIPLE' B);
    CONST(LEFT 'OF' B) 'AND'
    (C 'OF' B = "+"  'OR'  C 'OF' B = "*") 'AND'
    CONST(RIGHT 'OF' B)
);

'PROC' LEFT=('REF' 'FORMULA' E)'REF' 'FORMULA';
('FORMULA'(E)|('REF' 'TRIPLE' B);LEFT 'OF' B | ERROR; E);

'PROC' RIGHT=('REF' 'FORMULA' E)'REF' 'FORMULA';
('FORMULA'(E)|('REF' 'TRIPLE' B);RIGHT 'OF' B | ERROR; E);

'PROC' REARRANGE=('REF' 'FORMULA' E)'BOOL';
( 'BOOL' MODIF:='FALSE';
  'WHILE' LEFTASSOC(E) 'DO' MODIF:='TRUE' 'OD';
  ( 'FORMULA'(E)
  | ('REF' 'TRIPLE' B);
    ( 'REF' 'CHAR' C = C 'OF' B; 'INT' PC=PRIORITY(C);
      'REF' 'FORMULA' L = LEFT 'OF' B, R = RIGHT 'OF' B;
      MODIF:=REARRANGE(L) 'OR' REARRANGE(R);
      ( 'CHAR' CL=NODE(L); LESS(R,L,PC) 'AND' PC>4 'AND' PC<10 'AND'
        PRIORITY(CL) 'NE' PC 'AND' PC 'NE' 7 'AND'
        'NOT' (CL="1" 'AND' C="/"  'OR'  CL="0" 'AND' C="*")
      | ( C = "/" | E :=: 'HEAP' 'TRIPLE' := (E,"*",L); L:=1
        |:C = "*" | E :=: 'HEAP' 'TRIPLE' := (E,"+",L); L:=0
        | EXCHANGEI(L,R)
        );
        MODIF:='TRUE'
      );
      ( L
      | ('REF' 'TRIPLE' BL);
          ( 'REF' 'FORMULA' LBL = LEFT 'OF' BL, RBL = RIGHT 'OF' BL;
            'REF' 'CHAR' CBL = C 'OF' BL;
            PRIORITY(C 'OF' BL) = PC
          | ( IDENTICAL(RBL,R) 'AND' CBL 'NE' C 'AND' (PC=8 'OR' PC=9)
            | E:=LBL; MODIF:='TRUE'
            |:LESS(R,RBL,PC) 'AND' PC>4 'AND' PC<10
            | EXCHANGEC(CBL,C);EXCHANGEI(RBL,R);MODIF:='TRUE'
            )
          )
      );
      ( 'NOT' MODIF 'AND' PC=7
      | 'CHAR' NODL:=NODE(L);
        ( 'NOT' CONST(R)
        | L:='HEAP' 'TRIPLE':=(L,"*",R); R:=0; MODIF:='TRUE'
        |:NODL="+" 'OR' NODL="*"
```

```
        | ( 'FORMULA' RIGHTL:=RIGHT(L); CONST(RIGHTL)
          | NEGATIVE(NODL); R:='HEAP' 'TRIPLE':=(R,NODL,RIGHTL);
            L:=LEFT(L); MODIF:='TRUE'
          |:CHSIGN(L)
          | R:='HEAP' 'TRIPLE':=(0,"-",R);
            (C="<"|C=">"|:C=">"|C="<"); MODIF:='TRUE'
          )
        );
        ( 'NOT' MODIF 'AND' (C="|" 'OR' C="&")
        | ( CC:=NODE(R); CC="T" 'OR' CC="F"
          | E:=(C="&"|:(CC="T"|L|FALSE)|:(CC="T"|:TRUE|L));
            MODIF:='TRUE'
          )
        );
        ( 'NOT' MODIF 'AND' PC=7 'AND' CONST(L) 'AND' CONST(R)
        | 'CHAR' C1:=C; C:="-";
          'WHILE' REARRANGE(E); ADD(E) 'DO' 'SKIP' 'OD'; MODIF:='TRUE';
          E:=( 'FORMULA' (E)
             | ( 'INT' I):
               ( I>0
               | (C1=">" 'OR' C1="]" 'OR' C1="#"|:TRUE|FALSE)
               | (C1="=" 'OR' C1="[" 'OR' C1="]"|:TRUE|FALSE)
               ),
               ( 'REF' 'TRIPLE' ): (C1="<" 'OR' C1="#"|:TRUE|FALSE)
             )
        );
        ( 'NOT' MODIF
        | ( C="-" 'OR' C="/"
          | ( IDENTICAL(L,R)
            | E:=( C="-" | 0 | 1 ); MODIF:='TRUE'
            )
          )
        );
        ( 'NOT' MODIF  'AND' C="*"
        | ( R
          | ( 'INT' IR):
            ( L
            | ( 'INT' IL):
              ( IL 'GE' 0 'AND' IR 'GE' 0 | E:=IL*IR; MODIF:='TRUE')
            )
          )
        )
      )
    );
    (MODIF|:REARRANGE(E)|:'FALSE')
);
```

```
'PROC' PRBUF=('CHAR' C)'VOID';
( ( 'NOT'(BUF="0" 'AND' C="=" )
  ¦ PRINT(
          ( BUF="¦" ¦ "  ¦  "
          ¦;BUF="&" ¦ " & "
          ¦;BUF="(" ¦ "<="
          ¦;BUF=")" ¦ ">="
          ¦ BUF
          ))
  );
  BUF:=C
);

'PROC' PR=('FORMULA' E,'INT' PRIO)'VOID';
( E
¦ ( 'INT' V);
  ( V=TRUE 'OR' V=FALSE
  ¦ PRBUF("'"); PRBUF((V=TRUE¦"T"¦"F")); PRBUF("'")
  ¦ ( V<0
    ¦ PRBUF('REPR'-V)
    ¦ 'STRING' S=WHOLE(V,0);
      'FOR' I 'TO' 'UPB' S 'DO' PRBUF(S[I]) 'OD'
    )
  ),
  ( 'REF' 'TRIPLE' B)¦( 'CHAR' C= C'OF'B; 'INT' P=PRIORITY(C);
                        (P<PRIO¦PRBUF("("));
                        PR(LEFT 'OF' B, P);
                        PRBUF(C); PR(RIGHT 'OF' B, P+1);
                        (P<PRIO¦PRBUF(")"))
                      )
);

'PROC' IDENTICAL=('FORMULA' E,F)'BOOL';
( E
¦ ('INT' CE);
  ( F
  ¦ ('INT' CF); CE=CF
  ¦ 'FALSE'
  ),
  ('REF' 'TRIPLE' BE);
  ( F
  ¦ ('REF' 'TRIPLE' BF);
    ( C 'OF' BE = C 'OF' BF
    ¦ ( IDENTICAL(LEFT 'OF' BE, LEFT 'OF' BF)
      ¦ IDENTICAL(RIGHT 'OF' BE, RIGHT 'OF' BF)
      ¦ 'FALSE'
      )
    ¦ 'FALSE'
    )
  ¦ 'FALSE'
  )
);
```

```
'PROC' ZERONE=('REF' 'FORMULA' E)'BOOL';
( 'FORMULA'(E)
| ( 'REF' 'TRIPLE' B);
  ( 'CHAR' C = C 'OF' B; 'BOOL' PLUS= C="+", MIN= C="-",
    TIM= C="*", DIV= C="/"; 'BOOL' MODIF:='FALSE';
    'PROC' M=('FORMULA' I)'VOID':(E:=I; MODIF:='TRUE');
    ( RIGHT 'OF' B
    | ( 'INT' BR);
      ( 'BOOL' ZERO = BR=0, ONE = BR=1;
        ((PLUS 'OR' MIN) 'AND' ZERO 'OR' (TIM 'OR' DIV) 'AND' ONE
        | M(LEFT 'OF' B)
        |:TIM 'AND' ZERO | M(0)
        |:DIV 'AND' ZERO | M(='ABS'"?")
        )
      )
    );
    ( 'NOT' MODIF
    | ( LEFT 'OF' B
      | ( 'INT' BL);
        ( 'BOOL' ZERO = BL=0, ONE = BL=1;
          ( ZERO 'AND' PLUS  'OR'  ONE 'AND' TIM
          | M(RIGHT 'OF' B)
          |:ZERO 'AND' (TIM 'OR' DIV) | M(0)
          )
        )
      )
    );
    ( 'NOT' MODIF | MODIF:=ZERONE(LEFT 'OF' B));
    ( 'NOT' MODIF | MODIF:=ZERONE(RIGHT'OF' B));
    MODIF
  )
| 'FALSE'
);

'PROC' NEGATIVE=('REF' 'CHAR' C)'VOID': C:=(C="+"|"-"|"+");

'PROC' CHSIGN=('REF' 'FORMULA' E)'BOOL';
( 'FORMULA'(E)
| ( 'REF' 'TRIPLE' B);
  ( 'REF' 'CHAR' C = C 'OF' B,
    'REF' 'FORMULA' L = LEFT 'OF' B, R = RIGHT 'OF' B;
    C="-" 'OR' C="+"
  | ( C="-" 'AND' NODE(L)="0" | E:=R; 'TRUE'
    | (CHSIGN(L)|NEGATIVE(C);'TRUE'|'FALSE')
    )
  | 'FALSE'
  )
| 'FALSE'
);
```

```
'PROC' CHSGN=('REF' 'FORMULA' E, X)'BOOL';
( 'REF' 'FORMULA' I:=X, J:=E; 'CHAR' CH; 'INT' N:=0;
  'WHILE' CH:=NODE(I); CH="+" 'OR' CH="-"
  'DO' J:=I; N+:=1;
       I:=('FORMULA' (I)|('REF' 'TRIPLE' B): LEFT 'OF' B)
  'OD';
  ( CH="0" 'AND' NODE(J)="-"
  | I:=X; E :=: 'HEAP' 'TRIPLE' := (0,"-",E);
    'TO' N-1
    'DO'
      ( 'FORMULA' (I)
      | ('REF' 'TRIPLE' B):(NEGATIVE(C 'OF' B); I:=LEFT 'OF' B)
      )
    'OD';
    'REF' 'FORMULA'(I):=('FORMULA'(I)|('REF''TRIPLE' B):RIGHT 'OF' B);
    'TRUE'
  | 'FALSE'
  )
);

'PROC' NEGFAC=('REF' 'FORMULA' E)'BOOL';
( 'BOOL' MODIF:='FALSE';
  ( 'FORMULA'(E)
  | ( 'REF' 'TRIPLE' B):
    ( 'CHAR' C = C 'OF' B,
      'REF' 'FORMULA' L = LEFT 'OF' B, R = RIGHT 'OF' B;
      ( C="*" 'OR' C="/"
      | MODIF:=(CHSGN(E,L)|'TRUE'|CHSGN(E,R))
      );
      ( 'NOT' MODIF
      | MODIF:=(NEGFAC(L)|'TRUE'|NEGFAC(R))
      )
    )
  );
  MODIF
);

'PROC' COPYTREE=('FORMULA' E)'FORMULA';
( E
| ( 'INT' C):C,
  ( 'REF' 'TRIPLE' B):'HEAP' 'TRIPLE':=
    (COPYTREE(LEFT 'OF' B), C 'OF' B, COPYTREE(RIGHT 'OF' B))
);

'PROC' SUBST=('CHAR' V, 'FORMULA' E, 'FORMULA' F)'FORMULA';
# SUBSTITUTE E FOR V IN F #
( F
| ('INT' C):(C='ABS' V | COPYTREE(E) | C),
  ('REF' 'TRIPLE' B):'HEAP' 'TRIPLE':=
      (SUBST(V,E,LEFT 'OF' B), C 'OF' B, SUBST(V,E,RIGHT 'OF' B))
);
```

```
'PROC' NEGATE=('FORMULA' E)'FORMULA';
( E
| ( 'INT');('CHAR' CC=NODE(E); CC="T" | FALSE | M(CC="F");TRUE),
  ( 'REF' 'TRIPLE' B);
  ( 'REF' 'CHAR' C = C 'OF' B,
    'REF' 'FORMULA' L = LEFT 'OF' B, R = RIGHT 'OF' B; C:=
    ( C="|" 'OR' C="&"
    | L:=NEGATE(L); R:=NEGATE(R); (C="|" | "&" | "|")
    |:C= "<" | "]"
    |:C= ">" | "["
    |:C= "[" | ">"
    |:C= "]" | "<"
    |:C= "=" | "#"
    | M(C="#"); "="
    ); E
  )
);

'PROC' WP=('FORMULA' S,Q)'FORMULA';
( S
| ( 'REF' 'TRIPLE' B);
  ( 'REF' 'CHAR' C = C 'OF' B,
    'REF' 'FORMULA' L = LEFT 'OF' B, R = RIGHT 'OF' B;
    C = ";"
  | SUBST(NODE(L),R,Q)
  |:C = "."     # CONDITIONAL STATEMENT #
  | ( R
    | ( 'REF' 'TRIPLE' BR);
      ( 'REF' 'FORMULA' S = LEFT 'OF' BR, T = RIGHT 'OF' BR;
        'HEAP' 'TRIPLE':=
        ('HEAP' 'TRIPLE':=(COPYTREE(L),"&",WP(S,Q)), "|",
         'HEAP' 'TRIPLE':=(NEGATE(COPYTREE(L)),"&",WP(T,Q))
        )
      )
    )
  | ERROR; 'SKIP'
  )
);

'PROC' REDUCE=('REF' 'FORMULA' E)'BOOL';
( 'FORMULA' (E)
| ( 'REF' 'TRIPLE' B);
  ( 'REF' 'FORMULA' L = LEFT 'OF' B, R = RIGHT 'OF' B,
    'CHAR' C = C 'OF' B;
    C = "S"
  | ( L
    | ( 'REF' 'TRIPLE' BL);
      ( 'REF' 'FORMULA' LBL = LEFT 'OF' BL, RBL = RIGHT 'OF'BL,
        'CHAR' CBL = C 'OF' BL;
        E:=( CBL = ";"
```

```
            | 'HEAP' 'TRIPLE':=(LBL, "$", WP(RBL,R))
            | WP(L,R)
            );
          'TRUE'
        )
      | 'FALSE'
      )
    | 'FALSE'
    )
| 'FALSE'
);

'PROC' DISTR=('REF' 'FORMULA' E)'VOID';
( 'FORMULA'(E)
| ( 'REF' 'TRIPLE' B);
  ( 'REF' 'FORMULA' L = LEFT 'OF' B, R = RIGHT 'OF' B;
    DISTR(L); DISTR(R);
    ( C 'OF' B ="*"
    | ( ( 'FORMULA'(R)
        | ( 'REF' 'TRIPLE' BR);
          ( 'CHAR' CBR = C 'OF' BR;
            (CBR="+" 'OR' CBR="-") 'AND' NODE(L) 'NE' "/"
          | E:='HEAP' 'TRIPLE':=
            ('HEAP' 'TRIPLE':=(L,"*",LEFT 'OF' BR), CBR,
             'HEAP' 'TRIPLE':=(COPYTREE(L),"*",RIGHT 'OF' BR) );
            DISTR(E); 'TRUE'
          | 'FALSE'
          )
        | 'FALSE'
        )
      | 'SKIP'
      | ( 'FORMULA'(L)
        | ( 'REF' 'TRIPLE' BL);
          ( 'CHAR' CBL = C 'OF' BL;
            (CBL="+" 'OR' CBL="-") 'AND' NODE(R) 'NE' "/"
          | E:='HEAP' 'TRIPLE':=
            ('HEAP' 'TRIPLE':=(LEFT 'OF' BL,"*",R),CBL,
             'HEAP' 'TRIPLE':=(RIGHT 'OF' BL,"*",COPYTREE(R)));
            DISTR(E)
          )
        )
      )
    )
  )
);
```

```
'PROC' CONSFAC=('REF' 'FORMULA' E)'BOOL';
( 'FORMULA'(E)
| ('INT' I);
  ( I<0
  | E:='HEAP' 'TRIPLE':=(1,"*",I); 'TRUE' | 'FALSE'
  ),
  ('REF' 'TRIPLE' B);(C 'OF' B = "*"|CONSFAC(LEFT 'OF' B)|'FALSE')
);

'PROC' FACTONE=('FORMULA' E)'BOOL';
( 'BOOL' MODIF:='FALSE';
  ( E
  | ( 'REF' 'TRIPLE' B);
    ( 'REF' 'FORMULA' L = LEFT 'OF' B, R = RIGHT 'OF' B;
      'CHAR' C = C 'OF' B;
      ( C="+" 'OR' C="-"
      | MODIF:= CONSFAC(L) 'OR' CONSFAC(R)
      );
      ('NOT' MODIF|MODIF:=(FACTONE(L)|'TRUE'|FACTONE(R)))
    )
  );
  MODIF
);

'PROC' ALMOSTIDENT=('FORMULA' E,F)'BOOL';
( E
| ( 'INT' EI);
  ( F | ('INT' FI); EI>0 'AND' FI>0 | 'FALSE'),
  ( 'REF' 'TRIPLE' EB);
  ( F
  | ( 'REF' 'TRIPLE' FB);
    ( C 'OF' EB = C 'OF' FB 'AND' IDENTICAL(RIGHT'OF'EB,RIGHT'OF'FB)
    | ALMOSTIDENT(LEFT 'OF' EB, LEFT 'OF' FB)
    | 'FALSE'
    )
  | 'FALSE'
  )
);

'PROC' SUM=('INT' SIGNE,SIGNF,'FORMULA' E,F,'REF''INT' PLM)'FORMULA';
( 'HEAP' 'FORMULA' S := COPYTREE(E);
  'INT' FACTOR=SIGNE*LEFTLEAF(E)+SIGNF*LEFTLEAF(F);
  CHANGELEFT(S, 'ABS' FACTOR);
  PLM:=(FACTOR<0|-1|:FACTOR=0|0|1);
  S
);

'PROC' LEFTLEAF=('FORMULA' E)'INT';
( E | ('INT' I):I, ('REF' 'TRIPLE' B):LEFTLEAF(LEFT 'OF' B));

'PROC' CHANGELEFT=('REF' 'FORMULA' E, 'INT' C)'VOID';
('FORMULA'(E)|('INT'):E:=C,('REF' 'TRIPLE' B):CHANGELEFT(LEFT'OF'B,C));
```

```
'PROC' ADD=('REF' 'FORMULA' E)'BOOL';
( 'BOOL' MODIF:='FALSE'; 'INT' PLM; 'FORMULA' E1;
  ( 'FORMULA'(E)
  | ( 'REF' 'TRIPLE' B);
    ( 'REF' 'FORMULA' L = LEFT 'OF' B, R = RIGHT 'OF' B;
      'CHAR' C = C 'OF' B;
      ( C="+" 'OR' C="-"
      | ( ALMOSTIDENT(L,R)
        | E:=SUM(1,(C="+"|1|-1),L,R,PLM);
          ( PLM<0 | E:='HEAP' 'TRIPLE':=(0,"-",E)); MODIF:='TRUE'
        | ( 'FORMULA'(L)
          | ( 'REF' 'TRIPLE' BL);
            ( 'REF' 'FORMULA' LBL = LEFT'OF'BL, RBL = RIGHT'OF'BL;
              'CHAR' CBL = C 'OF' BL;
              CBL="+" 'OR' CBL="-"
            | ( ALMOSTIDENT(RBL,R)
              | E1:=SUM((CBL="+"|1|-1),(C="+"|1|-1),RBL,R,PLM);
                ( PLM=0 | E:=LBL
                | E:='HEAP' 'TRIPLE':=
                  (LBL,(PLM>0|"+"|"-"),E1)
                );
                MODIF:='TRUE'
              )
            )
          )
        )
      );
      ('NOT' MODIF | MODIF:=(ADD(L)|'TRUE'|ADD(R)))
    )
  );
  MODIF
);

'PROC' RRR='VOID': REARRANGE(ROOT);

PRINT(("INPUT :",NEWLINE)); READP(BUF);
'FORMULA' ROOT:=FORMPROC(1); RRR;
'WHILE' REDUCE(ROOT) 'DO' 'SKIP' 'OD'; RRR;
'WHILE' NEGFAC(ROOT) 'DO' RRR 'OD';
DISTR(ROOT); RRR;
'WHILE' ADD(ROOT) 'DO' RRR 'OD';
'WHILE' FACTONE(ROOT) 'DO' 'SKIP' 'OD'; RRR;
'WHILE' ADD(ROOT) 'DO' RRR 'OD';
'WHILE' ZERONE(ROOT) 'DO' 'SKIP' 'OD'; RRR;
PRINT((NEWLINE, NEWLINE,"OUTPUT:",NEWLINE));
BUF:=" "; PR(ROOT,1); PRBUF(" ");
'TO' 5 'DO' PRINT(NEWLINE) 'OD'
)
```

APPENDIX B: EXAMPLES.

```
INPUT :
A + (B=C)/(D/D+C=B=1) = A + 1 \

OUTPUT:
 0
```

```
INPUT :
(X+1)*(X+2)*(X+3)*(X+4) \

OUTPUT:
 X*X*X*X+10*X*X*X+35*X*X+50*X+24
```

```
INPUT :
( 3>4 ! A*A=B*B=(A+B)*(A=B) ! 2>5 ) &
( 2=3*4=10 ! 8=2*2 ! A=B ) &
C<D \

OUTPUT:
 C=D<0
```

```
INPUT :
X:=X+1; Y:=Y+2 $ X>5 & Y<20 \

OUTPUT:
 X>4 & Y<18
```

```
INPUT :
Q:=Q+1; R:=R-B $ A=Q*B+R \

OUTPUT:
 B*Q-A+R=0
```

```
INPUT :
X>5 . (X:=0 # X:=1) $ X=1 \

OUTPUT:
 X<=5
```

```
INPUT :
X<50
.
( X<40 . (X:=3 # X:=4)
# X<60 . (X:=5 # X:=6)
)
$ X=4
\

OUTPUT:
 X<50 & X>=40
```

ONTVANGEN 8 SEP. 1977