TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

Expression optimization for the APDL-compiler

Wijshoff, Marcel

Award date:
1992

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

EB 363

2080

Expression optimization

for the APDL-Compiler.

By : Marcel Wijshoff.

Graduation report.

Coach : ir F.P.M. Budzelaar
Supervisor : prof.ir. M.P.J. Stevens
TUE, 18-2-92.

The faculty of Electrical Engineering Eindhoven University of Technology does not accept any responsibility regarding the contents of graduation reports.

Summary

The Digital Systems Group at the department of electrical engineering, Technical University Eindhoven, is developing a compiler for the APDL-language. The compiler, called APDL-compiler, has to deal with expressions which are not bound by any limit regarding its size. We will introduce a method for the construction of a rulebasesystem, that is used to reduce the size of expressions.
After an extensive literary search it was clear that, the subject was not dealt with as most programmers belief that expressions are already simplified by the user. In our case however this is not always true. The compiler itself could introduce a lot of expressions that could be simplified to true of false, indicating a constant decision. This decision can be made at compile time resulting in a general gain of storage-space and execution-time.
The constructed system consists of a group of small rulebases. The rules in these rulebases are matched on the expressions we want to simplify. If there is a match the expression is altered, but the algebraic meaning of the expression remains the same.
Each rulebase has a special task, so performing a specified mutation on an expression.
We constructed a group of rulebases that transform the target expression to a canonical form called the leftsorted form. On this form we try to perform a reduction which results in the decrease of the number of operators.
This reduction is based on the distributive property:

$$(a \text{ op}_1 \text{ b}) \text{ op}_2 (c \text{ op}_1 \text{ b}) = (a \text{ op}_2 \text{ c}) \text{ op}_1 \text{ b}.$$

We see this property, used from left to right, reduces the number of operators with one. We could use this property numerous times ending up with a form that not matches the lefthand side form of the property. We could then use some simple reductionrules, based on the Unity, Zero and Cancelation of *Not*-property, to reduce the expression even more.
To come to this canonical form there are some steps that must be taken first. We have to get rid of *minus* and *not*-operators as they could block some of the steps in a later rulebase. *Minus* operators are easily rewritten to a multiplication by the constant -1. *Not*-operators are placed as close to the operand as possible. This results in a *not*-operator followed by a variable. We will see this construction as a whole 'new' variable. As *exclusive-or*-operators, when written out in an *AND* and *OR*, introduce another *not*-operator we have to expand this *XOR* even before we process the *not*-operator.
Then we expand all expressions in the form of the righthand side of the distributive

property, to the form of the lefthand side, thus using this property the other way around. For getting to the final canonical form we have to use a rulebase to sort the expression onto a leftsorted form. This form simply defines that there may not be a same kind of operator to the left of a dyadic operator. This rulebase uses mainly the associative and commutative properties.

To construct the rulebasesystem we had to introduce a system which makes it easier to detect in which order the rules must be placed and which method to use in applying the rulebase. The rulebasesystem constructed in this way is able to deal with a variety of expression as long as the input conditions are met. For this system there are only a few expressions that could mess up the system. Most of those expressions could be simply blocked by a rulebase which checks these input conditions.

Contents.

1. Introduction. - 3 -
   1.1. APDL-expressions form. - 4 -
   1.2. Key ideas and definition. - 4 -

2. Basic definitions. - 5 -
   2.1. Universal definition. - 5 -
   2.2. Definitions for our rulebasesystem. - 6 -
   2.3. Termination, unambiguously and completion. - 8 -

3. Pattern recognition. - 9 -
   3.1. Matching a pattern on expressions. - 9 -
   3.2. Matching patterns on an expression. - 10 -
   3.3. Expression levels. - 12 -

4. Rewriting. - 13 -
   4.1. From equation to rule. - 13 -
   4.2. From equations to rulebase, theory. - 15 -
   4.3. Consequence of a rewrite. - 17 -

5. Basic equations. - 19 -

6. The theoretical background. - 22 -
   6.1. Collapsing. - 23 -
   6.2. Sorting. - 26 -
   6.3. A canonical form. - 27 -
      6.3.1. Left sorted canonical form. - 29 -
   6.4. Expansion. - 29 -
   6.5. Introduction of constants. - 30 -
   6.6. Removing monadic operators. - 30 -
      6.6.1. Register expressions. - 31 -
      6.6.2. Integer expressions. - 32 -

7. Possibilities for implementation . . . . . . - 33 -
7.1. How to order the rulebase. . . . . . - 33 -
7.1.1. Definitions. . . . . . - 33 -
7.1.2. The problems. . . . . . - 35 -
7.1.3. Dependencies. . . . . . - 36 -
7.2. Global Rulebase ordering. . . . . . - 48 -
7.3. Constructing a rulebase for minus reduction. . . . . . - 50 -
7.4. Constructing a rulebase for not reduction/shifting. . . . . . - 52 -
7.5. Constructing a rulebase for constant introduction. . . . . . - 53 -
7.5.1. Integer expressions. . . . . . - 53 -
7.5.2. Register expressions. . . . . . - 54 -
7.6. Constructing a rulebase for expand. . . . . . - 55 -
7.6.1. Integer expressions. . . . . . - 55 -
7.6.2. Register expressions. . . . . . - 58 -
7.7. Construction of a sort algorithm. . . . . . - 59 -
7.7.1. Integer expressions. . . . . . - 60 -
7.7.2. Register expressions. . . . . . - 61 -
7.8. Construction of rulebase, resulting in a canonical form. . . . . . - 64 -
7.9. Construction of the collapse rulebase. . . . . . - 66 -
7.9.1. Construction of the table. . . . . . - 66 -

8. Practical point of view. . . . . . - 69 -
8.1. Rulebase structure. . . . . . - 69 -
8.2. Subsort in detail. . . . . . - 72 -

9. Summary. . . . . . - 76 -
9.1. Integer expressions. . . . . . - 76 -
9.2. Register expressions . . . . . . - 77 -

10. Conclusions. . . . . . - 79 -
11. Recommendations. . . . . . - 81 -
12. References. . . . . . - 82 -
Appendix 1 : Program source code
Appendix 2 : Rulebase source code

1. Introduction.

The Digital Systems Group of the department of electrical engineering, Technical University Eindhoven is developing an APDL-compiler. The source language of this APDL-compiler is a PASCAL like language.
The source-language of this compiler could hold assignments like:
"var := expression;"
or conditional statements like:
"if expression1 == expression2 then ... else ...;".

This source text could be compiled and produce a program. The compiler introduces some expressions itself. We thus get a program with a lot of expressions. If we run this program every expression is evaluated. An expression consists of elements which have to be known at that point of the program. If we try to evaluate an expression during compilation we could encounter elements which are not known, such as variables.

We could however try to simplify an expression as much as possible, so that the program will have less complicated expressions to evaluate during the running of this program. This will results in a faster program. A simpler expressions mostly means a smaller expressions so the program will decrease in size also.
This could even result in an evaluate of the condition expression1 == expression2 during compilation. This condition then results in an unconditional if-statement, which could either mean the part after 'then' is used or the part after 'else', so the other part which is not used could simply be deleted from the program.

We can "optimize" an expression by rewriting the original expression 'a' to an equivalent expression 'b'. One problem is the term "optimal", because an expression is only optimal in a certain context, e.g. '2*a' could be more optimal than 'a+a' if '*' is more efficient than '+'.
We assume here that an optimal expression is the expression with the least number of operators as possible.

1.1. APDL-expressions form.

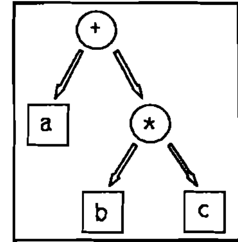

figure 1

An expression looks like a tree with monadic and dyadic operator nodes and terminals. In this tree every nonterminal is an operator, while the terminals are of the constant or variable type, e.g. the expression: a + (b * c) looks like figure 1.
Due to this dynamic structure it is easy to alter an expression in any way we like.

1.2. Key ideas and definition.

The key idea is a set of rewrite rules, which are descriptions of sets of expressions and their replacements.
E:a set of (linguistic, algebraic, symbolic) objects. (patterns)
R:a set of rules for which ($s \rightarrow_R t$): s,t ε E, a binary "reduction" relation $\rightarrow_R$ on E.
We write $s \rightarrow t$ instead of (s,t) ε $\rightarrow_R$ where possible.
It could be possible that $s \rightarrow t \rightarrow q$ and therefore we note this as $s \rightarrow^+ q$.[1]
We further introduce an ordering $\succ$ such that for two patterns s and t, $s \succ t$ means that pattern s has more operators than t.
Our aim is thus: $s \rightarrow^+ t$ and $s \succ t$ and this in such a way that t would be identical to a constant, which could lead to decisions now made at compile time, and so lead to an optimization of the program code. It could be possible that an expression is only partly rewriteble to a smaller expression, but this would also be an improvement.

---

[1]We could now make a set of rules like :

$s \rightarrow t$ 1.
$t \rightarrow q$ 2.

note 1 : we could introduce a third rule : $s \rightarrow q$ 3. We see that two transformations are then possible on the same expression which results in two different new expressions. We see that now the order of the rules is thus important because the rule which is placed first will be the one which is used while the second one is not!.)
note 2 : swapping rule 1 and 2 makes the set of rules miss the conversion $s \rightarrow t \rightarrow q$.

2. Basic definitions.

2.1. Universal definition.

We define:

P : a set of all linguistic, algebraic, symbolic objects. (we call these patterns).

Examples: { 'a const' , 'an expression' , 'a and b' , 'a const + a' , ...} ε*P*.

r : a binary "reduction" relation → on *P*. s → t ε *r* (s,t ε *P* and (s=t)≡true) we call a rule.

We can, using such a rule, alter an expression s in an expression t.

Examples : 'a + 0 → a' , 'a const or true → true' , '1 → a const' are rules.

R : a set of rules, and call this a rulebase. (Each rulebase is assigned an arbitrary unique name.)

Example:

rulebase example: {

'a + 0	→	a'	,
'a const or true	→	true'	,
'1	→	a const'	}

This is of course just an example rulebase. We want to show that there are no constrains to the contents of a rulebase.

E : a set of linguistic, algebraic, symbolic objects. (expressions).

Note: *e* ε *P* if and only if *e* ε *E*.

2.2. Definitions for our rulebasesystem.

We introduce here also a distinction between two types of expressions: an Integer expression, which returns an integer-type result, and a Register expression which returns a register-type result.

An expression is a string that is accepted by the following DFA:

Exp: var.
| const.
| Mon-op,Exp.
| dyadicoperator,Exp,Exp.
Var: name.
| name[exp].
Mon-op: '-' | '+' | rev | not.
Dya-op: '-' | '+' | '*' | or | and | xor | '>' | '<' | '=' | '>=' | '<=' | '<>' | '/'.

Of course checking on the correctness of the expression is done, but that is the task of the parser. We will in our part only consider valid expressions.

We have to restrict the definition of a rulebase, so it is useful to us:

R : an **ordered** set of rules which is called a rulebase. (Each rulebase is assigned an arbitrary unique name.)

We have now restricted rulebase R ε *R* to an ordered set of rules. The appearance of the rules is sequential. The rules don't necessarily have to be mutual exclusive. The order of the rules is thus very important. We will deal with this problem of order thoroughly later.

The rules are ordered: $s_1 \rightarrow t_1 .. s_e \rightarrow t_e$.
(e is the number of rules in the rulebase).

If we also make it possible for one rulebase R_m to call another rulebase R_c we can redefine a multi-related rulebase-system *M*:

M : a set of rulebases R_i ($i \varepsilon N$) ε *R* which have the possibility to call upon each other. (R_m : call R_c).

We have to introduce some command-words here in a rulebase to realize this special call function. (m and c stand for main and called rulebase, c and m do not have to be different rulebases, so it gives the possibility for recursion).
We see *M* is not an ordered set. A rulebase ε *M* is called by another rulebase by name, which is the reason for unique names for rulebases. Therefore it's not needed to order the set *M*. One special rulebase is reserved however to be used as a start rulebase.

Advantages:

- We can avoid useless matching of patterns which don't occur anyway in an expression.

- We could see every rulebase as a procedure. The start rulebase could be seen as the main program, while it is calling the "procedures" (rulebases). We can see the resembles in modern computer languages, making it possible to use this construction of the rulebase as a very flexible tool.

2.3. Termination, unambiguously and completion.

We cite Mohan and Srivas [11]: 'In any programming formalism, it is desirable that a function definition satisfies three properties:

P_1 : Evaluation of terms invoking the defined functions on basic operator term argument must *terminate*.

P_2 : The definition must be *unambiguous*,i.e., every evaluation of the same term must yield the same result.

P_3 : The definition must be *complete*,i.e.,evaluation must be possible for **every** invocation of the defined function on basic operator term arguments.'

Ad.P_1: In our multi-related-rulebase-system we use sequential rulebases. We thus have to satisfy property P_1 for every rulebase separately and thus fulfil the property P_1 for the whole system.

Ad.P_2: As our system is based on a sequential matching procedure, there is only one possible way an expression is processed to reach the end of the system. Therefore P_2 is always satisfied.

Ad.P_3: As we will see later we rewrite every expression to a canonical form, which is, if possible, unique for a group of expressions which have the same algebraic meaning. From that point on, the rewrite procedure is thus the same for every expression of that group. So P_3 is thus satisfied.

We have to keep these properties in mind as we construct our rulebasesystem.

3. Pattern recognition.

We want to make a rewrite system that recognises certain patterns in an expression. We can, if we have recognised such a pattern, act upon it and alter the expression to something we think is more useful. We will see that a pattern could be used to recognise more than one expression, which will lead to a reduction of the number of patterns. We will also see that different patterns can recognise the same expression, which could cause some problems regarding determinism, but we will deal with this problem later.

3.1. Matching a pattern on expressions.

The input of the expression-optimization is of course, an expression. Our rulebase *M*, however, is constructed basicly of patterns. We have to find a way to map a pattern on an expression.

Although *E* is a subcollection of *P* it is possible to project every $p \varepsilon P$ on $e \varepsilon E$.

$$\forall\ e \in E, \exists\ p \in P : \text{Proj}(p) = e\ ;$$

It is frequently possible to project *p* on more than one *e*, so on a subset of *E*.

$$\exists\ V \subset E, \exists\ p \in P : \text{Proj}(p) = V\ ;$$

This process of projecting *p* on *e* successfully is called matching.

Concluding from the above there could exist a subset *V* of *E* which could be matched by a single pattern *p* of *P* successfully.
An example:
p = 'A const';
e_0 = '0';
e_1 = '1';
e_2 = '2';
Here e_0, e_1 and e_2 are all matched with *p*.

3.2. Matching patterns on an expression.

Matching can be done at more than one level; we can see an expression in different ways (we look with a specific depthrange to it).
For example the expression: 'true or false' could be matched with:

'an expression'			(a)
'an expression'	'or'	'an expression'	(b)
'an expression'	'or'	'a constant'	(c)
'a constant'	'or'	'an expression'	(d)
'a constant'	'or'	'a constant'	(e)
'an expression'	'or'	'false'	(f)[1]
'true'	'or'	'an expression'	(g)
'a constant'	'or'	'false'	(h)
'true'	'or'	'a constant'	(i)
'true'	'or'	'false'	(j)

We see that one single expression can be matched by many different patterns.

Some of these patterns are useless for matching this expression, although we could for instance use (b) to simplify the two expressions on both sides first and then, as a result, simplify the whole expression by matching with (a). We note that the order in which the rewrite rules are placed is very important.

In the example we prefer (f) and (g) and do not need (h) because:

$$(f) \cup (g) = (h);$$

Further note the order of the example: if we use (a) we don't need the rest because (a) > (b..h), (a) gives a matching before b..h can do.

[1] Only a few patterns can make a match that will lead to a simplification of the expression. Only the last 5 (f..j) can distinguish that the expression is a possible candidate for simplification. A match in (e) could lead to an evaluation which could simplify the expression as well.

It is further important to note the change the rewrite rule implies. If a rule converts an expression to another expression this new one can still match another rule in the same rulebase. We have to take this possibility into account and act upon it. This possibility could be wanted or not. This is depending on the situation.
One possibility could be to start the rulebase over again, another is to exit the rulebase, doing nothing further at this stage.

We conclude that the subset V_i and V_j of E, which could be match by p_i and p_j of P, don't have to be disjunct.

In the graph, which is an example, there are 3 patterns involved:
p1 : 'a const' and TRUE
p2 : FALSE and TRUE
p3 : TRUE and 'a const'

and 4 expressions:
e1 : 3 and TRUE
e2 : FALSE and TRUE
e3 : TRUE and 4
e4 : TRUE and TRUE.

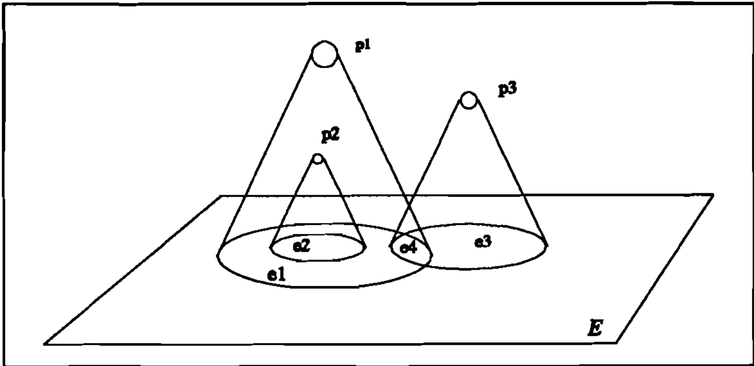

figure 2

We can imagine the expression space as a plane. The patterns are projected on it as a shade from a point projected by a light source.
We can see that indeed some patterns match more than one expression for example in the area of e1 there could also be: '4 and TRUE','5 and TRUE' and so on.
Also there are some expressions like e2 and e4 which are matched by more than one pattern (e2 by p1 and p2, e4 by p1 and p3.)

3.3. Expression levels.

Expressions can consist of more subexpressions, e.g., the expression a + b consist of two subexpressions: a,b.

In this way we can distinguish different levels in expressions.

An expression like (a + b) + c has an operator level range of 2, we have to go through 2 levels if we want to reach the last operator.

While searching a tree for a special pattern, we scan, with a pattern, through several levels. In the example above we scan through two levels. Sometimes this could be more or less.

Example: (a + b) + c is an expression which could be matched twice with the pattern a + b, once matching x + c (with x=a+b), and once on the subexpression a+b itself.

4. Rewriting.

4.1. From equation to rule.

Rewriting is based on the property of two expressions having the same algebraic meaning, stated in an equation.

$$s = t \ \{ s,t \varepsilon E \}$$

If we have such an equation, we could construct two rules:

$$\{ s,t \in E \}: s=t \Leftrightarrow s \rightarrow_R t \wedge t \rightarrow_R s;$$

Mostly only one rule is desired, so:

$$\{ s,t \in E \}: s=t \Rightarrow s \rightarrow_R t \vee t \rightarrow_R s;$$

In this way we can construct a directed rule.
We call the source *s* of a rule the lefthand side of a rule. We now define a function that we will need later:
LHS(r) = s iff $r : s \rightarrow_R t$.
RHS(r) = t iff $r : s \rightarrow_R t$.
Note: we only have a rule here which converts an expression into another, not a pattern. This implies that we have to make a rule for every possible mutation of an expression. We can however also try to pack more than one rule into a set of rules that have a common part, using patterns. We can then use this common part and try to make a more common rule that will process the complete set at once.

Consider a group(set) G of rules r_i (i ε [1..N]).
In G : e_i ε LHS(r_i) (e_i ε E).
In G : if $e_i = e_k$ => $r_i = r_k$ for all i,k ε [1..N] which implies:RHS(r_i)≡RHS(r_k) for all i,k ε [1..N].
For p ε P : Proj(p) = e_i for all i ε [1..N] <u>and</u> Proj(p) doesn't match outside the group G.

If these 3 conditions are satisfied we can replace G with only one rule containing p for all e_is.
Note 1: these considerations are also true if we take a $p \varepsilon P$ instead of $e \varepsilon E$.
Note 2: e_i can be a subexpression of a whole lefthand side of a rule.

An example:	equations:	0 * 1	=	0	(1)
		0 * 2	=	0	(2)
		0 * 3	=	0	(3)
		...			(4..n)
		0 * a	=	0	(n+1)
	rules:	0 * 1	→	0	(1)
		0 * 2	→	0	(2)
		0 * 3	→	0	(3)
		...			(4..n)
		0 * a	→	0	(n+1)

Rule (1) to (n) can be put into a group G and then the numbers 1 till n in the rules can be replaced by each other. For instance, if we take rule (3) and put the 3 of this rule on the place of the 1 in rule (1), we get again rule (3). If we assume n is the biggest integer possible in an expression we can take this group G together with the rest of the rules to:

0 * 'a const'	→	0	(1)
0 * a	→	0	(n+1)

We can again take these two rules together and construct one rule instead:

0 * 'expression'	→	0	(1)

In this way we can reduce the number of rules to a minimum. Rewriting can now be defined as applying rules so that the outcome of the rewrite is (sub)optimal.

4.2. From equations to rulebase, theory.

We want to have a method to convert a set of equations to a set of (directed) rules. In literature we can find some methods to extract a 'complete' rulebase out of a set of equations.

We define for this purpose *Eq* : a set of equations.

One method for creating a rulebase *R* out of a set of equations *Eq* is the one designed by Knuth and Bendix'70 (see H.Aït-Kaci & M.Nivat p39 [19] for further details).

We use the following inference rules which form a standard completion procedure:

(1) *Orient*: $(Eq \cup \{s \dot{=} t\}, R) \vdash (Eq, R \cup \{s \to t\})$ if: $s \succ t$.

(2) *Deduce*: $(Eq, R) \vdash (Eq \cup \{s = t\}, R)$ if: $s \leftarrow_R u \to_r t$.

(3) *Delete*: $(Eq \cup \{s = s\}, R) \vdash (Eq, R)$

(4) *Simplify*: $(Eq \cup \{s \dot{=} t\}, R) \vdash (Eq \cup \{u \dot{=} t\}, R)$ if: $s \to_R u$.

(5) *Compose*: $(Eq, R \cup \{s \to t\}) \vdash (Eq, R \cup \{s \to u\})$ if: $t \to_R u$.

(6) *Collapse*: $(Eq, R \cup \{s \to t\}) \vdash (Eq \cup \{v = t\}, R)$ if: $s \to_R u$ *by rule* $l \to r \in R$ *with* $s \rhd l$

The symbol $\rhd$ denotes the *specialization ordering, iff some subterm of s is an instance of l, but not vice versa.*[1]

(1) **Orient** turns an equation $s \leftrightarrow t$ that is orientable ($s > t$) into a rewrite rule. Since 1^- is greater that 1 for any reduction ordering (or an infinite derivation would be possible), the equation $1^- \leftrightarrow 1$ can only be oriented it the direction $1^- \to 1$.

(2) **Deduce** adds equational consequences to, E but only those that follow from back-to-back rewrites $s \leftarrow_R u$ and $u \to_R t$. For example, the rules $x.x^- \to 1$ and $1.x \to x$ can both be applied to the term 1.1^-. The first rewrites this term 1^-, from which the new equation $1^- \leftrightarrow 1$ can be deduced. As we will see, only consequences of certain 'critical' peaks need to be considered.

(3) **Delete** removes a trivial equation $s \to s$. An equation $x.1 \leftrightarrow x.1$, for example, would be candidate for deletion.

(4) **Simplify** rewrites either side of an equation $s \leftrightarrow t$. For example, given a rule $1.x \to x$, an

[1] If there is rewrite rule r applied to s and the result is t then t is an instance of s.

equation 1.1→1 would be replaced by 1↔1.

(5) **Compose** rewrites the right-hand side t of a rule s→t, if possible. For example, given a rule x^-→x, the rule (x.1).1→x^-.1 would be replaced by (x.1).1→x.1.

(6) **Collapse** reduces the left-hand side of a rule s→t and turns the result into an equation u↔t, but only when the rule l→r being applied to s is smaller in some sense (embodied in $\rhd$) than the rule being removed. In practice, we use the (proper *specialization* ordering as $\rhd$. In the ordering, s $\rhd$ l if a subterm s is an instance of l(but not vice-versa). For example, a rule x^-.y→x.y collapses to x.y↔x.y in the presence of a rule x^-→x. The age of the two rules may also be taken into account when each left-hand since in an instance of the other, making older rules smaller.

We write (Eq,R) $\vdash$ (Eq',R') to indicate that the pair (Eq',R') can be obtained from (E,R) by an application of an inference rule. A (possible infinite) sequence $(Eq_0,R_0) \vdash (Eq_1,R_1) \vdash$... is called a derivation from (Eq_0,R_0). The limit of a derivation is the pair (Eq^∞,R^∞) of the set $U_i \cap_{j\geq i} Eq_j$ of all persisting equations and the set $U_i \cap_{j\geq i} R_j$ of all persisting rules.

A completion procedure is a program that accepts as input a set of equations Eq_0, a Rewrite system R_0, and reduction ordering > containing R_0, and uses the above inference rules to generate a derivation from (Eq_0,R_0). We say that a completion procedure fails for a given input, if Eq^∞ <> 0. A completion is correct, if R^∞ is complete and $Eq^\infty = 0$.

This concludes the discussion of the theory. This procedure will produce a rulebase which is not ordered, that is all the rules are placed into a random order. The rulebase source-patterns thus have to be disjunct, e.g., if an expression is tried to be matched, there is at most one pattern that will match the expression. However, we want a rulebase that is matched sequentially, so some patterns don't have to be disjunct. Furthermore, we have a system with multiple rulebases, which is not so in the method above.
The theory above is constructed upon an ordering that we sometimes do not follow.
The definitions above are nevertheless useful for construction of a rulebase. We will therefore partly use this method. We will also introduce a method that will deal with the problem of ordering rules in a rulebase and rulebases themselves.

4.3. Consequence of a rewrite.

We have to consider the consequences of a rewrite. A rewrite alters the structure of the expression tree. Mostly the level range of the rewrite, so the levels of the alteration is equivalent to the operator level range. We have to consider that if a rule, consisting of a pattern and its rewrite pattern, is applied, the levels under it alter too. So if a rewrite rule is applied to a certain expression we have to keep in mind that the levels in the operator level range also change.

An example:

The expression: ((a + b) + c) * d

The rule: (a + b) * c → (a * c) + (b * c) : (displayed in figure 3)

Appliance:

((a + b) + c) * d →

((a+b) * d) + (c * d) →

((a * d) + (b * d)) + (c * d).

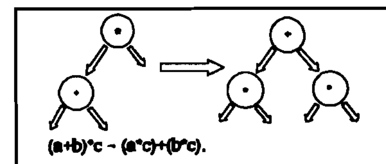

figure 3

We see that at first we couldn't apply the rule to the a,b,d couple. If however, we had applied it to the (a+b),c,d couple first, we could.

We altered the expression in a lower level by using the rewrite rule. This second rewrite, could result in a rewrite even lower.

This makes it likely to make a rulebase a top-down design, working from the root to the leaves.

In the figure we see again that the expression in a lower level changes as we apply a rule like this.

If, however, we do a rewrite at a level below, this could mean that, at the current level, now a pattern could be matched which couldn't be made if we didn't make the rewrite at the lower level.

A pattern that has an operator level range greater than one, so is depending on more than the current level, could be matched after the lower levels are rewritten.

An example:

The expression: ((a + b) * c) * d

The rule: (a + b) * c → (a * c) + (b * c) : (the same as above)

Appliance:

((a + b) * c) * d →

((a * c) + (b * c)) * d) →

((a * c) * d) + ((b * c) * d).

We can see that in this case we first have to apply the rule to the a,b,c couple and as a result we can use it again to the (a*c),(b*c),d couple, but this last is at a higher level. So using the first (top-down) technique we had not found the second rewrite. This makes it likely to use the bottom-up design in a rulebase, working from the leaves to the root. In the figure we see this also: changing the current level has consequences in the levels above.

Concluding, we have to use rules, which also have to be aware of alterations in the lower and upper levels. With each rule we introduce in a rulebase, we have to consider the consequences of the rewrite to the levels below and above. We will deal with this problem later.

5. Basic equations.

A rulebase is always based on an algebraic foundation. In our system we only take the most simple equations based on some simple algebraic properties:

1 the Associative property.	A.
2 the Distributive property.(left and right).	D.
3 the Commutative property.	C.
4 the Unity property of one.	U.
5 the Zero property of 0.	Z.
6 the cancelation of *NOT* (symbol of not: ¬).	N.
7 the expansion of *XOR*.	X.
8 the De Morgan rule.	M.
9 the *mInus* rules.	I.
10 the Evaluation of constants.	E.

Equations: (for the following equations, 'op' is a symmetrical operator !)

$(a\,op\,b)\,op\,c = a\,op\,(b\,op\,c).$	A_1
$(a + b) * c = (a * c) + (b * c).$	D_1
$c \wedge (a \vee b) = (a \vee c) \wedge (b \vee c).$	D_2
$c \vee (a \wedge b) = (a \wedge c) \vee (b \wedge c).$	D_3
$(a\ op\ b) = (b\ op\ a).$	C_1
$a * 1 = a.$	U_1
$a \wedge 1 = a.$	U_2
$a \vee 1 = 1.$	U_3
$a \oplus 1 = \neg a.$	U_4
$a * 0 = 0.$	Z_1
$a \wedge 0 = 0.$	Z_2
$a \vee 0 = a.$	Z_3
$a \oplus 0 = a.$	Z_4
$a + 0 = a.$	Z_5
$\neg\neg a = a.$	N_1
$\neg 1 = 0.$	N_2
$\neg 0 = 1.$	N_3
$a \oplus b = (a \wedge \neg b) \vee (\neg a \wedge b)$	X_1
$\neg(a \vee b) = \neg a \wedge \neg b.$	M_1
$\neg(a \wedge b) = \neg a \vee \neg b.$	M_2
$a - b = a + -b$	I_1
$-a = a * -1$	I_2
$C_1 // anyop // C_2 = C_3$	E

From these simple equations we can deduct some equations that are obvious to us but are quite timeconsuming if we have to deduct them with only the basic rules, e.g., we will try to construct a new equation with --a. Note that we, humans, mostly know which way we want to go. If the outcome of such a deduction is not clear, it could take some extensive effort even done by a computer. Note that these equations are not directed.

--a	=
-(-(a))	= (I_2)
(-(a))*-1	= (I_2)
(a*-1)*-1	= (A_1)
a*(-1*-1)	= (E)
a*1	= (U_1)
a	

We now deduced the equation --a = a. There are other ways to come to the same result

but there are, however, also some ways to come to an equation like for example --a=----a. The following deductions are used frequently and therefore done here only once and we can refer to them later.

$a\wedge\neg a=$	(if a=1)	$a\vee\neg a=$	(if a=1)
$1\wedge\neg 1=$	(N)	$1\vee\neg 1=$	(N)
$1\wedge 0=0$	(Z)	$1\vee 0=1$	(Z)
$a\wedge\neg a=$	(if a=0)	$a\vee\neg a=$	(if a=0)
$0\wedge\neg 0=$	(N)	$0\vee\neg 0=$	(N)
$0\wedge 1=0$	(U)	$0\vee 1=1$	(U)
concluding:	$a\wedge\neg a=0$	concluding:	$a\vee\neg a=1$

$a\wedge a=$	(Z)	$a\vee a=$	(U)
$(a\vee 0)\wedge(a\vee 0)=$	(D)	$(a\wedge 1)\vee(a\wedge 1)=$	(D)
$a\vee(0\wedge 0)=$	(Z)	$a\wedge(1\vee 1)=$	(U)
$a\vee 0=$	(Z)	$a\wedge 1=$	(U)
a		a	

6. The theoretical background.

The expressions offered to our rewrite system are of a variety of forms. Some of these expressions however have the same mathematical meaning. Our aim is as mentioned before, to reduce the number of operators in an expression, and if possible to reduce an expression to a constant expression.

To get an impression on the number of operators during the rewrite process, we look at figure 4, and see on the left side that a whole group of expressions is first forced to a canonical form, which is fairly unique. This gives an expression fewer possible forms which in its turn reduces the number of rules in a rulebase.

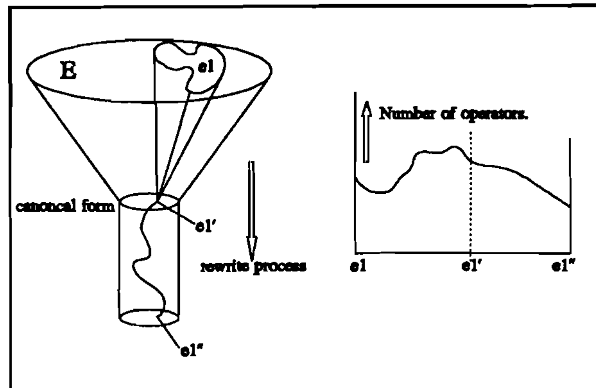

figure 4

Then this expression is further processed using a collapse mechanism discussed later.

On the right side we see that an expression could gain on the number of operators in the first part towards the canonical form, but later it will loose some operators as well. Mostly the number of operators at point e1'' is less that e1. E1' as the canonical form mostly holds more operators then e1'', as is shown in the figure. Note that the expression could have local minimums regarding the number of operators, but we have to avoid seeing these points as points where the expression could no longer be reduced.

This strategy results in some steps toward the canonical form, using different rulebases after each other. After a canonical form is reached there is a group of rulebases that reduces the number of operators step by step. Then the results are evaluated to search for some expressions that can still be reduced.

We will first handle the part where the number of operators is reduced. We then know what input conditions there are for this part, which are the output conditions of the part which produces the canonical form. We thus handle the part before construction of the canonical part after the collapse part. It's a sort of working back from end to the beginning of the process. This is due to in and output conditions that will be handled in a later chapter.

6.1. Collapsing.

The equations we use to reduce the number of operators are based on the distributive properties:

$$(a + b) * c = (a * c) + (b * c). \quad D_1$$
$$c \wedge (a \vee b) = (a \vee c) \wedge (b \vee c). \quad D_2$$
$$c \vee (a \wedge b) = (a \wedge c) \vee (b \wedge c). \quad D_3$$

These properties are used from the righthand side to the lefthand side. Using these properties the number of operators will diminish by one each time. We call this collapsing. One can consider a random search for these properties to use them on an expression but it is more effective to rewrite the expression into a canonical form first.
Further it is efficient to take the most common variable/subexpression in the whole expression and make this variable/subexpression the variable/subexpression that is taken out of the parenthesis first. We will call this the common variable.

Proof: Assume there is a most common variable *a*, which is n times in the expression while variable *b* is m times in the expression ($m \leq n$). If we take *a* out of parenthesis we gain maximal n-1 number of operators. If we take *b* out of parenthesis, we gain m-1 number of operators.
We can now consider tree cases:

1 all *b*s are in subexpressions where *a*s are too.

2 there are some *b*s that are in a subexpression with no *a*s.

3 all the *b*s are in subexpressions that contain no *a*s.

Case 1:

Let us assume we take *b* out of parenthesis, so we have a subexpression connected to *b* with m subterms. So now we can take *a* in the subterm out of parenthesis, and we can take *a* out of parenthesis outside of the subterm from *b*.

Total gain: m-1 (*b*) + m-1 (*a* inside) + n-m-1 (*a* remaining) = **m+n-3.**

If we take *a* out first (gain: n-1) we have all the *b*s inside the subterm and so we can take out m *b*s (gain: m-1).

Total gain: n-1 + m-1 = **n+m-2.**

Case 2:

We again try to take out *b* first.(gain: m-1), then we assume that there are r *a*s in the subterm of *b* so we can take them out too (gain: r-1). Then we can still take out *a*s outside the subterm (gain: n-r-1).

Total: m-1 + r-1 + n-r-1 = **n+m-3**.

If we take out *a* first (gain: n-1) and then take out the remaining *b*s in the subterm of *a* (gain: r-1), we can still take out m-r *b*s outside the subterm (gain: m-r-1).

Total: n-1 + r-1 + m-r-1 = **n+m-3**.

Case 3:

We take out *b* again first (gain: m-1) and in the subterm there are no *a*s, so outside the subterm we can take out *a* completely (gain: n-1).

Total: m-1 + n-1 = **n+m-2**.

Taking out *a* first has no consequence for the consideration above.

Total: **n+m-2**.

As we can see taking *a* out first has only an effect in case 1. In the other two cases it doesn't matter which we take first.

As the aim is to minimise the number of operators we consider this method a good one. Furthermore we have to consider the possibility that if we have taken out a variable out of parenthesis, we can update our expectations about what variable will make the most gain now. This means that the above presentations can be reconsidered.

Assume we have p variables a_i (i ε [1..p]). Variable a_i occurs O[i] times. We take a_m out of parenthesis:

$$\forall i: O[m] \geq O[i];$$

If we have taken a_m out of parenthesis we have to consider the most frequent variable again. (While taking a_m out of parenthesis we have to update O[m]).

We continue this process until there are no more variables left that are considered gaining anything.

Totally we have gained at this level:

$$\text{Gain} = \sum (\textit{number of the currently most frequent var}) - 1$$

In this way we each time achieve the maximal gain.

In the subterms (subexpressions) we see a new but similar expression which can be dealt with in the same way as described above.

Using this property we are assured we fulfil the goal of a maximal gain.

Note1: to use this property we have to make sure that the two subexpressions in question have to be adjacent. This is something we have to take care of in advance. We call this sorting.

Note2: we also have to satisfy the condition that the variable in question is the most right one of the expression.

Note3: To minimise the search for a matching pair of subexpression we assume a canonical form.

6.2. Sorting.

As stated in the collapse-chapter we want to make two nodes, in which the same variable is present, adjacent. This means we have, in some way, shift these two nodes towards each other, and make the collapsing possible. We call this sorting. We sort out some subtrees so they become adjacent. If we want to sort, we first have to prove the property: ('op' is a symmetrical operator)

(a op b) op c : a,b, and c can appear in any order.

Proof: (6 possible permutations)

(a op b) op c = (C)	1
(b op a) op c = (A)	2
b op (a op c) = (C)	
b op (c op a) = (A)	3
(b op c) op a = (C)	
(c op b) op a = (A)	4
c op (a op b) = (C)	
c op (b op a) = (C of 4)	5
a op (c op b)	6

Note that 'op' is the same operator each time.

If we have a tree of operators which are the same and if they are symmetrical operators we can sort them in any way we want using the property mentioned above. It states that we can exchange the position of two subexpression (a,b,c in the above property) if we want to.

This method is used to make two subexpressions adjacent, and so satisfying the condition 1 stated in the previous chapter.
This method can also be used to sort a subexpression and so satisfying condition 2 also.

Again we have to satisfy a few conditions before we can use this property.

1. - a canonical form to speed up the search through a tree and reduce the number of rules used.
2. - subexpressions are already sorted to speed up the variable match here.
3. - The variable in question must be known (chosen in advance).
4. - We have to eliminate monadic operators as much as possible.

From all above we conclude that a canonical form is essential.

6.3. A canonical form.

We want an expression as a canonical form to reduce the number of mutations of an expression, that is the number of possible forms that an expression can appear in.

E.g.: a + b + c + d can be represented in several forms:

(a + b) + (c + d) (1)

a + (b + (c + d)) (2)

((a + b) + c) + d (3)

(a + (b + c)) + d (4)

a + ((b + c) + d) (5)

Graphical it looks like figure 5.

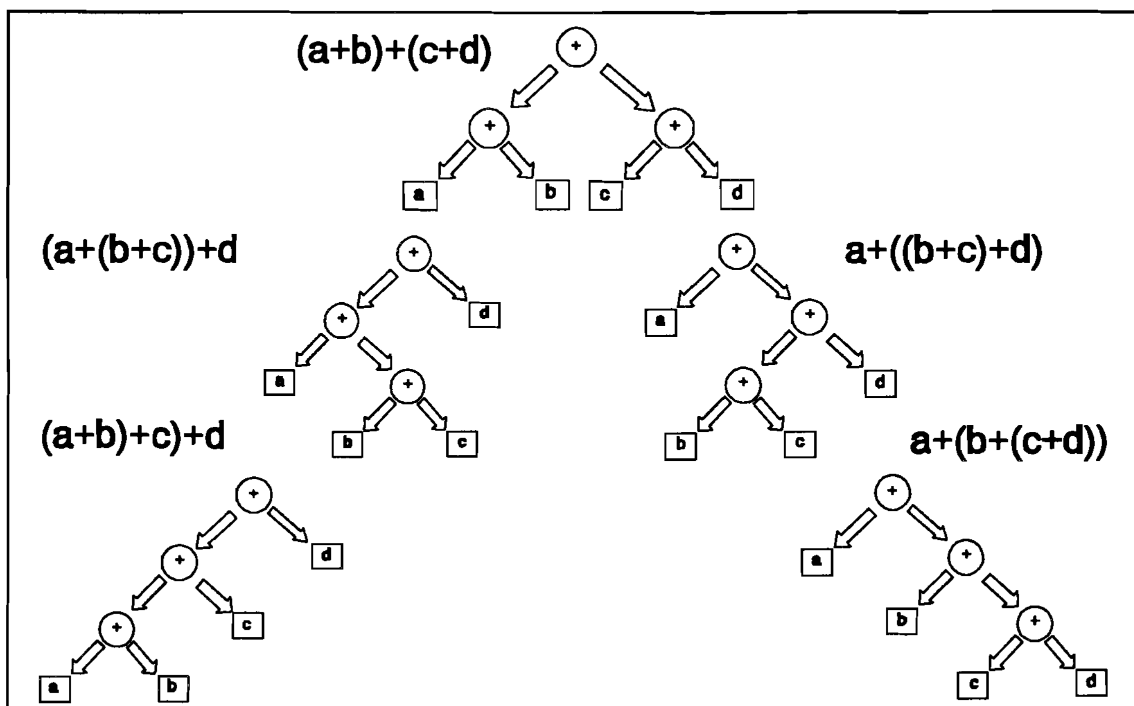

figure 5

If we see the number of '+' as n we could express the number of mutations of a single expression, M(n), as follows:

$$M(0)=1$$
$$M(1)=1$$
$$M(n)=\sum_{p=0}^{n-1} M(p).M(n-1-p) \quad \{for: n>1\}$$

We see that indeed:

M(3) =
M(0)M(2)+M(1)M(1)+M(2)M(0) =
M(0) {M(0)M(1)+M(1)M(0)} + M(1)M(1) + {M(0)M(1)+M(1)M(0)} M(0) =
1.{1+1} + 1.1 + {1+1}.1 = 5.

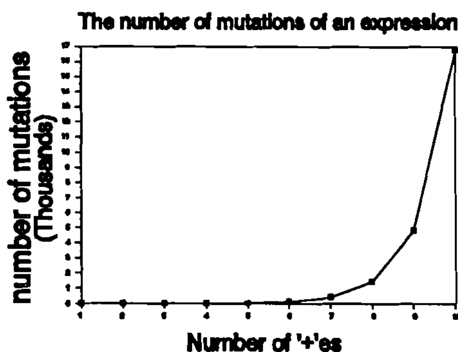

figure 6

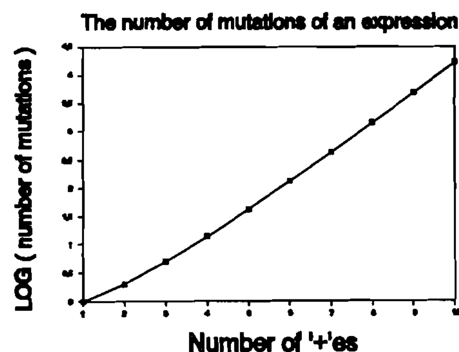

figure 7

In the two figures above we see that the number of mutations is exponentially related to the number of operators. The left one is used to indicate the rapid increase as we increase the number of operators. The right one is placed to show how fast the relation increases even on a logarithmic scale.

In the example stated earlier, this means that if we could reduce this expression, we had to have at least five patterns to match this expression and rewrite it to another. This could be more if we consider the variables incompatible and so in each tree we could set a variable at four different places, giving a total number of mutations: 5 * 4! = 5! = 120. This increases even more if the total number of operators increases, so for 4 operators we already have already 14 * 5! = 1680 possibilities, for 5: 42 * 6! = 30240.

To reduce the number of patterns we have to match, we declare the left sorted pattern as the canonical form. This is a specially defined form in which the operators are placed in

the tree.

If we first rewrite all expressions to this form we reduce the patterns in the rulebase from that point on.

Of course the number of mutations due to the variables are still present. This number is related to the number of operators (n): the number of mutations due to variables is (n+1)!, if all variables are disjunct. This is of a smaller order then M(n).

6.3.1. Left sorted canonical form.

We now define the left sorted canonical form and refer to it as the canonical form in the future. The following definitions only apply to dyadic operators.

1 The operatortype of a lefthand side of a dyadic operator is not the same as the operatortype of the dyadic operator itself.

2 If there are any constants in a expression (at the current level !) they have to be as far to the right as possible. This simplifies the search in a tree; we can detect a constant and skip it, remaining a constant free expression at this current level !

In the example at the beginning of the chapter, number 3 is the canonical form (((a + b) + c) + d). If for example b was a constant, the canonical form would be: (((a + c) + d) + b) [1].

For an optimal effect of the main equation (collapse) we first have to expand an expression to a form that also contributes to the canonical form.

6.4. Expansion.

Expand means using the distributive property to make an integer expression a 'sum' of 'products', and a register expression a 'or' tree of 'and' trees. (we do not regard *XOR* now because of its special characteristics and we already eliminated this operator.)

---

[1] there are still 3! = 6 mutations possible due to the variables a,c and d.

$$(a+b)*c = (a*c)+(b*c). \quad (1)$$

$$(a \vee b) \wedge c = (a \vee c) \wedge (b \vee c). \quad (2)$$

$$(a \wedge b) \vee c = (a \wedge c) \vee (b \wedge c). \quad (3)$$

We use these properties now from the lefthand side to the righthand side.
We immediately notice that this property used from left to right increases the number of operators by one each time. This is against the first main aim of this system.
We can defend this step by assuming that in the main collapse part the optimal solution is found. If the original expression is the optimal expression we would find this expression again as result of our system.

6.5. Introduction of constants.

As we saw the collapse rulebase expects that there is at least one constant at every node. This calls for a rulebase addconst, which takes care of this problem. It is simply a rulebase which checks if there is a constant at the most right place of a subexpression and if not, it introduces a *multiplication* with 1 for an integerexpression and a *and* with 1 for a registerexpression. Note that this has the opposite effect of sort, as it introduces a constant, while sort tries to get rid of it.

6.6. Removing monadic operators.

Here we encounter the first difference between an integer expression and a register expression.
In integer-expressions we only have to deal with monadic operators from the type minus, while with a register expression we come across more types of monadic operators such as *not* and *reverse* type.

6.6.1. Register expressions.

In register expressions we could encounter monadic operators, which are to be removed or at least be pushed as far as possible down the tree towards the operands.
Here we assume that the register operator reverse type is already as close to the operand as possible, We can easily introduce some rules, equivalent to the *not* operator, to push them down as well. For the *not* operator type we use the "De Morgan" property, and the distributive property.

$\neg$(a and b)	= $\neg$a or $\neg$b	(1)
$\neg$(a or b)	= $\neg$a and $\neg$b	(2)

We have to start at the root of an expression and work our way to the leaves, pushing the *not* operator out in front of us. This because the result of the current level is of great importance in the levels below.
Example:

$\neg$((a or b) or c)	=(using 1)
$\neg$(a or b) and $\neg$c	=(d)(using 1)
($\neg$a and $\neg$b) and $\neg$c.	

(d) indicates we are going down one level, and we find the *not* operator again, so we use the rules again at this level.
We introduce a simple rule that takes care of multiple *not*s after each other using N_1.

$\neg\neg$a	= a	(3)

6.6.2. Integer expressions.

Monadic minus operators and also dyadic minus operators can be eliminated using the property:

- a	= a * -1.	(1)
a - b	= a + b * -1	(2)

Proof:

(1) is an obvious result of equation I_2.

(2) a - b = (I_1)

a + - b = (I_2)

a + b * -1.

It doesn't matter at which point we start (root or leaves) as long as we check each part of the tree, because the result of this step is not used in a lower or higher level as is not so with the register expression.

If we start at the root, and we see a monadic operator, type minus, we use equation (1) to transform it. We then get an expression (called a in (1)), which has to be processed, and a constant expression -1, which is not processed. We see that even if we encounter:

--a, we only see -(b), with b=-a, so:

b * -1 (= -a * -1) and then one level down:

(a * -1) * -1.

This multiplication of -1*-1 will be dealt with later, in the sort part, so we leave it for now, but can mention that it is simply reduced to 1.

7. Possibilities for implementation .

We will now, as part of the implementation, first handle the problem of ordering a rulebase, which is a significant problem for rulebase design and implementation. We will then proceed investigating the order of the rulebases themselves.
This gives a method we can use to implement rules in a rulebase. We will do so with the rules we already found in the theoretical part, in the order in which they appear in the process of expression reduction.

7.1. How to order the rulebase.

We wonder if there is a mechanism for ordering a rulebase or at least describe a method of ordering a rulebase.
As we recall, the rulebase is sequential scanned until we have a match of the lefthand side of a rule and transform it to the righthand side of the same rule. After that we continue to scan the rulebase from the next rule on (sometimes we don't, so skip the rest of the rulebase, but that is an exception).

7.1.1. Definitions.

A rulebase has the following specialties in it:
- Mostly, the lefthand sides (s_i) of all the rules are disjunct to each other.
- If we have two rules r_i and r_j then if i<j this indicates that rule r_i will be placed before r_j in the rulebase.

This results in:
Rulebase ORDERED {

s_i	$\rightarrow t_i$	(r_i)
s_j	$\rightarrow t_j$	(r_j) }

We notice:
- If an expression matches s_i the expression is rewritten to a form of t_i.

- If an expression matches s_j, this could be the result of the expression itself which had the proper form as it entered the rulebase or rule i has transformed the expression so that it now can match rule j.

The connection between these two rules is obvious. If r_i produces a form that will match s_j or a subexpression of s_j this rule must be placed first.

For example:

Rulebase EXAMPLE1 {

--a	→ a	(1)
a + (b + c)	→ (a + b) + c	(2) }

Rules (1) and (2) are disjunct. The question is why do we have to place (1) before (2). First we see that if expressions a,b or c in (2) are of the form (--a) we still match rule (2) so this is NOT the reason (1) is placed before (2).

(1) Is placed before (2) for the part (b + c) that could be of the form: --(b + c) which will be handled by (1) so it is matched by (2), so an expression like 'a + --(b + c)' will be handled correctly. Note that rule (1) is used in a different (lower) level then (2). We thus have to use this rulebase bottom-up, working an expression form the leaves to the root, or in expressionterms form the inner parenthesis to the main expression.

As an example we want to investigate a simple rulebase used to push down the *NOT* operator to the operands as much as possible. The rules use in this rulebase are: (C stands for a constant)

¬¬a	→ a	(1)
¬(a + b)	→ ¬a * ¬b	(2)
¬(a * b)	→ ¬a + ¬b	(3)
a * 111..	→ a	(4)
a + 111..	→ 111..	(5)
a * 000..	→ 000..	(6)
a + 000..	→ a	(7)
C * a	→ a * C	(8)
C + a	→ a + C	(9)
C + C	→ C	(10) (C on RHS is evaluated C)
C * C	→ C	(11) (C on RHS is evaluated C)
¬C	→ C	(12) (C on RHS is evaluated C)

Note: the list order is arbitrary.

We can divide a pattern in subpatterns like:

$$s_i = s_{i1} * s_{i2} \quad \text{or}$$
$$s_{i1} + s_{i2} \quad \text{or}$$
$$\neg s_{i1}$$

E.g.: $s_1 = \neg\neg a$; $s_{11} = \neg a$; $s_{111} = a$. (note: $s_{111} = t_1$) so
$s_1 = \neg s_{11} = \neg\neg s_{111}$.

$s_2 = \neg(a + b)$; $s_{21} = a + b$; $s_{211} = a$, $s_{212} = b$ so
$s_2 = \neg s_{21} = \neg(s_{211} + s_{212})$.

We introduce an other definition here: $s_i = s_{i00..}$ supplementing the number of zeroes to the number of levels (in the expression/pattern) minus 1. So:
$s_{100} = \neg s_{110} = \neg\neg s_{111}$. In this way it's easier to indicate the level we are operating. If the index is smaller than another we are in a higher level than the other. So the smaller the index, the higher the level will be.

7.1.2. The problems.

We have to distinguish two problems here.

1. In which order do we place the rules in the rulebase ?
2. Which level is dealt with first ?

The second problem is a question whether to use a bottom-up or a top-down method. Bottom-up: we start at the leaves, work our way to the root, so the lowest level is dealt with first. In concreto this means, subpatterns with a higher index are dealt with first. Top-down is the other way around. Here the subpatterns with the lowest index are dealt with first. The only problem now is which method do we use.

The first problem is more complicated. We have to recognize if a consequence of one rule on another is in this level, in a level below, or in a level above (a higher level). A problem is that some rules only strip things off, but do not alter the expression basicly, e.g., rule (1) only strips off the two *not* operators, but leaves the rest of the expression intact. We could consider leaving this rule, as we come to the 'a' part two levels down anyway (as indicated by $s_{111} = t_{100}$). The problem lies in the consequences for this step in rule (2). If s_{111} is of the form s_{210} then one level down we get a match on rule (2) so an expression like ¬¬(a+b) will be transformed to ¬(¬a * ¬b), if rule (1) used it will become (a+b) directly, which we can not guaranty if we use an other order in these rules. So we have to consider the possibility that if this rule is not used (first), what rules will be used, even at another level.

7.1.3. Dependencies.

We see the problem is very complex. We will try to make a matrix, of the effects of some rules over others.

We have some possibilities.

-1 Effect of rule is in the current level, on other rule.	x
-2 Effect of rule is on a Lower level, on other rule.	L
-3 Effect of rule is on a Higher level, on other rule.	H
-4 Possible use of another rule on another level	p[x I L I H]
-5 Warning: rule could result in an Endless loop !	E
-6 Hazard: master rule could match something directly as does slave	Z
-7 No effect on other rule.	

We could express these possibilities in mathematical terms, using the $\rightarrow_{match}$ relation. This special relation tries to match a pattern on another pattern. We have defined a method to match expressions on patterns but we now see these patterns as an expression. To indicate the difference we use the $\rightarrow_{match}$ operator[1].

[1] note that ¬¬a $\rightarrow_{match}$ a but a $\rightarrow_{not\ match}$ ¬¬a !

x : $s_j \rightarrow_{match}$ t_i (will only be of influence for top-down.).
but not $s_j \rightarrow_{match} s_i$.

pL : $s_j \rightarrow_{match}$ $s_{i1} \mid s_{i2} \mid s_{i3}$ .. and no 'x'.
$s_j \rightarrow_{match}$ s_k ($i < k < i + 100..$) and no 'x'.

pH : $s_{j1} \mid s_{j2} \mid s_{j3}$ .. $\rightarrow_{match}$ t_i and no 'x'.
$s_k \rightarrow_{match}$ t_i ($j < k < j + 100..$) and no 'x'.

L : $s_j \rightarrow_{match}$ $t_{i1} \mid t_{i2} \mid t_{i3}$ ...
$s_j \rightarrow_{match}$ t_k ($i < k < i + 100..$).

H : $t_j \rightarrow_{match}$ $s_{j1} \mid s_{j2} \mid s_{j3}$ ...
$t_j \rightarrow_{match}$ s_k ($j < k < j + 100..$).

E : $s_i \rightarrow_{could\ match}$ t_j and s_j $\rightarrow_{could\ match}$ t_i.

Z : $s_i \rightarrow_{match}$ s_j and s_j $\rightarrow_{no\ match}$ s_i.
We could encounter combinations of these possibilities.

If we order the rules so that a dependent rule is used after the master rule, we get an ordered rulebase. In the matrix the xs have to be under the diagonal, e.g., if rule (3) is master over (4) (4 dependant of 3) we mark the place in the matrix (4,3) with a x (under (3,3)). The situation could occur that an x is on the diagonal, which indicates that a certain rule is dependant on itself. The rule could then be constructed to call upon itself until no match occurs any more, e.g., rule (1) is such a case: $s_{111} = t_1$, which indicates that an expression like '¬¬¬¬a' will be handled in two steps of rule (1) resulting in 'a'. The problem here is that if we use this rule bottom-up, we encounter '¬¬a' first (part of '¬¬¬¬a'), rewrite it to 'a' and later, two levels up, we encounter '¬¬a' again rewriting it to 'a'. We see here that the strategy (bottom-up / top-down) has influence on the rulebase.

If we have a **L** in the matrix this indicates that this master rule could result in a match on a lower level, on that particular dependent rule, so the rulebase for the master rule should

be used in the top-down method, to give the opportunity to match the dependent rule, e.g., rule (2) rewrites an expression like ¬(¬a+b) to (¬¬a*¬b). Now we can use rule (1) one level down on the ¬¬a part to rewrite it to a*¬b. Thus rule (2) used before (1) makes it possible to use rule (1) in a lower level, so indicated by an L at (1,2).

If an **H** occurs in the matrix this is a result of a master rule making it possible to use the dependent rule on a higher level, so this master rule should be used in the bottom-up method, so the master rule will be used before the dependent rule in a lower level, e.g., rule (5) produces a 111... string which in its turn can be used in rules (4) and (5) again, which is one level higher again. We indicate this with an H in the matrix.

All the entries with a '**p[L | H]**' have to be in the same method. So if at (2,1) there is a 'pL' in the matrix, this indicates that 1 must be performed in the same top-down sweep as 2, because if 1 is not used before 2 at this level, so in the same sweep, 2 could be used in a level down, before 1 at this level in the bottom-up sweep, e.g., an expression like ¬¬(a+b), we first use rule (2) (after on level down step) so we get ¬(¬a*¬b). Note we didn't encounter rule (1) because we get that one on the way back (bottom-up). To bring this one to a good end we have to use rule (3) again one level up, getting ¬¬a+¬¬b and now we can use rule (1), again one level down and resulting in a+b. We thus have a very weird run, up and down level, several times to bring this one to a good end.

The **E** is a special case only to indicate that a rule is dangerous if we use it to call a rewrite, e.g., rule (8) and (9) are never ending if they call themselves on an expression like 110+010 or 101*011.

A hazardous situation occurs if two source patterns are almost alike, e.g., 'a + const' and 'a + 1'. The target pattern doesn't even care in those cases. Here 'a + 1'(=i) will match on 'a + const' (=j). If we thus place rule j before i we get a situation that an expression like in rule i will not be handled, because rule j does all the rewriting, and so, assuming that the rule completely alters the expression, puts rule i out of business.

To get a good understanding of these indications some of them are represented as graphs for the dependencies of the first 3 rules, as they are the main rules of the rulebase-example:

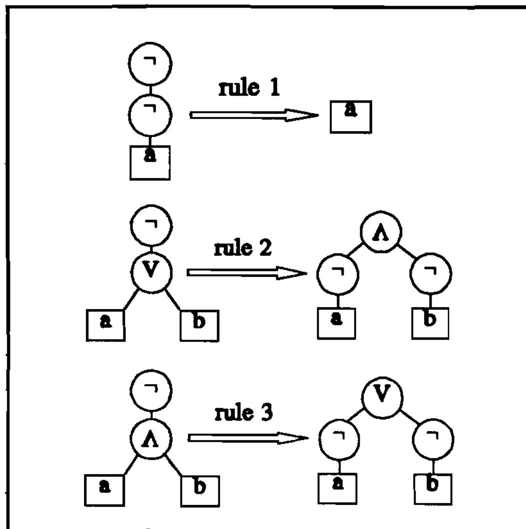

figure 8 : Original rules

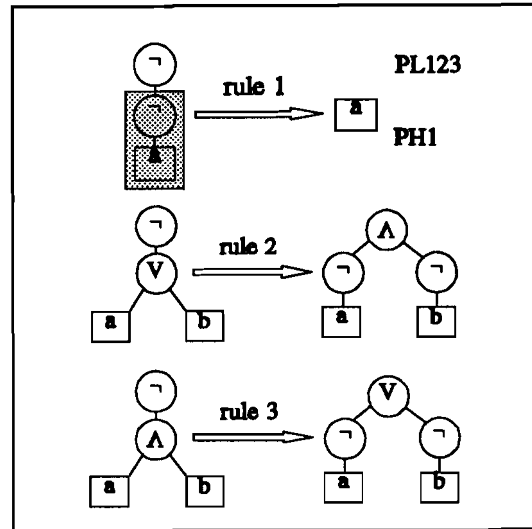

figure 9 : Possible dependencies

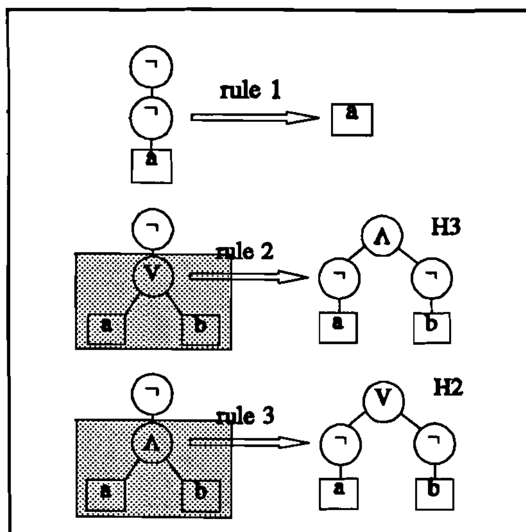

figure 10 : Higher dependencies

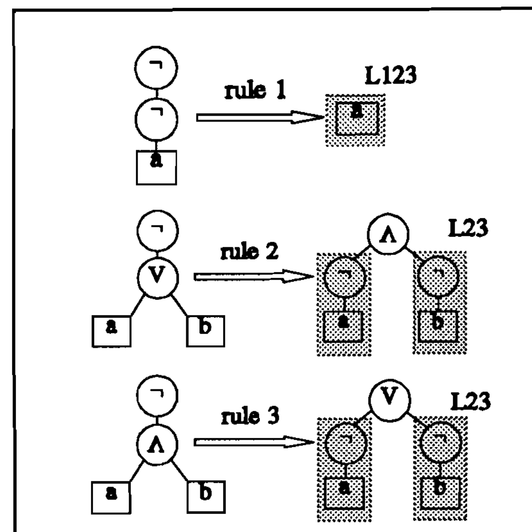

figure 11 : Lower dependencies

As an example to explain the graphs we look at figure 10 (higher dependencies). We see with rule 3 that the s_{310} part is marked. The text at the right corner of the rules states H2. This means that the t_{200} part matches the s_{310}, so there is a higher dependency of rule 2 in 3. This is then indicated by an H on the (3,2) spot.

For another example we look at figure 11 (lower dependencies). Here, in rule 2 the t_{211} part is marked. The indication is L3. This means that s_{300} will match t_{211} and so cause a L on the (3,2) spot also indicating a lower dependency.

Note that the definitions of the indications are limited. We do not look at the s_{221} part ('a') as this indicates a 'an expression' type that will match anything. The only exception is the case if the 'a' is the only part of a pattern. Therefore rule 1 is a special case and the t_1 is used in indication. We didn't make a figure of this case as it is obvious that 'a' (t_1) will match anything.

We will make a indication table of these first tree rules.

j↓,i→	1	2	3
1	x L	L	L
2	x L	L	L H
3	x L	L H	L

We first notice that in all the cells, there is an L. This suggest strongly that we use a top-down method for the 3 rules. Further are they at least downwards dependant on each other which indicates that it is a good idea to put them together in one rulebase, top-down method.

As we can see in the rule 1 column, this rule is dependant on all the others, including itself, and has therefore be implemented as a recursive rule. It calls upon itself provided that it makes a match. After that the expression no longer has the form of s_1 and so we can continue. We can now erase the xs form the table.

The possible rules are done before the lower level, so they can be erased also.

j↓,i→	1	2	3
1	L	L	L
2	L	L	LH
3	L	LH	L

Rule 2 could give erase to a hit on a higher level for rule 3, as rule 3 also does on rule 2, but this could only be possible if the original expression looks like ¬¬(a v b). This ¬¬ part

however is handled by rule 1, so it never occurs. We can not get this in the table. We, humans, have to decide this. We can see that this gives reason to the assumption that this procedure is not to be automated yet.

We will try to reconstruct an expression which will cause all this trouble. H(2,3) is a result of a rewrite of s_3. s_3 is rewritten to t_3 which will match s_{21}. We have thus $s_3 \rightarrow [t_3 \rightarrow_{match} s_{21}]$, and $s_2 = \neg s_{21}$, and so $s_2 \approx \neg t_3$ and concluding: $s_2 \leftarrow\approx \neg s_3 = \neg(\neg(a * b))$ (which will match s_1, so it doesn't occur).

We will refer to this procedure as backtracking, as it is used after we have taken a decision about the method (top-down / bottom-up).

We can now erase the Hs from the table as they are not applicable any more. We get a clean rulebase, only Ls in the table, so it is OK to use this rulebase, top-down, all rules together in one rulebase. Sometimes it could be possible that some of the Hs stay in the table. To avoid applying the rulebase over and over again, we can suggest two possible solutions:

1 Use some rules on the way backup (bottom-up method).

2 Introduce a rule which takes care of this particular situation.

This second solution could, however, alter the entire rulebase, which is not always desired. In every case we have to repeat the whole procedure of making a table, to construct the final rulebase.

We will now look at the backtracking part again.

For this purpose we will introduce a new definition here. We can for every rule define an area in the expression space E in which all the expressions are situated which may be offered to the rule. This part of the expression space we call the input expression space IES. After a rule there could be areas which are no longer present after the rule. We call this the output expression space OES.

We saw that some rules make the input space smaller, that is some expressions do no longer exist at the current level after we have applied a certain rule. This shows as we look at rule 1 of the example. The IES before this rule can be denoted as E the whole expression space. After rule 1 (recursively !) we have thus an OES of $E/\{\neg\neg a\}$, that is the original expression space without an expression that will match ¬¬a. We have to note that this is at the current level !

We thus see that after rule 1 we have an IES for rule 2 which is the output space of rule 1: $E/\{\neg\neg a\}$.

After 2 we thus have $E/\{\neg\neg a,\neg(a+b)\}$, after 3 $E/\{\neg\neg a,\neg(a+b),\neg(a*b)\}$.
Using backtracking we can find an input expression that causes the H in the dependency table namely ¬¬(a*b) which, as is now obvious, not in the input expression of rule 2 and 3. We can therefore wipe out the Hs in the table.
Note that the above is only permitted if we already have determined the order of the rules. We use backtracking to confirm our choice of top-down. If for some reason we don't get to a conformation we thus have to reconsider our choice we made, or introduce a rule that will take care of this situation. As a result of introducing a new rule we have to start the whole process over again.

We are now using all the rules from the example and try to construct a rulebase. We have to note that all the above steps represent an enormous amount of time, but we have filled the table any way.
We shuffle the rule order to get a nonprejudiced matrix. We then are going to sort the matrix. Sorting is done as follows:
We start with a top-down design. We can now forget the Hs in the matrix as we do not have the possibility to use these dependencies in any way.
We first make a dependency vector diagram. Here the arrows indicate a dependency. A pL indicates we have to use the source rule of the arrow before the target rule, because else the target rule will hit first, not giving the source rule a chance to do its job as we want it to.
Rules which have their whole column filled with xs we mark with a square, because in top-down, we have to call them recursively. We can then forget the whole column of xs and so simplify the vector diagram. We just can start with any rule, and work our way trough. After we have made the draft diagram, we rearrange it so we can draw some conclusions from it.

We can then rearrange the matrix according to the diagram.

j↓,i→	9	12	4	8	7	5	3	6	2	11	1	10
9	N	HpH	x		x	H		H	pL	HpH	x	HpH
12	H	HpH	x		x	H	L	H	L	HpH	x	HpH
4		pH	x	x	x	HpH	pL			pH	x	pH
8		HpH	x	N	x	H	pL	H		HpH	x	HpH
7	x	pH	x		x			HpH	pL	pH	x	pH
5	x	pH	x		x	HpH			pL	pH	x	pH
3			x	H	x		L		HL		x	
6		pH	x	x	x		pL	HpH		pH	x	pH
2	H		x		x		HL		L		x	
11		HpH	x		x	H	pL	H		HpH	x	HpH
1			x		x		L		L		x	
10		HpH	x		x	H		H	pL	HpH	x	HpH

First we have to determine which rules are used top-down, and which are used bottom-up. If a rule has a L on its diagonal, this indicates that the rule has some dependency of itself on a lower level. It is therefore logical to use this rule in a top-down method. On the other hand, if a rule has an H on its diagonal, this means it is best used in a bottom-up method.

A rule with a x on its diagonal, and mostly along the whole column, indicates a dependency on the current level and therefore the rule has to be used recursively anyway. We therefore clear this column of its xs, as we reconstruct the rule with such a recursive rewrite command, instantaneously.

We now have tree groups of rules, *A* for top-down, *B* with bottom-up, and C for which we don't know (yet).

In the example we get:

A: 2,3 ; *B*: 5,6,10,11,12 ; *C*: 1,4,7,8,9

p indicates rule p being recursive. The reason for this special case is because this rule

strips off two levels of an expression, and by doing so it skips the part of the matching process before it. It in a way steps into a 'new' level, e.g., (rule 1) ¬¬¬¬a is match as ¬¬(¬¬a) and rewritten to ¬¬a. but if no rewrite occurs this 'new' expression ¬¬a is not matched by all predecessors and so a part could be missed as we see in this example. If we use the bottom-up method we match from the inner part to the outer. We could find thus ¬¬a first not even knowing that the expression still holds two *not* operators above it. Now the expression is processed properly. As we have already placed this rule in the top-down part, we only have the solution of a rewrite command. This could be fairly local as this rule (1) is the only rule to be called again.
As we can see in the table, rules 2 and 3 are dependent on rule 1, indicated by the L on the row of rule 1, on a lower level. Further, the pL on the column of rule 1 indicates that rule 1 must be performed before rules 2 and 3 on a higher level. Concluding that rule 1 belongs to group *A*.
We have to notice the special rule 12, which is lower and higher dependent on some other rules. We therefore place this rule twice. One in group *A* and one in group *B*.
From the xs in the columns of rule 8 and 9, we see that rules 4,5,6, and 7 are dependant so, rules 8 and 9 must be before them. No further restrictions are made to these two rules. We can put them either in group *A* or (at the beginning of) group *B*. Placing them in group *B* automatic places 4 and 7 also in *B* as they must be after 8 and 9. This has the advantage that rule 4 and 7 don't have to be recursively rewritten because the 'a' part of the rule was already processed (bottom-up method). This in turn could speed up the rewrite process, as the number of recursive rewrites diminishes.
We get now thus:
A: 1,2,3,12. ; *B*: 4,5,6,7,8,9,10,11,12. ; *C*: empty.

As we remember we can split a rulebase in a top-down and a bottom-up part, using the recursive rewrite command. We now have thus a *A* group (top-down) and a *B* group (bottom-up). We can so reorder the table correspondingly.

	1	2	3	12	8	9	4	5	6	7	10	11	12
1	x	L	L				x			x			
2	x	L	HL			H	x			x			
3	x	HL	L		H		x			x			
12	x	L	L	HpH		H	x	H	H	x	HpH	HpH	HpH
8	x		pL	HpH	N		x	H	H	x	HpH	HpH	HpH
9	x	pL		HpH		N	x	H	H	x	HpH	HpH	HpH
4	x		pL	pH	x		x	HpH		x	pH	pH	pH
5	x	pL		pH		x	x	HpH		x	pH	pH	pH
6	x		pL	pH	x		x		HpH	x	pH	pH	pH
7	x	pL		pH		x	x		HpH	x	pH	pH	pH
10	x	pL		HpH			x	H	H	x	HpH	HpH	HpH
11	x		pL	HpH			x	H	H	x	HpH	HpH	HpH
12	x	L	L	HpH		H	x	H	H	x	HpH	HpH	HpH

We now have to sort the groups *A* and *B* so they behave as we expect of the rulebase.
Group *A*: Rule 1 is recursive rewritten so we can forget about the xs in that column. Also the xs in column 4 and 7 are not important for this group now. The Hs in the row of rule 12 are taken care of by using rule 12 twice.
The rules have no dependencies and so the rules may be mixed in order providing they stay in group *A*.
Group *B*: Here we can also disregard the xs on the columns 1,4 and 7 as they are no longer effective (in a bottom-up method the dependencies are no longer important as mentioned before). The rules 8 and 9 must be before 4,5,6 and 7 as we have seen before. This is also confirmed by the fact that the xs are under the diagonal. The only problem now is that the rules 10 and 11 have possibilities on rules 8,9,4,5, and 7. We therefore place these two rules before 8,9,4,5,6 and 7. This results in the final table.

	1	2	3	12	10	11	8	9	4	5	6	7	12
1	x	L	L						x			x	
2	x	L	HL					H	x			x	
3	x	HL	L				H		x			x	
12	x	L	L	HpH	HpH	HpH		H	x	H	H	x	HpH
10	x	pL		HpH	HpH	HpH			x	H	H	x	HpH
11	x		pL	HpH	HpH	HpH			x	H	H	x	HpH
8	x		pL	HpH	HpH	HpH	N		x	H	H	x	HpH
9	x	pL		HpH	HpH	HpH		N	x	H	H	x	HpH
4	x		pL	pH	pH	pH	x		x	HpH		x	pH
5	x	pL		pH	pH	pH		x	x	HpH		x	pH
6	x		pL	pH	pH	pH	x		x		HpH	x	pH
7	x	pL		pH	pH	pH		x	x		HpH	x	pH
12	x	L	L	HpH	HpH	HpH		H	x	H	H	x	HpH

We note also that all ps are also below the diagonal as are all the xs and thus the problem is solved. Further we have all the Ls in the top-down part (1-12) and all the Hs in the bottom-up part (12-7), so this criterion is satisfied as well.

As a result we thus get the following rulebase:

rulebase EXAMPLE1:

```
{
¬¬a              → REW a          (1)
¬( a + b )       → ¬a * ¬b        (2)
¬( a * b )       → ¬a + ¬b        (3)
¬C               → C              (12) (C on RHS is evaluated C)
RECURSIVE_REWRITE
C + C            → C              (10) (C on RHS is evaluated C)
C * C            → C              (11) (C on RHS is evaluated C)
C * a            → a * C          (8)
C + a            → a + C          (9)
a * 111..        → a              (4)
a + 111..        → 111..          (5)
a * 000..        → 000..          (6)
a + 000..        → a              (7)
¬C               → C              (12) (C on RHS is evaluated C)
}
```

7.2. Global Rulebase ordering.

If we want to order the rulebases, we have to use more or less the same rules as ordering the rulebase internally. Every rulebase can be described as a simple description, with an input expression space (IES) and an output expression space (OES).
For every rulebase there is an input condition, which could be described as an input expression space. We thus have to meet this input condition of the input expression space of the rulebase.
As we look for example at the push-*not*-rulebase, we see it lacks input conditions, or at least non that can be overcome easily, $e \varepsilon E \mid e =$ registerexpression. Therefore we could place this rulebase anywhere in the master rulebase, and call it on the only condition that the expression is of the registerexpression type. The output expression space however is in the area where there is no *not*-operator before another operator. The only thing after an *not*-operator is an operand or a constant. The output expression space also shows this: $E/\{$¬¬a,¬(a*b),¬(a+b),integerexpression$\}$.

For every rulebase we could set up a set of expression spaces, one for input, one for output.
Masterrulebase:

IES: E.
OES: E.

We see that for the masterrulebase there are no limitations.

PushNotRulebase:

IES: $E/\{$integerexpression.$\}$.
IES: $E/\{$¬¬a,¬(a*b),¬(a+b),integerexpression.$\}$.

As we already knew, this rulebase only limits the expression space more as preparation for other rulebases.

PushMinusrulebase:

IES: $E/\{$registerexpression.$\}$.
IES: $E/\{$-a,a-b,registerexpression.$\}$.

Also here we have a rulebase that only limits the expression space for further rulebases.
Note : $IES_{PushNotRulebase} \cup IES_{PushMinusrulebase} = IES_{Masterrulebase} = E$.

ExpandRulebase:

IES: $E/\{\neg\neg a, \neg(a*b), \neg(a+b), -a, a-b\}$.

OES: IES/{(a+b)*c,a*(b+c),(a or b)and c, a and(b or c)}

SortRulebase:

IES: $E/\{\neg\neg a, \neg(a*b), \neg(a+b), -a, a-b, (a+b)*c, a*(b+c)$,(a or b)and c, a and(b or c)}

OES: IES / {a op(b op c),C op a,(a op C)op b, C op C }

{note : op is still a symmetrical operator}

As we can easily see we are still narrowing the expression space. We work to a canonical form.

We now come to a rulebase Subsort which is a rulebase that can not be described with these parameters. This is due to the use of a algorithm that can not be described as a rulebase we defined.

We will however try to characterise it with an OES, but this can not cover all the possibilities. We will indicate this.

$IES_{Subsort} = OES_{Sort}$.

$OES_{Subsort} = OES_{Sort}$ / { a'+b+a' } (' marks the busy variable).

note that description is this is not complete.

We indicate that if there were two busy variables in an expression and they could be placed in such a way that they could be adjacent, they would have been. An expression like a'+b+a' would therefore not be in the $OES_{Subsort}$. Also an expression like a'+b+c+a' would not be in it. We could think of thousands of other expressions which would not be in the $OES_{Subsort}$. It is thus not possible to describe them in a way as we did before.

It is therefore also not possible to describe the whole rulebase in terms of expression spaces. The rulebase selects parts of an expression which are to be treated with an specially designed algorithm that will be discussed later.

As the $OES_{Subsort}$ could not be described, the $IES_{simplify}$ and $IES_{boolalg}$ could also not be described as they follow upon the subsort rulebase.

The OES of both these rulebases could be described in words: " If the rulebases use all

their possibilities, there will only be one reference of a variable in the expression[1]". There are of course expression for which this is not true, e.g., (a*b)+(b*c)+(c*a) will only come to: a*(b+c)+(b*c). That is why we used: 'all their possibilities'.

As we can see it is impossible to describe the IES and OES clearly. We will therefore only indicate them as well in words.

It is easy to see that:

$IES_{simplify} = IES_{boolalg} = OES_{Subsort}$

This is only partly true as simplify only deals with integer expressions and boolalg only deals with register expressions.

$OES_{simplify} = OES_{boolalg}$ with the same restrictions as with the IES.

The OES holds only expressions with the least number of operators which is possible to achieve with the used method.

7.3. Constructing a rulebase for minus reduction.

We have to construct a rulebase for minus operator reduction (to zero !).

The used properties are:

- a	= a * -1
a - b	= a + b * -1

So the rules we want to use first are:

- a	→ a * -1	(1).
a - b	→ a + b * -1	(2).

Resulting in a table like:

j↓,i→	1	2
1	-	-
2	-	-

[1] note that this is only true for expressions for which this is possible anyway.

From the table we conclude that there are no dependencies here so we simply construct the table as we like, e.g., top-down:

- a	→ a * -1	(orient)	(1).
a - b	→ a + b * -1	(orient)	(2).
REWRITE,CURRENT.		(recurs the tree)	(3).

Here we have an example of not following the rules for deducting a rulebase defined by Knuth and Bendix'70 (see H.Aït-Kaci & M.Nivat p39 [19]). The orient inference rule (1) states s > t. In both cases of the above properties this is not true. Nevertheless we construct the rule, keeping in mind, that if this expression is still present at the end of the process, we have to reverse the process. So implode minus introduces the following rules:

a * -1	→ - a	(4).
a + b * -1	→ a - b	(5).
a * -1 + b	→ b - a	(6).

Resulting in the following table.

j↓,i→	4	5	6
4	-	pL	pL
5	-	-	-
6	-	-	-

The only thing we have to be sure of is that rule 4 is used at a certain level before 4 and 6 on a lower level. The simplest solution is to use them top-down:

a * -1	→ - a	(orient)	(4).
a + b * -1	→ a - b	(orient)	(5).
a * -1 + b	→ b - a	(orient)	(6).
REWRITE,CURRENT.		(recurs the tree)	(7).

Note we have to use more rules here. Rule (6) is for a special case: if the -1 part is contained in the most left part of an expression.

Rule (6) is a result of rule (5) and the commutative property. We could deduce this rule also from a simplify step using Knuth and Bendix'70 [19].

We could change the order of the rules here a bit. For instance we could place rule (6) after (7) but in the form of: -a + b → b - a. This because we then have to consider the fact that rule (4) was already applied so all multiplications with minus 1 are then converted to a monadic minus operator with the rest of the expression under it.

A few examples. (later steps are already shown).

--a → (a*-1)*-1 → a*(-1*-1) → a*1 → a.

-a*3 → (a*-1)*3 → a*(-1*3) → a*-3 (note -3 is a constant as it is).

7.4. Constructing a rulebase for not reduction/shifting.

Using some register properties we can construct some properties which result in shifting a *not* operator.

$$
\begin{array}{lll}
(1): & \neg\neg a = a & (N) \\
(2): & \neg(a \wedge b) = \neg a \vee \neg b & (M) \\
(3): & \neg(a \vee b) = \neg a \wedge \neg b & (M) \\
(4): & \neg(a \oplus b) = & (X) \\
 & \neg((a \wedge \neg b) \vee (\neg a \wedge b)) = & (M) \\
 & (\neg(a \wedge \neg b)) \wedge (\neg(\neg a \wedge b)) = & (M) \\
 & (\neg a \vee \neg\neg b) \wedge (\neg\neg a \vee \neg b) = & (N) \\
 & (\neg a \vee b) \wedge (a \vee \neg b) = & (D) \\
 & ((\neg a \vee b) \wedge a) \vee ((\neg a \vee b) \wedge \neg b) = & (D) \\
 & (\neg a \wedge a) \vee (b \wedge a) \vee (\neg a \wedge \neg b) \vee (b \wedge \neg b) = & (\neg x \wedge x = 0) \\
 & 0 \vee (b \wedge a) \vee (\neg a \wedge \neg b) \vee 0 = & (Z) \\
 & 0 \vee (b \wedge a) \vee (\neg a \wedge \neg b) = & (C) \\
 & (b \wedge a) \vee (\neg a \wedge \neg b) \vee 0 = & (Z) \\
 & (b \wedge a) \vee (\neg a \wedge \neg b) = & (C) \\
 & (a \wedge b) \vee (\neg a \wedge \neg b) = & (N) \\
 & (\neg\ (\neg a) \wedge b) \vee ((\neg a) \wedge \neg b) = & (x) \\
 & \neg a \oplus b. &
\end{array}
$$

Resulting in the following rules:

$$
\begin{array}{ll}
\neg\neg a \rightarrow a & (1) \\
\neg(a \wedge b) \rightarrow \neg a \vee \neg b & (2) \\
\neg(a \vee b) \rightarrow \neg a \wedge \neg b & (3) \\
\neg(a \oplus b) \rightarrow \neg a \oplus b & (4)
\end{array}
$$

These are the basic rules for pushing a *not* operator to the operands. In this way we clear every level of an expression of *not* operators by moving them down one level. In the next

recursive step we find it back and try to push it again until we can't any more.
Only rules (1) and (4) are reducing the number of operators. The other two (2) and (3) must be reversed if no improvement has occurred.

$$\neg a \vee \neg b \rightarrow \neg(a \wedge b) \quad (2b)$$
$$(c \vee \neg a) \vee \neg b \rightarrow_{(C)} c \vee(\neg a \vee \neg b) \rightarrow c \vee \neg(a \wedge b) \quad (2c)$$
$$\neg a \wedge \neg b \rightarrow \neg(a \vee b) \quad (3b)$$
$$(c \wedge \neg a) \wedge \neg b \rightarrow_{(C)} c \wedge(\neg a \wedge \neg b) \rightarrow c \wedge \neg(a \vee b) \quad (3c)$$

The rules (2c) and (3c) are derived as shown. The only way these rules can be used is, if c is not a *not expression*.
These rules have to be applied from the leaves to the root, thus the reversed order of the push rules. We call this part 'implode_not'.

7.5. Constructing a rulebase for constant introduction.

The introduction of constants as described introduces some problems, as it is not depending on a presence of a constant but a '**not** presence of' a constant. Our rule thus has to be more suggestive as the other rules.

7.5.1. Integer expressions.

We want to introduce a constant to the most right side of all productterms. This implies a rule like:

a * 'not a const' → a * 'not a const' * 1. (1).

note 1: We may not use this recursively, as this would cause an introduction of a 1 each second invocation, which is not wanted.
note 2: We have to label the whole expression "a * 'not a const'" so we can refer to it if we have a hit. (we denote a label by 'L:' and a reference by 'L') We can thus denote rule 1 by:

L:(a*'not a const') → L * 1. (1).

From note 1 we can conclude that we can use this rule only by calling it from another rule:

a + L:'not a const' → a + call rule (1) L. (2).

We have to use a third rule to take care of the recursion, as rule (2) must be called also on the 'a' part. This however may not only be in case of a hit of rule (2) thus:

a + b → rewrite: (a) + b. (3).

note 3: We only rewrite the 'a' part, not the b part.
note 4: We could take rule 1 and 2 together to:

a + L:(a*'not a const') → a + L * 1 (4).

but we would still need rule (1) to deal with the last one of a or tree. We thus get only 3 rules: 1,3 and 4.

The advantage over using rules 1,2,and 4 is that we don't need another rulebase, while rule 2 calls rule 1,which is not the current rulebase. We would therefore need another rulebase for rule 1.
The order of these two rules is not important as rule 4 only adds something to an expression only once, while rule 3 takes care of getting from one level to another. The method top-down, or bottom-up is not important here.

7.5.2. Register expressions.

With register expression we have the same problem as with integer expressions,and thus only the rules are somewhat different.

a∨b	→	(rewrite: a)∨b.	(1).
a∨L: (b∧'notaconst')	→	a∨(L∧11..)	(2).
L: (b∧'notaconst')	→	L∧11..	(3).

We can easily see the analogy in these rules.

7.6. Constructing a rulebase for expand.

7.6.1. Integer expressions.

equations ε Eq:

a*(b+c) = a*b + a*c. (1)

(a+b)*c = a*c + b*c. (2)

Problem: which local strategy do we use ?

There are two solutions:

1. introduce a mechanism which keeps track of alterations and act upon it.
2. Start at the leaves, work to the root, and make sure the expression we leave behind is already expanded. (no match can be made in this part of the tree any more). Then if there is a match at this level that leads to an alteration, walk down tree again to the leaves until the alteration has no effect anymore.

An example.

The triangles represent parts of an expression tree. White areas are not yet visited. The procedure finishes if all parts are ready or not hit occurs. Ready means that the sub tree has the desired form, and not hit indicates that in that part there is no match, so the pattern we search for to be simplified is not found.

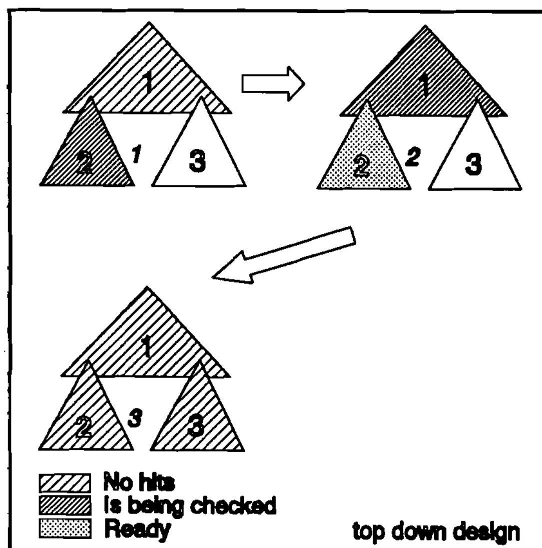

figure 12

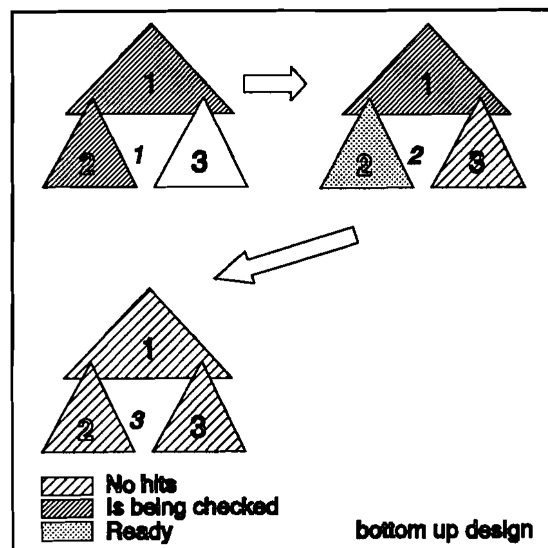

figure 13

The left (top-down) method tests part 1 (*1*), and finds nothing, goes on to part 2 and

finds a suitable pattern, which is altered. (2) Part 2 is altered and it is necessary to check part 1 again. Due to the alteration in part 2, there is now a match in part 1 (which has of course an overlapping match in part 2). Now the alteration is made to part 1 and we do the step to part 2 again. Check it, and find nothing to be simplified, continue to part 1 again (we don't check part 1 as part 2 altered nothing). Then check out part 3 and exit (3). The right (bottom-up) method walks down the tree to the leaves first. In the coming back part it starts checking. (1) As it hits something in part 2 it alters part 2[1]. Then part 3 is checked first (2). Then part 1, changing it and checking part 2 again, but no hits are made, so we exit (3).

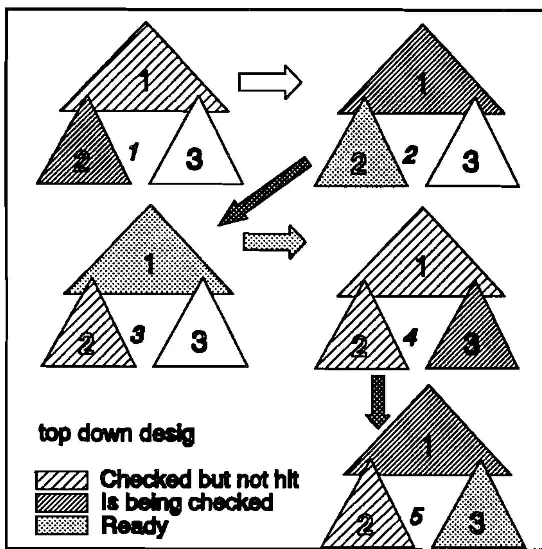

figure 14

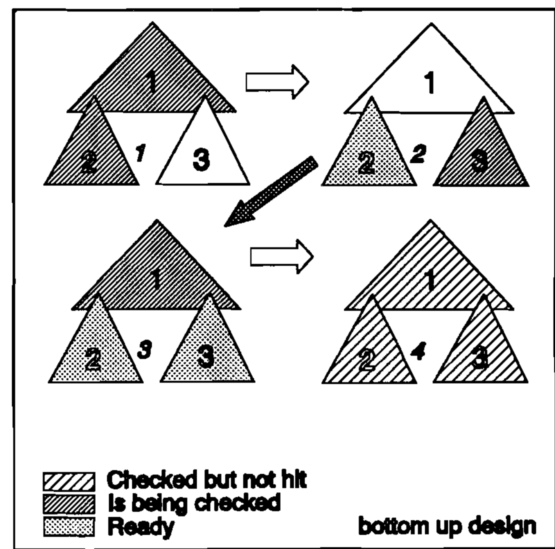

figure 15

We look at the same example again, but now in the case that part 3 also alters during the walk. The begin and end figures are the same although in the left figure, the final step isn't displayed because it's the same as on the right. We now see instantaneously that the first method is more complex than the second one. We see that the second method doesn't check part 1 first as part 1 is dependant on part 2 and 3. Therefore it is logical to use the second bottom-up method, because less checking is done. Further more there is no special mechanism needed to signal a alteration. We simply apply the rulebase again to the subexpressions if the root of a certain expression alters.

[1] part 1 is not checked now !

We could also try the Table method:

j↓,i→	2	3
2	L (2x) H	L (2x) H
3	L (2x) H	L (2x) H

We could use one of two methods (top-down or bottom-up) as we have Hs and Ls.
As we backtrack we will find that the input expression space is not limited enough to rule out some possibilities as we had with the shift-*not* rulebase.
Therefore we have to take all the Ls and Hs into account, e.g.
L (1,1): $a*((b+c)+d) \rightarrow_1 a*(b+c)+a*d \rightarrow_{L1} a*b+a*c+a*d$.
H (1,1): $a*b*(c+d) \rightarrow_{L1} a*(b*c + b*d) \rightarrow_1 a*b*c + a*b*d$.
Introduction of more rules would lead to an endless explosion of rules, as some rules are recursive, e.g., as we saw above with H(1,1) we could try to introduce this reduction as a new rule. But then what to do about: a*b*c*(d+e), which would introduce another rule. This could go on forever.
We could, however, try to solve this more elegantly. We use a bottom-up method solving the problems with the Hs in the table. We now only have to take care of the Ls. This could be handled by the rewrite command. If we have a hit on one of these 2 rules, and thus a rewrite, we could use this hit to rewrite the whole expression again, calling the current rulebase. In this way we go lower in the expression again until no more hit occurs and the lower dependency stops.
Using these considerations we can construct the following rulebase.

rewrite_recursively[1] (1)
a*(b+c) → rewrite:(a*b)+rewrite:(a*c)[2] (2)
(a+b)*c → rewrite:(a*c)+rewrite:(b*c)[3] (3)

As we look at these rules we note that in the rewrite after a hit, it is not necessary to walk

[1] This rule results in walking to the leaves first

[2] We only rewrite (and go down again (we were going up)) if we had an hit at the current level !.

[3] see above.

to the leaves again. Therefore we need two rulebases. One called the main, as described above, and one which only holds the rules (2) and (3), called the small rules. The rewrite in the main is thus calling the small, but small is only calling itself. In this way the small will finish when there is no more pattern match. This will speed up the process, because fewer matches are tried.

An example: ((b) is backup out of recursion)

(a+b)*c*d →	(1)
(a+b)*c , d →	(1)
a+b , c , d →	(1)
a , b , c , d →	(b)
a+b , c , d →	(b)
(a+b)*c , d →	(3)
a*c , b*c , d →	(b)
((a*c)+(b*c))*d →	(3)
(a*c)*d , (b*c)*d →	(b)
((a*b)*c)+((b*c)*d) →	(exit).

7.6.2. Register expressions.

Register expressions are handled the same. We use the same properties, and by using the same theory we can also construct a rulebase simply.
Only the + is swapped with 'or', and the * is swapped with the 'and'.

j↓,i→	2	3
2	L (2x) H	L (2x) H
3	L (2x) H	L (2x) H

rewrite recursive	(1)
a∧(b∨c)→ rewrite:(a∧b) ∨ rewrite:(a∧c)	(2)
(a∨b)∧c→ rewrite:(a∧c) ∨ rewrite:(b∧c)	(3)

for the rewrite command again a new small rulebase

7.7. Construction of a sort algorithm.

The aim is, as mentioned earlier, to use the properties:

D_1: (a op_2 b) op_1 c = (a op_1 c) op_2 (b op_1 c)
D_2: c op_1 (a op_2 b) = (a op_1 c) op_2 (b op_1 c)

These properties are formed to rules from righthand side to lefthand side.
Note that if we use the commutative property on the lefthand side of the second property we get the first property.
Concluding we say that property D_1 and D_2 are equivalent if op_1 is a symmetrical operator, and in our case it is. As this is a requirement for the property D_1 and D_2 to work we can combine these two properties to the rule:

(a op_1 c) op_2 (b op_1 c) $\rightarrow$ (a op_2 b) op_1 c (1)

We now have only one rule, but we have to note that the lefthand side of this rule only matches expressions with one main operator (op_2). The pattern must be extended using the associative property to:

{r op_2 (a op_1 c)} op_2 (b op_1 c) $\rightarrow$ {r op_2 (a op_2 b)} op_1 c (2)

We assume, as always, that op_1 and op_2 are symmetric operators.
As we see it is needed that for rule (1) and rule (2) there are some variables or expressions (represented by c in the rules) that are the same, and at the exact place. It's therefore needed to sort an expression again to ensure that the above is the case.
We now have to select the most common variable, which is represented in the most subexpressions. If we want to make such a decision, we first have to scan the expression and make a count of the present variables. We can also count in which subexpression the variable is present, and so minimising the search/scan time. If we have selected the most common, and so the most likely variable we can use the above rules.
We work from the root to the leaves, but first 'subsort' the subexpressions.

For these global rules we could already try to construct a table:

j↓,i→	1	2
1	LH	LH
2	LH	LH

The Hs are not desired here! We want to use these rules and assume that there is no higher level. Therefore we get a table with only Ls that indicates a top-down method.

7.7.1. Integer expressions.

We so find a tree consisting of a sum tree of producttrees. We thus first 'subsort' the producttrees using the sorting properties again. These state that a tree of the same symmetrical operators can be arranged in any order we like. The selected variable is placed at the most right place possible.

We now have all subexpressions of the form:

1. selected_var or
2. selected_var * const or
3. rest * selected_var or
4. rest * selected_var * const.

We see we have 4 appearances of this expression. As this pattern is present twice in the rules above we get a total number of rules:

2 (rules) * 4 (left-side) * 4 (right-side) = 32 rules.

We could diminish this by introducing a const multiplication of 1 (U_1). We reduce the appearances of the product to 2: possibility 1. and 3. are not possible any more. We so reduce the number of rules to: 2*2*2 = 8.

We see that we now get a rule like:

$$(a * c * C_1) + (b * c * C_2) \rightarrow \{(a * C_1) + (b * C_2)\} * c \qquad (1)$$

$$r + (a * c * C_1) + (b * c * C_2) \rightarrow r + \{(a * C_1) + (b * C_2)\} * c \qquad (2)$$

In these rules the 8 mutations are realised leaving out parts a and b respectively, or both.

We encounter an other problem. If we want to use this rule recursively, we leave the busy var (in the rule above represented by c) not in the right form. We want that every product has one constant at the left side, but with the c part we don't have this. Therefore we alter the right hand side of the rules that inserts an one.

$$(a * c * C_1) + (b * c * C_2) \rightarrow \{(a * C_1) + (b * C_2)\} * c * 1 \qquad (1)$$

$$r + (a * c * C_1) + (b * c * C_2) \rightarrow r + \{(a * C_1) + (b * C_2)\} * c * 1 \qquad (2)$$

Note that the gain we have now is zero: the number of operators is the same on the left- and right-hand-side. We have to admit that the last operator could be considered a dummy.

We can see the right-hand-side is again a part of the left-hand-side:

(1): $x * c * 1$; (2): $r + x * c * 1$. (we see $\{(a * C_1) + (b * C_2)\}$ as x)

We can thus use this part again in a recursive rewrite immediately, not using an other rulebase, until we run out of busy variables.

We have to note that the gain now is not 1 as we first wanted but 0. This is not a bad thing as we later strip of the multiplication by one, gaining 1 operator after all.

A special case occurs if both a and b are nonexistent so the rule (1 of the 8) is:

$(c * C_1) + (c * C_2) \rightarrow \{(C_1) + (C_2)\} * c * 1$.

We can now evaluate $\{(C_1) + (C_2)\}$ immediately and substitute it on the place of the one.

We get: $(c * C_1) + (c * C_2) \rightarrow c * \text{EVALUATE}\{(C_1) + (C_2)\}$.

7.7.2. Register expressions.

Here the same considerations are made as in the integer trees. We want to make a sum tree of product trees. The difference is that a variable or subexpression can be present as the plain variable (a) or as its negation (*not* a).

We now have several representations for a subexpressions. The difficulty of a non rewrite is also possible.

If we have for example an expression like:

$$(a\wedge b)\vee(c\wedge b) \qquad (1)$$
$$(a\wedge b)\vee(c\wedge\neg b) \qquad (2)$$
$$(a\wedge\neg b)\vee(c\wedge b) \qquad (3)$$
$$(a\wedge\neg b)\vee(c\wedge\neg b) \qquad (4)$$

We can easily verify that only (1) and (4) can be rewritten. (2) and (3) can only be rewritten in some special cases such as a=true or c=true.

As we remember that a and ¬a are considered two different variables we can also strip of rule (4) as the ¬(¬a) case doesn't occur, and this leaves use with only one main rule (1) as in the integer-tree case.

Here we introduce a variable true to be a good constant to be added if there is no constant expression in a subtree using the property U_2.

We have to take care of another special case. This is very important because these special cases could wipe out an entire subexpression making the simplification much easier.

The rules used are:

$$a\wedge a \rightarrow a \qquad (1)$$
$$a\wedge\neg a \rightarrow \text{false} \qquad (2)$$
$$a\vee\text{false} \rightarrow a \qquad (3)$$

We use rule (2) to reduce a whole subexpression to false and as this false is part of a sumtree, we can use rule (3) and eliminate the subexpression completely.

Rule (1) is only used to minimise the number of variables. Leaving (1) out, so not minimising this expression, and losing one operator, and later gaining it by getting it out of parenthesis, is not a good practice. Why leaving it while you can gain now ?

The main rules are:

1 = true (can be seen as all ones).

$r \vee (a \wedge c \wedge C_1) \vee (b \wedge c \wedge C_2) \rightarrow r \vee \{(a \wedge C_1) \vee (b \wedge C_2)\} \wedge c \wedge 1$ (1)

$(a \wedge c \wedge C_1) \vee (b \wedge c \wedge C_2) \rightarrow \{(a \wedge C_1) \vee (b \wedge C_2)\} \wedge c \wedge 1$ (2)

Specials:

$(c \wedge C_1) \vee (b \wedge c \wedge C_2) \rightarrow \{(C_1) \vee (b \wedge C_2)\} \wedge c \wedge 1$

$\rightarrow \{(b \wedge 1) \vee C_1\} \wedge c \wedge (C_1 \vee C_2)$ (3)

$(c \wedge C_1) \vee (c \wedge C_2) \rightarrow \{(C_1) \vee (C_2)\} \wedge c \wedge 1$

$\rightarrow c \wedge (C_1 \vee C_2)$ (4)

We didn't quote all the rules as they are of the same form as the ones above. From rule (3) we can easy derive 3 others, leaving out b instead of a, and placing a rest part r as in rule (1). We thus get 3 rules extra. Further, form rule (4) we can derive a rule including the rest part, giving one extra rule. We get a total of: 2 + 4 + 2 = 8 rules as said before. We note that two constant expressions together are simplified, returning one new constant expression.

We further note the rewrite of rule (3) and derivations. Here we can use a rule we can't use in integer expressions namely:

$$(a \wedge b) \vee c \rightarrow (a \vee c) \wedge (b \vee c) \quad \text{(a)}$$

In integer rules this rule would look like:

(a * b) + c → (a + c) * (b + c) which it NOT true (unless c = 0)!.

The only thing we gain by using this rule (a) in (3) above is the that we kick constants that are not 1 out of the lower levels. This could make it easier evaluating an expression on a lower level.

7.8. Construction of rulebase, resulting in a canonical form.

Also in the expand rulebase we have to match over more than one level so we again encounter the problem of which local strategy we have to use.

As in the expand rulebase we now also use the strategy of bottom-up, and if a rewrite occurs we follow the results down the tree again until we find a canonical form again.

We recall the aim again here. Canonical: left sorted form, if any constant expression then this is the most right as possible one.

Property:

a op (b op c)	=	(a op b) op c	A_1

Here op is a symmetrical operator, and the two ops are of the same type.

Rulebase:

Rewrite-current			(1)
a op (b op c)	→	rewrite: (a op b) op c	(2)
(const1 op a)	→	(a op const1)	(3)
(a op const1) op const2	→	a op evaluate:(const1 op const2)	(4)
(a op const1) op b)	→	(a op b) op const1	(5)

The last 3 rules take care of any constant expression in an string of symmetrical operators. Note the order in which these rules are placed.

Rule (1) descents down the tree to the leaves. Rule (2) uses the given property to make a leftsorted tree. (The rewrite is only done on this rule, so we make a special rulebase only containing rule (2)). The other rules (3..5) are only used once, assuming that the tree is already in a leftsorted form.

Rule (3) only takes care of any expression only having two subexpressions, which is an constant expression. The constant is put to the right only if op is a symmetrical operator. If this expression is part of a bigger expression, so this is a leave of an expression, we need rule (4) and (5) on a higher level.

Rule (4) assumes that there is a leftsorted structure from this level down, and detects that

there are two constant expressions that could be combined to one.

Rule (5) is one step behind rule (4). The reason is that an expression that hits on rule 4 also hits on 5, but not necessarily the other way around. For instance: expression (a + 3) + 5 hits on both, (a + 3) + b only on rule (5). Swapping the rules, gives in the end the same effect, but it is not necessary to rewrite (a+3)+5 to (a+5)+3 and that to a+8 if it can be done in one step: (a+3)+5 → a+8. using rule (4).

The thought behind rule (2) is it to leave a leftsorted tree behind, and if there is a change at the current level, we have to go back and retrace this change until it has no effect at the levels below the change.

We see that rule (3) is based on the commutative property, while rules (2),(4) and (5) are based on the associative property.

We can extend our rulebase with a rule concerning relational operators:

const1 relop a → a reversed_relop const1 (6)

We see that we here have to alter the operator. We reverse the relational operator in the same step as we reverse the operator. For instance 3>a can be rewritten to a<3 as '<' is the reverse of '>'.

7.9. Construction of the collapse rulebase.

The construction of the collapse rulebase was not easy. We had to construct a mechanism that numbered the node, so we could identify every node as part of table where the number of variables are kept.
Furthermore we had to introduce some commands to manipulate a table which looks like this:

	not	mark	node1	node2	node3	...
var 1						
var 2						
var 3						
...						

- the *not* field indicates if, in case of a register tree, the variable is present as a *not* variable of as a normal variable.
- the mark field is used to keep track of the variables already dealt with or not. It also indicates which variable is processed at the moment.
- the node number fields are used to indicate the number of occurrences a variable has under a certain node, for instance (a*a)*a, where the underlined node keeps the node number, the entrance at 'a' is 3.

Further there are some special fields containing the number of nodes, variables, and for direct access, the number of the variable currently processed, the busy variable.

7.9.1. Construction of the table.

The table is constructed in a normal sweep through the tree. We assume it is a *OR* tree of *AND* trees (register expression), or and *SUM* tree of *PRODUCT* trees (integer expression).
We go down a tree, which is left sorted, stepping down left until we don't find another

OR or *SUM* operator. While we step down, we look to the right and see an *AND* or a *PRODUCT* operator, which gets a node number, placed in the table. The tree for which this *AND* or *PRODUCT* operator is the root, we scan for variables that are also entered in the table.

After this sweep we can count the total appearances and the spread of a variable. The spread is used to indicate in how many different nodes each variable is present. In this way we can easily determine the busy variable that is the most likely variable to get a big gain on operators.

Then the tree is 'subsorted'. This is also a scan of the tree down the leftside until there are no more *OR* or *SUM* operator. We then look at the right side wether this subexpression contains the busy variable. If not we skip the node, else we take out the *OR* or *SUM* operator completely with its right subexpression, and place it on the new root expression that holds the output of the procedure. The rest of the original expression is restored so we can go on with the procedure. After we reached the end of the original expression we simply place the rest part of the original expression, which doesn't hold any busy variables, to the end of the new root expression. We now have an new root expression which from top to bottom holds first all the subexpressions with the busy variable and then the part with no busy variable.

What happens if we use a rule like:

$$r + (b * a) + (c * a) \quad \rightarrow \quad r + (b + c) * a.$$

This means that all the variables in the 'b' and 'c' part are from the current level not interesting any more and have to be eliminated from the table. Furthermore, the two multiplications are reduced to one, which indicates that two node entries in the table have to be reduced to one. It is fairly easy to see that the new node can have either one of the old numbers of the nodes reduced, while the other one can be made completely empty. Also the only entry left for the remaining node is the busy variable ('a').

All the other variables, contained in 'b' and 'c' are out of scoop for the current level. They will appear again in the part under the variable 'a'. This can, however not be done immediately, because in the 'r'(est) part, there could still be an 'a' that can be processed with the rule above at the current level.

Concluding we have to keep track of the number of busy variables. If this number

becomes less than two, thus one or zero, this means the variable has the whole part under it that is possible, or the variable is reduced completely out of the expression. We need thus a mechanism to detect this. This is done during the table update reducing two nodes to one.

After we have detected an end of reduction for the busy variable, we have to call the procedure for the rest part, for which the table is still true, and the part under 'a', the (ex) busy variable. For this part under the (ex) busy variable we have to construct a new table, and as we remember the rulebase is processed from left to right, we can assume the rest part is already finished.

All this leads to a construction using at least 3 rulebases.

- One containing the reduction rules as in the example above. We use the command rewrite after we have matched a certain pattern to call the second rulebases, we call 'loop'.

- The second rulebase 'loop' checks first if all the variables are processed, which leads to an end of this part of the reduction process.
  If this is not so we have to check if in the new expression, the busy variable is ready, so no more reduction is possible with this one. If so, we call, for the 'r'(est) part rulebase one. This after we have selected a new busy variable, and a subsort, which brings this variable in the position so it can be processed. (to select the new variable we need a fourth rulebase but we skip this part for simplicity). For the part under 'a' we have to call rulebase tree.

- The third rulebase calls an inititiation of the table, and then it does the same as for part 'r'(est) in rulebase two.

The three (four) rulebases are, although they are short, quite complex constructed.

8. Practical point of view.

8.1. Rulebase structure.

We will now look again at the structure of a rulebase to get an impression of it seen from a practical point of view.

As we recall from the introduction, we have an expression that looks like a normal algebraic expression. In APDL we represent an expression as a tree. This means that an operator is followed by its operands. We could compare this with a prefix coded expression, e.g.,

a + (b * c)   is represented by: +a*bc, but

(b * c) + a   is represented by: +*bca.

A rulebase consists of patterns, which must be matched against an expression. This brings the problem of matching a pattern in normal algebraic form, with an expression in prefix notation form. To overcome this problem, all rules (patterns) are represented in a prefix notation also. Further is every operator, split into two parts. One parts states which operator type it is, so a monadic- or a dyadic operator, while the second part defines the operation the operator represents. A source pattern is directly followed by the targetpattern. We will place a remark between them to separate them for us humans.

As the program is written in the progam language C We will place these remarks between '/*' and '*/' as is usual in the C). In this way a rule like:

' z + (a*b*C1) + (c*b*C2) → z + (a*C1 + c*C2) * b * 1 '

or in prefix :'++ z ** a b C1 ** c b C2 → + z **+* a C1 * c C2 b 1 ' , looks like:

```
' DYADOP,INTADDOP,
    DYADOP,INTADDOP,
      EXP1,                       /* z */
      DYADOP,MULOP,
        DYADOP,MULTOP,
          EXP2,                   /* a */
          EXP3,                   /* b */
        CONST1,                   /* C1 */
    DYADOP,MULOP,
      DYADOP,MULOP,
        EXP4,                     /* c */
        ISEXP3,                   /* is it also b ? */
    CONST2,                       /* C2 */
/* to: now comes the target expression */
  DYADOP,INTADDOP,
    EXP1,                         /* z */
    DYADOP,MULOP,
      DYADOP,MULOP,
        DYADOP,INTADDOP,
          DYADOP,MULOP,
            EXP2,                 /* a */
            CONST1,               /* C1 */
          DYADOP,MULOP,
            EXP4,                 /* c */
            CONST2,               /* C2 */
        EXP3,                     /* b */
      INT1,                       /* 1 */
  END,                            /* marks end of a rule */'
```

We see that for a human it is quite difficult to read such a rule. That is why we made a habit of it to place a remark at the top of a rule that states the rule in readable form, and some remarks in the rule where needed.
For changing the expression level, we normally use the rewrite command. This command calls an rulebase, stated by name after the command and the expression that is used there

after. An example: If we want an expression like a + b to be rewritten by a rulebase example1, we get in the rulebase the following line:

```
' DYADOP,INTADDOP,              /* + */
    EXP1,                       /* a */
    EXP2,                       /* b */
/* to */
  REWRITE,EXAMPLE2,             /* call rulebase EXAMPLE2 */
  DYADOP,INTADDOP,              /* + */
    EXP1,                       /* a */
    EXP2,                       /* b */
  END,                          /* end of rule */
'
```

We see we first have to get a match to get a call on an other rulebase.
If we want to use rules at different levels we have to have a method for getting up and down one level in an expression.
We can only call down, as this is the way an expression is build. Going up means first getting down to the leaves. We will show how this is done.

```
MONADOP,EXP1, /* to */   MONADOP,
                              REWRITE,CURRENT,EXP1,
                         END,

DYADOP,EXP1,EXP2, /* to */ DYADOP,
                              REWRITE,CURRENT,EXP1,
                              REWRITE,CURRENT,EXP2,
                         END,
```

The keyword 'CURRENT' is used to call the current rulebase again. Using these two rules we can get to the leaves of an expression tree. We note the possibility of placing these rules at the beginning of a rulebase or at the end having the effect of using the rest of the rules from the leaves to the root or the other way around.
As we use these rules in almost every rulebase we call these two rules:
Recursive_rewrite.

8.2. Subsort in detail.

We will look at the subsort procedure again, but now in more detail. The aim of this rulebase is to make the select variable, the 'busy var', adjacent.

From the sort rulebase we get an expression tree that is left sorted. This means an expression looks like mentioned in Figure 16. We have already made an output expression pointer which will hold the output expression from this procedure. The star marks the subtree with the variable we want to sort to the right. Note that the aim is to set these variables to the right only.

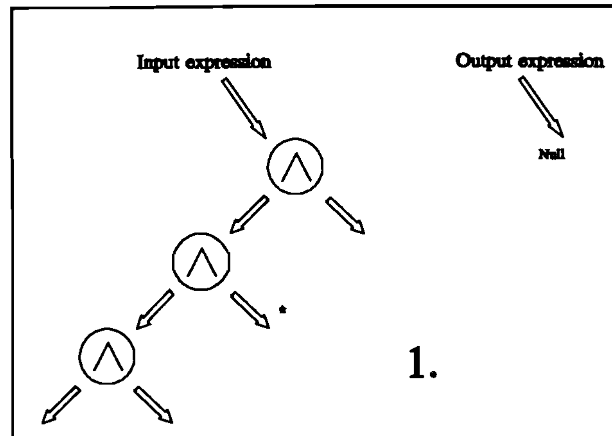

Figure 16

In Figure 17 we already moved a pointer from the input expression pointer to the node which holds the marked variable to the right, called the aux pointer. Further we have put a tmp pointer to the node just above of the aux pointer. We will now take out the node, pointed to by the aux pointer completely and place it in the output expression.

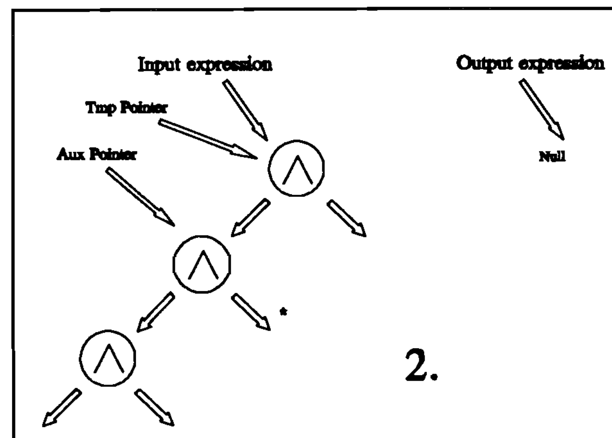

Figure 17

As we see in Figure 18 we first clip out the selected node from the input expression. Note that for this we need the tmp pointer. This already completes the part on the input expression. We thus don't need the tmp pointer anymore and will see it no more.

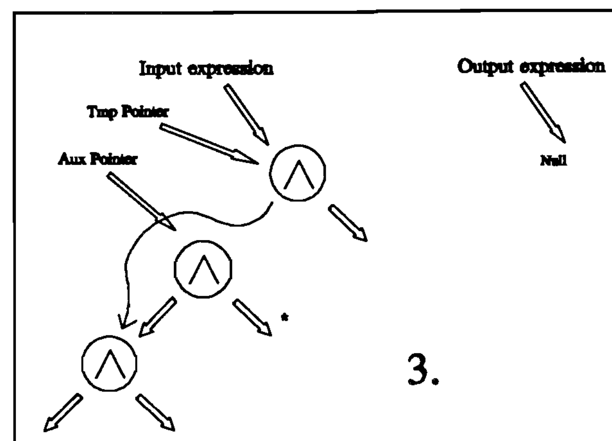

Figure 18

In Figure 19 we see that we will now start implanting the marked var plus node into the outputexpression. We will first take over the rest of the output expression by pointing with the left side of the marked node to the rest of the output expression. We will there after insert the marked node completely in the output expression by taking the outputexpression and point it to the marked node (Figure 20). The result can be reordered so we can see the result clearly(Figure 21).

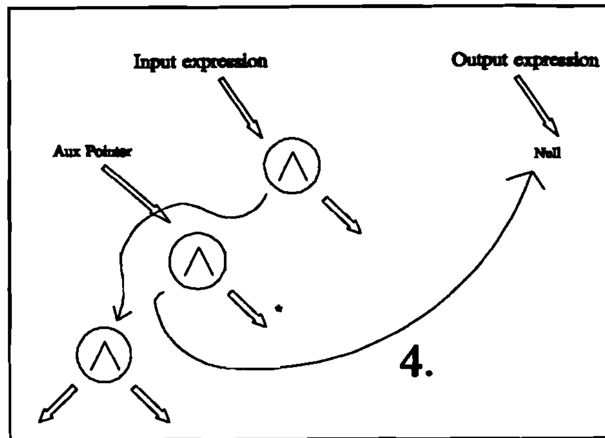

Figure 19

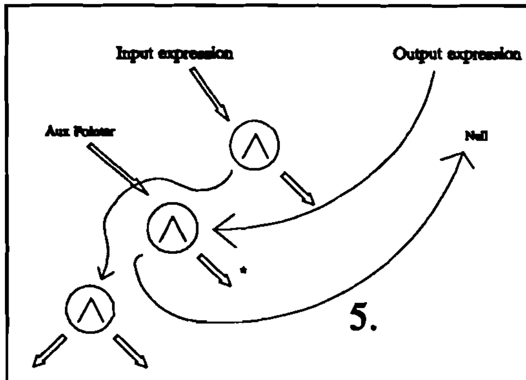

Figure 20

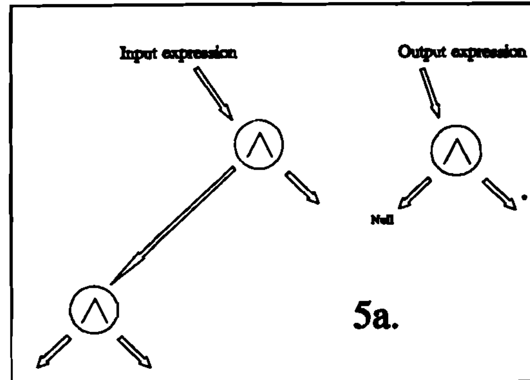

Figure 21

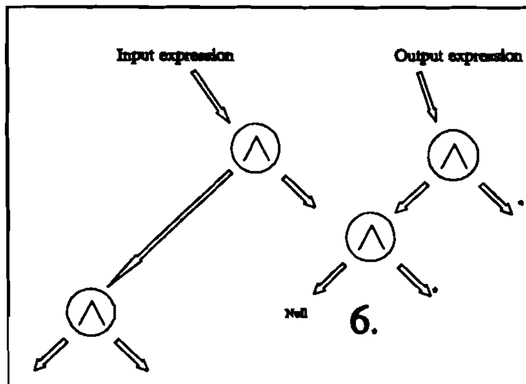

Figure 22

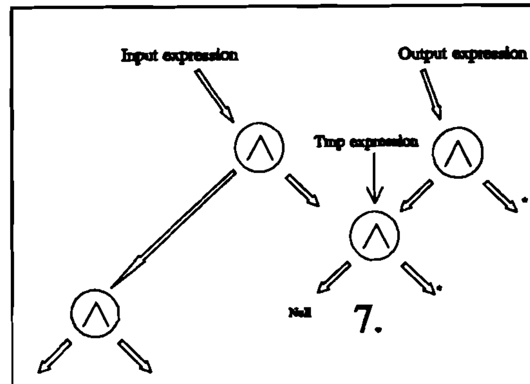

Figure 23

We will assume we found another marked node and this one is placed also on the outputexpression (Figure 22). We will now append the rest of the input expression at the end of the output expression. Therefore we first place a tmp pointer to the end of the outputexpression (Figure 23).

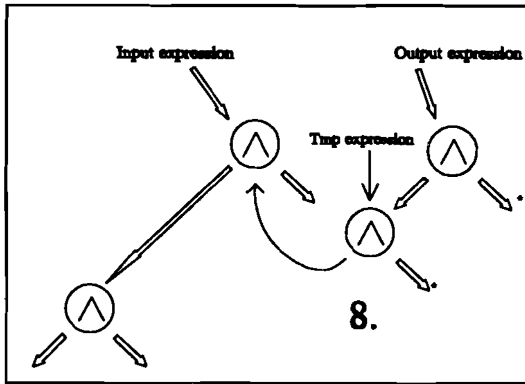

Figure 24

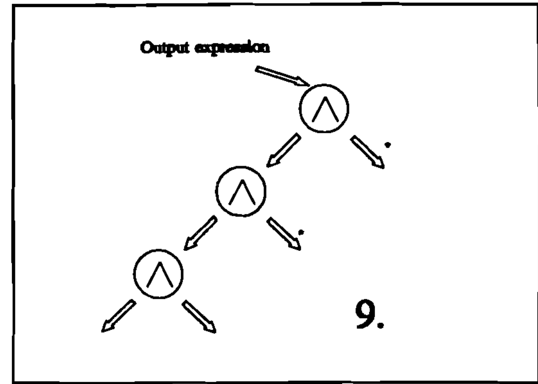

Figure 25

We then simply replace the null pointer (initial setting of the outputexpression) with the inputexpression-pointer (Figure 24).
In this way we created an outputexpression that has the marked nodes at the beginning of the expression and all the marked nodes are adjacent (Figure 25).

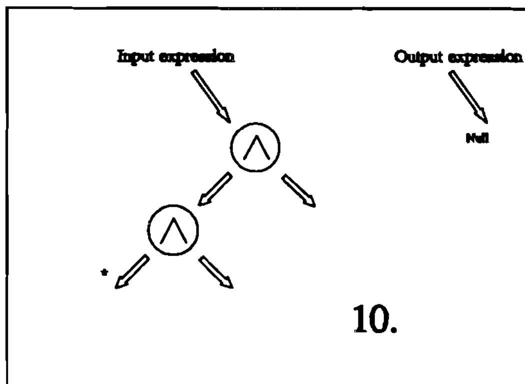

Figure 26

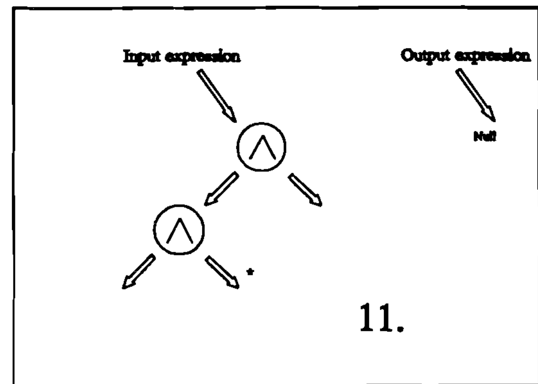

Figure 27

We will look at one special case, but there are more. The marked expression is not at the right side of a node but on the left side (Figure 26). We simple swap the left and righthand side of this node as the node represents a symmetrical operator, and we thus get an expression (Figure 27) which is simply used with the method mention before.
Another special cases occurs when the first node holds a constant expression to the right. In this case we store this node completely with a pointer defined specially for holding constants. We then process the rest of the expression as we described and at the end insert the constantexpression at the beginning of the outputexpression, which leads again to en expression with first a constant expression and then the rest.

Of course there are more special case, such as an expression with only one variable or just a constant expression. These however can be dealt with fairly easy at the beginning of the procedure. In the rulebase which calls this procedure, we have to call it on the *AND* and *Product* trees first, and so sorting the busy var to the right. After this is done we call the procedure again on the *OR* and *SUM* trees. Information about the marking of a node, we could simply be extracted of the table made before.

Register expressions are special treated regarding the complement of variables as they are placed as near as possible to the variable itself. After the situation in Figure 22 we therefore select the complement of the busy var and repeat the whole process on the input expression.

9. Summary.

We have now handled all the parts of the rulebasesystem. To get an impression of the whole system we will mention the parts here in the order they appear in the rewrite process.

9.1. Integer expressions.

- Eliminate all minus operators converting them to a multiplication with -1.

- We then expand the expression to a *SUM* tree of *PRODUCT* trees.

- While there are variables left not processed:

  - Rewrite all to a canonical form. (left sorted).

  - Introduce the required constants for rule reduction.

  - Then find out the most common variable. (busy variable).

  - Make the busy var the right most var in the tree and sort all busy vars in the subexpression to that side too.

  - Use the rules:

    z + (x * a * C1) + (y * a * C2) → z + ((x * C1) + (y * C2)) * a.

    - use this rule as long as there are any subtrees left with this busy var, if no more left: rewrite z and subtree of a separately.

- Try to remove the constants introduced earlier.

- Try to reverse some expansions that did not result in a simplification.
- rewrite ends.

For the register expressions it is pretty much the same.

9.2. Register expressions

- Eliminate all *not* / *reverse* operators, by pushing them to the operands as far as possible.
  We can then look at a variable with such a prefix as one new variable.

- We then expand the expression to a *OR* tree of *AND* trees.

- While there are variables left not processed:

  - Rewrite all to a canonical form. (left sorted).

  - Introduce the required constants for rule reduction.

  - Then find out the most common variable. (busy variable).

  - Make the busy var the right most var in the tree and sort all busy vars in the subexpression to that side too. Search for the complement of the busy var and place it as second to the right. We will use this later.

  - Use the rules:
    z or(x and a and C1)or(y and a and C2)→z or(x and C1)or(y and C2)and a.
     - use this rule as long as there are any subtrees left with this busy var, if no more left: rewrite z and subtree of a separately.

    With the register algebra there are some special cases concerning cancellations:

$a \wedge 0$	$\rightarrow 0$	(1)
$a \wedge 1$	$\rightarrow a$	(2)
$a \wedge a$	$\rightarrow a$	(3)
$a \wedge \neg a$	$\rightarrow 0$	(4)
$a \vee 0$	$\rightarrow a$	(5)
$a \vee 1$	$\rightarrow 1$	(6)
$a \vee a$	$\rightarrow a$	(7)
$a \vee \neg a$	$\rightarrow 1$	(8)

We could use these last rules at the beginning of the simplification process also, saving time and processing time. We have, however, keep in mind the number of mutations we could encounter, searching for these special cases. If we are at the point were we have an canonical form, we can wait for the collapse part and work form the inside, where these special cases are more accessible anyway, because of the strip-off of the variables, leaving the constants.

- Try to remove the constants introduced earlier.

- Try to reverse some expansions that did not result in a simplification.

- Rewrite ends.

10. Conclusions.

The definitions made for rulebases, the process of matching and rewriting are clear. This enlarges the insight in matching and rewrite problems.
We can only present one heuristic solution to the rewriting / reduction problem, as this is a NP-complete problem.
We try to catch as many expressions as we can but there will always be some expressions that have simpler forms than our rulebasesystem will produce.

We have introduced a method for ordering a rulebase. The construction of the dependency table could be automated and the group forming could be left to computers as well. Some decisions are still to be made by humans however, because the computer can't overlook a rulebase's task.

The rulebase system is very efficient in rewriting more or less all the offered expressions, to a simpler form if this is possible. It is therefore useful for compilers and other processes that work with expressions. The only restriction to an expression is that it is offered in the described dynamic structure, or it has to be transformed into such a form. The APDL-compiler does transform any expression into this desired dynamic structure.
Every rewritesystem has some weak points, as it is a heuristic solution. These weak points can be taken care of easily in the rulebasesystem by introducing some special treatments for the expressions which course the problem.

In the rulebasesystem with multiple rulebases, it is simple to introduce new command-words, which can be used as a name for a new rulebase, but also to implement some rewrite-routines that are not possible in the rulebasesystem itself. There are no limitations to the use of new command-words.
The rulebase is simple to use and, if desired, could be expanded for future use. This could also be of benefit for other applications.
Our rulebase system is working, although it sometime takes a lot of time expanding, especially *XOR* expressions. It could be of benefit to try to implement rules that simplify *XOR* expressions first before they are expanded to *AND* and *OR* expressions, because every *XOR* expandation introduces two times as much operands as there are in the original expression.

The system will reduce most expressions to constant expressions if possible. This is the first goal for this system, as the compiler can then eliminate some parts of the program that is compiled. This will always be of benefit, also to other applications.

11. Recommendations.

This system is based on a static number of rules, that are used sequentially. It could be of use to store every reduction, if successful, so it could be reproduced later. We could now create a dynamic rulebase, and it would look more like artificial intelligence. We could start with this system, although modified for the learning of new 'reduction rules'. The process of learning would be the storing of input expressions and evaluate the output expressions against it. If the system finds a reduction worth will this 'new rule' is kept in the system.

The advantages of this system are that it could reduce very complex expressions very quickly if recognized. If a user has some sort of habit of bringing up the same expression repeatedly, we could keep a profile rulebase for this user. Such a profile rulebase is a sort of preference rulebase, that is checked before the 'normal' rulebase is used. This could increase the power of a reduction system, as the more frequently used expressions are recognized immediately and reduced with only one step.

The disadvantage of such a system could be that the number of rules explodes, so not only more computerstorage space would be needed, but also more computerpower, as the matching process would take much more time. These could however be controlled with a limited learning period. Or a more efficient reduction selection, e.g., rules with a gain of only one or two would not be admitted in the profile rulebase, but rules with more gain would.
Expressions which look like each other are not recognized although they could be reduced in a very similar way.

Another problem is the order of the rulebase. We have brought up some suggestions for ordering a rulebase but we have to decide what strategy to follow, and how to order the rules. Only very simple rulebases could be the result of automated rulebase ordering. I think that it is possible to make a sort of preselection program, which could help extensive. It could construct the table, and order the rules into groups.

12. References.

[1] L.Aiello : Using meta-theoretic reasoning to do algebra.*5th Conf an automatic deduction (ed:Bibel) (RC:DBC 80 AUT) 1980, p1-12.*

Contents : Experiments with FOL, showing improvements in proofs.

[2] J.C.Beatty : An Axiomatic Approach to Code Optimization for Expressions. *J ACM 19(4) (Oct 72) 613-639.*

Contents : For parallel computers, only one reference to a variable, minimal delay(parallel) , minimal code generation as Sethi and Ullman.

[3] G.A.Blaauw : Optimization of Relational Expressions Using A Logical Analogon. *IBM J RES Develop Vol 27 no 5 (Sep 83) 497-519.*

[4] B.Buchberger : Algebraic Simplification. *Computer Algebra. Symbolic and Algebraic Computation.* (RC: CAC 82 COM).

Contents : Problem, definitions , Canonical simplification, reduction, critical pairs(Knuth- Bendix).

[5] B.F.Caviness : On Canonical Forms and Simplification. *J ACM Vol 17 no 2 (Apr 70) 385-396.*

Contents : Canonical forms,decidebility,exponential expressions.

[6] K.M.Chilukuri : Function definitions in term rewriting and applicative programming.*Information and control 71 (1986) 186-217*

Contents : Normal forms, rewrite rules, formal description, conditional defs, transformations between formalisms.

[7] J.P.Fitch : On Algebraic Simplification. *Computer J Vol 16 no 1 23-27.*

Contents : Brief review, compactification , intelligibility , identity (=0 ?) , Classification of transformations, proof of canonical forms.

[8] D.J.Frailey : Expression Optimization Using Unary Complement Operators. *Sigplan Notices Jul 1970 67-85.*

Contents : Search for redundancy in a easily detectible way. Complement operators. (-a -> a(comp -)), canonical forms, common subexpressions detection (in different expressions !). Cancelling (a-a , x/x etc). Detecting complements.

[9] P.A.V.Hall : Optimization of Single Expressions in a Relation Data Base System. *IBM J RES Develop (May 76) 244-257.*

[10] J-M Hullot : Canonical forms and unification. *5th Conf an automatic deduction (ed:Bibel) (RC:DBC 80 AUT) 1980, p318-334.*

Contents : K&B deduction, T-unification, some improvements.

[11] C.K.Mohan and M.K.Srivas : Function definitions on Term Rewriting and Applicative Programming. *Information and control 71, p186-217 (1986).*

Contents : (Un)conditional Term Rewriting, guaranteed termination.

[12] J.Moses : Algebraic Simplification : A Guide for the Perplexed. *Comm ACM Vol 14 no 8 (Aug 71) 527-537.*

Contents : Guide, Substitution methods , Class subdivisions, labelling according to Brown, Expression swell , canonical simplifiers (with unsolvables !).

[13] J.E.Sammet : Bibliography 11. *Computer Reviews (ACM) Jul-Aug 1866 B1-B31.*

[14] J.E.Sammet : Introduction to Formac. *IEEE Trans on Elect Computers Aug 1964 386-394.*

Contents : Description of FORMAC.

[15] J.E.Sammet : Survey of formula Manipulation. *C ACM vol 9 no 8 (aug 66) 555-569.*

Contents : Integration,differentiation, simplification. Refers to FORMAC and MATHLAB. Global function of different systems. BIBLIOGRAFY.

[16] J.E.Sammet : Formula Manipulation by Computer. *Advances in Computers Vol 8 47-102.*
(E: DAC 63 ADV).
Contents : Summary of FORMAC.

[17] R.G.Tobey : Automatic Simplification in Formac. *Proc - FJCC 1965 37-53.*
Contents : Rules for the use of FORMAC, derivations.
Only rules could be interesting.

[18] R.G.Tobey : Experience with FORMAC Algorithm Design. *Comm ACM Vol 9 no 3 (Aug 66) 589-597.*
Contents : FORMAC.

Books

[19] H.Aït-Kaci & M.Nivat : Resolution in algebraic structures vol 2 Rewrite techniques, 89 Ac.Press (W:BDD89 RES).
Contents : Rewrite techniques : theory.

[20] R.v.Book : Formal Language Theory.

[21] S.D.Danielopoulos : Algebraic Simplification in fortral , 84.
Contents : Makes tuples and simplifies canonical.

[22] P.A.V.Hall : Optimization of Single Expressions in a Relation Data Base System. *IBM J RES Develop (May 76) 244-257.*
Contents : Only some rules but restricted to dbases.

[23] M.Jantzen : Confluent String rewriting , 88 . (RC : DBN 99 JAN).
Contents : Compiler directed substitution.

[24] J.P.Jouannaud : Rewriting techniques and applications, 87 Ac.Press (W:DBF 87 REW).
Contents : Rewrite techniques : theory.

[25] J.W.Klop : Term rewriting systems from Church_Rosser to Knuth-Bendix and beyond. *Computer Science/Department of software technology Rep:CS-R9012, May-1990.*
Contents : A short survey covering abstract rewriting, Combinatory Logic, orthogonal systems, strategies, critical pair completion, and some extended rewriting formats.

[26] A.Middeldorp : Modular properties of term rewriting (W:DBN 90 MID). 1990.

[27] E.W.Ng : Symbolic and Algebraic Computation.(RC: DGP 79 SYM) EUROSAM '79, An International Symp. Marseille, France June '79.
Contents : Mostly practical application on simplification.

[28] Proceedings of the 1977 MACSYMA Users'conference (NASA). (RC: DPG 77 MAC). More writers.
Contents : Several applications for MACSYMA.

[29] R.Socher : Simplification and reduction for automated theorem proving , 90. (CM : APD 90 SOC).
Contents : Boolean Simplification.

[30] W.Wulf : Fundamental Structures of computer Science (RC book). 549-564.
Contents : Substitution and parameter matching.

```
  1| 0| #define MAXVARS  20           /* ps no checking is done ! */
  2| 0| #define MAXNODES 50           /* ps no checking is done ! */
  3| 0| #define MARK     0            /* marked field */
  4| 0| #define SPREAD   1            /* spreadfunction = how many nodes contain var */
  5| 0| #define MAIN     2            /* how many times a var is present in the total exp */
  6| 0| #define MINNODE  MAIN+1       /* first free node number */
  7| 0| #define marked count[MARK]    /* countarr[0] = marked used */
  8| 0| #define spread count[SPREAD]
  9| 0| #define CLEAR 0
 10| 0| #define BUSY  1
 11| 0| #define DONE  2
 12| 0|
 13| 0| typedef struct counttypestruct counttype;
 14| 0| struct countarrstruct
 15| 7|       {
 16| 7|       char *varname ;        /* pointer to name of var */
 17| 7|       int notflag ;          /* flag which indicates a var as a not var ( register expression only
   |  | )*/
 18| 7|       int count[MAXNODES]; /* array of nodes, indicating the number of occurences */
 19| 7|       );                     /* 0 is a markfield , 1 is the main root expression, 2... are
   |  | subnodes *
 20| 0| struct counttypestruct
 21| 3|    {
 22| 3|    struct countarrstruct countarr[MAXVARS]; /* as array of vars */
 23| 3|    int countnum;              /* number of vars in table */
 24| 3|    int node;                  /* number of nodes in table */
 25| 3|    int busyvar;               /* the bussy var */
 26| 3|    };
 27| 0|
 28| 0| typedef struct expstatstruct expstat;
 29| 0| struct expstatstruct
 30| 3|    {
 31| 3|    rulebasetype *rulebase,*ruleptr;
 32| 3|    expressionnode *exp1,**parentexp1,
 33|18|                   *exp2,**parentexp2,
 34|18|                   *exp3,**parentexp3,
 35|18|                   *exp4,**parentexp4,
 36|18|                   *const1,**parentconst1,
 37|18|                   *const2,**parentconst2;
 38| 3|    int node1,node2,node3,node4,nn,killnode;
 39| 3|    /* nn = nodenumber for rebuild , commands PUSHNN,POPNN*/
 40| 3|    int exp1cnt,exp2cnt,exp3cnt,exp4cnt,const1cnt,const2cnt;
 41| 3|    operatortype op1,op2,op3;
 42| 3|    };
 43| 0|
 44| 3|    int match_hits;  /* used to count the hitrate on match */
 45| 3|    int match_trys;  /* used to cuont the hitrate on match try's */
 46| 3|    int notflag;     /* used to signal a set/reset on matching two vars ! for a and !a */
 47| 3|    /* these 3 are declared extern in header, for use in reducexp.c */
 48| 0|
 49| 3|    int hit = FALSE;             /* global flag to indicate a hit of a rule */
 50| 3|    int allhad = FALSE;          /* global flag used to indicate if all vars are processed *
 51| 3|    counttype *cntarr = NULL;    /* global pointer to table */
 52| 3|    int decreasevar = 0;         /* used to count the number of reductions to update table */
 53| 3|    int size = 0;                /* used to remember the number of bits are in a register */
 54| 3|    int flag = FALSE;
 55| 0|
 56| 0| /*************************************************************************************************
 57| 1|  * printtable : prints the list ( table ) nothing else                                          *
 58| 1|  *************************************************************************************************
 59| 1|  */
 60| 0| void print_table( counttype *countarrptr )
 61| 2|   {
 62| 0| #define MRCOUT debugfile
 63| 2|   int no,var;
 64| 6|       iosprintf(MRCOUT," vars : %d , nodes : %d , busyvar : %d ( %2s )\n",
 65|15|                countarrptr->countnum,
 66|15|                countarrptr->node,
 67|15|                countarrptr->busyvar,
 68|15|                countarrptr->countarr[countarrptr->busyvar].varname);
 69| 0|
 70| 3|    iosprintf(MRCOUT,"                                   ");
 71| 3|    for ( no = 0 ; no < countarrptr->node ; no++)
 72| 6|       iosprintf(MRCOUT," %2d ",no);
 73| 3|    iosprintf(MRCOUT,".\n");
 74| 0|
```

```
 75| 3|    for ( var = 0 ; var < countarrptr->countnum ; var++ )
 76| 6|      {
 77| 6|      iosprintf(MRCOUT," var %2d ( %2s ) ( not : %d ): ",
 78|13|              var,
 79|13|              countarrptr->countarr[var].varname,
 80|13|              countarrptr->countarr[var].notflag);
 81| 6|      for ( no = 0 ; no < countarrptr->node ; no++)
 82|10|          iosprintf(MRCOUT," %2d ",countarrptr->countarr[var].count[no]);
 83| 6|      iosprintf(MRCOUT,".\n");
 84| 6|      }
 85| 3|    }
 86| 0|
 87| 0| /*****************************************************************************************
 88| 1|  * boolean areequalvars(expressionnode *exp1 , expressionnode *curexp);                  *
 89| 1|  * test if two vars are equal                                                           *
 90| 1|  ****************************************************************************************
 91| 1|  */
 92| 0| boolean areequalvars(expressionnode *exp1 , expressionnode *curexp)
 93| 3|    {
 94| 3|    if ( curexp -> nodetype == MONADICOPERATOR &&
 95|10|         exp1 -> nodetype == MONADICOPERATOR )
 96| 7|       return
 97|14|              ( curexp -> exp.monadicoperator.operator == exp1 -> exp.monadicoperator.operator ) &
   |  | &
 98|14|              areequalvars(exp1 -> exp.monadicoperator.subexpression,curexp -> exp.
   |  | monadicoperator.subexpression);
 99| 3|    if ( curexp -> nodetype == VAR && exp1 -> nodetype == VAR )
100| 7|       return ( curexp -> exp.var.refvar == exp1 -> exp.var.refvar );
101| 3|    return FALSE;
102| 3|    }
103| 0|
104| 0| /*****************************************************************************************
105| 1|  * isbusyvar: returns true of false if in a expression                                  *
106| 1|  ****************************************************************************************
107| 1|  */
108| 0| int isbusyvar(expressionnode *curexp , counttype *countarrptr,int notf)
109| 0| {
110| 0|
111| 5|     if ( curexp -> nodetype == MONADICOPERATOR && curexp -> exp.monadicoperator.operator ==
   |  | NOTOP)
112|12|            return isbusyvar(curexp->exp.monadicoperator.subexpression,countarrptr,!notf);
113| 5|     return (   curexp -> nodetype == VAR
114|13|             && curexp ->exp.var.refvar -> elemname == countarrptr->countarr[countarrptr->busyvar]
   |  | .varname
115|13|             && countarrptr->countarr[countarrptr->busyvar].notflag == notf );
116| 0| }
117| 0| /*****************************************************************************************
118| 1|  * countvar : count var in expression and puts them in list allocated also and return as a *
119| 1|  * pointer to this list                                                                  *
120| 1|  ****************************************************************************************
121| 1|  */
122| 0| counttype *countvar(expressionnode *curexp )
123| 2|   {
124| 2|   operatortype savedyadop = NULL;
125| 2|   int var,node;
126| 2|   counttype *countarrptr;
127| 0|
128| 2|   /* test if table exist, if not create one, should be created although */
129| 2|   if ( cntarr == NULL )  cntarr = mmalloc( counttype ); /* make first list */
130| 2|   if ( cntarr == NULL ) fatalerror("couldn't alloc countarray ( ");
131| 0|
132| 2|   countarrptr = cntarr;
133| 2|   /* init to zero */
134| 2|   for ( var = 0 ; var < MAXVARS ; var++)
135| 7|       {
136| 7|       countarrptr -> countarr[var].notflag = FALSE;
137| 7|       countarrptr -> countarr[var].varname = NULL;
138| 7|       for ( node = 0 ; node < MAXNODES ; node++) countarrptr -> countarr[var].count[node] = 0;
139| 7|       }
140| 2|   countarrptr -> node = MINNODE; /* init */
141| 2|   countarrptr -> countnum = 0;   /* init */
142| 2|   countarrptr -> busyvar = 0;    /* init */
143| 0|
144| 2|   if ( curexp->nodetype == MONADICOPERATOR )
145|12|           {
146|12|           return countvar(curexp -> exp.monadicoperator.subexpression);
```

```
| 147|12|            }
| 148| 0|
| 149| 2|   savedyadop = curexp -> exp.dyadicoperator.operator ;
| 150| 2|   curexp -> nodenumber = MAIN ;
| 151| 0|
| 152| 2|   while ( curexp->nodetype == DYADICOPERATOR && curexp -> exp.dyadicoperator.operator ==
|    |  | savedyadop )
| 153| 4|     {
| 154| 4|     curexp -> nodenumber = MAIN ; /* number them too */
| 155| 4|     curexp -> exp.dyadicoperator.rightexpression -> nodenumber = countarrptr -> node;
| 156| 4|     countlocalvar(curexp -> exp.dyadicoperator.rightexpression , countarrptr,FALSE);
| 157| 4|     countarrptr -> node++ ; /* next node */
| 158| 4|     /* for ( var = 0 ; var < MAXVARS ; var++) countarrptr -> countarr[var].count[countarrptr ->
|    |  | node] = 0; */
| 159| 4|     curexp = curexp -> exp.dyadicoperator.leftexpression;
| 160| 4|     }
| 161| 2|   /* last node on left */
| 162| 4|     curexp -> nodenumber = countarrptr -> node;
| 163| 4|     countlocalvar(curexp , countarrptr, FALSE);
| 164| 4|     countarrptr -> node++; /* next node */
| 165| 4|     /* for ( var = 0 ; var < MAXVARS ; var++) countarrptr -> countarr[var].count[countarrptr ->
|    |  | node] = 0; */
| 166| 0| return countarrptr;
| 167| 0| }
| 168| 0|
| 169| 0| /*******************************************************************************************
| 170| 1|  * countlocal : count localy vars in expression and puts them in the list pointed by        *
| 171| 1|  *******************************************************************************************
| 172| 1|  */
| 173| 0| countlocalvar(expressionnode *curexp , counttype *countarrptr, int notflag)
| 174| 2|   {
| 175| 2|   int i,found;
| 176| 0|
| 177| 0|
| 178| 2|   switch ( curexp->nodetype )
| 179| 4|     {
| 180| 4|     case MONADICOPERATOR : if ( curexp->exp.monadicoperator.operator == NOTOP )
| 181|32|                                 notflag = TRUE ; /* set notflag */
| 182|27|                            curexp->exp.monadicoperator.subexpression->nodenumber = curexp->
|    |  | nodenumber;
| 183|27|                            countlocalvar(curexp->exp.monadicoperator.subexpression,countarrptr,
|    |  | notflag);
| 184|27|                            break;
| 185| 4|     case DYADICOPERATOR  : curexp->exp.dyadicoperator.leftexpression->nodenumber  = curexp->
|    |  | nodenumber;
| 186|27|                            curexp->exp.dyadicoperator.rightexpression->nodenumber = curexp->
|    |  | nodenumber;
| 187|27|                            countlocalvar(curexp->exp.dyadicoperator.leftexpression,countarrptr,
|    |  | notflag);
| 188|27|                            countlocalvar(curexp->exp.dyadicoperator.rightexpression,countarrptr,
|    |  | notflag);
| 189|27|                            break;
| 190| 4|     case CONSTANT        : break ; /* no count */
| 191| 4|     case VAR             : for ( i = 0 , found = FALSE ; i < countarrptr -> countnum ; i++ )
| 192|33|                                  if ( curexp -> exp.var.refvar -> elemname == countarrptr ->
|    |  | countarr[i].varname &&
| 193|38|                                      countarrptr -> countarr[i].notflag == notflag )
| 194|38|                                      {
| 195|38|                                      countarrptr -> countarr[i].count[countarrptr -> node]++;
| 196|38|                                      found = TRUE;
| 197|38|                                      break;
| 198|38|                                      }
| 199|28|                             if ( !found )
| 200|32|                                 /* new var found -> init all above to zero */
| 201|32|                                 {
| 202|32|                                 countarrptr -> countarr[countarrptr -> countnum].varname = curexp
|    |  | -> exp.var.refvar -> elemname;
| 203|32|                                 countarrptr -> countarr[countarrptr -> countnum].notflag =
|    |  | notflag;
| 204|32|                                 countarrptr -> countarr[countarrptr -> countnum].count[
|    |  | countarrptr -> node] = 1;
| 205|32|                                 countarrptr -> countarr[countarrptr -> countnum].marked = CLEAR;
| 206|32|                                 countarrptr -> countnum++;
| 207|32|                                 }
| 208|28|                             break;
| 209| 4|     default              : fatalerror(" Countvar :unknown type ");
```

```
210| 4|     }
211| 2|   }
212| 0|
213| 0| /*****************************************************************************************
214| 1|  * sortarray sort array on key1, 1 and 2 , returns countarrptr                           *
215| 1|  *****************************************************************************************
216| 1|  */
217| 0| sortarray( int key1 , int key2 , counttype *countarrptr)
218| 0| {
219| 0| /* count num points to unused place so we use it as tmp */
220| 0| int i,j,k;
221| 9|          for ( i = 0 ; i < countarrptr -> countnum ; i++)
222|14|               for ( j = i+1 ; j < countarrptr -> countnum ; j++)
223|19|                    if ( countarrptr -> countarr[i].count[key1] < countarrptr -> countarr[j].count[
   |  | key1] ||
224|22|                       ( countarrptr -> countarr[i].count[key1] == countarrptr -> countarr[j].
   |  | count[key1] &&
225|24|                         countarrptr -> countarr[i].count[key2] <  countarrptr -> countarr[j].
   |  | count[key2] ))
226|25|                          { /* swap them */
227|25|                          countarrptr -> countarr[countarrptr -> countnum].varname   = countarrptr
   |  | -> countarr[i].varname;
228|25|                          countarrptr -> countarr[countarrptr -> countnum].notflag   = countarrptr
   |  | -> countarr[i].notflag;
229| 0|
230|25|                          countarrptr -> countarr[i].varname = countarrptr -> countarr[j].varname;
231|25|                          countarrptr -> countarr[i].notflag = countarrptr -> countarr[j].notflag;
232| 0|
233|25|                          countarrptr -> countarr[j].varname = countarrptr -> countarr[countarrptr
   |  | -> countnum].varname;
234|25|                          countarrptr -> countarr[j].notflag = countarrptr -> countarr[countarrptr
   |  | -> countnum].notflag;
235| 0|
236|25|                          for ( k = 0 ; k < countarrptr -> node ; k++)
237|30|                               {
238|30|                               countarrptr -> countarr[countarrptr -> countnum].count[k] =
   |  | countarrptr -> countarr[i].count[k];
239|30|                               countarrptr -> countarr[i].count[k]       = countarrptr ->
   |  | countarr[j].count[k];
240|30|                               countarrptr -> countarr[j].count[k]       = countarrptr ->
   |  | countarr[countarrptr -> countnum].count[k];
241|30|                               }
242|25|                          }
243| 0| }
244| 0|
245| 0| /*****************************************************************************************
246| 1|  * make_gloabl_count : makes undate on main count in list *countarrptr ( doesn't clear marks) *
247| 1|  *****************************************************************************************
248| 1|  */
249| 0| make_global_count( counttype *countarrptr )
250| 0| {
251| 3|    int no,var;
252| 3|    for ( var = 0 ; var < countarrptr -> countnum ; var++ )
253| 6|       {
254| 6|       countarrptr -> countarr[var].count[MAIN] = 0;
255| 6|       countarrptr -> countarr[var].count[SPREAD] = 0;
256| 6|       for ( no = MINNODE ; no < countarrptr -> node ; no++)
257|10|           {
258|10|           if ( countarrptr -> countarr[var].count[no] != 0)
259|16|                {
260|16|                countarrptr -> countarr[var].count[MAIN] += countarrptr -> countarr[var].count[no]
   |  | ;
261|16|                countarrptr -> countarr[var].count[SPREAD]++;
262|16|                }
263|10|           }
264| 6|       }
265| 0| }
266| 0|  *****************************************************************************************
267| 1|  * select_var : select var ( next clear var ), if only one ref skip                      *
268| 1|  *****************************************************************************************
269| 1|  *
270| 0| void select_var( counttype *countarrptr )
271| 0| {
272| 0| int i;
273| 7|        for ( i = 0 ; i < countarrptr -> countnum ; i++)
274|12|             if ( countarrptr -> countarr[i].marked == BUSY ) countarrptr -> countarr[i].marked =
```

```
|    |  | DONE;
| 275| 3|
| 276| 7|        if ( countarrptr -> countarr[0].marked == CLEAR ) countarrptr -> countarr[0].marked = BUSY;
| 277| 7|        else
| 278|11|            {
| 279|11|            if ( countarrptr->countarr[countarrptr->busyvar].marked == BUSY )
| 280|14|               countarrptr->countarr[countarrptr->busyvar].marked = DONE;
| 281|11|            for ( i = 0 ; i < countarrptr -> countnum && countarrptr -> countarr[i].marked == DONE
|    |  | ; i++);
| 282|11|            if ( i != countarrptr -> countnum )
| 283|20|                     {
| 284|20|                     countarrptr -> countarr[i].marked = BUSY;
| 285|20|                     countarrptr -> busyvar = i;
| 286|20|                     if ( countarrptr -> countarr[i].spread < 2 &&  countarrptr -> countarr[i].
|    |  | count[MAIN] < 2) /* only one or none !*/
| 287|23|                        {
| 288|23|                        countarrptr -> countarr[i].marked = DONE; /* mark it as done */
| 289|23|                        select_var( countarrptr );      /* try next */
| 290|23|                        }
| 291|20|                     }
| 292|12|             else
| 293|20|                     allhad = TRUE;
| 294|12|             }
| 295| 0| }
| 296| 0| /****************************************************************************************
| 297| 1|  * newnode(int node1 , int node2 , expressionnode *outexp , countarrptr );                *
| 298| 1|  * kills two nodes ( 1 and 2 ) in table and replaces them with one ( only busy var entrance ) *
| 299| 1|  * we asume that node1 delivers the overal new operator and thus the new node number      *
| 300| 1|  ****************************************************************************************
| 301| 1|  */
| 302| 0| newnode( int node1 , int node2 , expressionnode *outexp , counttype *countarrptr)
| 303| 3|    {
| 304| 3|    int var;
| 305| 3|    for ( var = 0 ; var < countarrptr -> countnum ; var++)
| 306| 8|         {
| 307| 8|         countarrptr -> countarr[var].count[node1] = 0;
| 308| 8|         countarrptr -> countarr[var].count[node2] = 0;
| 309| 8|         }
| 310| 3|    countarrptr -> countarr[countarrptr->busyvar].count[node1]  = 1;
| 311| 3|    outexp -> nodenumber = node1; /* take over node number */
| 312| 0|
| 313| 3|    make_global_count( cntarr );
| 314| 0|
| 315| 3|    if ( countarrptr -> countarr[countarrptr->busyvar].spread < 2)
| 316| 6|       hit = TRUE;
| 317| 3|    }
| 318| 0|
| 319| 0| /****************************************************************************************
| 320| 1|  * decrease_var : gets one busy var of list, ( because of reduction ) but also kills another  *
| 321| 1|  * node : so if we decrease a var we loos 1 node ( set all to 0 ) , and we put only one var  *
| 322| 1|  * on the other one. e.g. z + (a*x*C1) + (b*x*C2) -> z + ((a*C1 + b*C2))*x               *
| 323| 1|  * we kill node 'a', 'b' wil be new x ( only x is present ), update globals              *
| 324| 1|  ****************************************************************************************
| 325| 1|  */
| 326| 0| int decrease_var( expressionnode *curexp , counttype *countarrptr )
| 327| 0| {
| 328| 0| int var;
| 329| 0| int busynode;
| 330| 5|      if ( curexp -> nodetype == DYADICOPERATOR )
| 331|11|            {
| 332|11|            busynode = curexp -> nodenumber;
| 333|11|            for ( var = 0 ; var < countarrptr -> countnum && countarrptr -> countarr[var].marked !=
|    |  |  BUSY ; var++);
| 334|11|            if ( var != countarrptr -> countnum ) fatalerror(" decrease var finds no BUSY var ");
| 335| 0|
| 336|11|            countarrptr -> countarr[var].count[busynode] --;
| 337|11|            countarrptr -> countarr[var].count[MAIN] --;
| 338|11|            if ( countarrptr -> countarr[var].count[busynode] == 0 ) /* more vars in current node *
|    |  | /
| 339|17|                  {
| 340|17|                  countarrptr -> countarr[var].count[busynode] --;
| 341|17|                  countarrptr -> countarr[var].marked = DONE;
| 342|17|                  }
| 343|11|            }
| 344| 0|
| 345| 0| }
```

```
346| 0|
347| 0| /******************************************************************************************
348| 1|  * getvarname: get main name out of an expression ( returns -> varname )                  *
349| 1|  ******************************************************************************************
350| 1|  */
351| 0|
352| 0| char *getvarname(expressionnode *curexp, int *notflag)
353| 0| {
354| 3|    while ( curexp -> nodetype == MONADICOPERATOR || curexp -> nodetype == DYADICOPERATOR )
355|16|                 {
356|16|                 if ( curexp -> nodetype == MONADICOPERATOR )
357|26|                          {
358|27|                           if ( curexp -> exp.monadicoperator.operator == ( operatortype ) NOTOP )
   |  |
359|79|                                                                                   *notflag = TRUE;
360|26|                          curexp = curexp -> exp.monadicoperator.subexpression;
361|26|                          }
362|16|                 if  ( curexp -> nodetype == DYADICOPERATOR )
363|26|                          { /* if right is const then take left way else right way */
364|26|                          if ( curexp -> exp.dyadicoperator.rightexpression -> nodetype ==
   |  | CONSTANT )
365|33|                                 curexp = curexp -> exp.dyadicoperator.leftexpression ;
366|26|                          else
367| 0|
368|33|                                 curexp = curexp -> exp.dyadicoperator.rightexpression ;
369|26|                          }
370|16|                 }
371| 0|
372| 3|    if ( curexp -> nodetype != VAR )
373| 8|         fatalerror(" getvarname didn't encounter var where it chould !");
374| 3|    else return curexp -> exp.var.refvar -> elemname;
375| 0| }
376| 0|
377| 0| /******************************************************************************************
378| 1|  * sortsubtree : sort the tree in a particular way : als the same vars next to each other  *
379| 1|  ******************************************************************************************/
380| 0|
381| 0| expressionnode *sortsubtree(expstat *curstat , counttype * countarrptr , int second )
382| 3|    {
383| 3|    expressionnode *curexpression = NULL;
384| 3|    expressionnode *auxexpression = NULL;
385| 3|    expressionnode *backexpression = NULL;
386| 3|    expressionnode *newrootexpression = NULL , *constexpression = NULL;
387| 3|    char *tmp1,*busyvarname;      /* pointers to names to compare them */
388| 3|    int i,j,notflag,busyvarcount, constsaved = FALSE , nodeno;
389| 3|    operatortype savedyadop = NULL;
390| 3|
391| 3|    curexpression = curstat -> exp1;
392| 3|    curstat -> exp1 = NULL;
393| 0|
394| 3|    /* sorted, so now we continue :we take all busy var to right */
395| 3|
396| 3|    /* test if var we want to sort is present else skip */
397| 3|    busyvarname = countarrptr -> countarr[countarrptr->busyvar].varname;
398| 3|    busyvarcount = countarrptr -> countarr[countarrptr->busyvar].count[curexpression -> nodenumber]
   |  | ;
399| 3|    nodeno = curexpression -> nodenumber;
400| 0|
401| 3|    /* there are situations where this number can not be found ( e.g (a < b ) and ( b < a ) */
402| 0|
403| 3|    /* selction : if no busy in current node or if local = global ?????????????????? */
404| 0|
405| 3|    if ( busyvarcount == 0 ) return curexpression;
406| 0|
407| 3|    /* save first operator to check if the tree is ended and there is another dyadop */
408| 3|    if ( curexpression -> nodetype == DYADICOPERATOR )
409| 8|         savedyadop = curexpression -> exp.dyadicoperator.operator;
410| 3|    else return curexpression;
411| 3|    /* only one var possible, check for const if returned */
412| 0|
413| 3|    /* handle const to put them in constexpression and append it at the end */
414| 3|    constsaved = FALSE; /* init */
415| 3|    if ( curexpression -> exp.dyadicoperator.rightexpression -> nodetype == CONSTANT )
416| 6|       {
417| 6|       constexpression = curexpression ; /* --> *---const */
418| 6|       curexpression = curexpression -> exp.dyadicoperator.leftexpression ; /* rest */
```

```
|419| 6|        constsaved = TRUE;
|420| 6|        }
|421| 2|
|422| 3|     /* check if this is also still part of legal tree */
|423| 3|     if ( curexpression -> nodetype == DYADICOPERATOR && savedyadop != curexpression -> exp.
|   |  | dyadicoperator.operator)
|424| 7|         if ( constsaved == TRUE )
|425|12|             return constexpression;
|426| 7|         else
|427| 9|           {
|428| 9|           curexpression -> nodenumber = nodeno;
|429| 9|           return curexpression;
|430| 9|           }
|431| 0|
|432| 3|     auxexpression = curexpression ;/* init */
|433| 0|
|434| 3|     while( auxexpression -> nodetype == DYADICOPERATOR && auxexpression -> exp.dyadicoperator.
|   |  | operator == savedyadop && busyvarcount > 0 )
|435|13|              {
|436|13|              notflag = FALSE;
|437|13|              notflag = FALSE;
|438|13|              if ( getvarname( auxexpression -> exp.dyadicoperator.rightexpression , &notflag )
|439|18|                   == countarrptr -> countarr[countarrptr->busyvar].varname &&
|440|18|                   notflag == countarrptr -> countarr[countarrptr -> busyvar].notflag )
|441|21|                      {
|442|21|                      busyvarcount--;
|443|21|                      if ( busyvarcount < 0 ) fatalerror (" trying to set more chars then there
|   |  | are(1a) ");
|444|21|                      if ( auxexpression == curexpression ) /* still first */
|445|24|                         {
|446|24|                         auxexpression = auxexpression -> dyadleftexp;
|447|24|                         curexpression -> dyadleftexp = newrootexpression;
|448|24|                         newrootexpression = curexpression;
|449|24|                         curexpression = auxexpression;
|450|24|                         }
|451|21|                      else /* somewhere in between */
|452|24|                         {
|453|24|                         backexpression = curexpression;
|454|24|                         while ( backexpression -> dyadleftexp != auxexpression )
|455|30|                               backexpression = backexpression -> dyadleftexp;
|456|24|                         backexpression -> dyadleftexp = auxexpression -> dyadleftexp;
|457|24|                         auxexpression -> dyadleftexp = newrootexpression;
|458|24|                         newrootexpression = auxexpression;
|459|24|                         auxexpression = backexpression;
|460|24|                         }
|461|21|                      }
|462|13|              else
|463|21|                      auxexpression = auxexpression -> dyadleftexp;
|464|13|              } /* while */
|465| 7|         /* we have to check last elem on left leaf of tree ( we only checked right ) */
|466| 0|
|467| 7|         if ( (busyvarcount > 0) ) /* must be one left */
|468|10|           {
|469|10|           notflag = FALSE ;
|470|10|           if ( getvarname( auxexpression , &notflag ) == countarrptr -> countarr[countarrptr->
|   |  | busyvar].varname &&
|471|15|                notflag == countarrptr -> countarr[countarrptr -> busyvar].notflag)
|472|14|               {
|473|14|               busyvarcount--;
|474|14|               if ( busyvarcount < 0 ) fatalerror(" trying to give more busy vars then there are (
|   |  | 1b)");
|475|14|               if ( auxexpression == curexpression )  * only curexp left *
|476|18|                   {
|477|18|                   if ( newrootexpression != NULL )
|478|22|                       {
|479|22|                       backexpression = newrootexpression;
|480|22|                       while ( backexpression->dyadleftexp != NULL )backexpression =
|   |  | backexpression->dyadleftexp;
|481|22|                       backexpression -> dyadleftexp = curexpression;
|482|22|                       }
|483|18|                   else newrootexpression = curexpression;
|484|18|                   }
|485|14|               else
|486|18|                   {
|487|18|                   backexpression = curexpression;
|488|18|                   while ( backexpression->dyadleftexp!=auxexpression )backexpression=
```

```
|    |  | backexpression->dyadleftexp;
| 489|18|                     backexpression -> dyadleftexp = backexpression -> dyadrightexp;
| 490|18|                     backexpression -> dyadrightexp  = auxexpression;
| 491|18|                     auxexpression = backexpression ;
| 492|18|                     if ( auxexpression == curexpression ) /* only curexp left */
| 493|22|                         {
| 494|22|                         if ( newrootexpression == NULL ) newrootexpression = curexpression;
| 495|22|                         else
| 496|25|                            {
| 497|25|                            backexpression = newrootexpression;
| 498|25|                            while ( backexpression->dyadleftexp != NULL )backexpression =
|    |  | backexpression->dyadleftexp;
| 499|25|                            backexpression -> dyadleftexp = curexpression;
| 500|25|                            }
| 501|22|                         }
| 502|18|                     else
| 503|22|                         {
| 504|22|                         backexpression = curexpression;
| 505|22|                         while ( backexpression->dyadleftexp!=auxexpression )backexpression=
|    |  | backexpression->dyadleftexp;
| 506|22|                         backexpression -> dyadleftexp = auxexpression -> dyadleftexp;
| 507|22|                         auxexpression -> dyadleftexp = newrootexpression;
| 508|22|                         newrootexpression = auxexpression;
| 509|22|                         backexpression = newrootexpression;
| 510|22|                         while ( backexpression->dyadleftexp != NULL )backexpression =
|    |  | backexpression->dyadleftexp;
| 511|22|                         backexpression -> dyadleftexp = curexpression;
| 512|22|                         }
| 513|18|                     }
| 514|14|                 }
| 515|10|             }
| 516| 7|       else
| 517|10|           {
| 518|10|           if ( newrootexpression == NULL ) printf( " NULL ! \n");
| 519|10|           if ( newrootexpression -> nodetype == DYADICOPERATOR )
| 520|14|               {
| 521|14|               backexpression = newrootexpression;
| 522|14|               while ( backexpression->dyadleftexp != NULL )
| 523|23|                        backexpression = backexpression->dyadleftexp;
| 524| 0|
| 525|14|               backexpression -> dyadleftexp = curexpression;
| 526|14|               }
| 527|10|           }
| 528| 0|
| 529| 3|    /* we want to set p and not p adjacent so we call subsort again */
| 530| 3|    /* this  only if curexp and newrootexp are not the same , else they are already adjacent */
| 531| 0|
| 532| 3|    if ( newrootexpression != curexpression && second == 0)
| 533| 7|        if ( newrootexpression->nodetype == DYADICOPERATOR &&
| 534|12|             newrootexpression->exp.dyadicoperator.operator == ANDOP)
| 535|15|                {
| 536|15|                int oldbusyvar,newbusyvar;
| 537|15|                expstat tmpcurstat;
| 538|15|                /* search new busyvar if any */
| 539|15|                oldbusyvar = countarrptr -> busyvar;
| 540|15|                for ( newbusyvar = oldbusyvar + 1 ;
| 541|24|                         countarrptr -> countarr[newbusyvar].varname != busyvarname &&
| 542|24|                         newbusyvar < countarrptr -> countnum;
| 543|21|                      newbusyvar++);
| 544|15|                if ( ! ( newbusyvar >= countarrptr -> countnum ||
| 545|15|                countarrptr -> countarr[newbusyvar].count[nodeno] == 0) ) /* not found */
| 546|18|                   {
| 547|18|                    /* now we mark oldbusy var as DONE ( temp ) and select newbusyvar as busy */
| 548|18|                   countarrptr -> countarr[oldbusyvar].marked = DONE;
| 549|18|                   countarrptr -> countarr[newbusyvar].marked = BUSY;
| 550|18|                   countarrptr -> busyvar = newbusyvar;
| 551|18|                   /* save pointer above curexp */
| 552|18|                   backexpression = newrootexpression;
| 553|18|                   while ( backexpression->dyadleftexp != curexpression )
| 554|50|                                                    backexpression = backexpression->dyadleftexp;
| 555| 0|
| 556|18|                   /* sort subtree with not operator exp */
| 557|18|                   tmpcurstat.exp1 = curexpression;
| 558|18|                   curexpression = sortsubtree(&tmpcurstat , countarrptr , 1 ); /* 1 = detect
|    |  | recursion */
| 559|18|                   backexpression->dyadleftexp = curexpression;
```

```
 560|17|
 561|18|                 /* reset to old settings */
 562|18|                 countarrptr -> busyvar = oldbusyvar;
 563|18|                 countarrptr -> countarr[oldbusyvar].marked = BUSY;
 564|18|                 countarrptr -> countarr[newbusyvar].marked = CLEAR;
 565| 0|
 566|18|                 }
 567|15|              }
 568| 0|
 569| 3|   /* if const save before we set it back */
 570| 3|   if ( constsaved == TRUE )
 571| 6|      {
 572| 6|      constexpression -> dyadleftexp = newrootexpression;
 573| 6|      newrootexpression = constexpression;
 574| 6|      }
 575| 0|
 576| 3|   if ( second != 1 )
 577| 6|      {
 578| 3|   if ( newrootexpression->nodetype == DYADICOPERATOR &&
 579| 8|        newrootexpression->exp.dyadicoperator.operator == ANDOP &&
 580| 8|        countarrptr -> countarr[countarrptr->busyvar].count[nodeno] > 0 )
 581|13|             {
 582|13|             decreasevar = 0; /* init */
 583|13|             if (debuglevel & 0x8) iosprintf(debugfile,"\n[BOOLAND  : ");
 584|13|             newrootexpression = rewriteexpression(newrootexpression,boolandrules);
 585|13|             decompileexpression(debugfile,newrootexpression);
 586|13|             iosprintf(debugfile,"\n");
 587|13|             /* update table */
 588|13|             if ( decreasevar > 0 )
 589|19|                   /* decrease_var holds number of busyvar are reduced */
 590|19|                   countarrptr -> countarr[countarrptr->busyvar].count[nodeno] -= decreasevar;
 591|19|                   make_global_count(countarrptr);
 592|13|             decreasevar = 0;
 593|13|             }
 594| 6|      }
 595| 0| if ( nodeno > MAXNODES || nodeno < 0 ) iosprintf (debugfile,"\n ERROR2 !!! : nodeno = %d \n",
    |  | nodeno);
 596| 4|     newrootexpression -> nodenumber = nodeno;
 597| 0|
 598| 0| if ( nodeno > MAXNODES || nodeno < 0 ) iosprintf (debugfile,"\n ERROR3 !!! : nodeno = %d \n",
    |  | nodeno);
 599| 0|
 600| 3|   return newrootexpression;
 601| 0| }
 602| 0|
 603| 0| /*************************************************************************/
 604| 0|
 605| 0| /* makes a constant with all zero/ones lengts same as most left expression in
 606| 3|    original exp1 , i is 0 for all zero's, 1 for all ones !!! */
 607| 0| expressionnode *regall(int i)
 608| 3|   {
 609| 3|   expressionnode *auxexpression = constructexpressionnode();
 610| 3|   if ( size != 0 )
 611| 7|       {
 612| 7|       auxexpression->sign = UNKNOWN;
 613| 7|       auxexpression->complement = FALSE;
 614| 7|       auxexpression->reduced = 0;
 615| 0|
 616| 7|       if ( i == 0 ) auxexpression->constval = reg_construct( size , ZERO , NOFORMAT );
 617| 7|       else auxexpression->constval = reg_construct( size , ONE , NOFORMAT );
 618| 0|
 619| 7|       auxexpression->restype = constructtype();
 620| 0|
 621| 7|       auxexpression->nodetype=CONSTANT;
 622| 7|       auxexpression->restype->type = REG;
 623| 7|       auxexpression->restype->typ.reg.size = size;
 624| 7|       auxexpression->restype->typ.reg.repform = NOFORMAT;
 625| 0|
 626| 7|       }
 627| 3|   else  fatalerror("regall(0/ 1) : not a reg type to construct ");
 628| 3|   return auxexpression;
 629| 3|   }
```

```
  1| 0| /*-----------------------------------------------------------------------
  2| 0|
  3| 3|    module name   : rulebase
  4| 3|    creation date : 90/06/19
  5| 3|    history       : from reduce expression, only rulebase to prevent
  6|19|                    recompilation of rulebase + files.
  7|19|                    16/07/91
  8|19|                    introduced ability to use different rulbases.
  9| 0|
 10| 0| 14/08/91 :  New made : -a --> a * -1 !! and all rules are now addapted !
 11| 0|
 12| 3|    module mady by Marcel Wijshoff , copied from reducexp.c !
 13| 0|
 14| 3|    purpose
 15| 0|
 16| 6|       the rulebase consists of two parts:
 17| 6|       * a rule-based rewrite system
 18| 6|       * a heuristic term combining system
 19| 0|
 20| 3|    NEW RULEBASE SYSTEM ! 16/07/91
 21| 3|    LAST UPDATE : 04/02/92.
 22| 0|
 23| 3|    for more see rulbase.h !
 24| 0|
 25| 0| _______________________________________________________________________*/
 26| 0|
 27| 0| #include "def.h"
 28| 0|
 29| 0| #define MOD_RULEBASE
 30| 0| #include "rulebase.h"
 31| 0|
 32| 0| /* MACRO for rucursive decent in tree, uses current rulebase */
 33| 0| #define RECURSIVE_REWRITE(A)  ASOP1,MONADOP,ANYOP,EXP1,MONADOP,OP1,REWRITE,(A),EXP1,END,ASOP1,
   |  | DYADOP,ANYOP,EXP1,\
 34|26|                              EXP2,DYADOP,OP1,REWRITE,(A),EXP1,REWRITE,(A),EXP2,END,ASOP1,TRIADOP,
   |  | ANYOP,EXP1,\
 35|26|                              EXP2,EXP3,TRIADOP,OP1,REWRITE,(A),EXP1,REWRITE,(A),EXP2,REWRITE,(A),
   |  | EXP3,END
 36| 0|  * 42 */
 37| 0|
 38| 0| #define EVALUATE_CONSTS   ASEXP1,MONADOP,ANYOP,ACONST,EVALUATE,EXP1,EXIT,END,ASEXP1,DYADOP,ANYOP,
   |  | ACONST,ACONST,\
 39|26|                              EVALUATE,EXP1,EXIT,END,ASEXP1,TRIADOP,ANYOP,ACONST,ACONST,ACONST,
   |  | EVALUATE,EXP1,EXIT,END
 40| 0|
 41| 0| #define REW REWRITE,SIMPLIFY,REWRITE,SORTSUBTREE,REWRITE,SORT
 42| 0| #define REWREG REWRITE,BOOLALG,REWRITE,SORTSUBTREE,REWRITE,SORT
 43| 0|
 44| 0|
 45| 0| rulebasetype masterrules[] =
 46| 3|    {
 47| 0|
 48| 3|    EXP1,RESET_FLAG,EXP1,END,
 49| 3|    /* reset flag used for recursion !*/
 50| 0|
 51| 7|        ASEXP1,REGEXP,REWRITE,
 52| 3|    EXPANDXOR, /* used to simplify xor expressions */
 53| 7|        EXP1,END,
 54| 0|
 55| 7|        ASEXP1,INTEXP,REWRITE,
 56| 3|    SHIFTMINUS,
 57| 7|        EXP1,END,
 58| 0|
 59| 7|        ASEXP1,REGEXP,REWRITE,
 60| 3|    SHIFTNOT,
 61| 7|        EXP1,END,
 62| 0|
 63| 7|        EXP1,PRINT,EXP1,END,
 64| 0|
 65| 7|        EXP1,REWRITE,
 66| 3|    EXPAND,
 67| 7|        EXP1,END,
 68| 0|
 69| 7|        EXP1,PRINT,EXP1,END,
 70| 0|
 71| 7|        EXP1,REWRITE,
```

```
  72| 3|    SORT,
  73| 7|        EXP1,END,
  74| 3|
  75| 7|        EXP1,PRINT,EXP1,END,
  76| 0|
  77| 3|    ASEXP1,ACONST,EXP1,EXIT,END,
  78| 0|
  79| 3|    /* instert const in and tree's to limit mutations */
  80| 7|        EXP1,REWRITE,
  81| 1|    INSERT_CONST,
  82| 7|        EXP1,END,
  83| 6|
  84| 3|    CREATE_TABLE,END,
  85| 0|
  86| 3|    /* make a count for sort sub tree */
  87| 3|    MAKE_VAR_COUNT,END,
  88| 0|
  89| 3|    PRINT_TABLE,END,
  90| 0|
  91| 7|        ASEXP1,IFNOTALL_HAD,
  92| 3|    REWRITE,SORTSUBTREE,
  93| 7|        EXP1,END,
  94| 3|
  95| 7|        EXP1,PRINT,EXP1,END,
  96| 0|
  97| 3|    INITHIT,END,
  98| 0|
  99| 7|        ASEXP1,REGEXP,REWRITE,
 100| 3|    BOOLALG,
 101| 7|        EXP1,END,
 102| 0|
 103| 3|    INITHIT,END,
 104| 0|
 105| 7|        ASEXP1,INTEXP,REWRITE,
 106| 3|    SIMPLIFY,
 107| 7|        EXP1,END,
 108| 0|
 109| 7|        EXP1,PRINT,EXP1,END,
 110| 0|
 111| 7|        ASEXP1,REGEXP,REWRITE,
 112| 3|    IMPLODEREG,
 113| 7|        EXP1,END,
 114| 0|
 115| 0|
 116| 3|    ASEXP1,IF_SET_FLAG,
 117| 3|    /* test if flag set anywhere which results in a rewrite */
 118| 3|    REWRITE,MASTER,EXP1,
 119| 3|    EXIT,END,
 120| 3|    /* call the whole bunch again , and EXIT !*/
 121| 0|
 122| 0| /*  dont need this ,but we re-use the name to
 123| 4|     make some small rules stick regarding 0 and 1 */
 124| 7|        ASEXP1,INTEXP,REWRITE,
 125| 3|    IMPLODEINT,
 126| 7|        EXP1,END,
 127| 0|
 128| 0|
 129| 7|        EXP1,PRINT,EXP1,END,
 130| 0|
 131| 7|        ASEXP1,REGEXP,
 132| 3|    REWRITE,IMPLODENOT,
 133| 3|    REWRITE,SORTNOT,
 134| 7|        EXP1,END,
 135| 0|
 136| 7|        ASEXP1,INTEXP,REWRITE,
 137| 3|    IMPLODEMINUS,
 138| 7|        EXP1,END,
 139| 0|
 140| 7|        EXP1,PRINT,EXP1,END,
 141| 0|
 142| 3|    KILL_TABLE,END,
 143| 0| /*********************************************************************************/
 144| 3|    END
 145| 3|    };
 146| 0|
 147| 0| rulebasetype dummy[]= {
```

```
148| 0| /**********************************************************************/
149|23|
150| 3|    END};
151| 0|
152| 0|
153| 0| /**************************************************************************************
154| 1|  *  rulebase shiftminus shifts a minus down to the operands and                      *
155| 1|  *  inserts constants / simple rules on comeback                                    *
156| 1|  *************************************************************************************/
157| 0|
158| 0| rulebasetype shiftminusrules[] =
159| 3|    {
160| 0|
161| 3|    /* - a -> a * -1 */
162| 3|    MONADOP,MINUSOP,
163| 9|          EXP1,
164| 3|    /* rewrite to */
165| 3|    DYADOP,MULTOP,
166|10|           EXP1,
167|10|           INTM1,
168| 3|    END,
169| 3|
170| 3|    /* a - b  -> a + ( b * -1) */
171| 3|    DYADOP,INTSUBOP,
172| 6|       EXP1,
173| 6|       EXP2,
174| 3|    /*Equivalent to:*/
175| 3|    DYADOP,INTADDOP,
176| 6|       EXP1,
177| 6|       DYADOP,MULTOP,
178| 9|          EXP2,
179| 9|          INTM1,
180| 3|    END,
181| 0|
182| 3|    RECURSIVE_REWRITE(CURRENT),
183| 3|
184| 3|    EVALUATE_CONSTS,
185| 0|
186| 3|    END
187| 3|    };
188| 0|
189| 0| /**********************************************************************
190| 1|  * rulebase EXPANDXOR                                                 *
191| 1|  **********************************************************************/
192| 0| rulebasetype expandxorrules[] =
193| 3|    {
194| 3|    DYADOP,XOROP,
195| 5|      EXP1,
196| 5|      EXP2,
197| 3|    /* to */
198| 3|    DYADOP,OROP,
199| 5|      DYADOP,ANDOP,
200| 7|        MONADOP,NOTOP,EXP1,
201| 7|        EXP2,
202| 5|      DYADOP,ANDOP,
203| 7|        EXP1,
204| 7|        MONADOP,NOTOP,EXP2,
205| 3|    END,
206| 0|
207| 3|    RECURSIVE_REWRITE(CURRENT),
208| 0|
209| 3|    END,
210| 3|    };
211| 0|
212| 0| /**********************************************************************
213| 1|  * rulebase SHIFTNOT                                                  *
214| 1|  * pushes 'not' in boolean tree down to operants                      *
215| 1|  **********************************************************************/
216| 0| rulebasetype shiftnotrules[] =
217| 3|    {
218| 3|    /* expanding xor's her at once !!! */
219| 3|    /* a xor b --> (a and !b) or (!a and b) *
220| 0|  * doesn't occur any more
221| 3|    DYADOP,XOROP,
222| 5|      EXP1,
223| 5|      EXP2,
```

```
 224| 0|
 225| 3|     DYADOP,OROP,
 226| 5|       DYADOP,ANDOP,
 227| 7|         EXP1,
 228| 7|         MONADOP,NOTOP,EXP2,
 229| 5|       DYADOP,ANDOP,
 230| 7|         MONADOP,NOTOP,EXP1,
 231| 7|         EXP2,
 232| 3|     END,
 233| 0| */
 234| 3|     /*   sort operands for monadic operators: REVERSE NOT a --> NOT REVERSE a */
 235| 3|     MONADOP,REVERSEOP,
 236| 6|        MONADOP,NOTOP,
 237| 9|           EXP1,
 238| 3|     /*Equivalent to:*/
 239| 3|     REWRITE,CURRENT,MONADOP,NOTOP,
 240| 6|        MONADOP,REVERSEOP,
 241| 9|           EXP1,
 242| 3|     END,
 243| 0| /* 12 */
 244| 0|
 245| 3|     /*   REVERSE REVERSE a --> a   */
 246| 3|     MONADOP,REVERSEOP,
 247| 6|        MONADOP,REVERSEOP,
 248| 9|           EXP1,
 249| 3|     /*Equivalent to:*/
 250| 3|     REWRITE,CURRENT,
 251| 3|     EXP1,
 252| 3|     END,
 253| 0| /* 20 */
 254| 0|
 255| 3|     /*   NOT NOT a --> a   */
 256| 3|     MONADOP,NOTOP,
 257| 6|        MONADOP,NOTOP,
 258| 9|           EXP1,
 259| 3|     /*Equivalent to:*/
 260| 3|     REWRITE,CURRENT,
 261| 3|     EXP1,
 262| 3|     END,
 263| 0| /* 28 */
 264| 0|
 265| 3|     /*   NOT (a relop b) --> a comprelop b   */
 266| 3|     MONADOP,NOTOP,
 267| 6|        ASOP1,DYADOP,RELOP,
 268| 9|           EXP1,
 269| 9|           EXP2,
 270| 3|     /*Equivalent to:*/
 271| 3|     DYADOP,COMPRELOP1,
 272| 6|        EXP1,
 273| 6|        EXP2,
 274| 3|     END,
 275| 0| /* 40 */
 276| 0|
 277| 0|
 278| 3|     /*   !(a AND b) --> !a OR  !b  */
 279| 3|     MONADOP,NOTOP,
 280| 6|        DYADOP,ANDOP,
 281| 9|           EXP1,
 282| 9|           EXP2,
 283| 3|     /*Equivalent to:*/
 284| 3|     DYADOP,OROP,
 285| 6|        MONADOP,NOTOP,EXP1,
 286| 6|        MONADOP,NOTOP,EXP2,
 287| 3|     END,
 288| 0| /* 55 */
 289| 0|
 290| 3|     /*   !(a OR b) --> !a AND !b   */
 291| 3|     MONADOP,NOTOP,
 292| 6|        DYADOP,OROP,
 293| 9|           EXP1,
 294| 9|           EXP2,
 295| 3|     /* Equivalent to */
 296| 3|     DYADOP,ANDOP,
 297| 6|        MONADOP,NOTOP,EXP1,
 298| 6|        MONADOP,NOTOP,EXP2,
 299| 3|     END,
```

```
300| 0| /* 70 */
301| 0|
302| 3|    RECURSIVE_REWRITE(CURRENT),
303| 0|
304| 0| /* on way back eval const */
305| 0|
306| 3|    EVALUATE_CONSTS,
307|31|
308| 3|    /*   sort operands for symmetrical dyadic operators:
309| 8|         const op a --> a op const   */
310| 3|    ASOP1,DYADOP,SYMOP,
311| 6|       CONST1,
312| 6|       EXP2.
313| 3|    /*Equivalent to:*/
314| 3|    DYADOP,OP1,
315| 6|       EXP2,
316| 6|       CONST1,
317| 3|    END,
318| 0|
319| 3|    /*   a and #1...1b --> a   */
320| 3|    DYADOP,ANDOP,
321| 6|       EXP1,
322| 6|       REGALLONES,
323| 3|    /*Equivalent to:*/
324| 3|    EXP1,
325| 3|    END,
326| 0|
327| 3|    /*   a and #0...0b --> #0...0b   */
328| 3|    DYADOP,ANDOP,
329| 6|       EXP,
330| 6|       ASEXP1,
331| 6|       REGALLZEROES,
332| 3|    /*Equivalent to:*/
333| 3|    EXP1,
334| 3|    EXIT,END,
335| 0|
336| 3|    /*   a or #1...1b --> #1...1b   */
337| 3|    DYADOP,OROP,
338| 6|       EXP,
339| 6|       ASEXP1,
340| 6|       REGALLONES,
341| 3|    /*Equivalent to:*/
342| 3|    EXP1,
343| 3|    EXIT,END,
344| 0|
345| 3|    /*   a or #0...0b --> a   */
346| 3|    DYADOP,OROP,
347| 6|       EXP1,
348| 6|       REGALLZEROES,
349| 3|    /*Equivalent to:*/
350| 3|    EXP1,
351| 3|    END,
352| 0|
353| 3|    END
354| 3|    };
355| 0|
356| 0|
357| 0| rulebasetype sortrule[] =
358| 3|    {
359| 0|
360| 0| /* sort left ( : exp ?  (c ? d) --> ( exp ? c) ? d  ( where ? is sym op )*
361| 3|    ASOP1,DYADOP,SYMOP,
362| 8|          EXP1,
363| 9|          DYADOP,ISOP1,
364|10|            EXP2,
365|10|            EXP3,
366| 3|    /* !! rewrite recursive to */
367| 3|    REWRITE,CURRENT,
368| 3|    DYADOP,OP1,
369| 6|       DYADOP,OP1,
370| 9|          EXP1,
371| 9|          EXP2,
372| 6|       EXP3
373| 4|     END,
374| 4|     END
375| 4|     };
```

```
376| 0|
377| 0| /*******************************************************************
378| 1|  * rulebase SORTRULES                                              *
379| 1|  * rules for sorting trees into left sorted position .             *
380| 1|  * we aply rules a long as it is nessasary on one node before      *
381| 1|  * we recurse into depth ! so after one node is frocessed          *
382| 1|  * on the right node there is no longer the same opperator         *
383| 1|  * as the root !.                                                  *
384| 1|  *******************************************************************/
385| 0|
386| 0| rulebasetype sortrules[] =
387| 3|    {
388| 3|
389| 3|    /* initial sortrule
390| 6|       sort left ! : exp ?  (c ? d) --> ( exp ? c) ? d  ( where ? is sym op )*/
391| 3|    ASOP1,DYADOP,SYMOP,
392| 8|         EXP1,
393| 8|         DYADOP,ISOP1,
394|10|           EXP2,
395|10|           EXP3,
396| 0| /* !! rewrite recursive to */
397| 0| REWRITE,SORTRULE,
398| 3|    DYADOP,OP1,
399| 6|       DYADOP,OP1,
400| 9|          EXP1,
401| 9|          EXP2,
402| 6|       EXP3,
403| 4|     END,
404| 0|
405| 3|    RECURSIVE_REWRITE(CURRENT),
406| 0|
407| 3|    /* evaluate dyadop with constantes */
408| 3|    ASEXP1,DYADOP,ANYOP,ACONST,ACONST,EVALUATE,EXP1,EXIT,END,
409| 0|
410| 3|    /*   sort operands for symmetrical dyadic operators:
411| 8|         const op a --> a op const   */
412| 3|    ASOP1,DYADOP,SYMOP,
413| 6|       CONST1,
414| 6|       EXP2,
415| 3|    /*Equivalent to:*/
416| 3|    DYADOP,OP1,
417| 6|       EXP2,
418| 6|       CONST1,
419| 3|    END,
420| 0|
421| 3|    /* (a ? const1) ? const2 --> a ? (const3 = const1 ? const 2) , ? is symop */
422| 3|    ASOP1,DYADOP,SYMOP,
423| 6|       DYADOP,ISOP1,
424| 8|         EXP1,CONST1,
425| 6|       CONST2,
426| 3|    /* rewrite to */
427| 3|    DYADOP,OP1,
428| 6|       EXP1,
429| 6|       EVALUATE,DYADOP,OP1,
430| 8|         CONST1,
431| 8|         CONST2,
432| 3|    END,
433| 0|
434| 3|    /* ( exp1 ? c ) ? exp2 --> (exp1 ? exp2) ? c */
435| 3|    ASOP1,DYADOP,SYMOP,
436| 6|       DYADOP,ISOP1,
437|11|            EXP1,
438|11|            CONST1,
439| 6|       EXP2,
440| 3|    /* equ */
441| 3|    DYADOP,OP1,
442| 6|       DYADOP,OP1,
443|11|            EXP1,
444|11|            EXP2,
445| 6|       CONST1,
446| 3|    END,
447| 0|
448| 3|    /* const relop exp --> exp revrelop const */
449| 3|    ASOP1,DYADOP,RELOP,
450| 6|       CONST1,
451| 6|       EXP1.
```

```
452| 3|    /* to *
453| 3|    DYADOP,REVRELOP1,
454| 6|       EXP1,
455| 6|       CONST1,
456| 3|    END,
457| 0|
458| 3|    END
459| 3|    };
460| 2|
461| 0| /*****************************************************************
462| 1|  * rulebase EXPANDRULES                                          *
463| 1|  * rules only to expand tree to a canonical form :              *
464| 1|  *       (a*b*...) + (a*b*.. ) + .... ( sum of products )       *
465| 1|  *****************************************************************/
466| 0| rulebasetype expandrules[] = {
467| 0|
468| 4|     RECURSIVE_REWRITE(CURRENT),
469| 0|
470| 3|    /* do not swap !; else two rules are needed */
471| 3|    /* a*(b+c) -> a*b+c*a   EXPAND              */
472| 3|    DYADOP,MULTOP,
473| 7|        EXP1,
474| 7|        DYADOP,INTADDOP,
475|10|           EXP2,
476|10|           EXP3,
477| 3|    /* TO */
478| 3|    DYADOP,INTADDOP,
479| 5|      REWRITE,EXPANDINT,
480| 7|        DYADOP,MULTOP,
481|10|           EXP1,
482|10|           EXP2,
483| 5|      REWRITE,EXPANDINT,
484| 7|        DYADOP,MULTOP,
485|10|           EXP1,
486|10|           EXP3,
487| 4|     END,
488| 3|
489| 0|
490| 3|    /* (a+b)*c -> a*c + b*c  EXPAND */
491| 3|    DYADOP,MULTOP,
492| 7|        DYADOP,INTADDOP,
493|10|           EXP1,
494|10|           EXP2,
495| 7|        EXP3,
496| 3|    /* TO */
497| 3|    DYADOP,INTADDOP,
498| 5|      REWRITE,EXPANDINT,
499| 7|        DYADOP,MULTOP,
500|11|            EXP1,
501|11|            EXP3,
502| 5|      REWRITE,EXPANDINT,
503| 7|        DYADOP,MULTOP,
504|11|            EXP2,
505|11|            EXP3,
506| 4|     END,
507|12|
508| 4|     /* a and ( b or c ) --> a and b or a and c == EXPAND bool*/
509| 4|     DYADOP,ANDOP,
510| 7|        EXP1,
511| 7|        DYADOP,OROP,
512|10|           EXP2,
513|10|           EXP3,
514| 4|     /* to */
515| 4|     DYADOP,OROP,
516| 5|      REWRITE,EXPANDREG,
517| 7|        DYADOP,ANDOP,
518|10|           EXP1,
519|10|           EXP2,
520| 5|      REWRITE,EXPANDREG,
521| 7|        DYADOP,ANDOP,
522|10|           EXP1,
523|10|           EXP3,
524| 4|     END,
525| 0|
526| 4|     /* ( a or b ) and c --> a and c or b and c == EXPAND bool */
527| 4|     DYADOP,ANDOP,
```

```
| 528| 7|         DYADOP,OROP,
| 529|10|            EXP1,
| 530|10|            EXP2,
| 531| 7|         EXP3,
| 532| 4|      /* to */
| 533| 4|      DYADOP,OROP,
| 534| 5|       REWRITE,EXPANDREG,
| 535| 7|         DYADOP,ANDOP,
| 536|10|            EXP1,
| 537|10|            EXP3,
| 538| 5|       REWRITE,EXPANDREG,
| 539| 7|         DYADOP,ANDOP,
| 540|10|            EXP2,
| 541|10|            EXP3,
| 542| 4|      END,
| 543| 0|
| 544| 4|      EVALUATE_CONSTS,
| 545| 0|
| 546| 4|      END };
| 547| 0|
| 548| 0|/*************************************************************************
| 549| 1| * rulebase EXPANDINTRULES                                              *
| 550| 1| * ints only                                                            *
| 551| 1| ************************************************************************/
| 552| 0|rulebasetype expandintrules[] = {
| 553| 0|
| 554| 3|     /* do not swap !; else two rules are needed */
| 555| 3|     /* a*(b+c) -> a*b+c*a   EXPANDINT           */
| 556| 3|     DYADOP,MULTOP,
| 557| 7|         EXP1,
| 558| 7|         DYADOP,INTADDOP,
| 559|10|            EXP2,
| 560|10|            EXP3,
| 561| 3|     /* TO */
| 562| 3|     DYADOP,INTADDOP,
| 563| 5|       REWRITE,CURRENT,
| 564| 7|         DYADOP,MULTOP,
| 565|10|            EXP1,
| 566|10|            EXP2,
| 567| 5|       REWRITE,CURRENT,
| 568| 7|         DYADOP,MULTOP,
| 569|10|            EXP1,
| 570|10|            EXP3,
| 571| 4|      END,
| 572| 3|
| 573| 0|
| 574| 3|     /* (a+b)*c -> a*c + b*c  EXPAND INT*/
| 575| 3|     DYADOP,MULTOP,
| 576| 7|         DYADOP,INTADDOP,
| 577|10|            EXP1,
| 578|10|            EXP2,
| 579| 7|         EXP3,
| 580| 3|     /* TO */
| 581| 3|     DYADOP,INTADDOP,
| 582| 5|       REWRITE,CURRENT,
| 583| 7|         DYADOP,MULTOP,
| 584|11|             EXP1,
| 585|11|             EXP3,
| 586| 5|       REWRITE,CURRENT,
| 587| 7|         DYADOP,MULTOP,
| 588|11|             EXP2,
| 589|11|             EXP3,
| 590| 4|      END,
| 591| 4|      END };
| 592| 0|
| 593| 0|/*************************************************************************
| 594| 1| * rulebase EXPANDREGRULES                                              *
| 595| 1| * REGS only                                                            *
| 596| 1| ************************************************************************
| 597| 0|rulebasetype expandregrules[] = {
| 598| 0|
| 599| 4|      /* a and ( b or c ) --> a and b or a and c == EXPAND bool*/
| 600| 4|      DYADOP,ANDOP,
| 601| 7|         EXP1,
| 602| 7|         DYADOP,OROP,
| 603|10|            EXP2,
```

```
| 604|10|          EXP3,
| 605| 4|     /* to */
| 606| 4|     DYADOP,OROP,
| 607| 5|      REWRITE,CURRENT,
| 608| 7|        DYADOP,ANDOP,
| 609|10|          EXP1,
| 610|10|          EXP2,
| 611| 5|      REWRITE,CURRENT,
| 612| 7|        DYADOP,ANDOP,
| 613|10|          EXP1,
| 614|10|          EXP3,
| 615| 4|     END,
| 616| 0|
| 617| 4|     /* ( a or b ) and c --> a and c or b and c == EXPAND bool */
| 618| 4|     DYADOP,ANDOP,
| 619| 7|        DYADOP,OROP,
| 620|10|          EXP1,
| 621|10|          EXP2,
| 622| 7|        EXP3,
| 623| 4|     /* to */
| 624| 4|     DYADOP,OROP,
| 625| 5|      REWRITE,CURRENT,
| 626| 7|        DYADOP,ANDOP,
| 627|10|          EXP1,
| 628|10|          EXP3,
| 629| 5|      REWRITE,CURRENT,
| 630| 7|        DYADOP,ANDOP,
| 631|10|          EXP2,
| 632|10|          EXP3,
| 633| 4|     END,
| 634| 0|
| 635| 4|     END };
| 636| 0|
| 637| 0| /**************************************************************
| 638| 1|  * rulebase INSERT_CONST                                       *
| 639| 1|  * inserts constantes in and tree 's                           *
| 640| 1|  **************************************************************/
| 641| 0| rulebasetype insert_construles[] = {
| 642| 0| /* walk down a OR tree left sides and if the first right factor of
| 643| 3|    tree underlying AND tree is not a const then inser one :
| 644| 3|    a --> a and 111..
| 645| 0| */
| 646| 0|
| 647| 0| /* register */
| 648| 3|    /* a or b --> REW a or check if b has to be added with 1 */
| 649| 3|    DYADOP,OROP,
| 650| 5|      EXP1,
| 651| 5|      EXP2,
| 652| 3|    /* to */
| 653| 3|    DYADOP,OROP,
| 654| 5|      REWRITE,CURRENT,EXP1,
| 655| 5|      REWRITE,CHECK_CONST,EXP2,
| 656| 3|    EXIT,
| 657| 3|    END,
| 658| 0|
| 659| 0| /* integer */
| 660| 3|    /* a + b --> REW a , check b */
| 661| 3|    DYADOP,INTADDOP,
| 662| 5|      EXP1,
| 663| 5|      EXP2,
| 664| 3|    /* to */
| 665| 3|    DYADOP,INTADDOP,
| 666| 5|      REWRITE,CURRENT,EXP1,
| 667| 5|      REWRITE,CHECK_CONST,EXP2,
| 668| 3|    EXIT,
| 669| 3|    END,
| 670| 0|
| 671| 0| /* both */
| 672| 3|     * testing last one */
| 673| 3|    EXP1,
| 674| 3|    /* to */
| 675| 3|    REWRITE,CHECK_CONST,EXP1,
| 676| 3|    EXIT,
| 677| 3|    END,
| 678| 0|
| 679| 3|    END } ;
```

```
| 680| 0|
| 681| 0| /**************************************************************
| 682| 1|  * rulebase CHECK_CONST                                       *
| 683| 1|  **************************************************************/
| 684| 0| rulebasetype check_construles[] = {
| 685| 4|
| 686| 0| /* both register and integer */
| 687| 4|     /* if constante : exit at once */
| 688| 4|     ASEXP1,ACONST,
| 689| 4|     /* TO */
| 690| 4|     EXP1,
| 691| 4|     EXIT,
| 692| 4|     END,
| 693| 0|
| 694| 0| /* register */
| 695| 4|     /* a and const --> a and const --> quit */
| 696| 4|     ASEXP1,
| 697| 4|     DYADOP,ANDOP,
| 698| 7|        EXP,
| 699| 7|        ACONST,
| 700| 4|     /* TO */
| 701| 4|     EXP1,EXIT, /* quit without doing anything */
| 702| 4|     END,
| 703| 0|
| 704| 0| /* integer */
| 705| 4|     /* a * const --> a * const -> quit */
| 706| 4|     ASEXP1,
| 707| 4|     DYADOP,MULTOP,
| 708| 7|        EXP,
| 709| 7|        ACONST,
| 710| 4|     /* to */
| 711| 4|     EXP1,EXIT, /* and exit */
| 712| 4|     END,
| 713| 8|
| 714| 0| /* register */
| 715| 4|     ASEXP1,SAVESIZE,REGEXP, /* could cause errormessage in
| 716| 11|                               debug file if not regexp */
| 717| 4|     /* to */
| 718| 4|     DYADOP,ANDOP,
| 719| 7|        EXP1,
| 720| 7|        REGALLONES,
| 721| 4|     EXIT,   /* quick exit */
| 722| 4|     END,
| 723| 0|
| 724| 0| /* interget is all thats left ! */
| 725| 4|     EXP1,
| 726| 4|     /* to */
| 727| 4|     DYADOP,MULTOP,
| 728| 7|        EXP1,
| 729| 7|        INT1,
| 730| 4|     END,
| 731| 0|
| 732| 4|     END};
| 733| 0|
| 734| 0| /**************************************************************
| 735| 1|  * rulebase BOOLAND                                           *
| 736| 1|  * expects only and tree  called in last part of subsort      *
| 737| 1|  * doesn't decent tree ! , only if hit                        *
| 738| 1|  **************************************************************/
| 739| 0| rulebasetype boolandrules[] = {
| 740| 1|
| 741| 0| /*
| 742| 0| 10      !a and  a --> 00...
| 743| 0| 11       a  and !a --> 00...
| 744| 0|
| 745| 0| 13    ( x and  a ) and          !a   --> 00...
| 746| 0| 15    ( x and !a ) and           a   --> 00...
| 747| 1|
| 748| 0| 16             a   and           a   -->         a
| 749| 0| 17    ( x and  a ) and           a   --> x and  a
| 750| 0| *
| 751| 0|
| 752| 3|    PUSHNU,END, /* save current node number */
| 753| 0|
| 754| 3|    RECURSIVE_REWRITE(CURRENT),
| 755| 5|
```

```
| 756| 3|    EVALUATE_CONSTS,
| 757| 0|
| 758| 3|    /* CONST and A -> A and CONST */
| 759| 3|    ASNODE1,
| 760| 3|    DYADOP,ANDOP,
| 761| 6|       CONST1,
| 762| 6|       EXP1,
| 763| 3|    /* TO */
| 764| 3|    PUTNODE1,
| 765| 3|    DYADOP,ANDOP,
| 766| 5|      EXP1,
| 767| 5|      CONST1,
| 768| 3|    END,
| 769| 0|
| 770| 3|    /* ( A and CONST ) and B -> ( A and B ) and CONST */
| 771| 3|    ASNODE1,
| 772| 3|    DYADOP,ANDOP,
| 773| 5|      DYADOP,ANDOP,
| 774| 7|        EXP1,
| 775| 7|        CONST1,
| 776| 5|      EXP2,
| 777| 3|    /* TO */
| 778| 3|    PUTNODE1,
| 779| 3|    DYADOP,ANDOP,
| 780| 5|      PUTNODE1,
| 781| 5|      DYADOP,ANDOP,
| 782| 7|        EXP1,
| 783| 7|        EXP2,
| 784| 5|      CONST1,
| 785| 3|    END,
| 786| 0|
| 787| 3|    /* ( z and a ) and a --> z and a */
| 788| 3|    ASNODE1,
| 789| 3|    DYADOP,ANDOP,
| 790| 6|       DYADOP,ANDOP,
| 791| 9|          EXP2,
| 792| 9|          EXP1,
| 793| 6|       RESET_NOTFLAG, /* init for matching vars */
| 794| 6|       ISVAR1,
| 795| 3|    /* TO */
| 796| 3|    DECREASE_VAR,
| 797| 3|    REWRITE,CURRENT,
| 798| 6|       PUTNODE1,
| 799| 6|       DYADOP,ANDOP,
| 800| 9|          EXP2,
| 801| 9|          EXP1,
| 802| 3|    END,
| 803| 3|
| 804| 3|    /* ( z and  a ) and !a   --> 00... */
| 805| 3|    ASKILLNODE, /* save nodeno to update table */
| 806| 3|    DYADOP,ANDOP,
| 807| 6|       DYADOP,ANDOP,
| 808|10|           EXP,
| 809|10|           EXP1,
| 810| 6|       RESET_NOTFLAG, /* init for matching vars */
| 811| 6|       MONADOP,NOTOP,SAVESIZE,ISVAR1,
| 812| 3|    /* to */
| 813| 3|    KILL_NODE, /* reset varcount in table */
| 814| 3|    SET_FLAG,  /* flag set to use master again */
| 815| 6|       REGALLZEROES,
| 816| 3|    EXIT,
| 817| 3|    END,
| 818| 0|
| 819| 3|    /* ( x and !a ) and a   --> 00.. */
| 820| 3|    ASKILLNODE,
| 821| 3|    DYADOP,ANDOP,
| 822| 6|       DYADOP,ANDOP,
| 823|10|           EXP,
| 824|10|           MONADOP,NOTOP,EXP1,
| 825| 6|       SET_NOTFLAG, /* init for matching vars */
| 826| 6|       SAVESIZE,
| 827| 6|       ISVAR1,
| 828| 3|    /* to */
| 829| 3|    KILL_NODE,
| 830| 3|    SET_FLAG,  /* flag set to use master again */
| 831| 6|       REGALLZEROES,
```

```
| 832| 3|    EXIT,
| 833| 3|    END,
| 834| 0|
| 835| 3|    /* !a and a --> 00...   */
| 836| 3|    ASKILLNODE,
| 837| 3|    DYADOP,ANDOP,
| 838| 6|       MONADOP,NOTOP,EXP1,
| 839| 6|       SET_NOTFLAG, /* init for matching vars */
| 840| 6|       SAVESIZE,ISVAR1,
| 841| 3|    /* to:*/
| 842| 3|    KILL_NODE,
| 843| 3|    SET_FLAG,  /* flag set to use master again */
| 844| 6|       REGALLZEROES,
| 845| 3|    EXIT,
| 846| 3|    END,
| 847| 0|
| 848| 3|    /*  a and !a --> 00...   */
| 849| 3|    ASKILLNODE,
| 850| 3|    DYADOP,ANDOP,
| 851| 6|       EXP1,
| 852| 6|       SET_NOTFLAG, /* init for matching vars */
| 853| 6|       MONADOP,NOTOP,
| 854| 8|         SAVESIZE,ISVAR1,
| 855| 3|    /*Equivalent to:*/
| 856| 3|    KILL_NODE,
| 857| 3|    SET_FLAG,  /* flag set to use master again */
| 858| 6|       REGALLZEROES,
| 859| 3|    END,
| 860| 0|
| 861| 0|
| 862| 3|    /*  a AND a --> a   */
| 863| 3|    ASNODE1,
| 864| 3|    DYADOP,ANDOP,
| 865| 6|       EXP1,
| 866| 6|       RESET_NOTFLAG, /* init for matching vars */
| 867| 6|       ISVAR1,
| 868| 3|    /*Equivalent to:*/
| 869| 3|    DECREASE_VAR,
| 870| 3|    PUTNODE1,
| 871| 6|       EXP1,
| 872| 3|    END,
| 873| 3|
| 874| 3|    POPNN,END,
| 875| 0|
| 876| 0| EXP1,PRINT,EXP1,END,
| 877| 0|
| 878| 3|    END };
| 879| 0|
| 880| 0|
| 881| 0|
| 882| 0| /*****************************************************************
| 883| 1|  * rulebase BOOLLOOP                                             *
| 884| 1|  *****************************************************************/
| 885| 0| rulebasetype boollooprules[] = {
| 886| 0|
| 887| 3|    EXP1,PRINT,EXP1,END,
| 888| 0|
| 889| 3|    ASEXP1,IFALL_HAD,EXP1,EXIT,END,
| 890| 4|
| 891| 3|    /* 4 cases : z or ( ..x.. and y(=ready) and C ) -> rew z ( new var ) and rew x ( new table )
| 892|16|                z or (           y(=ready) and C ) -> rew z ( new var )
| 893|21|                     ( ..x.. and y(=ready) and C ) -> rew x ( new table )
| 894|21|                     (           y(=ready) and C ) -> exit       */
| 895| 0|
| 896| 3|    /* z or ( ..x.. and y and C ) --> Rew z or ( Rew x and y and C) */
| 897| 3|    ASNODE1
| 898| 3|    DYADOP,OROP,
| 899| 6|       EXP1,
| 900| 6|       DYADOP,ANDOP, /* number don't care because of rebuild anyway */
| 901| 8|         DYADOP,ANDOP,
| 902|11|            EXP2,
| 903|11|            EXP3,
| 904| 8|         ASCONST1,IFHIT, /* new busy var selected, hit flag set ! */
| 905| 3|    /* to */
| 906| 3|    PUTNODE1,
| 907| 3|    DYADOP,OROP,
```

```
908| 6|       REWRITE,BOOLALG,REWRITE,SORTSUBTREE,EXP1,
909| 6|       DYADOP,ANDOP,
910| 8|         DYADOP,ANDOP,
911| 9|          REWRITE,INITBOOL,EXP2, /* sort , subsort .. */
912| 9|          EXP3,       /* ready so leave it  */
913| 6|       CONST1,
914| 3|    END,
915| 0|
916| 3|    /* z or ( y and C ) --> Rew z or ( y and C) */
917| 3|    ASNODE1,
918| 3|    DYADOP,OROP,
919| 6|       EXP1,
920| 6|       DYADOP,ANDOP,
921|11|            EXP3,
922|11|            ASCONST1,IFHIT, /* new busy var selected, hit flag set ! */
923| 3|    /* to */
924| 3|    PUTNODE1,
925| 3|    DYADOP,OROP,
926| 6|       REWRITE,BOOLALG,REWRITE,SORTSUBTREE,EXP1,
927| 8|         DYADOP,ANDOP,
928| 9|          EXP3,       /* ready so leave it  */
929| 9|          CONST1,
930| 3|    END,
931| 0|
932| 3|    /* ( ...x... and y and C ) --> ( Rew x and y and C ) */
933| 3|    DYADOP,ANDOP,
934| 5|      DYADOP,ANDOP,
935| 7|        EXP1,
936| 7|        EXP2,
937| 5|      ASCONST1,IFHIT, /* new busy var selected , hit flag set !*/
938| 3|    /* to *
939| 3|    DYADOP,ANDOP,
940| 6|       DYADOP,ANDOP,
941| 9|          REWRITE,INITBOOL,EXP1, /* sort , subsort .. */
942| 9|          EXP2,       /* ready so leave it  */
943| 6|       CONST1,
944| 3|    END,
945| 0|
946| 3|    ASEXP1,IFALL_HAD,EXP1,EXIT,END,
947| 0|
948| 3|    ASEXP1,IFHIT,REWRITE,LOCALBOOL,EXP1,EXIT,END,
949| 0|
950| 3|    ASEXP1,IFALL_HAD,EXP1,EXIT,END,
951| 0|
952| 3|    EXP1,REWRITE,BOOLALG,EXP1,END,
953| 0|
954| 3|    END };
955| 0|
956| 0| /*********************************************************************
957| 1|  * rulebasetype LOCALBOOL                                            *
958| 1|  *********************************************************************/
959| 0| rulebasetype localboolrules[] =
960| 3|    {
961| 3|    EXP1,PRINT,EXP1,END,
962| 0|
963| 3|    SELECT_VAR,END,
964| 1|
965| 3|    ASEXP1,IFALL_HAD,EXP1,EXIT,END,
966| 0|
967| 7|        EXP1,
968| 3|    REWRITE,SORT,
969| 7|        EXP1,END,
970| 0|
971| 7|        EXP1,
972| 3|    REWRITE,SORTSUBTREE,
973| 7|        EXP1,END,
974| 0|
975| 7|        ASEXP1,INTEXP,REWRITE,
976| 3|    BOOLALG,
977| 7|        EXP1,END,
978| 0|
979| 3|    END };
980| 0|
981| 0| /*********************************************************************
982| 1|  * rulebasetype INITBOOL                                             *
983| 1|  *********************************************************************/
```

```
| 984| 0| rulebasetype initboolrules[] =
| 985| 3|    {
| 986| 0|
| 987| 3|    EXP1,PRINT,EXP1,END,
| 988| 0|
| 989| 3|    EXP1,REWRITE,EXPAND,EXP1,END,
| 990| 0|
| 991| 3|    EXP1,REWRITE,SORT,EXP1,END,
| 992| 0|
| 993| 0| /*
| 994| 7|        ASEXP1,REGEXP,REWRITE,
| 995| 3|    INSERT_CONST,
| 996| 7|        EXP1,END,
| 997| 0| */
| 998| 0|
| 999| 3|    MAKE_VAR_COUNT,END,
|1000| 1|
|1001| 3|    PRINT_TABLE,END,
|1002| 2|
|1003| 3|    EXP1,PRINT,EXP1,END,
|1004| 0|
|1005| 7|        ASEXP1,IFNOTALL_HAD,
|1006| 3|    REWRITE,SORTSUBTREE,
|1007| 7|        EXP1,END,
|1008| 3|
|1009| 7|        EXP1,REWRITE,
|1010| 3|    BOOLALG,
|1011| 7|        EXP1,END,
|1012| 0|
|1013| 3|    END };
|1014| 0|
|1015| 0| /**************************************************************
|1016| 1|  * rulebase BOOLALGRULES                                      *
|1017| 1|  * special rulebase for boolalgebra                           *
|1018| 1|  **************************************************************
|1019| 0| */
|1020| 0|
|1021| 0| rulebasetype boolalgrules[] = {
|1022| 3|
|1023| 3|    INITHIT,END,
|1024| 0|
|1025| 3|    EXP1,PRINT,EXP1,END,
|1026| 0|
|1027| 0| #include "relop.inc"
|1028| 0|
|1029| 0| /*
|1030| 0|
|1031| 0| First we introduce a const to every and node if non present
|1032| 0| and assume this is done in the master routine : insertconst
|1033| 0| In this way we aviod, the nonsense rules and mutations,
|1034| 0| We reduce the number of mutations by 4 !!!! And the !a /a combination
|1035| 0| is now a specail case !
|1036| 0|
|1037| 0| Main two rules now :
|1038| 0|
|1039| 0| I :        ( b and a and c1 ) or ( c and a and c2 ) -->       ( ( b and c1 ) or ( c and c2 ) ) and a
|1040| 0| II:  d or ( b and a and c1 ) or ( c and a and c2 ) --> d or ( ( b and c1 ) or ( c and c2 ) ) and a
|1041| 0| We only have mutaions conserning the b's and c's so 4 mutions of these 2 rules makes 8 rules :
|1042| 0|
|1043| 0| ****************************************************************************************************
|    |  | ***************
|1044| 0| not e: All regs may be of lenght x so :: var a,x,y,C1,C2 : reg[x] ;
|1045| 0|
|1046| 0| 40          ( x and  a and C1 ) or ( y and a and C2 ) -->       ((( x and C1 ) or ( y and C2 )) and
|    |  | a and 11.. )
|1047| 0| 41          (        a and C1 ) or ( y and a and C2 ) -->       (((       C1 ) or ( y and C2 )) and
|    |  | a and 11.. )
|1048|63|                                                                  \______________________________
|    |  | _______
|1049|62|                                                                 ( ( y or C1 ) and ( C1 or C2 ) )
|    |  | and a and 11
|1050|63|                                                                  \______________________________
|    |  | __'
|1051|62|                                                                 ( ( y or C1 ) and a EVAL :( C1 or
|    |  | C2 ) )
|1052|63|                                                                  \______________________/
```

```
|1053|62|                                                         ( ( y or C1 ) and a and C3
|1054| 0|
|1055| 0| 42        ( x and  a and C1 ) or (       a and C2 ) -->     {(( x and C1 ) or (      C2 )) and
|    |  | a and 11.. }
|1056|63|                                                          as above !!!!!
|1057|62|                                                         ( ( x or C2 ) and a and C3
|1058| 0|
|1059| 0| 43        (        a and C1 ) or (       a and C2 ) -->     {((      C1 ) or (      C2 )) and
|    |  | a and 11.. }
|1060|63|                                                          \________________________.
|1061|66|                                                             evaluat this and place it left
|1062|65|                                                            a and ( eval : C1 or C2 )
|1063| 0|
|1064| 0| 50    z or ( x and  a and C1 ) or ( y and a and C2 ) --> z or {(( x and C1 ) or ( y and C2 )) and
|    |  | a and 11.. }
|1065| 0| 51    z or (        a and C1 ) or ( y and a and C2 ) --> z or {((      C1 ) or ( y and C2 )) and
|    |  | a and 11.. }
|1066| 0| 52    z or ( x and  a and C1 ) or (       a and C2 ) --> z or {(( x and C1 ) or (      C2 )) and
|    |  | a and 11.. }
|1067| 0| 53    z or (        a and C1 ) or (       a and C2 ) --> z or {((      C1 ) or (      C2 )) and
|    |  | a and 11.. }
|1068|63|                                                          \________________________/
|1069|66|                                                             evaluat this and place it left
|1070|57|                                                     z or (  a and ( eval : C1 or C2 ))
|1071| 0|
|1072| 0| *********************************************************************************************
|    |  | *************
|1073| 0| Some specall cases :
|1074| 0|
|1075| 0| 45        (        !a        ) or ( y and a and C2 ) -->     {((    !a    ) or ( y and C2 ))}
|    |  |
|1076| 0| 47        (        !a        ) or (       a and C2 ) -->     {((    !a    ) or (       C2 ))}
|1077| 0|
|1078| 0| 50        ( x and  !a and C1 ) or (       a        ) -->     {(( x and C1 ) or (    a     ))}
|1079| 0| 51        (        !a and C1 ) or (       a        ) -->     {((      C1 ) or (    a     ))}
|1080| 0|
|1081| 0| 53        (        !a        ) or ( y and a        ) -->     {((    !a    ) or ( y        ))}
|1082| 0| 54        ( x and  !a        ) or (       a        ) -->     {(( x        ) or (    !a    ))}
|1083| 0| 55        (        !a        ) or (       a        ) -->     {((    1     ) or (    !a    ))}
|    |  |       ->       1
|1084| 0|
|1085| 0| 61    z or (        !a        ) or ( y and a and C2 ) --> z or {((    !a    ) or ( y and C2 ))}
|    |  |
|1086| 0| 63    z or (        !a        ) or (       a and C2 ) --> z or {((    !a    ) or (       C2 ))}
|1087| 0|
|1088| 0| 66    z or ( x and  !a and C1 ) or (       a        ) --> z or {(( x and C1 ) or (    a     ))}
|1089| 0| 67    z or (        !a and C1 ) or (       a        ) --> z or {((      C1 ) or (    a     ))}
|1090| 0|
|1091| 0| 69    z or (        !a        ) or ( y and a        ) --> z or {((    !a    ) or ( y        ))}
|1092| 0| 70    z or ( x and  !a        ) or (       a        ) --> z or {(( x        ) or (    !a    ))}
|1093| 0| 71    z or (        !a        ) or (       a        ) --> z or {((    1     ) or (    !a    ))}
|    |  |       ->       1
|1094| 0|
|1095| 0| 45    (         a        ) or ( y and !a and C2 )    -->     {((    a     ) or ( y and C2 ))}
|1096| 0| 45    (         a        ) or ( y and !a and C2 )    -->     {((    a     ) or ( y and C2 ))}
|1097| 0| 47    (         a        ) or (       !a and C2 )    -->     {((    a     ) or (       C2 ))}
|1098| 0|
|1099| 0| 50    ( x and  a and C1 ) or (       !a        )    -->     {(( x and C1 ) or (    !a    ))}
|1100| 0| 51    (        a and C1 ) or (       !a        )    -->     {((      C1 ) or (    !a    ))}
|1101|94|
|1102| 0| 53    (         a        ) or ( y and !a       )    -->     {((    a     ) or ( y        ))}
|1103| 0| 54    ( x and  a         ) or (       !a       )    -->     {(( x        ) or (    !a    ))}
|1104| 0| 55    (         a        ) or (       !a       )    -->     {((    a     ) or (    !a    ))}
|    |  |       ->       1
|1105| 0|
|1106| 0| 61    z or (        a        ) or ( y and !a and C2 ) --> z or {((    a     ) or ( y and C2 ))}
|1107| 0| 63    z or (        a        ) or (       !a and C2 ) --> z or {((    a     ) or (       C2 ))}
|1108|58|
|1109| 0| 66    z or ( x and  a and C1 ) or (       !a        ) --> z or {(( x and C1 ) or (    !a    ))}
|1110| 0| 67    z or (        a and C1 ) or (       !a        ) --> z or {((      C1 ) or (    !a    ))}
|1111| 0|
|1112| 0| 69    z or (        a        ) or ( y and !a       ) --> z or {((    a     ) or ( y        ))}
|1113| 0| 70    z or ( x and  a        ) or (       !a       ) --> z or {(( x        ) or (    !a    ))}
|1114| 0| 71    z or (        a        ) or (       !a       ) --> z or {((    a     ) or (    !a    ))}
|    |  |        ->       1
|1115| 0|
```

```
|1116| 0| ****************************************************************************************
|    |  | **************/
|1117| 0| /* ******************** NEW IMPLANT ************************ */
|1118| 0| /* 40        ( x and  a and C1 ) or ( y and a and C2 ) -->     {(( x and C1 ) or ( y and C2 ))
|    |  | and a and 11.. } */
|1119| 2|   DYADOP,OROP,
|1120| 4|     ASNODE1,
|1121| 4|     DYADOP,ANDOP,
|1122| 7|        DYADOP,ANDOP,
|1123|10|           EXP2,
|1124|10|           EXP1,
|1125| 7|        CONST1,
|1126| 4|     ASNODE2,
|1127| 4|     DYADOP,ANDOP,
|1128| 7|        DYADOP,ANDOP,
|1129|10|           EXP3,
|1130|10|           RESET_NOTFLAG, /* init for matching vars */
|1131|10|           ISVAR1,
|1132| 7|        CONST2,
|1133| 3|    /* TO */
|1134| 0| REWRITE,BOOLLOOP,
|1135| 3|    NEWNODE,
|1136| 3|    DYADOP,ANDOP,
|1137| 4|     DYADOP,ANDOP,
|1138| 6|       DYADOP,OROP,
|1139| 9|          DYADOP,ANDOP,
|1140|12|             EXP2,
|1141|12|             CONST1,
|1142| 9|          DYADOP,ANDOP,
|1143|12|             EXP3,
|1144|12|             CONST2,
|1145| 6|       EXP1,
|1146| 4|     REGALLONES,
|1147| 3|    END,
|1148| 1|
|1149| 0| /* 41        (         a and C1 ) or ( y and a and C2 ) -->     {((        C1 ) or ( y and C2 ))
|    |  | and a and 11.. }
|1150| 1|  *                                                          (  y or C1 ) and a and C3 == ( C1
|    |  | or C2 )              */
|1151| 2|   DYADOP,OROP,
|1152| 4|     ASNODE1,
|1153| 4|     DYADOP,ANDOP,
|1154| 7|        EXP1,
|1155| 7|        CONST1,
|1156| 4|     ASNODE2,
|1157| 4|     DYADOP,ANDOP,
|1158| 7|        DYADOP,ANDOP,
|1159|10|           EXP3,
|1160|10|           RESET_NOTFLAG, /* init for matching vars */
|1161|10|           ISVAR1,
|1162| 7|        CONST2,
|1163| 3|    /* TO */
|1164| 0| REWRITE,BOOLLOOP,
|1165| 3|    NEWNODE,
|1166| 3|    DYADOP,ANDOP,
|1167| 4|     DYADOP,ANDOP,
|1168| 6|       DYADOP,OROP,
|1169|12|             EXP3,
|1170|12|             CONST1,
|1171| 6|       EXP1,
|1172| 4|     EVALUATE,
|1173| 6|       DYADOP,OROP,
|1174| 8|         CONST1,
|1175| 8|         CONST2,
|1176| 3|    END,
|1177| 0|
|1178| 0| /* 42        ( x and  a and C1 ) or (       a and C2 ) -->     {(( x and C1 ) or (       C2 ))
|    |  | and a and 11.. }
|1179|60|                                                        ( X or C2 ) and a and C3 == ( C1 or
|    |  | C2 )             */
|1180| 2|   DYADOP,OROP,
|1181| 4|     ASNODE1,
|1182| 4|     DYADOP,ANDOP,
|1183| 7|        DYADOP,ANDOP,
|1184|10|           EXP2,
|1185|10|           EXP1,
```

```
|1186| 7|         CONST1,
|1187| 4|      ASNODE2,
|1188| 4|      DYADOP,ANDOP,
|1189| 7|         RESET_NOTFLAG, /* init for matching vars */
|1190| 7|         ISVAR1,
|1191| 7|         CONST2,
|1192| 3|     /* TO */
|1193| 0| REWRITE,BOOLLOOP,
|1194| 3|    NEWNODE,
|1195| 3|    DYADOP,ANDOP,
|1196| 4|     DYADOP,ANDOP,
|1197| 6|       DYADOP,OROP,
|1198|12|             EXP2,
|1199|12|             CONST2,
|1200| 6|       EXP1,
|1201| 4|      EVALUATE,
|1202| 6|        DYADOP,OROP,
|1203| 8|          CONST1,
|1204| 8|          CONST2,
|1205| 3|    END,
|1206| 0|
|1207| 0| /* 43         (        a and C1 ) or (        a and C2 ) -->       (((        C1 ) or (        C2 ))
|    |  | and a and 11.. )
|1208| 0| *                                                                          a and eval c1 or C2 */
|1209| 2|   DYADOP,OROP,
|1210| 4|     ASNODE1,
|1211| 4|     DYADOP,ANDOP,
|1212| 7|        EXP1,
|1213| 7|        CONST1,
|1214| 4|     ASNODE2,
|1215| 4|     DYADOP,ANDOP,
|1216| 7|        RESET_NOTFLAG, /* init for matching vars */
|1217| 7|        ISVAR1,
|1218| 7|        CONST2,
|1219| 3|    /* TO */
|1220| 0| REWRITE,BOOLLOOP,
|1221| 3|    NEWNODE,
|1222| 3|    DYADOP,ANDOP,
|1223| 6|       EXP1,
|1224| 6|       EVALUATE,
|1225| 8|         DYADOP,OROP,
|1226|11|            CONST1,
|1227|11|            CONST2,
|1228| 3|    END,
|1229| 0|
|1230| 0| /* 50    z or ( x and  a and C1 ) or ( y and a and C2 ) --> z or ((( x and C1 ) or ( y and C2 ))
|    |  | and a and 11.. ) */
|1231| 2|   ASNODE3,
|1232| 2|   DYADOP,OROP,
|1233| 4|     DYADOP,OROP,
|1234| 6|       EXP4,
|1235| 7|        ASNODE1,
|1236| 7|        DYADOP,ANDOP,
|1237|10|           DYADOP,ANDOP,
|1238|13|              EXP2,
|1239|13|              EXP1,
|1240|10|           CONST1,
|1241| 7|        ASNODE2,
|1242| 7|        DYADOP,ANDOP,
|1243|10|           DYADOP,ANDOP,
|1244|13|              EXP3,
|1245|13|              RESET_NOTFLAG, /* init for matching vars */
|1246|13|              ISVAR1,
|1247|10|           CONST2,
|1248| 3|    /* TO */
|1249| 0| REWRITE,BOOLLOOP,
|1250| 3|    PUTNODE3,
|1251| 3|    DYADOP,OROP,
|1252| 6|       EXP4,
|1253| 4|     NEWNODE,
|1254| 4|     DYADOP,ANDOP,
|1255| 6|       DYADOP,ANDOP,
|1256| 9|          DYADOP,OROP,
|1257|12|             DYADOP,ANDOP,
|1258|15|                EXP2,
|1259|15|                CONST1,
```

```
|1260|12|            DYADOP,ANDOP,
|1261|15|               EXP3,
|1262|15|               CONST2,
|1263| 9|         EXP1,
|1264| 5|     REGALLONES,
|1265| 3|   END,
|1266| 0|
|1267| 0|/* 51    z or (        a and C1 ) or ( y and a and C2 ) --> z or (((        C1 ) or ( y and C2 ))
|    |  |and a and 11.. ) */
|1268| 2|  ASNODE3,
|1269| 2|  DYADOP,OROP,
|1270| 4|    DYADOP,OROP,
|1271| 6|      EXP4,
|1272| 7|       ASNODE1,
|1273| 7|       DYADOP,ANDOP,
|1274|10|          EXP1,
|1275|10|          CONST1,
|1276| 7|       ASNODE2,
|1277| 7|       DYADOP,ANDOP,
|1278|10|          DYADOP,ANDOP,
|1279|13|             EXP3,
|1280|13|             RESET_NOTFLAG, /* init for matching vars */
|1281|13|             ISVAR1,
|1282|10|          CONST2,
|1283| 3|   /* TO */
|1284| 0|REWRITE,BOOLLOOP,
|1285| 3|   PUTNODE3,
|1286| 3|   DYADOP,OROP,
|1287| 6|      EXP4,
|1288| 3|   NEWNODE,
|1289| 3|   DYADOP,ANDOP,
|1290| 4|    DYADOP,ANDOP,
|1291| 6|      DYADOP,OROP,
|1292|12|            EXP3,
|1293|12|            CONST1,
|1294| 6|      EXP1,
|1295| 4|    EVALUATE,
|1296| 6|      DYADOP,OROP,
|1297| 8|        CONST1,
|1298| 8|        CONST2,
|1299| 3|   END,
|1300| 0|
|1301| 0|/* 52    z or ( x and  a and C1 ) or (        a and C2 ) --> z or ((( x and C1 ) or (        C2 ))
|    |  |and a and 11.. ) */
|1302| 2|  ASNODE3,
|1303| 2|  DYADOP,OROP,
|1304| 4|    DYADOP,OROP,
|1305| 6|      EXP4,
|1306| 7|       ASNODE1,
|1307| 7|       DYADOP,ANDOP,
|1308|10|          DYADOP,ANDOP,
|1309|13|             EXP2,
|1310|13|             EXP1,
|1311|10|          CONST1,
|1312| 7|       ASNODE2,
|1313| 7|       DYADOP,ANDOP,
|1314|10|          RESET_NOTFLAG, /* init for matching vars */
|1315|10|          ISVAR1,
|1316|10|          CONST2,
|1317| 3|   /* TO */
|1318| 0|REWRITE,BOOLLOOP,
|1319| 3|   PUTNODE3,
|1320| 3|   DYADOP,OROP,
|1321| 6|      EXP4
|1322| 0|REWRITE,BOOLLOOP,
|1323| 3|   NEWNODE,
|1324| 3|   DYADOP,ANDOP,
|1325| 4|    DYADOP,ANDOP,
|1326| 6|      DYADOP,OROP,
|1327|12|            EXP2,
|1328|12|            CONST2,
|1329| 6|      EXP1,
|1330| 4|    EVALUATE,
|1331| 6|      DYADOP,OROP,
|1332| 8|        CONST1,
|1333| 8|        CONST2,
```

```
|1334| 3|    END,
|1335| 0|
|1336| 0| /* 54    z or (        a and C1 ) or (        a and C2 ) --> z or (((        C1 ) or (        C2 ))
|    |  | and a and l1.. ) */
|1337| 2|   ASNODE3,
|1338| 2|   DYADOP,OROP,
|1339| 4|     DYADOP,OROP,
|1340| 6|       EXP4,
|1341| 7|        ASNODE1,
|1342| 7|        DYADOP,ANDOP,
|1343|10|           EXP1,
|1344|10|           CONST1,
|1345| 7|        ASNODE2,
|1346| 7|        DYADOP,ANDOP,
|1347|10|           RESET_NOTFLAG, /* init for matching vars */
|1348|10|           ISVAR1,
|1349|10|           CONST2,
|1350| 3|    /* TO *
|1351| 0| REWRITE,BOOLLOOP,
|1352| 3|    PUTNODE3,
|1353| 3|    DYADOP,OROP,
|1354| 6|       EXP4,
|1355| 0| REWRITE,BOOLLOOP,
|1356| 3|    NEWNODE,
|1357| 3|    DYADOP,ANDOP,
|1358| 6|       EXP1,
|1359| 6|       EVALUATE,
|1360| 8|         DYADOP,OROP,
|1361|11|            CONST1,
|1362|11|            CONST2,
|1363| 3|    END,
|1364| 0|
|1365| 0| /********************** END OF NEW INSERTED PART , NOW THE SPECIAL CASES CAN FOLLOW **********
|1366| 0|
|1367| 3|    /*   exp1 relop1 exp2 op3 exp2 relop2 exp1 -->
|1368| 8|         exp1 relop1 exp2 op3 exp1 reverserelop2 exp2   */
|1369| 3|    ASOP3,DYADOP,ANYOP,
|1370| 6|       ASOP2,DYADOP,RELOP,
|1371| 9|          EXP1,
|1372| 9|          EXP2,
|1373| 6|       ASOP1,DYADOP,RELOP,
|1374| 9|          ISEXP2,
|1375| 9|          ISEXP1,
|1376| 3|    /*Equivalent to:*/
|1377| 3|    DYADOP,OP3,
|1378| 6|       DYADOP,OP2,
|1379| 9|          EXP1,
|1380| 9|          EXP2,
|1381| 6|       DYADOP,REVRELOP1,
|1382| 9|          EXP1,
|1383| 9|          EXP2,
|1384| 3|    END,
|1385| 0|
|1386| 3|    /*   exp1 relop1 exp2 dyadlogop exp1 relop2 exp2 --> special case 1   */
|1387| 3|    ASOP1,DYADOP,LOGOP,
|1388| 6|       ASOP2,DYADOP,RELOP,
|1389| 9|          ASEXP1,INTEXP,
|1390| 9|          EXP2,
|1391| 6|       ASOP3,DYADOP,RELOP,
|1392| 9|          ISEXP1,
|1393| 9|          ISEXP2,
|1394| 3|    /*Equivalent to:*/
|1395| 3|    SPECCASE1,EXP1,EXP2, /* These last two entries list the use of SpecCase 1 ! *
|1396| 3|    END,
|1397| 0|
|1398| 3|    /*   exp1 relop1 intconst1 dyadlogop exp1 relop2 intconst1 --> special case 2   */
|1399| 3|    ASOP1,DYADOP,LOGOP,
|1400| 6|       ASOP2,DYADOP,RELOP,
|1401| 9|          EXP1,
|1402| 9|          ASCONST1,INTCONST,
|1403| 6|       ASOP3,DYADOP,RELOP,
|1404| 9|          ISEXP1,
|1405| 9|          ASCONST2,INTCONST,
|1406| 3|    /*Equivalent to:*/
|1407| 3|    SPECCASE2,EXP1,CONST1,CONST2, /* These last two entries list the use of SpecCase 2 ! */
|1408| 3|    END,
```

```
|1409| 0|
|1410| 3|    /*   exp1 relop1 regconst1 dyadlogop exp1 relop2 regconst2 --> special case 3   */
|1411| 3|    ASOP1,DYADOP,LOGOP,
|1412| 6|       ASOP2,DYADOP,RELOP,
|1413| 9|          EXP1,
|1414| 9|          ASCONST1,REGCONST,
|1415| 6|       ASOP3,DYADOP,RELOP,
|1416| 9|          ISEXP1,
|1417| 9|          ASCONST2,REGCONST,
|1418| 3|    /*Equivalent to:*/
|1419| 3|    SPECCASE3,EXP1,CONST1,CONST2, /* These last two entries list the use of SpecCase 3 ! */
|1420| 3|    END,
|1421| 3|
|1422| 3|    END_VAR,END,
|1423| 0|
|1424| 3|    END };
|1425| 0|
|1426| 0| /********************************************************************
|1427| 1|  * rulebasetype SIMLOOPRULES                                        *
|1428| 1|  ********************************************************************/
|1429| 0| rulebasetype simlooprules[] =
|1430| 3|    {
|1431| 3|
|1432| 3|    EXP1,PRINT,EXP1,END,
|1433| 0|
|1434| 3|    ASEXP1,IFALL_HAD,EXP1,EXIT,END,
|1435| 3|
|1436| 3|    /* z + ( ( ..x..) * y * C1 ) --> Rew z + ( (Rew x) * y * C1) */
|1437| 3|    ASNODE1,
|1438| 3|    DYADOP,INTADDOP,
|1439| 6|       EXP1,
|1440| 6|       DYADOP,MULTOP,
|1441| 8|         DYADOP,MULTOP,
|1442|11|            EXP2,
|1443|11|            EXP3,
|1444| 8|         ASCONST1,IFHIT,       /* new busy var selected */
|1445| 3|    /* to *
|1446| 3|    PUTNODE1,
|1447| 3|    DYADOP,INTADDOP,
|1448| 6|       REWRITE,SIMPLIFY,REWRITE,SORTSUBTREE,EXP1,
|1449| 6|       DYADOP,MULTOP,
|1450| 8|         DYADOP,MULTOP,
|1451|10|           REWRITE,INIT,EXP2, /* sort , subsort .. */
|1452|10|           EXP3,              /* simpleso leave it  */
|1453| 8|         CONST1,
|1454| 3|    END,
|1455| 0|
|1456| 3|    /* z + ( y * C1 ) --> Rew z + ( y * C1) */
|1457| 3|    ASNODE1,
|1458| 3|    DYADOP,INTADDOP,
|1459| 6|       EXP1,
|1460| 6|       DYADOP,MULTOP,
|1461| 8|         EXP2,
|1462| 8|         ASCONST1,IFHIT,       /* new busy var selected */
|1463| 3|    /* to */
|1464| 3|    PUTNODE1,
|1465| 3|    DYADOP,INTADDOP,
|1466| 6|       REWRITE,SIMPLIFY,REWRITE,SORTSUBTREE,EXP1,
|1467| 6|       DYADOP,MULTOP,
|1468| 8|         EXP2,                 /* simpleso leave it  */
|1469| 8|         CONST1,
|1470| 3|    END,
|1471| 0|
|1472| 0|
|1473| 0|
|1474| 3|    /* ( (...x...) * y * C1 ) --> ( (Rew x) * y * C1 ) */
|1475| 3|    DYADOP,MULTOP,
|1476| 5|      DYADOP,MULTOP,
|1477| 7|        EXP1,
|1478| 7|        EXP2,
|1479| 5|      ASCONST1,IFHIT,          /* new busy var selected */
|1480| 3|    /* to */
|1481| 3|    DYADOP,MULTOP,
|1482| 5|      DYADOP,MULTOP,
|1483| 7|        REWRITE,INIT,EXP1,   /* sort , subsort .. */
|1484| 7|        EXP2,                /* simpleso leave it  */
```

```
|1485| 5|       CONST1,
|1486| 3|     END,
|1487| 0|
|1488| 4|     ASEXP1,IFALL_HAD,EXP1,EXIT,END,
|1489| 0|
|1490| 4|     ASEXP1,IFHIT,REWRITE,LOCALSIM,EXP1,EXIT,END,
|1491| 0|
|1492| 3|     ASEXP1,IFALL_HAD,EXP1,EXIT,END,
|1493| 0|
|1494| 3|     EXP1,REWRITE,SIMPLIFY,EXP1,END,
|1495| 0|
|1496| 0| /*   ASEXP1,IFNOTALL_HAD,REWRITE,CURRENT,EXP1,END,*/
|1497| 0|
|1498| 3|     END };
|1499| 0|
|1500| 0| /*********************************************************************
|1501| 1|  * rulebasetype LOCALSIM                                             *
|1502| 1|  *********************************************************************/
|1503| 0| rulebasetype localsimrules[] =
|1504| 3|     {
|1505| 3|     SELECT_VAR,END,
|1506| 0|
|1507| 3|     /* a + 0 --> a */
|1508| 4|     DYADOP,INTADDOP,
|1509| 6|        EXP1,INT0,
|1510| 3|     /*  to */
|1511| 3|     EXP1,END,
|1512| 0|
|1513| 3|     /* a * 0 --> 0 */
|1514| 4|     ASKILLNODE,
|1515| 3|     DYADOP,MULTOP,
|1516| 6|        EXP1,INT0,
|1517| 3|     /* to */
|1518| 3|     KILL_NODE,
|1519| 3|     INT0,EXIT,END,
|1520| 0|
|1521| 3|     ASEXP1,IFALL_HAD,EXP1,EXIT,END,
|1522| 0|
|1523| 7|         EXP1,
|1524| 4|     REWRITE,SORTSUBTREE,
|1525| 7|         EXP1,END,
|1526| 3|
|1527| 7|         ASEXP1,INTEXP,REWRITE,
|1528| 3|     SIMPLIFY,
|1529| 7|         EXP1,END,
|1530| 0|
|1531| 3|     END };
|1532| 0|
|1533| 0| /*********************************************************************
|1534| 1|  * rulebasetype INIT                                                 *
|1535| 1|  *********************************************************************/
|1536| 0| rulebasetype initrules[] =
|1537| 3|     {
|1538| 3|     EXP1,REWRITE,EXPAND,EXP1,END,
|1539| 0|
|1540| 3|     EXP1,REWRITE,SORT,EXP1,END,
|1541| 0|
|1542| 3|     MAKE_VAR_COUNT,END,
|1543| 2|
|1544| 3|     EXP1,PRINT,EXP1,END,
|1545| 0|
|1546| 7|         ASEXP1,IFNOTALL_HAD,
|1547| 3|     REWRITE,SORTSUBTREE,
|1548| 7|         EXP1,END,
|1549| 3|
|1550| 7|         ASEXP1,INTEXP,REWRITE,
|1551| 3|     SIMPLIFY,
|1552| 7|         EXP1,END,
|1553| 0|
|1554| 3|     END };
|1555| 0|
|1556| 0|  *********************************************************************
|1557| 1|  * rulebasetype SIMPLIFYRULES                                        *
|1558| 1|  * this rulebase defines the rewrite rules for expression            *
|1559| 1|  * simplification. the rulebase is applied to an expression          *
|1560| 1|  * using rewriteexpression(), which resides in module rewritex.      *
```

```
|1561| 1|  *********************************************************************/
|1562| 0| rulebasetype simplifyrules[] =
|1563| 3|    {
|1564| 4|
|1565| 3|    RESET_NOTFLAG,INITHIT,END, /* reseting to avoid problems */
|1566| 4|
|1567| 3|    EXP1,PRINT,EXP1,END,
|1568| 0|
|1569| 3|    /*  const op a --> a op const   */
|1570| 3|    ASOP1,DYADOP,SYMOP,
|1571| 6|       CONST1,
|1572| 6|       EXP2,
|1573| 3|    /* to *,
|1574| 3|    DYADOP,OP1,
|1575| 6|       EXP2,
|1576| 6|       CONST1,
|1577| 3|    END,
|1578| 0|
|1579| 3|    /* a + 0 --> a */
|1580| 3|    DYADOP,INTADDOP,
|1581| 6|       EXP1,INTO,
|1582| 3|    /*  to */
|1583| 3|    EXP1,END,
|1584| 0|
|1585| 3|    /* a * 0 --> 0 */
|1586| 3|    ASKILLNODE,
|1587| 3|    DYADOP,MULTOP,
|1588| 6|       EXP1,INTO,
|1589| 3|    /* to *
|1590| 3|    KILL_NODE,
|1591| 3|    INTO,EXIT,END,
|1592| 0|
|1593| 0| /* new made rules because constat insert ! only the rules stay
|1594| 0|
|1595| 0| 1     z + x * a * C1 + y * a * C2     -->     z + ( x * C1 + y * C2 ) * a * 1
|1596| 0| 2         x * a * C1 + y * a * C2     -->         ( x * C1 + y * C2 ) * a * 1
|1597| 0|
|1598| 0| 3     z + x * a * C1 +     a * C2     -->     z + ( x * C1 +     C2 ) * a * 1
|1599| 0| 4         x * a * C1 +     a * C2     -->         ( x * C1 +     C2 ) * a * 1
|1600| 0|
|1601| 0| 5     z +     a * C1 + y * a * C2     -->     z + ( y * C2 +     C1 ) * a * 1
|1602| 0| 6             a * C1 + y * a * C2     -->         ( y * C2 +     C1 ) * a * 1
|1603| 0|
|1604| 0| 7     z +     a * C1 +     a * C2     -->     z +               a * ( C1+C2 )
|1605| 0| 8             a * C1 +     a * C2     -->                       a * ( C1+C2 )
|1606| 3|    exp2 exp3 exp1 C1 exp4 exp1 C2
|1607| 0| */
|1608| 0|
|1609| 0| /* 1 z + x * a * C1 + y * a * C2     -->     z + ( x * C1 + y * C2 ) * a * 1 */
|1610| 3|    ASNODE3,
|1611| 3|    DYADOP,INTADDOP,
|1612| 6|       DYADOP,INTADDOP,
|1613| 9|          EXP2,
|1614| 9|          ASNODE1,DYADOP,MULTOP,
|1615|12|             DYADOP,MULTOP,
|1616|15|                EXP3,
|1617|15|                NOTCONST,EXP1,
|1618|12|             CONST1,
|1619| 6|       ASNODE2,DYADOP,MULTOP,
|1620| 9|          DYADOP,MULTOP,
|1621|12|             EXP4,
|1622|12|             ISVAR1,
|1623| 9|          CONST2,
|1624| 3|    /* to *
|1625| 0| REWRITE,SIMLOOP,
|1626| 3|    PUTNODE3,
|1627| 3|    DYADOP,INTADDOP,
|1628| 6|       EXP2,
|1629| 6|       NEWNODE,
|1630| 6|       DYADOP,MULTOP,
|1631| 8|         DYADOP,MULTOP,
|1632|10|           DYADOP,INTADDOP,
|1633|12|             DYADOP,MULTOP,
|1634|16|                 EXP3,
|1635|16|                 CONST1,
|1636|12|             DYADOP,MULTOP,
```

```
|1637|16|                    EXP4,
|1638|16|                    CONST2,
|1639|10|              EXP1,
|1640| 8|            INT1,
|1641| 0| END,
|1642| 0|
|1643| 0|
|1644| 0|
|1645| 0| /* 2    x * a * C1 + y * a * C2     -->          ( x * C1 + y * C2 ) * a * 1 */
|1646| 3|    DYADOP,INTADDOP,
|1647| 9|          ASNODE1,DYADOP,MULTOP,
|1648|12|             DYADOP,MULTOP,
|1649|15|                EXP3,
|1650|15|                NOTCONST,EXP1,
|1651|12|             CONST1,
|1652| 6|       ASNODE2,DYADOP,MULTOP,
|1653| 9|          DYADOP,MULTOP,
|1654|12|             EXP4,
|1655|12|             ISVAR1,
|1656| 9|          CONST2,
|1657| 3|    /* to */
|1658| 0| REWRITE,SINLOOP,
|1659| 6|       NEWNODE,
|1660| 6|       DYADOP,MULTOP,
|1661| 8|         DYADOP,MULTOP,
|1662|10|           DYADOP,INTADDOP,
|1663|12|             DYADOP,MULTOP,
|1664|16|                 EXP3,
|1665|16|                 CONST1,
|1666|12|             DYADOP,MULTOP,
|1667|16|                 EXP4,
|1668|16|                 CONST2,
|1669|10|           EXP1,
|1670| 8|         INT1,
|1671| 0| END,
|1672| 0|
|1673| 0|
|1674| 6|
|1675| 0| /* 3 z + x * a * C1 +      a * C2      -->      z + ( x * C1 +      C2 ) * a * 1 */
|1676| 3|    ASNODE3,
|1677| 3|    DYADOP,INTADDOP,
|1678| 6|       DYADOP,INTADDOP,
|1679| 9|          EXP2,
|1680| 9|          ASNODE1,DYADOP,MULTOP,
|1681|12|             DYADOP,MULTOP,
|1682|15|                EXP3,
|1683|15|                NOTCONST,EXP1,
|1684|12|             CONST1,
|1685| 6|       ASNODE2,DYADOP,MULTOP,
|1686| 9|          ISVAR1,
|1687| 9|          CONST2,
|1688| 3|    /* to */
|1689| 0| REWRITE,SINLOOP,
|1690| 3|    PUTNODE3,
|1691| 3|    DYADOP,INTADDOP,
|1692| 6|       EXP2,
|1693| 6|       NEWNODE,
|1694| 6|       DYADOP,MULTOP,
|1695| 8|         DYADOP,MULTOP,
|1696|10|           DYADOP,INTADDOP,
|1697|12|             DYADOP,MULTOP,
|1698|16|                 EXP3,
|1699|16|                 CONST1,
|1700|12|             CONST2,
|1701|10|           EXP1,
|1702| 8|         INT1,
|1703| 0| END,
|1704| 0|
|1705| 0|
|1706| 0|  /* 4  x * a * C1 +      a * C2      -->            ( x * C1 +      C2 ) * a * 1 */
|1707| 3|    DYADOP,INTADDOP,
|1708| 9|          ASNODE1,DYADOP,MULTOP,
|1709|12|             DYADOP,MULTOP,
|1710|15|                EXP3,
|1711|15|                NOTCONST,EXP1,
|1712|12|             CONST1,
```

```
|1713| 6|       ASNODE2,DYADOP,MULTOP,
|1714| 9|          ISVAR1,
|1715| 9|          CONST2,
|1716| 3|    /* to */
|1717| 0| REWRITE,SIMLOOP,
|1718| 6|       NEWNODE,
|1719| 6|       DYADOP,MULTOP,
|1720| 8|         DYADOP,MULTOP,
|1721|10|           DYADOP,INTADDOP,
|1722|12|             DYADOP,MULTOP,
|1723|16|                 EXP3,
|1724|16|                 CONST1,
|1725|13|              CONST2,
|1726|10|           EXP1,
|1727| 8|         INT1,
|1728| 0| END,
|1729| 0|
|1730| 0| /* 5 z +      a * C1 + y * a * C2      -->      z + ( y * C2 +      C1 ) * a * 1 */
|1731| 3|    ASNODE3.
|1732| 3|    DYADOP,INTADDOP,
|1733| 6|       DYADOP,INTADDOP,
|1734| 9|          EXP2,
|1735| 9|          ASNODE1,DYADOP,MULTOP,
|1736|12|             NOTCONST,EXP1,
|1737|12|             CONST1,
|1738| 6|       ASNODE2,DYADOP,MULTOP,
|1739| 9|          DYADOP,MULTOP,
|1740|12|             EXP4,
|1741|12|             ISVAR1,
|1742| 9|          CONST2,
|1743| 3|    /* to *
|1744| 0| REWRITE,SIMLOOP,
|1745| 3|    PUTNODE3,
|1746| 3|    DYADOP,INTADDOP,
|1747| 6|       EXP2,
|1748| 6|       NEWNODE,
|1749| 6|       DYADOP,MULTOP,
|1750| 8|         DYADOP,MULTOP,
|1751|10|           DYADOP,INTADDOP,
|1752|12|             DYADOP,MULTOP,
|1753|16|                 EXP4,
|1754|16|                 CONST2,
|1755|12|             CONST1,
|1756|10|           EXP1,
|1757| 8|         INT1,
|1758| 0| END,
|1759| 0|
|1760| 0| /* 6  a * C1 + y * a * C2      -->         ( y * C2 +      C1 ) * a * 1 */
|1761| 3|    DYADOP,INTADDOP,
|1762| 9|          ASNODE1,DYADOP,MULTOP,
|1763|12|             NOTCONST,EXP1,
|1764|12|             CONST1,
|1765| 6|       ASNODE2,DYADOP,MULTOP,
|1766| 9|          DYADOP,MULTOP,
|1767|12|             EXP4,
|1768|12|             ISVAR1,
|1769| 9|          CONST2,
|1770| 3|    /* to */
|1771| 0| REWRITE,SIMLOOP,
|1772| 6|       NEWNODE,
|1773| 6|       DYADOP,MULTOP,
|1774| 8|         DYADOP,MULTOP,
|1775|10|           DYADOP,INTADDOP,
|1776|12|             DYADOP,MULTOP,
|1777|16|                 EXP4,
|1778|16|                 CONST2,
|1779|12|             CONST1,
|1780|10|           EXP1,
|1781| 8|         INT1,
|1782| 0| END,
|1783| 0|
|1784| 0|  * 7 z +      a * C1 +      a * C2      -->      z +              a * ( C1+C2 ) */
|1785| 3|    ASNODE3.
|1786| 3|    DYADOP,INTADDOP,
|1787| 6|       DYADOP,INTADDOP,
|1788| 9|          EXP2,
```

```
|1789| 9|         ASNODE1,DYADOP,MULTOP,
|1790|12|            NOTCONST,EXP1,
|1791|12|            CONST1,
|1792| 9|         ASNODE2,DYADOP,MULTOP,
|1793|12|            ISVAR1,
|1794|12|            CONST2,
|1795| 3|   /* to */
|1796| 0|REWRITE,SINLOOP,
|1797| 3|   PUTNODE1,
|1798| 3|   DYADOP,INTADDOP,
|1799| 6|      EXP2
|1800| 6|      NEWNODE,
|1801| 6|      DYADOP,MULTOP,
|1802| 8|        EXP1,
|1803| 8|        EVALUATE,
|1804|10|          DYADOP,INTADDOP,
|1805|16|                CONST1,
|1806|16|                CONST2,
|1807| 0|END,
|1808| 0|
|1809| 0|/* 8 a * C1 +     a * C2      -->                         a * ( C1+C2 ) */
|1810| 3|   DYADOP,INTADDOP,
|1811| 9|         ASNODE1,DYADOP,MULTOP,
|1812|12|            NOTCONST,EXP1,
|1813|12|            CONST1,
|1814| 9|         ASNODE2,DYADOP,MULTOP,
|1815|12|            ISVAR1,
|1816|12|            CONST2,
|1817| 3|   /* to *
|1818| 0|REWRITE,SINLOOP,
|1819| 6|      NEWNODE,
|1820| 6|      DYADOP,MULTOP,
|1821| 9|         EXP1,
|1822| 9|         EVALUATE,
|1823| 9|         DYADOP,INTADDOP,
|1824|16|                CONST1,
|1825|16|                CONST2,
|1826| 4|    END,
|1827| 0|
|1828| 0|/*   RECURSIVE_REWRITE(CURRENT), */
|1829| 0|
|1830| 0|/*   DYADOP,INTADDOP,
|1831| 6|      EXP1.
|1832| 6|      EXP2.
|1833| 3|   DYADOP,INTADDOP,
|1834| 6|      REWRITE,CURRENT,EXP1,
|1835| 6|      EXP2,
|1836| 3|   END,
|1837| 0|*/
|1838| 3|   ASNODE1,
|1839| 3|   DYADOP,INTADDOP,
|1840| 5|     EXP1,
|1841| 5|     CONST1,
|1842| 3|   /* to */
|1843| 3|   PUTNODE1,
|1844| 3|   DYADOP,INTADDOP,
|1845| 5|     REWRITE,CURRENT,EXP1,
|1846| 5|     CONST1,
|1847| 3|   END,
|1848| 2|
|1849|11|
|1850| 3|   END_VAR,END,
|1851| 0|
|1852| 3|   END
|1853| 3|   };
|1854| 0|
|1855| 0| ****************************************************************************
|1856| 1| * rulebase ADDSUBSORTTREE                                                  *
|1857| 1| ****************************************************************************
|1858| 0|rulebasetype addsortsubtreerules[]={
|1859| 0|
|1860| 3|   /* a ADD or a MUL  */
|1861| 4|
|1862| 3|   PUSHNN,END,
|1863| 0|
|1864| 3|   ASNODE1.DYADOP,INTADDOP,
```

```
|1865| 6|        EXP1, /* a ADD or a MUL if only one ADD to go */
|1866| 6|        EXP2, /* a MUL or a terminal */
|1867| 3|     /*  to  */
|1868| 3|     PUTNODE1,DYADOP,INTADDOP,
|1869| 6|        REWRITE,CURRENT,EXP1,
|1870| 6|        REWRITE,MULSORTSUBTREE,EXP2,
|1871| 3|     END,
|1872| 0|
|1873| 3|     /* INTEGER mul last one */
|1874| 3|     ASEXP1,
|1875| 3|     DYADOP,MULTOP,
|1876| 6|        EXP,
|1877| 6|        EXP,
|1878| 3|     /* SORT IT */
|1879| 3|     REWRITE,MULSORTSUBTREE,EXP1,END,
|1880| 0|
|1881| 0| /*
|1882| 7|         EXP1,PRINT,EXP1,END,
|1883| 0| */
|1884| 0|
|1885| 2|    POPNN,END,
|1886| 0|
|1887| 2|    END);
|1888| 0|
|1889| 0| /*************************************************************************
|1890| 1|  * rulebase MULSUBSORTTREE                                                *
|1891| 1|  ************************************************************************/
|1892| 0| rulebasetype mulsortsubtreerules[]={
|1893| 4|
|1894| 3|     PUSHNN,END,
|1895| 0|
|1896| 3|     /* check if INTEGER mul and proces if it is */
|1897| 3|     ASEXP1,
|1898| 3|     DYADOP,MULTOP,
|1899| 6|        EXP,
|1900| 6|        EXP,
|1901| 3|     /* SORT IT */
|1902| 3|     SORTSUBTREE,EXP1,END,
|1903| 0|
|1904| 0| /*
|1905| 7|         EXP1,PRINT,EXP1,END,
|1906| 0| */
|1907| 3|     POPNN,END,
|1908| 0|
|1909| 3|     END);
|1910| 0|
|1911| 0| /*************************************************************************
|1912| 1|  * rulebase ORSUBSORTTREE                                                 *
|1913| 1|  ************************************************************************/
|1914| 0| rulebasetype orsortsubtreerules[] ={
|1915| 5|
|1916| 3|     PUSHNN,END,
|1917| 0|
|1918| 3|      /* an OR or an AND  */
|1919| 3|     ASNODE1.DYADOP,OROP,
|1920| 6|        EXP1, /* OR or AND if only one OR */
|1921| 6|        EXP2, /* AND or terminal */
|1922| 3|     /* to */
|1923| 3|     PUTNODE1,DYADOP,OROP,
|1924| 6|        REWRITE,CURRENT,EXP1,
|1925| 6|        REWRITE,ANDSORTSUBTREE,EXP2,
|1926| 3|     END,
|1927| 0|
|1928| 3|      /* REG AND tree last one */
|1929| 3|     ASEXP1,
|1930| 3|     DYADOP,ANDOP,
|1931| 6|        EXP,
|1932| 6|        EXP,
|1933| 3|     /* SORT IT */
|1934| 3|     REWRITE.ANDSORTSUBTREE,EXP1,END,
|1935| 0|
|1936| 0| /*
|1937| 7|         EXP1,PRINT,EXP1,END,
|1938| 0| */
|1939| 3|     POPNN,END,
|1940| 0|
```

```
|1941| 3|    END);
|1942| 0|
|1943| 0| /*****************************************************************************
|1944| 1|  * rulebase XORSUBSORTTREE                                                   *
|1945| 1|  *****************************************************************************/
|1946| 0| rulebasetype xorsortsubtreerules[] ={
|1947|13|
|1948| 4|     PUSHNN,END,
|1949| 0|
|1950| 4|     /* an OR or an AND  */
|1951| 3|    ASNODE1,DYADOP,XOROP,
|1952| 6|       EXP2, /* OR or AND if only one OR */
|1953| 6|       EXP1, /* AND or terminal */
|1954| 3|    /* to */
|1955| 3|    PUTNODE1,DYADOP,XOROP,
|1956| 6|       REWRITE,CURRENT,EXP2,
|1957| 6|       SORTSUBTREE,EXP1,
|1958| 3|    END,
|1959| 0|
|1960| 3|    /* REG AND tree last one */
|1961| 3|    EXP1,
|1962| 3|    /* SORT IT */
|1963| 3|    SORTSUBTREE,EXP1,END,
|1964| 0|
|1965| 0| /*
|1966| 7|        EXP1,PRINT,EXP1,END,
|1967| 0| */
|1968| 3|    POPNN,END,
|1969| 0|
|1970| 3|    END);
|1971| 0|
|1972| 0|
|1973| 0|
|1974| 0| /*****************************************************************************
|1975| 1|  * rulebase ANDSUBSORTTREE                                                   *
|1976| 1|  *****************************************************************************/
|1977| 0| rulebasetype andsortsubtreerules[]={
|1978| 0|
|1979| 3|    PUSHNN,END,
|1980| 0|
|1981| 3|    /* check if and */
|1982| 3|    ASEXP1,
|1983| 3|    DYADOP,ANDOP,
|1984| 6|       EXP,
|1985| 6|       EXP,
|1986| 3|    /* SORT IT */
|1987| 3|    SORTSUBTREE,EXP1,END,
|1988|12|
|1989| 0| /*
|1990| 7|        EXP1,PRINT,EXP1,END,
|1991| 0| */
|1992| 3|    POPNN,END,
|1993| 0|
|1994| 3|    END);
|1995| 0|
|1996| 0| /*****************************************************************************
|1997| 1|  * rulebase SUBSORTTREE                                                      *
|1998| 1|  * used to sort subtrees which are only of one dyadic operant                *
|1999| 1|  *****************************************************************************/
|2000| 0| rulebasetype sortsubtreerules[] = {
|2001| 3|
|2002| 3|    /* we expect : a ADD or a OR , the main root thing */
|2003| 3|    PUSHNN,END,
|2004| 3|
|2005| 3|    EXP1,PRINT,EXP1,END,
|2006|10|
|2007| 3|    SELECT_VAR,END,
|2008| 0|
|2009| 3|    ASEXP1,IFALL_HAD,EXP1,EXIT,END,
|2010| 0|
|2011| 3|    /* first search subtree INTEGER ADD*/
|2012| 3|    ASNODE1,DYADOP,INTADDOP,
|2013| 6|       EXP1, /* a ADD or a MUL if only one ADD */
|2014| 6|       EXP2, /* a MUL or a terminal */
|2015| 3|    /*  to  */
|2016| 3|    PUTNODE1,DYADOP,INTADDOP,
```

```
|2017| 6|        REWRITE,ADDSORTSUBTREE,EXP1,
|2018| 6|        REWRITE,MULSORTSUBTREE,EXP2,
|2019| 3|     END,
|2020| 0|
|2021| 3|     /* first search subtree REG OR */
|2022| 3|     ASNODE1,DYADOP,OROP,
|2023| 6|        EXP1, /* OR or AND if only one OR */
|2024| 6|        EXP2, /* AND or terminal */
|2025| 3|     /* to */
|2026| 3|     PUTNODE1,DYADOP,OROP,
|2027| 6|        REWRITE,ORSORTSUBTREE,EXP1,
|2028| 6|        REWRITE,ANDSORTSUBTREE,EXP2,
|2029| 3|     END,
|2030| 3|
|2031| 3|     /* first search xor tree */
|2032| 3|     ASNODE1,DYADOP,XOROP,
|2033| 6|        EXP2,
|2034| 6|        EXP1,
|2035| 3|     /* TO */
|2036| 3|     PUTNODE1,DYADOP,XOROP,
|2037| 7|         REWRITE,XORSORTSUBTREE,EXP2,
|2038| 7|         SORTSUBTREE,EXP1,
|2039| 3|     END,
|2040| 0|
|2041| 3|     /* if only one node : AND or MUL, exit after it ! */
|2042| 0|
|2043| 3|     /* first search subtree INTEGER mult*/
|2044| 3|     ASEXP1,ASNODE1,
|2045| 3|     DYADOP,MULTOP,
|2046| 6|        EXP,
|2047| 6|        EXP,
|2048| 3|     /* SORT IT */
|2049| 3|     PUTNODE1,REWRITE,MULSORTSUBTREE,EXP1,END,
|2050| 0|
|2051| 3|     ASEXP1,ASNODE1,
|2052| 3|     DYADOP,ANDOP,
|2053| 6|        EXP,
|2054| 6|        EXP,
|2055| 3|     /* SORT IT */
|2056| 3|     PUTNODE1,REWRITE,ANDSORTSUBTREE,EXP1,END,
|2057| 0|
|2058| 3|     /* sort and tree */
|2059| 3|     ASEXP1,ASNODE1,
|2060| 3|     DYADOP,INTADDOP,
|2061| 6|        EXP,
|2062| 6|        EXP,
|2063| 3|     /*  SORT IT */
|2064| 3|     SORTSUBTREE, /* is command now ! */
|2065| 3|     PUTNODE1,EXP1,END,
|2066| 0|
|2067| 3|     /* sort or tree */
|2068| 3|     ASEXP1,ASNODE1,
|2069| 3|     DYADOP,OROP,
|2070| 6|        EXP,
|2071| 6|        EXP,
|2072| 3|     /* SORT IT */
|2073| 3|     SORTSUBTREE, /* is command now ! */
|2074| 3|     PUTNODE1,EXP1,END,
|2075| 0|
|2076| 3|     /*  search sorted xor tree */
|2077| 3|     ASEXP1,ASNODE1,
|2078| 3|     DYADOP,XOROP,
|2079| 6|        EXP,
|2080| 6|        EXP,
|2081| 3|     /* TO */
|2082| 3|     PUTNODE1,SORTSUBTREE,EXP1,
|2083| 3|     END,
|2084| 0|
|2085| 3|     EXP1,PRINT,EXP1,END,
|2086|14|
|2087| 3|     POPUN,END,
|2088| 0|
|2089| 3|     END
|2090| 3|     };
|2091| 0|
|2092| 0| /***************************************************************************************
```

```
|2093| 1|  * rulebase IMPLODEREG                                                       *
|2094| 1|  * inverse expand on regs !                                                  *
|2095| 1|  ****************************************************************************/
|2096| 0| rulebasetype imploderegrules[] = {
|2097| 3|
|2098| 3|    RECURSIVE_REWRITE(CURRENT),
|2099| 0|
|2100| 3|    /*   sort operands for symmetrical dyadic operators:
|2101| 0|         const op a --> a op const    */
|2102| 3|    ASOP1,DYADOP,SYMOP,
|2103| 6|       CONST1,
|2104| 6|       EXP2,
|2105| 3|    /*Equivalent to:*/
|2106| 3|    DYADOP,OP1,
|2107| 6|       EXP2,
|2108| 6|       CONST1,
|2109| 3|    END,
|2110| 0|
|2111| 3|    /*   a and #0...0b --> #0...0b    */
|2112| 3|    DYADOP,ANDOP,
|2113| 6|       EXP,
|2114| 6|       ASEXP1,REGALLZEROES,
|2115| 3|    /*Equivalent to:*/
|2116| 3|    EXP1,
|2117| 3|    EXIT,END,
|2118| 0|
|2119| 3|    /*   a or #0...0b --> a    */
|2120| 3|    DYADOP,OROP,
|2121| 6|       EXP1,
|2122| 6|       REGALLZEROES,
|2123| 3|    /*Equivalent to:*/
|2124| 3|    EXP1,
|2125| 3|    END,
|2126| 0|
|2127| 3|    /*   a or #1...1b --> #1...1b    */
|2128| 3|    DYADOP,OROP,
|2129| 6|       EXP,
|2130| 6|       ASEXP1,REGALLONES,
|2131| 3|    /*Equivalent to:*/
|2132| 3|    EXP1,
|2133| 3|    END,
|2134| 0|
|2135| 3|    /*   a and #1...1b --> a    */
|2136| 3|    DYADOP,ANDOP,
|2137| 6|       EXP1,
|2138| 6|       REGALLONES,
|2139| 3|    /*Equivalent to:*/
|2140| 3|    EXP1,
|2141| 3|    END,
|2142| 0|
|2143| 3|    END };
|2144| 0|
|2145| 0| /****************************************************************************
|2146| 1|  * rulebase IMPLODEINT                                                       *
|2147| 1|  * reverse expand on ints                                                    *
|2148| 1|  ****************************************************************************/
|2149| 0| rulebasetype implodeintrules[] = {
|2150| 0|
|2151| 3|    RECURSIVE_REWRITE(CURRENT),
|2152| 0|
|2153| 3|    /*  const op a --> a op const    */
|2154| 3|    ASOP1,DYADOP,SYMOP,
|2155| 6|       CONST1,
|2156| 6|       EXP2,
|2157| 3|    /* to */
|2158| 3|    DYADOP,OP1,
|2159| 6|       EXP2,
|2160| 6|       CONST1,
|2161| 3|    END,
|2162| 0|
|2163| 3|    /* a + 0 --> a */
|2164| 3|    DYADOP,INTADDOP,
|2165| 6|       EXP1,INT0,
|2166| 3|    /* to */
|2167| 3|    EXP1,END,
|2168| 0|
```

```
|2169| 3|    /* a * 0 --> 0 */
|2170| 3|    DYADOP,MULTOP,
|2171| 6|       EXP1,INT0,
|2172| 3|    /* to */
|2173| 3|    INT0,EXIT,END,
|2174| 0|
|2175| 3|    /* a * 1 --> a */
|2176| 3|    DYADOP,MULTOP,
|2177| 6|       EXP1,INT1,
|2178| 3|    /* to *
|2179| 3|    EXP1,END,
|2180| 3|
|2181| 0|
|2182| 3|    EVALUATE_CONSTS,
|2183| 0|
|2184| 3|    END );
|2185| 0|
|2186| 0| /*****************************************************************************
|2187| 1|  * rulebase IMPLODEMINUS                                                    *
|2188| 1|  * pushes minus back up                                                     *
|2189| 1|  *****************************************************************************/
|2190| 0| rulebasetype implodeminusrules[] = {
|2191| 0|
|2192| 3|    /* ( a * -1) -->  - a */
|2193| 3|    DYADOP,MULTOP,
|2194| 8|         EXP1,
|2195| 8|         INTM1,
|2196| 3|    /* to *
|2197| 3|    MONADOP,MINUSOP,
|2198| 8|         EXP1,
|2199| 3|    END,
|2200| 0|
|2201| 3|    RECURSIVE_REWRITE(CURRENT),
|2202| 0|
|2203| 3|    /* a + - b ) --> a - b   2x !!*/
|2204| 3|    DYADOP,INTADDOP,
|2205| 6|       EXP1,
|2206| 6|       MONADOP,MINUSOP,
|2207| 9|          EXP2,
|2208| 3|    /* to */
|2209| 3|    DYADOP,INTSUBOP,
|2210| 6|       EXP1,
|2211| 6|       EXP2,
|2212| 3|    END,
|2213| 0|
|2214| 3|    /* ( - a   ) + b --> b - a */
|2215| 3|    DYADOP,INTADDOP,
|2216| 6|       MONADOP,MINUSOP,
|2217|10|           EXP2,
|2218| 6|       EXP1,
|2219| 3|    /* to */
|2220| 3|    DYADOP,INTSUBOP,
|2221| 6|       EXP1,
|2222| 6|       EXP2,
|2223| 3|    END,
|2224| 0|
|2225| 0| END );
|2226| 0|
|2227| 0| /*****************************************************************************
|2228| 1|  * rulebase IMPLODENOT                                                      *
|2229| 1|  * pushes not bacs up                                                       *
|2230| 1|  *****************************************************************************/
|2231| 0| rulebasetype implodenotrules[] = {
|2232| 0|
|2233| 3|    RECURSIVE_REWRITE(CURRENT),
|2234| 0|
|2235| 3|    /* not a and b or a and not b --> a xor b */
|2236| 3|    DYADOP,OROP,
|2237| 6|       DYADOP,ANDOP,
|2238| 9|          MONADOP,NOTOP,
|2239|12|             EXP1,
|2240| 9|          EXP2,
|2241| 6|       DYADOP,ANDOP,
|2242| 9|          ISEXP1,
|2243| 9|          MONADOP,NOTOP,
|2244|12|             ISEXP2,
```

```
|2245| 3|    /*Equivalent to:*/
|2246| 3|    DYADOP,XOROP,
|2247| 6|       EXP1,
|2248| 6|       EXP2,
|2249| 3|    END,
|2250| 0|
|2251| 3|    /* b and not a or not b and a --> a xor b */
|2252| 3|    DYADOP,OROP,
|2253| 6|       DYADOP,ANDOP,
|2254| 9|          MONADOP,NOTOP,EXP1,
|2255| 9|          EXP2,
|2256| 6|       DYADOP,ANDOP,
|2257| 9|          ISEXP1,
|2258| 9|          MONADOP,NOTOP,ISEXP2,
|2259| 3|    /*Equivalent to:*/
|2260| 3|    DYADOP,XOROP,
|2261| 6|       EXP1,
|2262| 6|       EXP2,
|2263| 3|    END,
|2264| 0|
|2265| 3|    /* swapped also ! */
|2266| 3|    /* not b and a or b and not a --> a xor b */
|2267| 3|    DYADOP,OROP,
|2268| 6|       DYADOP,ANDOP,
|2269| 9|          EXP1,
|2270| 9|          MONADOP,NOTOP,EXP2,
|2271| 6|       DYADOP,ANDOP,
|2272| 9|          MONADOP,NOTOP,ISEXP1,
|2273| 9|          ISEXP2,
|2274| 3|    /*Equivalent to:*/
|2275| 3|    DYADOP,XOROP,
|2276| 6|       EXP1,
|2277| 6|       EXP2,
|2278| 3|    END,
|2279| 0|
|2280| 0|
|2281| 3|    /* a and b or not (a or b) --> not (a xor b) */
|2282| 3|    DYADOP,OROP,
|2283| 6|       DYADOP,ANDOP,
|2284| 9|          EXP1,
|2285| 9|          EXP2,
|2286| 6|       MONADOP,NOTOP,
|2287| 9|          DYADOP,OROP,
|2288|12|             ISEXP1,
|2289|12|             ISEXP2,
|2290| 3|    /*Equivalent to:*/
|2291| 3|    MONADOP,NOTOP,
|2292| 6|       DYADOP,XOROP,
|2293| 9|          EXP1,
|2294| 9|          EXP2,
|2295| 3|    END,
|2296| 3|    /* a and b or not (b or a) --> not (a xor b) */
|2297| 3|    DYADOP,OROP,
|2298| 6|       DYADOP,ANDOP,
|2299| 9|          EXP1,
|2300| 9|          EXP2,
|2301| 6|       MONADOP,NOTOP,
|2302| 9|          DYADOP,OROP,
|2303|12|             ISEXP2,
|2304|12|             ISEXP1,
|2305| 3|    /*Equivalent to:*/
|2306| 3|    MONADOP,NOTOP,
|2307| 6|       DYADOP,XOROP,
|2308| 9|          EXP1,
|2309| 9|          EXP2,
|2310| 3|    END,
|2311| 0|
|2312| 0|
|2313| 3|    /* !a & !b --> !( a | b ) */
|2314| 3|    DYADOP,ANDOP,
|2315| 6|       MONADOP,NOTOP,
|2316| 9|          EXP1,
|2317| 6|       MONADOP,NOTOP,
|2318| 9|          EXP2,
|2319| 3|    /* Equivalent to: */
|2320| 3|    MONADOP,NOTOP,
```

```
|2321| 6|       DYADOP,OROP,
|2322| 9|          EXP1,
|2323| 9|          EXP2,
|2324| 3|    END,
|2325| 0|
|2326| 3|    /*    !a ¦ !b --> ! ( a & b )   */
|2327| 3|    DYADOP,OROP,
|2328| 6|       MONADOP,NOTOP,
|2329| 9|          EXP1,
|2330| 6|       MONADOP,NOTOP,
|2331| 9|          EXP2,
|2332| 3|    /* Equivalent to: */
|2333| 3|    MONADOP,NOTOP,
|2334| 6|       DYADOP,ANDOP,
|2335| 9|          EXP1,
|2336| 9|          EXP2,
|2337| 3|    END,
|2338| 0|
|2339| 3|    /* ( x & !a ) & !b -- >  !( a ¦ b ) & x */
|2340| 3|    DYADOP,ANDOP,
|2341| 6|       DYADOP,ANDOP,
|2342| 9|          EXP1,
|2343| 9|          MONADOP,NOTOP,EXP2,
|2344| 6|       MONADOP,NOTOP,EXP3,
|2345| 3|    /* to: */
|2346| 3|    DYADOP,ANDOP,
|2347| 6|       MONADOP,NOTOP,
|2348| 9|          DYADOP,OROP,
|2349|12|             EXP2,
|2350|12|             EXP3,
|2351| 6|       EXP1.
|2352| 3|    END,
|2353| 0|
|2354| 3|    /* ( x ¦ !a ) ¦ !b --> ! ( a & b ) ¦ x */
|2355| 3|    DYADOP,OROP,
|2356| 6|       DYADOP,OROP,
|2357| 9|          EXP1,
|2358| 9|          MONADOP,NOTOP,EXP2,
|2359| 6|       MONADOP,NOTOP,EXP3,
|2360| 3|    /* to */
|2361| 3|    DYADOP,OROP,
|2362| 6|       MONADOP,NOTOP,
|2363| 9|          DYADOP,ANDOP,
|2364|12|             EXP2,
|2365|12|             EXP3,
|2366| 6|       EXP1,
|2367| 3|    END,
|2368| 0|
|2369| 0| END };
|2370| 0|
|2371| 0| /*****************************************************************************
|2372| 1|  * rulebase PREPROCESRULES                                                  *
|2373| 1|  * not used yet                                                             *
|2374| 1|  ****************************************************************************/
|2375| 0|
|2376| 0| rulebasetype preprocesrules[] = {
|2377| 0|
|2378| 0|
|2379| 0| /* from simpl */
|2380| 3|    /* (a +' const1) +' const2 --> a +' (const1 +' const 2) */
|2381| 3|    ASOP1,DYADOP,SYMOP,
|2382| 6|       DYADOP,ISOP1,
|2383| 9|         EXP1,CONST1,
|2384| 6|       CONST2,
|2385| 3|    /* rewrite to */
|2386| 3|    DYADOP,OP1,
|2387| 6|       EXP1,
|2388| 6|       EVALUATE,DYADOP,OP1,
|2389| 8|         CONST1,
|2390| 8|         CONST2,
|2391| 3|    END,
|2392| 0|
|2393| 3|    /* ( a + ( b * C1 )) * C2 -->  a * C2 + ( b * ( C1*C2) ) */
|2394| 3|    DYADOP,MULTOP,
|2395| 7|        DYADOP,INTADDOP,
|2396|11|            EXP1,
```

```
|2397|13|           DYADOP,MULTOP,
|2398|14|              EXP2,
|2399|14|              CONST1,
|2400| 7|       CONST2,
|2401| 3|   /* to */
|2402| 3|   DYADOP,INTADDOP,
|2403| 7|       DYADOP,MULTOP,
|2404|10|          EXP1,
|2405|10|          CONST1,
|2406| 7|       DYADOP,MULTOP,
|2407|10|          EXP2,
|2408|10|          EVALUATE,
|2409|13|             DYADOP,MULTOP,
|2410|16|                CONST1,
|2411|16|                CONST2,
|2412| 3|   END,
|2413| 3|   /* --a --> a */
|2414| 3|   MONADOP,MINUSOP,
|2415| 5|     MONADOP,MINUSOP,
|2416| 8|        EXP1,
|2417| 3|   /* to */
|2418| 3|   EXP1,END,
|2419| 0|
|2420| 0| /*72*/
|2421| 0|
|2422| 0| END };
|2423| 0|
|2424| 0| rulebasetype sortnotrules[] = {
|2425| 0| /***************************************************************************
|2426| 1|  * rulebase SORTNOTRULES                                                  *
|2427| 1|  * used to sort all not variables/expression to the left side             *
|2428| 1|  * result : !a and !b and !c and d and e and f                            *
|2429| 1|  **************************************************************************/
|2430| 6|
|2431| 3|   RECURSIVE_REWRITE(CURRENT),
|2432| 4|
|2433| 3|   /* a and !b --> !b and a */
|2434| 3|   DYADOP,ASOP1,SYMOP,
|2435| 6|      EXP1,
|2436| 6|      ASEXP2,MONADOP,NOTOP,EXP,
|2437| 3|   /* to */
|2438| 3|   DYADOP,OP1,
|2439| 6|      EXP2,
|2440| 6|      EXP1,
|2441| 3|   END,
|2442| 0|
|2443| 3|   /* ( a and b )  and !c --> REW:( a and !c ) and b */
|2444| 3|   DYADOP,ASOP1,SYMOP,
|2445| 6|      DYADOP,ISOP1,
|2446| 9|         EXP1,
|2447| 9|         ASEXP2,NOTNOT,
|2448| 6|      ASEXP3,MONADOP,NOTOP,EXP,
|2449| 3|   /* to */
|2450| 3|   DYADOP,OP1,
|2451| 6|      REWRITE,CURRENT,
|2452| 9|         DYADOP,OP1,
|2453|12|            EXP1,
|2454|12|            EXP3,
|2455| 6|      EXP2,
|2456| 3|   END,
|2457| 0| /* extra out of expand xor */
|2458| 0| /* name doesn't fit anymore
|2459| 3|   now these are the simplification rules for xor*
|2460| 0|
|2461| 3|   DYADOP,XOROP,
|2462| 6|      EXP1
|2463| 6|      EXP2.
|2464| 3|    * to *
|2465| 3|   DYADOP,XOROP,
|2466| 6|      REWRITE,CURRENT, EXP1,
|2467| 6|      EXP2.
|2468| 3|   END,
|2469| 4|
|2470| 4|    INITHIT,END, /* reset hit flag */
|2471| 0|
|2472| 4|    /* a xor a --> 0 */
```

```
|2473| 4|      ASKILLNODE,
|2474| 4|      DYADOP,XOROP,
|2475| 7|         EXP1,
|2476| 7|         RESET_NOTFLAG, /* init for matching vars */
|2477| 7|         SAVESIZE,
|2478| 7|         ISVAR1,
|2479| 4|      /* to */
|2480| 4|      HIT,
|2481| 4|      KILL_NODE,
|2482| 4|      REGALLZEROES,
|2483| 4|      EXIT,END,
|2484| 0|
|2485| 4|      /* a xor not a --> 1 */
|2486| 4|      ASKILLNODE,
|2487| 4|      DYADOP,XOROP,
|2488| 7|         EXP1,
|2489| 7|         RESET_NOTFLAG, /* init for matching vars */
|2490| 7|         MONADOP,NOTOP,SAVESIZE,ISVAR1,
|2491| 4|      /* to */
|2492| 4|      HIT,
|2493| 4|      KILL_NODE,
|2494| 4|      REGALLONES,
|2495| 4|      EXIT,END,
|2496| 0|
|2497| 4|      /* NOT a xor a --> 0 */
|2498| 4|      ASKILLNODE,
|2499| 4|      DYADOP,XOROP,
|2500| 7|         MONADOP,NOTOP,EXP1,
|2501| 7|         SET_NOTFLAG, /* init for matching vars */
|2502| 7|         SAVESIZE,
|2503| 7|         ISVAR1,
|2504| 4|      /* to */
|2505| 4|      HIT,
|2506| 4|      KILL_NODE,
|2507| 4|      REGALLONES,
|2508| 4|      EXIT,END,
|2509| 0|
|2510| 4|      /* ...a... xor ...a... --> 0 */
|2511| 4|      ASKILLNODE,
|2512| 4|      DYADOP,XOROP,
|2513| 7|         EXP1,
|2514| 7|         SET_NOTFLAG,
|2515| 7|         SAVESIZE,
|2516| 7|         ISEXP1,
|2517| 4|      /* to */
|2518| 4|      HIT,
|2519| 4|      KILL_NODE,
|2520| 4|      REGALLZEROES,
|2521| 4|      EXIT,END,
|2522| 0|
|2523| 4|      /* z xor a xor a --> z xor 0 --> z */
|2524| 4|      DYADOP,XOROP,
|2525| 7|         DYADOP,XOROP,
|2526|10|            EXP2,
|2527|10|            EXP1,
|2528| 7|         RESET_NOTFLAG, /* init for matching vars */
|2529| 7|         SAVESIZE,
|2530| 7|         ISVAR1,
|2531| 4|      /* to */
|2532| 4|      DYADOP,XOROP,
|2533| 7|         EXP2,
|2534| 7|         REGALLZEROES,
|2535| 4|      END,
|2536| 0|
|2537| 4|      /* z xor not a xor a --> z xor 1 --> not z */
|2538| 4|      DYADOP,XOROP,
|2539| 7|         DYADOP,XOROP,
|2540|10|            EXP2,
|2541|10|            MONADOP,NOTOP,EXP1,
|2542| 7|         SET_NOTFLAG, /* init for matching vars *
|2543| 7|         ISVAR1,
|2544| 4|      /* to */
|2545| 4|      MONADOP,NOTOP,EXP2,END,
|2546| 0|
|2547| 4|      /* z xor a xor not a --> z xor 1 --> not z */
|2548| 4|      DYADOP,XOROP,
```

1332/ 1


```
|2549|  7|        DYADOP,XOROP,
|2550| 10|           EXP2,
|2551| 10|           EXP1,
|2552|  7|        RESET_NOTFLAG, /* init for matching vars */
|2553|  7|        MONADOP,NOTOP,ISVAR1,
|2554|  4|     /* to */
|2555|  4|     MONADOP,NOTOP,EXP2,END,
|2556|  0|
|2557|  4|     /* z xor ...a... xor ...a... --> z xor 0 --> z */
|2558|  4|     DYADOP,XOROP,
|2559|  7|        DYADOP,XOROP,
|2560| 10|           EXP2,
|2561| 10|           EXP1,
|2562|  7|        RESET_NOTFLAG, /* init for matching vars */
|2563|  7|        ISEXP1,
|2564|  4|     /* to */
|2565|  4|     EXP2,END,
|2566|  3|    ASEXP1,IFHIT,REWRITE,CURRENT,REWREG,EXP1,END,
|2567|  3|    END };
|2568|  0|
```