REDUCE

User's Manual

Free Version

Anthony C. Hearn
Santa Monica, CA, USA

http://reduce-algebra.sourceforge.net/

May 13, 2012

Contents

Abstract

This document provides the user with a description of the algebraic programming system REDUCE. The capabilities of this system include:

1. expansion and ordering of polynomials and rational functions,
2. substitutions and pattern matching in a wide variety of forms,
3. automatic and user controlled simplification of expressions,
4. calculations with symbolic matrices,
5. arbitrary precision integer and real arithmetic,
6. facilities for defining new functions and extending program syntax,
7. analytic differentiation and integration,
8. factorization of polynomials,
9. facilities for the solution of a variety of algebraic equations,
10. facilities for the output of expressions in a variety of formats,
11. facilities for generating numerical programs from symbolic input,
12. Dirac matrix calculations of interest to high energy physicists.

Acknowledgment

The production of this version of the manual has been the result of the contributions of a large number of individuals who have taken the time and effort to suggest improvements to previous versions, and to draft new sections. Particular thanks are due to Gerry Rayna, who provided a draft rewrite of most of the first half of the manual. Other people who have made significant contributions have included John Fitch, Martin Griss, Stan Kameny, Jed Marti, Herbert Melenk, Don Morrison, Arthur Norman, Eberhard Schrüfer, Larry Seward and Walter Tietze. Finally, Richard Hitt produced a TEX version of the REDUCE 3.3 manual, which has been a useful guide for the production of the LATEX version of this manual.

Chapter 1

Introductory Information

REDUCE is a system for carrying out algebraic operations accurately, no matter how complicated the expressions become. It can manipulate polynomials in a variety of forms, both expanding and factoring them, and extract various parts of them as required. REDUCE can also do differentiation and integration, but we shall only show trivial examples of this in this introduction. Other topics not considered include the use of arrays, the definition of procedures and operators, the specific routines for high energy physics calculations, the use of files to eliminate repetitious typing and for saving results, and the editing of the input text.

Also not considered in any detail in this introduction are the many options that are available for varying computational procedures, output forms, number systems used, and so on.

REDUCE is designed to be an interactive system, so that the user can input an algebraic expression and see its value before moving on to the next calculation. For those systems that do not support interactive use, or for those calculations, especially long ones, for which a standard script can be defined, REDUCE can also be used in batch mode. In this case, a sequence of commands can be given to REDUCE and results obtained without any user interaction during the computation.

In this introduction, we shall limit ourselves to the interactive use of REDUCE, since this illustrates most completely the capabilities of the system. When REDUCE is called, it begins by printing a banner message like:

```
REDUCE 3.8, 15-Jul-2003 ...
```

where the version number and the system release date will change from time to time. It then prompts the user for input by:

```
1:
```

You can now type a REDUCE statement, terminated by a semicolon to indicate the end of the expression, for example:

```
(x+y+z)^2;
```

This expression would normally be followed by another character (a **Return** on an ASCII keyboard) to "wake up" the system, which would then input the expression, evaluate it, and return the result:

```
 2                    2           2
X  + 2*X*Y + 2*X*Z + Y  + 2*Y*Z + Z
```

Let us review this simple example to learn a little more about the way that REDUCE works. First, we note that REDUCE deals with variables, and constants like other computer languages, but that in evaluating the former, a variable can stand for itself. Expression evaluation normally follows the rules of high school algebra, so the only surprise in the above example might be that the expression was expanded. REDUCE normally expands expressions where possible, collecting like terms and ordering the variables in a specific manner. However, expansion, ordering of variables, format of output and so on is under control of the user, and various declarations are available to manipulate these.

Another characteristic of the above example is the use of lower case on input and upper case on output. In fact, input may be in either mode, but output is usually in lower case. To make the difference between input and output more distinct in this manual, all expressions intended for input will be shown in lower case and output in upper case. However, for stylistic reasons, we represent all single identifiers in the text in upper case.

Finally, the numerical prompt can be used to reference the result in a later computation.

As a further illustration of the system features, the user should try:

```
for i:= 1:40 product i;
```

The result in this case is the value of 40!,

```
815915283247897734345611269596115894272000000000
```

You can also get the same result by saying

```
factorial 40;
```

Since we want exact results in algebraic calculations, it is essential that integer arithmetic be performed to arbitrary precision, as in the above example. Furthermore, the FOR statement in the above is illustrative of a whole range of combining forms that REDUCE supports for the convenience of the user.

Among the many options in REDUCE is the use of other number systems, such as multiple precision floating point with any specified number of digits — of use if roundoff in, say, the 100^{th} digit is all that can be tolerated.

In many cases, it is necessary to use the results of one calculation in succeeding calculations. One way to do this is via an assignment for a variable, such as

```
u := (x+y+z)^2;
```

If we now use U in later calculations, the value of the right-hand side of the above will be used.

The results of a given calculation are also saved in the variable WS (for WorkSpace), so this can be used in the next calculation for further processing.

For example, the expression

```
df(ws,x);
```

following the previous evaluation will calculate the derivative of (x+y+z)^2 with respect to X. Alternatively,

```
int(ws,y);
```

would calculate the integral of the same expression with respect to y.

REDUCE is also capable of handling symbolic matrices. For example,

```
matrix m(2,2);
```

declares m to be a two by two matrix, and

```
m := mat((a,b),(c,d));
```

gives its elements values. Expressions that include M and make algebraic sense may now be evaluated, such as 1/m to give the inverse, 2*m - u*m^2 to give us another matrix and det(m) to give us the determinant of M.

REDUCE has a wide range of substitution capabilities. The system knows about elementary functions, but does not automatically invoke many of their well-known properties. For example, products of trigonometrical functions

are not converted automatically into multiple angle expressions, but if the user wants this, he can say, for example:

```
     (sin(a+b)+cos(a+b))*(sin(a-b)-cos(a-b))
 where cos(~x)*cos(~y) = (cos(x+y)+cos(x-y))/2,
       cos(~x)*sin(~y) = (sin(x+y)-sin(x-y))/2,
       sin(~x)*sin(~y) = (cos(x-y)-cos(x+y))/2;
```

where the tilde in front of the variables `X` and `Y` indicates that the rules apply for all values of those variables. The result of this calculation is

```
        -(COS(2*A) + SIN(2*B))
```

See also the user-contributed packages ASSIST (chapter 15.5), CAMAL (chapter 15.11) and TRIGSIMP (chapter 15.61).

Another very commonly used capability of the system, and an illustration of one of the many output modes of REDUCE, is the ability to output results in a FORTRAN compatible form. Such results can then be used in a FORTRAN based numerical calculation. This is particularly useful as a way of generating algebraic formulas to be used as the basis of extensive numerical calculations.

For example, the statements

```
on fort;
df(log(x)*(sin(x)+cos(x))/sqrt(x),x,2);
```

will result in the output

```
 ANS=(-4.*LOG(X)*COS(X)*X**2-4.*LOG(X)*COS(X)*X+3.*
 . LOG(X)*COS(X)-4.*LOG(X)*SIN(X)*X**2+4.*LOG(X)*
 . SIN(X)*X+3.*LOG(X)*SIN(X)+8.*COS(X)*X-8.*COS(X)-8.
 . *SIN(X)*X-8.*SIN(X))/(4.*SQRT(X)*X**2)
```

These algebraic manipulations illustrate the algebraic mode of REDUCE. REDUCE is based on Standard Lisp. A symbolic mode is also available for executing Lisp statements. These statements follow the syntax of Lisp, e.g.

```
symbolic car '(a);
```

Communication between the two modes is possible.

With this simple introduction, you are now in a position to study the material in the full REDUCE manual in order to learn just how extensive the range of facilities really is. If further tutorial material is desired, the seven REDUCE Interactive Lessons by David R. Stoutemyer are recommended. These are

normally distributed with the system.

Chapter 2

Structure of Programs

A REDUCE program consists of a set of functional commands which are evaluated sequentially by the computer. These commands are built up from declarations, statements and expressions. Such entities are composed of sequences of numbers, variables, operators, strings, reserved words and delimiters (such as commas and parentheses), which in turn are sequences of basic characters.

2.1 The REDUCE Standard Character Set

The basic characters which are used to build REDUCE symbols are the following:

1. The 26 letters `a` through `z`
2. The 10 decimal digits `0` through `9`
3. The special characters _ ! " $ % ' ( ) * + , - . / : ; < > = { } <blank>

With the exception of strings and characters preceded by an exclamation mark, the case of characters is ignored: depending of the underlying LISP they will all be converted internally into lower case or upper case: `ALPHA`, `Alpha` and `alpha` represent the same symbol. Most implementations allow you to switch this conversion off. The operating instructions for a particular implementation should be consulted on this point. For portability, we shall limit ourselves to the standard character set in this exposition.

2.2 Numbers

There are several different types of numbers available in REDUCE. Integers consist of a signed or unsigned sequence of decimal digits written without a decimal point, for example:

```
-2, 5396, +32
```

In principle, there is no practical limit on the number of digits permitted as exact arithmetic is used in most implementations. (You should however check the specific instructions for your particular system implementation to make sure that this is true.) For example, if you ask for the value of 2^{2000} you get it displayed as a number of 603 decimal digits, taking up nine lines of output on an interactive display. It should be borne in mind of course that computations with such long numbers can be quite slow.

Numbers that aren't integers are usually represented as the quotient of two integers, in lowest terms: that is, as rational numbers.

In essentially all versions of REDUCE it is also possible (but not always desirable!) to ask REDUCE to work with floating point approximations to numbers again, to any precision. Such numbers are called *real*. They can be input in two ways:

1. as a signed or unsigned sequence of any number of decimal digits with an embedded or trailing decimal point.

2. as in 1. followed by a decimal exponent which is written as the letter `E` followed by a signed or unsigned integer.

e.g. `32.  +32.0 0.32E2` and `320.E-1` are all representations of 32.

The declaration `SCIENTIFIC_NOTATION` controls the output format of floating point numbers. At the default settings, any number with five or less digits before the decimal point is printed in a fixed-point notation, e.g., `12345.6`. Numbers with more than five digits are printed in scientific notation, e.g., `1.234567E+5`. Similarly, by default, any number with eleven or more zeros after the decimal point is printed in scientific notation. To change these defaults, `SCIENTIFIC_NOTATION` can be used in one of two ways. `SCIENTIFIC_NOTATION` *m*;, where *m* is a positive integer, sets the printing format so that a number with more than *m* digits before the decimal point, or *m* or more zeros after the decimal point, is printed in scientific notation. `SCIENTIFIC_NOTATION` {*m*,*n*}, with *m* and *n* both positive integers, sets the format so that a number with more than *m* digits before the decimal point, or *n* or more zeros after the decimal point is printed in scientific notation.

CAUTION: The unsigned part of any number may *not* begin with a decimal point, as this causes confusion with the `CONS` (.) operator, i.e., NOT ALLOWED: `.5 -.23 +.12`; use `0.5 -0.23 +0.12` instead.

2.3 Identifiers

Identifiers in REDUCE consist of one or more alphanumeric characters (i.e. alphabetic letters or decimal digits) the first of which must be alphabetic. The maximum number of characters allowed is implementation dependent, although twenty-four is permitted in most implementations. In addition, the underscore character (_) is considered a letter if it is *within* an identifier. For example,

```
a az p1 q23p  a_very_long_variable
```

are all identifiers, whereas

```
_a
```

is not.

A sequence of alphanumeric characters in which the first is a digit is interpreted as a product. For example, `2ab3c` is interpreted as `2*ab3c`. There is one exception to this: If the first letter after a digit is `E`, the system will try to interpret that part of the sequence as a real number, which may fail in some cases. For example, `2E12` is the real number $2.0 * 10^{12}$, `2e3c` is 2000.0*C, and `2ebc` gives an error.

Special characters, such as $-$, *, and blank, may be used in identifiers too, even as the first character, but each must be preceded by an exclamation mark in input. For example:

```
light!-years    d!*!*n          good! morning
!$sign          !5goldrings
```

CAUTION: Many system identifiers have such special characters in their names (especially * and =). If the user accidentally picks the name of one of them for his own purposes it may have catastrophic consequences for his REDUCE run. Users are therefore advised to avoid such names.

Identifiers are used as variables, labels and to name arrays, operators and procedures.

Restrictions

The reserved words listed in another section may not be used as identifiers. No spaces may appear within an identifier, and an identifier may not extend over a line of text. (Hyphenation of an identifier, by using a reserved character as a hyphen before an end-of-line character is possible in some versions of REDUCE).

2.4 Variables

Every variable is named by an identifier, and is given a specific type. The type is of no concern to the ordinary user. Most variables are allowed to have the default type, called *scalar*. These can receive, as values, the representation of any ordinary algebraic expression. In the absence of such a value, they stand for themselves.

Reserved Variables

Several variables in REDUCE have particular properties which should not be changed by the user. These variables include:

`E` Intended to represent the base of the natural logarithms. `log(e)`, if it occurs in an expression, is automatically replaced by 1. If `ROUNDED` is on, `E` is replaced by the value of E to the current degree of floating point precision.

`I` Intended to represent the square root of -1. `i^2` is replaced by -1, and appropriately for higher powers of `I`. This applies only to the symbol `I` used on the top level, not as a formal parameter in a procedure, a local variable, nor in the context `for i:= ...`

`INFINITY` Intended to represent ∞ in limit and power series calculations for example. Note however that the current system does *not* do proper arithmetic on ∞. For example, `infinity + infinity` is `2*infinity`.

`NIL` In REDUCE (algebraic mode only) taken as a synonym for zero. Therefore `NIL` cannot be used as a variable.

`PI` Intended to represent the circular constant. With `ROUNDED` on, it is replaced by the value of π to the current degree of floating point precision.

T Should not be used as a formal parameter or local variable in procedures, since conflict arises with the symbolic mode meaning of T as *true*.

Other reserved variables, such as LOW_POW, described in other sections, are listed in Appendix A.

Using these reserved variables inappropriately will lead to errors.

There are also internal variables used by REDUCE that have similar restrictions. These usually have an asterisk in their names, so it is unlikely a casual user would use one. An example of such a variable is K!* used in the asymptotic command package.

Certain words are reserved in REDUCE. They may only be used in the manner intended. A list of these is given in the section "Reserved Identifiers". There are, of course, an impossibly large number of such names to keep in mind. The reader may therefore want to make himself a copy of the list, deleting the names he doesn't think he is likely to use by mistake.

2.5 Strings

Strings are used in WRITE statements, in other output statements (such as error messages), and to name files. A string consists of any number of characters enclosed in double quotes. For example:

```
"A String".
```

Lower case characters within a string are not converted to upper case.

The string "" represents the empty string. A double quote may be included in a string by preceding it by another double quote. Thus "a""b" is the string a"b, and """" is the string ".

2.6 Comments

Text can be included in program listings for the convenience of human readers, in such a way that REDUCE pays no attention to it. There are two ways to do this:

1. Everything from the word COMMENT to the next statement terminator, normally ; or $, is ignored. Such comments can be placed anywhere a

blank could properly appear. (Note that `END` and >> are *not* treated as `COMMENT` delimiters!)

2. Everything from the symbol % to the end of the line on which it appears is ignored. Such comments can be placed as the last part of any line. Statement terminators have no special meaning in such comments. Remember to put a semicolon before the % if the earlier part of the line is intended to be so terminated. Remember also to begin each line of a multi-line % comment with a % sign.

2.7 Operators

Operators in REDUCE are specified by name and type. There are two types, infix and prefix. Operators can be purely abstract, just symbols with no properties; they can have values assigned (using `:=` or simple `LET` declarations) for specific arguments; they can have properties declared for some collection of arguments (using more general `LET` declarations); or they can be fully defined (usually by a procedure declaration).

Infix operators have a definite precedence with respect to one another, and normally occur between their arguments. For example:

```
a + b - c                    (spaces optional)
x<y and y=z                  (spaces required where shown)
```

Spaces can be freely inserted between operators and variables or operators and operators. They are required only where operator names are spelled out with letters (such as the `AND` in the example) and must be unambiguously separated from another such or from a variable (like `Y`). Wherever one space can be used, so can any larger number.

Prefix operators occur to the left of their arguments, which are written as a list enclosed in parentheses and separated by commas, as with normal mathematical functions, e.g.,

```
cos(u)
df(x^2,x)
q(v+w)
```

Unmatched parentheses, incorrect groupings of infix operators and the like, naturally lead to syntax errors. The parentheses can be omitted (replaced by a space following the operator name) if the operator is unary and the argument is a single symbol or begins with a prefix operator name:

`cos y`	means cos(y)
`cos (-y)`	- parentheses necessary
`log cos y`	means log(cos(y))
`log cos (a+b)`	means log(cos(a+b))

but

`cos a*b`	means (cos a)*b
`cos -y`	is erroneous (treated as a variable "cos" minus the variable y)

A unary prefix operator has a precedence higher than any infix operator, including unary infix operators. In other words, REDUCE will always interpret `cos y + 3` as `(cos y) + 3` rather than as `cos(y + 3)`.

Infix operators may also be used in a prefix format on input, e.g., `+(a,b,c)`. On output, however, such expressions will always be printed in infix form (i.e., `a + b + c` for this example).

A number of prefix operators are built into the system with predefined properties. Users may also add new operators and define their rules for simplification. The built in operators are described in another section.

Built-In Infix Operators

The following infix operators are built into the system. They are all defined internally as procedures.

```
<infix operator>::= where|:=|or|and|member|memq|=|neq|eq|
                    >=|>|<=|<|+|-|*|/|^|**|.
```

These operators may be further divided into the following subclasses:

```
<assignment operator>   ::= :=
<logical operator>      ::= or|and|member|memq
<relational operator>   ::= =|neq|eq|>=|>|<=|<
<substitution operator> ::= where
<arithmetic operator>   ::= +|-|*|/|^|**
<construction operator> ::= .
```

`MEMQ` and `EQ` are not used in the algebraic mode of REDUCE. They are explained in the section on symbolic mode. `WHERE` is described in the section on substitutions.

In previous versions of REDUCE, *not* was also defined as an infix operator. In the present version it is a regular prefix operator, and interchangeable with *null*.

For compatibility with the intermediate language used by REDUCE, each special character infix operator has an alternative alphanumeric identifier associated with it. These identifiers may be used interchangeably with the corresponding special character names on input. This correspondence is as follows:

```
:= setq          (the assignment operator)
= equal
>= geq
> greaterp
<= leq
< lessp
+ plus
- difference    (if unary, minus)
* times
/ quotient      (if unary, recip)
^ or ** expt    (raising to a power)
.  cons
```

Note: `NEQ` is used to mean *not equal*. There is no special symbol provided for it.

The above operators are binary, except `NOT` which is unary and + and * which are nary (i.e., taking an arbitrary number of arguments). In addition, - and / may be used as unary operators, e.g., /2 means the same as 1/2. Any other operator is parsed as a binary operator using a left association rule. Thus `a/b/c` is interpreted as `(a/b)/c`. There are two exceptions to this rule: `:=` and . are right associative. Example: `a:=b:=c` is interpreted as `a:=(b:=c)`. Unlike ALGOL and PASCAL, `^` is left associative. In other words, `a^b^c` is interpreted as `(a^b)^c`.

The operators <, <=, >, >= can only be used for making comparisons between numbers. No meaning is currently assigned to this kind of comparison between general expressions.

Parentheses may be used to specify the order of combination. If parentheses are omitted then this order is by the ordering of the precedence list defined by the right-hand side of the `<infix operator>` table at the beginning of this section, from lowest to highest. In other words, `WHERE` has the lowest precedence, and . (the dot operator) the highest.

Chapter 3

Expressions

REDUCE expressions may be of several types and consist of sequences of numbers, variables, operators, left and right parentheses and commas. The most common types are as follows:

3.1 Scalar Expressions

Using the arithmetic operations `+ - * / ^` (power) and parentheses, scalar expressions are composed from numbers, ordinary "scalar" variables (identifiers), array names with subscripts, operator or procedure names with arguments and statement expressions.

Examples:

```
x
x^3 - 2*y/(2*z^2 - df(x,z))
(p^2 + m^2)^(1/2)*log (y/m)
a(5) + b(i,q)
```

The symbol ** may be used as an alternative to the caret symbol (`^`) for forming powers, particularly in those systems that do not support a caret symbol.

Statement expressions, usually in parentheses, can also form part of a scalar expression, as in the example

```
w + (c:=x+y) + z .
```

When the algebraic value of an expression is needed, REDUCE determines it, starting with the algebraic values of the parts, roughly as follows:

Variables and operator symbols with an argument list have the algebraic values they were last assigned, or if never assigned stand for themselves. However, array elements have the algebraic values they were last assigned, or, if never assigned, are taken to be 0.

Procedures are evaluated with the values of their actual parameters.

In evaluating expressions, the standard rules of algebra are applied. Unfortunately, this algebraic evaluation of an expression is not as unambiguous as is numerical evaluation. This process is generally referred to as "simplification" in the sense that the evaluation usually but not always produces a simplified form for the expression.

There are many options available to the user for carrying out such simplification. If the user doesn't specify any method, the default method is used. The default evaluation of an expression involves expansion of the expression and collection of like terms, ordering of the terms, evaluation of derivatives and other functions and substitution for any expressions which have values assigned or declared (see assignments and LET statements). In many cases, this is all that the user needs.

The declarations by which the user can exercise some control over the way in which the evaluation is performed are explained in other sections. For example, if a real (floating point) number is encountered during evaluation, the system will normally convert it into a ratio of two integers. If the user wants to use real arithmetic, he can effect this by the command `on rounded;`. Other modes for coefficient arithmetic are described elsewhere.

If an illegal action occurs during evaluation (such as division by zero) or functions are called with the wrong number of arguments, and so on, an appropriate error message is generated.

3.2 Integer Expressions

These are expressions which, because of the values of the constants and variables in them, evaluate to whole numbers.

Examples:

```
2,      37 * 999,       (x + 3)^2 - x^2 - 6*x
```

are obviously integer expressions.

```
j + k - 2 * j^2
```

is an integer expression when J and K have values that are integers, or if not integers are such that "the variables and fractions cancel out", as in

```
k - 7/3 - j + 2/3 + 2*j^2.
```

3.3 Boolean Expressions

A boolean expression returns a truth value. In the algebraic mode of REDUCE, boolean expressions have the syntactical form:

```
<expression> <relational operator> <expression>
```

or

```
<boolean operator> (<arguments>)
```

or

```
<boolean expression> <logical operator>
<boolean expression>.
```

Parentheses can also be used to control the precedence of expressions.

In addition to the logical and relational operators defined earlier as infix operators, the following boolean operators are also defined:

`EVENP(U)`	determines if the number `U` is even or not;
`FIXP(U)`	determines if the expression `U` is integer or not;
`FREEOF(U,V)`	determines if the expression `U` does not contain the kernel `V` anywhere in its structure;
`NUMBERP(U)`	determines if `U` is a number or not;
`ORDP(U,V)`	determines if `U` is ordered ahead of `V` by some canonical ordering (based on the expression structure and an internal ordering of identifiers);
`PRIMEP(U)`	true if `U` is a prime object, i.e., any object other than 0 and plus or minus 1 which is only exactly divisible by itself or a unit.

Examples:

```
j<1
x>0  or  x=-2
numberp x
fixp x and evenp x
numberp x and x neq 0
```

Boolean expressions can only appear directly within IF, FOR, WHILE, and UNTIL statements, as described in other sections. Such expressions cannot be used in place of ordinary algebraic expressions, or assigned to a variable.

NB: For those familiar with symbolic mode, the meaning of some of these operators is different in that mode. For example, NUMBERP is true only for integers and reals in symbolic mode.

When two or more boolean expressions are combined with AND, they are evaluated one by one until a *false* expression is found. The rest are not evaluated. Thus

```
numberp x and numberp y and x>y
```

does not attempt to make the x>y comparison unless X and Y are both verified to be numbers.

Similarly, evaluation of a sequence of boolean expressions connected by OR stops as soon as a *true* expression is found.

NB: In a boolean expression, and in a place where a boolean expression is expected, the algebraic value 0 is interpreted as *false*, while all other algebraic values are converted to *true*. So in algebraic mode a procedure can be written for direct usage in boolean expressions, returning say 1 or 0 as its value as in

```
procedure polynomialp(u,x);
   if den(u)=1 and deg(u,x)>=1 then 1 else 0;
```

One can then use this in a boolean construct, such as

```
if polynomialp(q,z) and not polynomialp(q,y) then ...
```

In addition, any procedure that does not have a defined return value (for example, a block without a RETURN statement in it) has the boolean value *false*.

3.4 Equations

Equations are a particular type of expression with the syntax

```
<expression> = <expression>.
```

In addition to their role as boolean expressions, they can also be used as arguments to several operators (e.g., SOLVE), and can be returned as values.

Under normal circumstances, the right-hand-side of the equation is evaluated but not the left-hand-side. This also applies to any substitutions made by the SUB operator. If both sides are to be evaluated, the switch EVALLHSEQP should be turned on.

To facilitate the handling of equations, two selectors, LHS and RHS, which return the left- and right-hand sides of a equation respectively, are provided. For example,

```
lhs(a+b=c) -> a+b
```

and

```
rhs(a+b=c) -> c.
```

3.5 Proper Statements as Expressions

Several kinds of proper statements deliver an algebraic or numerical result of some kind, which can in turn be used as an expression or part of an expression. For example, an assignment statement itself has a value, namely the value assigned. So

```
2 * (x := a+b)
```

is equal to 2*(a+b), as well as having the "side-effect" of assigning the value a+b to X. In context,

```
y := 2 * (x := a+b);
```

sets X to a+b and Y to 2*(a+b).

The sections on the various proper statement types indicate which of these statements are also useful as expressions.

Chapter 4

Lists

A list is an object consisting of a sequence of other objects (including lists themselves), separated by commas and surrounded by braces. Examples of lists are:

```
{a,b,c}

{1,a-b,c=d}

{{a},{{b,c},d},e}.
```

The empty list is represented as

```
{}.
```

4.1 Operations on Lists

Several operators in the system return their results as lists, and a user can create new lists using braces and commas. Alternatively, one can use the operator LIST to construct a list. An important class of operations on lists are MAP and SELECT operations. For details, please refer to the chapters on MAP, SELECT and the FOR command. See also the documentation on the ASSIST package.

To facilitate the use of lists, a number of operators are also available for manipulating them. `PART(<list>,n)` for example will return the n^{th} element of a list. `LENGTH` will return the length of a list. Several operators are also defined uniquely for lists. For those familiar with them, these operators in fact mirror the operations defined for Lisp lists. These operators are as follows:

4.1.1 LIST

The operator LIST is an alternative to the usage of curly brackets. LIST accepts an arbitrary number of arguments and returns a list of its arguments. This operator is useful in cases where operators have to be passed as arguments. E.g.,

```
list(a,list(list(b,c),d),e);        ->  {{a},{{b,c},d},e}
```

4.1.2 FIRST

This operator returns the first member of a list. An error occurs if the argument is not a list, or the list is empty.

4.1.3 SECOND

`SECOND` returns the second member of a list. An error occurs if the argument is not a list or has no second element.

4.1.4 THIRD

This operator returns the third member of a list. An error occurs if the argument is not a list or has no third element.

4.1.5 REST

`REST` returns its argument with the first element removed. An error occurs if the argument is not a list, or is empty.

4.1.6 . (Cons) Operator

This operator adds ("conses") an expression to the front of a list. For example:

```
a . {b,c}      ->   {a,b,c}.
```

4.1.7 APPEND

This operator appends its first argument to its second to form a new list. *Examples:*

```
append({a,b},{c,d})      ->     {a,b,c,d}
append({{a,b}},{c,d})    ->     {{a,b},c,d}.
```

4.1.8 REVERSE

The operator REVERSE returns its argument with the elements in the reverse order. It only applies to the top level list, not any lower level lists that may occur. Examples are:

```
reverse({a,b,c})         ->     {c,b,a}
reverse({{a,b,c},d})     ->     {d,{a,b,c}}.
```

4.1.9 List Arguments of Other Operators

If an operator other than those specifically defined for lists is given a single argument that is a list, then the result of this operation will be a list in which that operator is applied to each element of the list. For example, the result of evaluating log{a,b,c} is the expression {LOG(A),LOG(B),LOG(C)}.

There are two ways to inhibit this operator distribution. Firstly, the switch LISTARGS, if on, will globally inhibit such distribution. Secondly, one can inhibit this distribution for a specific operator by the declaration LISTARGP. For example, with the declaration `listargp log`, log{a,b,c} would evaluate to LOG({A,B,C}).

If an operator has more than one argument, no such distribution occurs.

4.1.10 Caveats and Examples

Some of the natural list operations such as *member* or *delete* are available only after loading the package *ASSIST*.

Please note that a non-list as second argument to CONS (a "dotted pair" in LISP terms) is not allowed and causes an "invalid as list" error.

```
a := 17 . 4;

***** 17 4 invalid as list
```

Also, the initialization of a scalar variable is not the empty list - one has to set list type variables explicitly, as in the following example:

```
load_package assist;
```

```
procedure lotto (n,m);
 begin scalar list_1_n, luckies, hit;
    list_1_n := {};
    luckies := {};
    for k:=1:n do list_1_n := k . list_1_n;
    for k:=1:m do
      << hit := part(list_1_n,random(n-k+1) + 1);
         list_1_n := delete(hit,list_1_n);
         luckies := hit . luckies >>;
    return luckies;
 end;                          % In Germany, try lotto (49,6);
```

Another example: Find all coefficients of a multivariate polynomial with respect to a list of variables:

```
procedure allcoeffs(q,lis); % q : polynomial, lis: list of vars
   allcoeffs1 (list q,lis);

procedure allcoeffs1(q,lis);
  if lis={} then q else
    allcoeffs1(foreach qq in q join coeff(qq,first lis),
               rest lis);
```

Chapter 5

Statements

A statement is any combination of reserved words and expressions, and has the syntax

```
<statement> ::= <expression>|<proper statement>
```

A REDUCE program consists of a series of commands which are statements followed by a terminator:

```
<terminator> ::= ;|$
```

The division of the program into lines is arbitrary. Several statements can be on one line, or one statement can be freely broken onto several lines. If the program is run interactively, statements ending with ; or $ are not processed until an end-of-line character is encountered. This character can vary from system to system, but is normally the **Return** key on an ASCII terminal. Specific systems may also use additional keys as statement terminators.

If a statement is a proper statement, the appropriate action takes place.

Depending on the nature of the proper statement some result or response may or may not be printed out, and the response may or may not depend on the terminator used.

If a statement is an expression, it is evaluated. If the terminator is a semicolon, the result is printed. If the terminator is a dollar sign, the result is not printed. Because it is not usually possible to know in advance how large an expression will be, no explicit format statements are offered to the user. However, a variety of output declarations are available so that the output can be produced in different forms. These output declarations are explained in Section 8.3.3.

The following sub-sections describe the types of proper statements in REDUCE.

5.1 Assignment Statements

These statements have the syntax

```
<assignment statement> ::= <expression> := <expression>
```

The `<expression>` on the left side is normally the name of a variable, an operator symbol with its list of arguments filled in, or an array name with the proper number of integer subscript values within the array bounds. For example:

```
a1 := b + c
h(1,m) := x-2*y              (where h is an operator)
k(3,5) := x-2*y              (where k is a 2-dim. array)
```

More general assignments such as `a+b := c` are also allowed. The effect of these is explained in Section 10.2.5.

An assignment statement causes the expression on the right-hand-side to be evaluated. If the left-hand-side is a variable, the value of the right-hand-side is assigned to that unevaluated variable. If the left-hand-side is an operator or array expression, the arguments of that operator or array are evaluated, but no other simplification done. The evaluated right-hand-side is then assigned to the resulting expression. For example, if `A` is a single-dimensional array, `a(1+1) := b` assigns the value `B` to the array element `a(2)`.

If a semicolon is used as the terminator when an assignment is issued as a command (i.e. not as a part of a group statement or procedure or other similar construct), the left-hand side symbol of the assignment statement is printed out, followed by a "`:=`", followed by the value of the expression on the right.

It is also possible to write a multiple assignment statement:

```
<expression> := ... := <expression> := <expression>
```

In this form, each `<expression>` but the last is set to the value of the last `<expression>`. If a semicolon is used as a terminator, each expression except the last is printed followed by a "`:=`" ending with the value of the last expression.

5.1.1 Set Statement

In some cases, it is desirable to perform an assignment in which *both* the left- and right-hand sides of an assignment are evaluated. In this case, the SET statement can be used with the syntax:

```
SET(<expression>,<expression>);
```

For example, the statements

```
j := 23;
set(mkid(a,j),x);
```

assigns the value X to A23.

5.2 Group Statements

The group statement is a construct used where REDUCE expects a single statement, but a series of actions needs to be performed. It is formed by enclosing one or more statements (of any kind) between the symbols << and >>, separated by semicolons or dollar signs - it doesn't matter which. The statements are executed one after another.

Examples will be given in the sections on IF and other types of statements in which the << ... >> construct is useful.

If the last statement in the enclosed group has a value, then that is also the value of the group statement. Care must be taken not to have a semicolon or dollar sign after the last grouped statement, if the value of the group is relevant: such an extra terminator causes the group to have the value NIL or zero.

5.3 Conditional Statements

The conditional statement has the following syntax:

```
<conditional statement> ::=
   IF <boolean expression> THEN <statement> [ELSE <statement>]
```

The boolean expression is evaluated. If this is *true*, the first <statement> is executed. If it is *false*, the second is.

Examples:

```
if x=5 then a:=b+c else d:=e+f

if x=5 and numberp y
   then <<ff:=q1; a:=b+c>>
   else <<ff:=q2; d:=e+f>>
```

Note the use of the group statement.
Conditional statements associate to the right; i.e.,

```
IF <a> THEN <b> ELSE IF <c> THEN <d> ELSE <e>
```

is equivalent to:

```
IF <a> THEN <b> ELSE (IF <c> THEN <d> ELSE <e>)
```

In addition, the construction

```
IF <a> THEN IF <b> THEN <c> ELSE <d>
```

parses as

```
IF <a> THEN (IF <b> THEN <c> ELSE <d>).
```

If the value of the conditional statement is of primary interest, it is often called a conditional expression instead. Its value is the value of whichever statement was executed. (If the executed statement has no value, the conditional expression has no value or the value 0, depending on how it is used.)

Examples:

```
a:=if x<5 then 123 else 456;
b:=u + v^(if numberp z then 10*z  else 1) + w;
```

If the value is of no concern, the ELSE clause may be omitted if no action is required in the *false* case.

```
if x=5 then a:=b+c;
```

Note: As explained in Section 3.3,a if a scalar or numerical expression is used in place of the boolean expression – for example, a variable is written there – the *true* alternative is followed unless the expression has the value 0.

5.4 FOR Statements

The FOR statement is used to define a variety of program loops. Its general syntax is as follows:

$$\texttt{FOR}\left\{\begin{array}{l}\langle var\rangle\ \texttt{:=}\ \langle number\rangle\ \left\{\begin{array}{c}\texttt{STEP}\ \langle number\rangle\ \texttt{UNTIL}\\ \texttt{:}\end{array}\right\}\langle number\rangle \\ \texttt{EACH}\ \langle var\rangle\ \left\{\begin{array}{c}\texttt{IN}\\ \texttt{ON}\end{array}\right\}\ \langle list\rangle\end{array}\right\}\langle action\rangle\ \langle exprn\rangle$$

where

$$\langle action\rangle\ \texttt{::= do|product|sum|collect|join.}$$

The assignment form of the FOR statement defines an iteration over the indicated numerical range. If expressions that do not evaluate to numbers are used in the designated places, an error will result.

The FOR EACH form of the FOR statement is designed to iterate down a list. Again, an error will occur if a list is not used.

The action DO means that <exprn> is simply evaluated and no value kept; the statement returning 0 in this case (or no value at the top level). COLLECT means that the results of evaluating <exprn> each time are linked together to make a list, and JOIN means that the values of <exprn> are themselves lists that are joined to make one list (similar to CONC in Lisp). Finally, PRODUCT and SUM form the respective combined value out of the values of <exprn>.

In all cases, <exprn> is evaluated algebraically within the scope of the current value of <var>. If <action> is DO, then nothing else happens. In other cases, <action> is a binary operator that causes a result to be built up and returned by FOR. In those cases, the loop is initialized to a default value (0 for SUM, 1 for PRODUCT, and an empty list for the other actions). The test for the end condition is made before any action is taken. As in Pascal, if the variable is out of range in the assignment case, or the <list> is empty in the FOR EACH case, <exprn> is not evaluated at all.

Examples:

1. If A, B have been declared to be arrays, the following stores 5^2 through 10^2 in A(5) through A(10), and at the same time stores the cubes in the B array:

```
for i := 5 step 1 until 10 do <<a(i):=i^2; b(i):=i^3>>
```

2. As a convenience, the common construction

```
STEP 1 UNTIL
```

may be abbreviated to a colon. Thus, instead of the above we could write:

```
for i := 5:10 do <<a(i):=i^2; b(i):=i^3>>
```

3. The following sets C to the sum of the squares of 1,3,5,7,9; and D to the expression x*(x+1)*(x+2)*(x+3)*(x+4):

```
c := for j:=1 step 2 until 9 sum j^2;
d := for k:=0 step 1 until 4 product (x+k);
```

4. The following forms a list of the squares of the elements of the list {a,b,c}:

```
for each x in {a,b,c} collect x^2;
```

5. The following forms a list of the listed squares of the elements of the list {a,b,c} (i.e., {{A^2},{B^2},{C^2}}):

```
for each x in {a,b,c} collect {x^2};
```

6. The following also forms a list of the squares of the elements of the list {a,b,c}, since the JOIN operation joins the individual lists into one list:

```
for each x in {a,b,c} join {x^2};
```

The control variable used in the FOR statement is actually a new variable, not related to the variable of the same name outside the FOR statement. In other words, executing a statement for i:= ... doesn't change the system's assumption that $i^2 = -1$. Furthermore, in algebraic mode, the value of the control variable is substituted in <exprn> only if it occurs explicitly in that expression. It will not replace a variable of the same name in the value of that expression. For example:

```
b := a; for a := 1:2 do write b;
```

prints A twice, not 1 followed by 2.

5.5 WHILE ... DO

The FOR ...DO feature allows easy coding of a repeated operation in which the number of repetitions is known in advance. If the criterion for repetition is more complicated, WHILE ...DO can often be used. Its syntax is:

```
WHILE <boolean expression> DO <statement>
```

The `WHILE ...DO` controls the single statement following `DO`. If several statements are to be repeated, as is almost always the case, they must be grouped using the $<< \ldots >>$ or `BEGIN ...END` as in the example below.

The `WHILE` condition is tested each time *before* the action following the `DO` is attempted. If the condition is false to begin with, the action is not performed at all. Make sure that what is to be tested has an appropriate value initially.

Example:

Suppose we want to add up a series of terms, generated one by one, until we reach a term which is less than 1/1000 in value. For our simple example, let us suppose the first term equals 1 and each term is obtained from the one before by taking one third of it and adding one third its square. We would write:

```
ex:=0; term:=1;
while num(term - 1/1000) >= 0  do
        <<ex := ex+term; term:=(term + term^2)/3>>;
ex;
```

As long as `TERM` is greater than or equal to (>=) 1/1000 it will be added to `EX` and the next `TERM` calculated. As soon as `TERM` becomes less than 1/1000 the `WHILE` test fails and the `TERM` will not be added.

5.6 REPEAT ... UNTIL

`REPEAT ...UNTIL` is very similar in purpose to `WHILE ...DO`. Its syntax is:

```
REPEAT <statement> UNTIL <boolean expression>
```

(PASCAL users note: Only a single statement – usually a group statement – is allowed between the `REPEAT` and the `UNTIL`.)

There are two essential differences:

1. The test is performed *after* the controlled statement (or group of statements) is executed, so the controlled statement is always executed at least once.

2. The test is a test for when to stop rather than when to continue, so its "polarity" is the opposite of that in `WHILE ...DO`.

As an example, we rewrite the example from the WHILE ...DO section:

```
ex:=0; term:=1;
repeat <<ex := ex+term; term := (term + term^2)/3>>
   until num(term - 1/1000) < 0;
ex;
```

In this case, the answer will be the same as before, because in neither case is a term added to EX which is less than 1/1000.

5.7 Compound Statements

Often the desired process can best (or only) be described as a series of steps to be carried out one after the other. In many cases, this can be achieved by use of the group statement. However, each step often provides some intermediate result, until at the end we have the final result wanted. Alternatively, iterations on the steps are needed that are not possible with constructs such as WHILE or REPEAT statements. In such cases the steps of the process must be enclosed between the words BEGIN and END forming what is technically called a *block* or *compound* statement. Such a compound statement can in fact be used wherever a group statement appears. The converse is not true: BEGIN ...END can be used in ways that << ... >> cannot.

If intermediate results must be formed, local variables must be provided in which to store them. *Local* means that their values are deleted as soon as the block's operations are complete, and there is no conflict with variables outside the block that happen to have the same name. Local variables are created by a SCALAR declaration immediately after the BEGIN:

```
scalar a,b,c,z;
```

If more convenient, several SCALAR declarations can be given one after another:

```
scalar a,b,c;
scalar z;
```

In place of SCALAR one can also use the declarations INTEGER or REAL. In the present version of REDUCE variables declared INTEGER are expected to have only integer values, and are initialized to 0. REAL variables on the other hand are currently treated as algebraic mode SCALARs.

CAUTION: INTEGER, REAL and SCALAR declarations can only be given immediately after a BEGIN. An error will result if they are used after other statements

in a block (including ARRAY and OPERATOR declarations, which are global in scope), or outside the top-most block (e.g., at the top level). All variables declared SCALAR are automatically initialized to zero in algebraic mode (NIL in symbolic mode).

Any symbols not declared as local variables in a block refer to the variables of the same name in the current calling environment. In particular, if they are not so declared at a higher level (e.g., in a surrounding block or as parameters in a calling procedure), their values can be permanently changed.

Following the SCALAR declaration(s), if any, write the statements to be executed, one after the other, separated by delimiters (e.g., ; or $) (it doesn't matter which). However, from a stylistic point of view, ; is preferred.

The last statement in the body, just before END, need not have a terminator (since the BEGIN ...END are in a sense brackets confining the block statements). The last statement must also be the command RETURN followed by the variable or expression whose value is to be the value returned by the procedure. If the RETURN is omitted (or nothing is written after the word RETURN) the procedure will have no value or the value zero, depending on how it is used (and NIL in symbolic mode). Remember to put a terminator after the END.

Example:

Given a previously assigned integer value for N, the following block will compute the Legendre polynomial of degree N in the variable X:

```
begin scalar seed,deriv,top,fact;
   seed:=1/(y^2 - 2*x*y +1)^(1/2);
   deriv:=df(seed,y,n);
   top:=sub(y=0,deriv);
   fact:=for i:=1:n product i;
   return top/fact
end;
```

5.7.1 Compound Statements with GO TO

It is possible to have more complicated structures inside the BEGIN ...END brackets than indicated in the previous example. That the individual lines of the program need not be assignment statements, but could be almost any other kind of statement or command, needs no explanation. For example, conditional statements, and WHILE and REPEAT constructions, have an obvious role in defining more intricate blocks.

If these structured constructs don't suffice, it is possible to use labels and

GO TOs within a compound statement, and also to use RETURN in places within the block other than just before the END. The following subsections discuss these matters in detail. For many readers the following example, presenting one possible definition of a process to calculate the factorial of N for preassigned N will suffice:

Example:

```
begin scalar m;
    m:=1;
 l: if n=0 then return m;
    m:=m*n;
    n:=n-1;
    go to l
end;
```

5.7.2 Labels and GO TO Statements

Within a BEGIN ...END compound statement it is possible to label statements, and transfer to them out of sequence using GO TO statements. Only statements on the top level inside compound statements can be labeled, not ones inside subsidiary constructions like << ... >>, IF ... THEN ..., WHILE ... DO ..., etc.

Labels and GO TO statements have the syntax:

```
<go to statement> ::= GO TO <label> | GOTO <label>
<label> ::= <identifier>
<labeled statement> ::= <label>:<statement>
```

Note that statement names cannot be used as labels.

While GO TO is an unconditional transfer, it is frequently used in conditional statements such as

```
if x>5 then go to abcd;
```

giving the effect of a conditional transfer.

Transfers using GO TOs can only occur within the block in which the GO TO is used. In other words, you cannot transfer from an inner block to an outer block using a GO TO. However, if a group statement occurs within a compound statement, it is possible to jump out of that group statement to a point within the compound statement using a GO TO.

5.7.3 RETURN Statements

The value corresponding to a BEGIN ...END compound statement, such as a procedure body, is normally 0 (NIL in symbolic mode). By executing a RETURN statement in the compound statement a different value can be returned. After a RETURN statement is executed, no further statements within the compound statement are executed.

Examples:

```
return x+y;
return m;
return;
```

Note that parentheses are not required around the x+y, although they are permitted. The last example is equivalent to return 0 or return nil, depending on whether the block is used as part of an expression or not.

Since RETURN actually moves up only one block level, in a sense the casual user is not expected to understand, we tabulate some cautions concerning its use.

1. RETURN can be used on the top level inside the compound statement, i.e. as one of the statements bracketed together by the BEGIN ...END
2. RETURN can be used within a top level $<< \ldots >>$ construction within the compound statement. In this case, the RETURN transfers control out of both the group statement and the compound statement.
3. RETURN can be used within an IF ...THEN ...ELSE ...on the top level within the compound statement.

NOTE: At present, there is no construct provided to permit early termination of a FOR, WHILE, or REPEAT statement. In particular, the use of RETURN in such cases results in a syntax error. For example,

```
begin scalar y;
   y := for i:=0:99 do if a(i)=x then return b(i);
   ...
```

will lead to an error.

Chapter 6

Commands and Declarations

A command is an order to the system to do something. Some commands cause visible results (such as calling for input or output); others, usually called declarations, set options, define properties of variables, or define procedures. Commands are formally defined as a statement followed by a terminator

```
<command> ::= <statement> <terminator>
<terminator> ::= ;|$
```

Some REDUCE commands and declarations are described in the following sub-sections.

6.1 Array Declarations

Array declarations in REDUCE are similar to FORTRAN dimension statements. For example:

```
array a(10),b(2,3,4);
```

Array indices each range from 0 to the value declared. An element of an array is referred to in standard FORTRAN notation, e.g. `A(2)`.

We can also use an expression for defining an array bound, provided the value of the expression is a positive integer. For example, if `X` has the value 10 and `Y` the value 7 then `array c(5*x+y)` is the same as `array c(57)`.

If an array is referenced by an index outside its range, an error occurs. If the array is to be one-dimensional, and the bound a number or a variable (not a more general expression) the parentheses may be omitted:

```
array a 10, c 57;
```

The operator LENGTH applied to an array name returns a list of its dimensions.

All array elements are initialized to 0 at declaration time. In other words, an array element has an *instant evaluation* property and cannot stand for itself. If this is required, then an operator should be used instead.

Array declarations can appear anywhere in a program. Once a symbol is declared to name an array, it can not also be used as a variable, or to name an operator or a procedure. It can however be re-declared to be an array, and its size may be changed at that time. An array name can also continue to be used as a parameter in a procedure, or a local variable in a compound statement, although this use is not recommended, since it can lead to user confusion over the type of the variable.

Arrays once declared are global in scope, and so can then be referenced anywhere in the program. In other words, unlike arrays in most other languages, a declaration within a block (or a procedure) does not limit the scope of the array to that block, nor does the array go away on exiting the block (use CLEAR instead for this purpose).

6.2 Mode Handling Declarations

The ON and OFF declarations are available to the user for controlling various system options. Each option is represented by a *switch* name. ON and OFF take a list of switch names as argument and turn them on and off respectively, e.g.,

```
on time;
```

causes the system to print a message after each command giving the elapsed CPU time since the last command, or since TIME was last turned off, or the session began. Another useful switch with interactive use is DEMO, which causes the system to pause after each command in a file (with the exception of comments) until a **Return** is typed on the terminal. This enables a user to set up a demonstration file and step through it command by command.

As with most declarations, arguments to ON and OFF may be strung together separated by commas. For example,

```
off time,demo;
```

will turn off both the time messages and the demonstration switch.

We note here that while most ON and OFF commands are obeyed almost instantaneously, some trigger time-consuming actions such as reading in necessary modules from secondary storage.

A diagnostic message is printed if ON or OFF are used with a switch that is not known to the system. For example, if you misspell DEMO and type

```
on demq;
```

you will get the message

```
***** DEMQ not defined as switch.
```

6.3 END

The identifier END has two separate uses.

1) Its use in a BEGIN ...END bracket has been discussed in connection with compound statements.

2) Files to be read using IN should end with an extra END; command. The reason for this is explained in the section on the IN command. This use of END does not allow an immediately preceding END (such as the END of a procedure definition), so we advise using ;END; there.

6.4 BYE Command

The command BYE; (or alternatively QUIT;) stops the execution of REDUCE, closes all open output files, and returns you to the calling program (usually the operating system). Your REDUCE session is normally destroyed.

6.5 SHOWTIME Command

SHOWTIME; prints the elapsed time since the last call of this command or, on its first call, since the current REDUCE session began. The time is normally given in milliseconds and gives the time as measured by a system clock. The operations covered by this measure are system dependent.

6.6 DEFINE Command

The command DEFINE allows a user to supply a new name for any identifier or replace it by any well-formed expression. Its argument is a list of expressions of the form

```
<identifier> = <number>|<identifier>|<operator>|
                <reserved word>|<expression>
```

Example:

```
define be==,x=y+z;
```

means that BE will be interpreted as an equal sign, and X as the expression y+z from then on. This renaming is done at parse time, and therefore takes precedence over any other replacement declared for the same identifier. It stays in effect until the end of the REDUCE run.

The identifiers ALGEBRAIC and SYMBOLIC have properties which prevent DEFINE from being used on them. To define ALG to be a synonym for ALGEBRAIC, use the more complicated construction

```
put('alg,'newnam,'algebraic);
```

Chapter 7

Built-in Prefix Operators

In the following subsections are descriptions of the most useful prefix operators built into REDUCE that are not defined in other sections (such as substitution operators). Some are fully defined internally as procedures; others are more nearly abstract operators, with only some of their properties known to the system.

In many cases, an operator is described by a prototypical header line as follows. Each formal parameter is given a name and followed by its allowed type. The names of classes referred to in the definition are printed in lower case, and parameter names in upper case. If a parameter type is not commonly used, it may be a specific set enclosed in brackets $\{ \ldots \}$. Operators that accept formal parameter lists of arbitrary length have the parameter and type class enclosed in square brackets indicating that zero or more occurrences of that argument are permitted. Optional parameters and their type classes are enclosed in angle brackets.

7.1 Numerical Operators

REDUCE includes a number of functions that are analogs of those found in most numerical systems. With numerical arguments, such functions return the expected result. However, they may also be called with non-numerical arguments. In such cases, except where noted, the system attempts to simplify the expression as far as it can. In such cases, a residual expression involving the original operator usually remains. These operators are as follows:

7.1.1 ABS

`ABS` returns the absolute value of its single argument, if that argument has

a numerical value. A non-numerical argument is returned as an absolute value, with an overall numerical coefficient taken outside the absolute value operator. For example:

```
abs(-3/4)      ->  3/4
abs(2a)        ->  2*ABS(A)
abs(i)         ->  1
abs(-x)        ->  ABS(X)
```

7.1.2 CEILING

This operator returns the ceiling (i.e., the least integer greater than the given argument) if its single argument has a numerical value. A non-numerical argument is returned as an expression in the original operator. For example:

```
ceiling(-5/4) ->  -1
ceiling(-a)   ->  CEILING(-A)
```

7.1.3 CONJ

This returns the complex conjugate of an expression, if that argument has an numerical value. A non-numerical argument is returned as an expression in the operators `REPART` and `IMPART`. For example:

```
conj(1+i)      -> 1-I
conj(a+i*b)    -> REPART(A) - REPART(B)*I - IMPART(A)*I
 - IMPART(B)
```

7.1.4 FACTORIAL

If the single argument of `FACTORIAL` evaluates to a non-negative integer, its factorial is returned. Otherwise an expression involving `FACTORIAL` is returned. For example:

```
factorial(5)  ->  120
factorial(a)  ->  FACTORIAL(A)
```

7.1.5 FIX

This operator returns the fixed value (i.e., the integer part of the given argument) if its single argument has a numerical value. A non-numerical argu-

ment is returned as an expression in the original operator. For example:

```
fix(-5/4)   ->  -1
fix(a)      ->  FIX(A)
```

7.1.6 FLOOR

This operator returns the floor (i.e., the greatest integer less than the given argument) if its single argument has a numerical value. A non-numerical argument is returned as an expression in the original operator. For example:

```
floor(-5/4)   ->  -2
floor(a)      ->  FLOOR(A)
```

7.1.7 IMPART

This operator returns the imaginary part of an expression, if that argument has an numerical value. A non-numerical argument is returned as an expression in the operators REPART and IMPART. For example:

```
impart(1+i)   -> 1
impart(a+i*b) -> REPART(B) + IMPART(A)
```

7.1.8 MAX/MIN

MAX and MIN can take an arbitrary number of expressions as their arguments. If all arguments evaluate to numerical values, the maximum or minimum of the argument list is returned. If any argument is non-numeric, an appropriately reduced expression is returned. For example:

```
max(2,-3,4,5) ->  5
min(2,-2)     ->  -2.
max(a,2,3)    ->  MAX(A,3)
min(x)        ->  X
```

MAX or MIN of an empty list returns 0.

7.1.9 NEXTPRIME

NEXTPRIME returns the next prime greater than its integer argument, using a probabilistic algorithm. A type error occurs if the value of the argument is

not an integer. For example:

```
nextprime(5)       ->  7
nextprime(-2)      ->  2
nextprime(-7)      -> -5
nextprime 1000000 -> 1000003
```

whereas `nextprime(a)` gives a type error.

7.1.10 RANDOM

`random(`n`)` returns a random number r in the range $0 \leq r < n$. A type error occurs if the value of the argument is not a positive integer in algebraic mode, or positive number in symbolic mode. For example:

```
random(5)          ->    3
random(1000)       ->  191
```

whereas `random(a)` gives a type error.

7.1.11 RANDOM_NEW_SEED

`random_new_seed(`n`)` reseeds the random number generator to a sequence determined by the integer argument n. It can be used to ensure that a repeatable pseudo-random sequence will be delivered regardless of any previous use of `RANDOM`, or can be called early in a run with an argument derived from something variable (such as the time of day) to arrange that different runs of a REDUCE program will use different random sequences. When a fresh copy of REDUCE is first created it is as if `random_new_seed(1)` has been obeyed.

A type error occurs if the value of the argument is not a positive integer.

7.1.12 REPART

This returns the real part of an expression, if that argument has an numerical value. A non-numerical argument is returned as an expression in the operators `REPART` and `IMPART`. For example:

```
repart(1+i)   -> 1
repart(a+i*b) -> REPART(A) - IMPART(B)
```

7.1.13 ROUND

This operator returns the rounded value (i.e, the nearest integer) of its single argument if that argument has a numerical value. A non-numeric argument is returned as an expression in the original operator. For example:

```
round(-5/4)   ->  -1
round(a)      ->  ROUND(A)
```

7.1.14 SIGN

SIGN tries to evaluate the sign of its argument. If this is possible SIGN returns one of 1, 0 or -1. Otherwise, the result is the original form or a simplified variant. For example:

```
sign(-5)      ->  -1
sign(-a^2*b)  ->  -SIGN(B)
```

Note that even powers of formal expressions are assumed to be positive only as long as the switch COMPLEX is off.

7.2 Mathematical Functions

REDUCE knows that the following represent mathematical functions that can take arbitrary scalar expressions as their single argument:

```
ACOS ACOSH ACOT ACOTH ACSC ACSCH ASEC ASECH ASIN ASINH
ATAN ATANH ATAN2 CI COS COSH COT COTH CSC CSCH DILOG EI EXP
HYPOT LN LOG LOGB LOG10 SEC SECH SI SIN SINH SQRT TAN TANH
```

where LOG is the natural logarithm (and equivalent to LN), and LOGB has two arguments of which the second is the logarithmic base.

The derivatives of all these functions are also known to the system.

REDUCE knows various elementary identities and properties of these functions. For example:

```
cos(-x) = cos(x)            sin(-x) = - sin (x)
cos(n*pi) = (-1)^n          sin(n*pi) = 0
log(e)  = 1                 e^(i*pi/2) = i
log(1)  = 0                 e^(i*pi) = -1
log(e^x) = x                e^(3*i*pi/2) = -i
```

Beside these identities, there are a lot of simplifications for elementary functions defined in the REDUCE system as rulelists. In order to view these, the SHOWRULES operator can be used, e.g.

```
      SHOWRULES tan;

{tan(~n*arbint(~i)*pi + ~(~ x)) => tan(x) when fixp(n),

 tan(~x)

  => trigquot(sin(x),cos(x)) when knowledge_about(sin,x,tan)

 ,

      ~x + ~(~ k)*pi
 tan(----------------)
    ~d

     x                              k    1
  =>  - cot(---) when x freeof pi and abs(---)=---,
     d                              d    2

      ~(~ w) + ~(~ k)*pi           w + remainder(k,d)*pi
 tan(--------------------) => tan(----------------------)
    ~(~ d)                          d

       k
 when w freeof pi and ratnump(---) and fixp(k)
       d

   k
  and abs(---)>=1,
   d

 tan(atan(~x)) => x,

     2
 df(tan(~x),~x) => 1 + tan(x) }
```

For further simplification, especially of expressions involving trigonometric functions, see the TRIGSIMP package documentation.

Functions not listed above may be defined in the special functions package SPECFN.

The user can add further rules for the reduction of expressions involving these operators by using the LET command.

In many cases it is desirable to expand product arguments of logarithms, or collect a sum of logarithms into a single logarithm. Since these are inverse operations, it is not possible to provide rules for doing both at the same time and preserve the REDUCE concept of idempotent evaluation. As an alternative, REDUCE provides two switches EXPANDLOGS and COMBINELOGS to carry out these operations. Both are off by default, and are subject to the value of the switch PRECISE. This switch is on by default and prevents modifications that may be false in a complex domain. Thus to expand LOG(3*Y) into a sum of logs, one can say

```
ON EXPANDLOGS; LOG(3*Y);
```

whereas to expand LOG(X*Y) into a sum of logs, one needs to say

```
OFF PRECISE; ON EXPANDLOGS; LOG(X*Y);
```

To combine this sum into a single log:

```
OFF PRECISE; ON COMBINELOGS; LOG(X) + LOG(Y);
```

These switches affect the logarithmic functions LOG10 (base 10) and LOGB (arbitrary base) as well.

At the present time, it is possible to have both switches on at once, which could lead to infinite recursion. However, an expression is switched from one form to the other in this case. Users should not rely on this behavior, since it may change in the next release.

The current version of REDUCE does a poor job of simplifying surds. In particular, expressions involving the product of variables raised to non-integer powers do not usually have their powers combined internally, even though they are printed as if those powers were combined. For example, the expression

```
x^(1/3)*x^(1/6);
```

will print as

```
SQRT(X)
```

but will have an internal form containing the two exponentiated terms. If you now subtract `sqrt(x)` from this expression, you will *not* get zero. Instead, the confusing form

```
SQRT(X) - SQRT(X)
```

will result. To combine such exponentiated terms, the switch `COMBINEEXPT` should be turned on.

The square root function can be input using the name `SQRT`, or the power operation `^(1/2)`. On output, unsimplified square roots are normally represented by the operator `SQRT` rather than a fractional power. With the default system switch settings, the argument of a square root is first simplified, and any divisors of the expression that are perfect squares taken outside the square root argument. The remaining expression is left under the square root. Thus the expression

```
 sqrt(-8a^2*b)
```

becomes

```
2*a*sqrt(-2*b).
```

Note that such simplifications can cause trouble if `A` is eventually given a value that is a negative number. If it is important that the positive property of the square root and higher even roots always be preserved, the switch `PRECISE` should be set on (the default value). This causes any non-numerical factors taken out of surds to be represented by their absolute value form. With `PRECISE` on then, the above example would become

```
2*abs(a)*sqrt(-2*b).
```

However, this is incorrect in the complex domain, where the $\sqrt{x^2}$ is not identical to $|x|$. To avoid the above simplification, the switch `PRECISE_COMPLEX` should be set on (default is off). For example:

```
on precise_complex; sqrt(-8a^2*b);
```

yields the output

```
             2
2*sqrt( - 2*a *b)
```

The statement that REDUCE knows very little about these functions applies only in the mathematically exact `off rounded` mode. If `ROUNDED` is on, any of the functions

```
ACOS ACOSH ACOT ACOTH ACSC ACSCH ASEC ASECH ASIN ASINH
ATAN ATANH ATAN2 COS COSH COT COTH CSC CSCH EXP HYPOT
LN LOG LOGB LOG10 SEC SECH SIN SINH SQRT TAN TANH
```

when given a numerical argument has its value calculated to the current degree of floating point precision. In addition, real (non-integer valued) powers of numbers will also be evaluated.

If the `COMPLEX` switch is turned on in addition to `ROUNDED`, these functions will also calculate a real or complex result, again to the current degree of floating point precision, if given complex arguments. For example, with `on rounded,complex;`

```
2.3^(5.6i)    ->   -0.0480793490914 - 0.998843519372*I
cos(2+3i)     ->   -4.18962569097 - 9.10922789376*I
```

7.3 DF Operator

The operator `DF` is used to represent partial differentiation with respect to one or more variables. It is used with the syntax:

```
DF(EXPRN:algebraic[,VAR:kernel<,NUM:integer>]):algebraic.
```

The first argument is the expression to be differentiated. The remaining arguments specify the differentiation variables and the number of times they are applied.

The number `NUM` may be omitted if it is 1. For example,

`df(y,x)` = $\partial y/\partial x$
`df(y,x,2)` = $\partial^2 y/\partial x^2$
`df(y,x1,2,x2,x3,2)` = $\partial^5 y/\partial x_1^2\,\partial x_2 \partial x_3^2$.

The evaluation of `df(y,x)` proceeds as follows: first, the values of `Y` and `X` are found. Let us assume that `X` has no assigned value, so its value is `X`. Each term or other part of the value of `Y` that contains the variable `X` is differentiated by the standard rules. If `Z` is another variable, not `X` itself, then its derivative with respect to `X` is taken to be 0, unless `Z` has previously been declared to `DEPEND` on `X`, in which case the derivative is reported as the symbol `df(z,x)`.

7.3.1 Switches influencing differentiation

Consider `df(u,x,y,z)`. If non of `x,y,z` are equal to `u` then the order of differentiation is commuted into a canonical form, unless the switch `NOCOMMUTEDF` is turned on (default is off). if at least one of `x,y,z` is equal to `u` then the order of differentiation is *not* commuted and the derivative is emphnot simplified to zero, unless the switch `COMMUTEDF` is turned on. It is off by default.

If `COMMUTEDF` is off and the switch `SIMPNONCOMDF` is on then simplify as follows:

```
DF(U,X,U)         ->  DF(U,X,2) / DF(U,X)
DF(U,X,N,U)       ->  DF(U,X,N+1) / DF(U,X)
```

provided `U` depends only on the one variable `X`. This simplification removes the non-commutative aspect of the derivative.

If the switch `EXPANDDF` is turned on then REDUCE uses the chain rule to expand symbolic derivatives of indirectly dependent variables provided the result is unambiguous, i.e. provided there is no direct dependence. It is off by default. Thus, for example, given

```
DEPEND F,U,V; DEPEND {U,V},X;
ON EXPANDDF;
DF(F,X)           -> DF(F,U)*DF(U,X) + DF(F,V)*DF(V,X)
```

whereas after

```
DEPEND F,X;
```

`DF(F,X)` does not expand at all (since the result would be ambiguous and the algorithm would loop).

Turning on the switch `ALLOWDFINT` allows ”differentiation under the integral sign”, i.e.

```
DF(INT(Y, X), V) -> INT(DF(Y, V), X)
```

if this results in a simplification. If the switch `DFINT` is also turned on then this happens regardless of whether the result simplifies. Both switches are off by default.

7.3.2 Adding Differentiation Rules

The LET statement can be used to introduce rules for differentiation of user-defined operators. Its general form is

```
FOR ALL <var1>,...,<varn>
     LET DF(<operator><varlist>,<vari>)=<expression>
```

where <varlist> ::= (<var1>,...,<varn>), and <var1>,...,<varn> are the dummy variable arguments of <operator>.

An analogous form applies to infix operators.

Examples:

```
for all x let df(tan x,x)= 1 + tan(x)^2;
```

(This is how the tan differentiation rule appears in the REDUCE source.)

```
for all x,y let df(f(x,y),x)=2*f(x,y),
df(f(x,y),y)=x*f(x,y);
```

Notice that all dummy arguments of the relevant operator must be declared arbitrary by the FOR ALL command, and that rules may be supplied for operators with any number of arguments. If no differentiation rule appears for an argument in an operator, the differentiation routines will return as result an expression in terms of DF. For example, if the rule for the differentiation with respect to the second argument of F is not supplied, the evaluation of df(f(x,z),z) would leave this expression unchanged. (No DEPEND declaration is needed here, since f(x,z) obviously "depends on" Z.)

Once such a rule has been defined for a given operator, any future differentiation rules for that operator must be defined with the same number of arguments for that operator, otherwise we get the error message

```
Incompatible DF rule argument length for <operator>
```

7.4 INT Operator

INT is an operator in REDUCE for indefinite integration using a combination of the Risch-Norman algorithm and pattern matching. It is used with the syntax:

```
INT(EXPRN:algebraic,VAR:kernel):algebraic.
```

This will return correctly the indefinite integral for expressions comprising polynomials, log functions, exponential functions and tan and atan. The arbitrary constant is not represented. If the integral cannot be done in closed terms, it returns a formal integral for the answer in one of two ways:

1. It returns the input, `INT(...,...)` unchanged.
2. It returns an expression involving `INT`s of some other functions (sometimes more complicated than the original one, unfortunately).

Rational functions can be integrated when the denominator is factorizable by the program. In addition it will attempt to integrate expressions involving error functions, dilogarithms and other trigonometric expressions. In these cases it might not always succeed in finding the solution, even if one exists.

Examples:

```
int(log(x),x) ->  X*(LOG(X) - 1),
int(e^x,x)    ->  E**X.
```

The program checks that the second argument is a variable and gives an error if it is not.

Note: If the `int` operator is called with 4 arguments, REDUCE will implicitly call the definite integration package (DEFINT) and this package will interpret the third and fourth arguments as the lower and upper limit of integration, respectively. For details, consult the documentation on the DEFINT package.

7.4.1 Options

The switch `TRINT` when on will trace the operation of the algorithm. It produces a great deal of output in a somewhat illegible form, and is not of much interest to the general user. It is normally off.

If the switch `FAILHARD` is on the algorithm will terminate with an error if the integral cannot be done in closed terms, rather than return a formal integration form. `FAILHARD` is normally off.

The switch `NOLNR` suppresses the use of the linear properties of integration in cases when the integral cannot be found in closed terms. It is normally off.

7.4.2 Advanced Use

If a function appears in the integrand that is not one of the functions `EXP`, `ERF`, `TAN`, `ATAN`, `LOG`, `DILOG` then the algorithm will make an attempt to

integrate the argument if it can, differentiate it and reach a known function. However the answer cannot be guaranteed in this case. If a function is known to be algebraically independent of this set it can be flagged transcendental by

```
flag('(trilog),'transcendental);
```

in which case this function will be added to the permitted field descriptors for a genuine decision procedure. If this is done the user is responsible for the mathematical correctness of his actions.

The standard version does not deal with algebraic extensions. Thus integration of expressions involving square roots and other like things can lead to trouble. A contributed package that supports integration of functions involving square roots is available, however (ALGINT, chapter 15.1). In addition there is a definite integration package, DEFINT(chapter 15.16).

7.4.3 References

A. C. Norman & P. M. A. Moore, "Implementing the New Risch Algorithm", Proc. 4th International Symposium on Advanced Comp. Methods in Theor. Phys., CNRS, Marseilles, 1977.

S. J. Harrington, "A New Symbolic Integration System in Reduce", Comp. Journ. 22 (1979) 2.

A. C. Norman & J. H. Davenport, "Symbolic Integration — The Dust Settles?", Proc. EUROSAM 79, Lecture Notes in Computer Science 72, Springer-Verlag, Berlin Heidelberg New York (1979) 398-407.

7.5 LENGTH Operator

`LENGTH` is a generic operator for finding the length of various objects in the system. The meaning depends on the type of the object. In particular, the length of an algebraic expression is the number of additive top-level terms its expanded representation.

Examples:

```
length(a+b)     ->  2
length(2)       ->  1.
```

Other objects that support a length operator include arrays, lists and matrices. The explicit meaning in these cases is included in the description of these objects.

7.6 MAP Operator

The MAP operator applies a uniform evaluation pattern to all members of a composite structure: a matrix, a list, or the arguments of an operator expression. The evaluation pattern can be a unary procedure, an operator, or an algebraic expression with one free variable.

It is used with the syntax:

```
MAP(U:function,V:object)
```

Here object is a list, a matrix or an operator expression. Function can be one of the following:

1. the name of an operator for a single argument: the operator is evaluated once with each element of object as its single argument;
2. an algebraic expression with exactly one free variable, that is a variable preceded by the tilde symbol. The expression is evaluated for each element of object, where the element is substituted for the free variable;
3. a replacement rule of the form var => rep where var is a variable (a kernel without a subscript) and rep is an expression that contains var. Rep is evaluated for each element of object where the element is substituted for var. Var may be optionally preceded by a tilde.

The rule form for function is needed when more than one free variable occurs.

Examples:

```
map(abs,{1,-2,a,-a})  ->  {1,2,ABS(A),ABS(A)}
map(int(~w,x), mat((x^2,x^5),(x^4,x^5))) ->

[  3     6 ]
[ x     x  ]
[----  ----]
[ 3     6  ]
[          ]
[  5     6 ]
[ x     x  ]
[----  ----]
[ 5     6  ]

map(~w*6, x^2/3 = y^3/2 -1) -> 2*X^2=3*(Y^3-2)
```

You can use MAP in nested expressions. However, you cannot apply MAP to a non-composed object, e.g. an identifier or a number.

7.7 MKID Operator

In many applications, it is useful to create a set of identifiers for naming objects in a consistent manner. In most cases, it is sufficient to create such names from two components. The operator MKID is provided for this purpose. Its syntax is:

```
MKID(U:id,V:id|non-negative integer):id
```

for example

```
mkid(a,3)       -> A3
mkid(apple,s)   -> APPLES
```

while `mkid(a+b,2)` gives an error.

The SET operator can be used to give a value to the identifiers created by MKID, for example

```
set(mkid(a,3),3);
```

will give A3 the value 2.

7.8 PF Operator

PF(<exp>,<var>) transforms the expression <exp> into a list of partial fractions with respect to the main variable, <var>. PF does a complete partial fraction decomposition, and as the algorithms used are fairly unsophisticated (factorization and the extended Euclidean algorithm), the code may be unacceptably slow in complicated cases.

Example: Given `2/((x+1)^2*(x+2))` in the workspace, `pf(ws,x);` gives the result

```
    2      - 2         2
{-------,-------,--------------} .
  X + 2   X + 1    2
  X  + 2*X + 1
```

If you want the denominators in factored form, use `off exp;`. Thus, with `2/((x+1)^2*(x+2))` in the workspace, the commands `off exp; pf(ws,x);` give the result

```
    2      - 2       2
{-------,-------,----------} .
  X + 2   X + 1          2
  (X + 1)
```

To recombine the terms, `FOR EACH ...SUM` can be used. So with the above list in the workspace, `for each j in ws sum j;` returns the result

```
     2
     ------------------
     2
      (X + 2)*(X + 1)
```

Alternatively, one can use the operations on lists to extract any desired term.

7.9 SELECT Operator

The `SELECT` operator extracts from a list, or from the arguments of an n-ary operator, elements corresponding to a boolean predicate. It is used with the syntax:

```
SELECT(U:function,V:list)
```

`Function` can be one of the following forms:

1. the name of an operator for a single argument: the operator is evaluated once with each element of `object` as its single argument;
2. an algebraic expression with exactly one free variable, that is a variable preceded by the tilde symbol. The expression is evaluated for each element of ⟨*object*⟩, where the element is substituted for the free variable;
3. a replacement rule of the form ⟨*var* => *rep*⟩ where `var` is a variable (a kernel without subscript) and `rep` is an expression that contains `var`. `Rep` is evaluated for each element of `object` where the element is substituted for `var`. `var` may be optionally preceded by a tilde.

The rule form for `function` is needed when more than one free variable occurs.

The result of evaluating `function` is interpreted as a boolean value corresponding to the conventions of REDUCE. These values are composed with the leading operator of the input expression.

Examples:

```
select( ~w>0 , {1,-1,2,-3,3}) -> {1,2,3}
select(evenp deg(~w,y),part((x+y)^5,0):=list)
       -> {X^5 ,10*X^3*Y^2 ,5*X*Y^4}
select(evenp deg(~w,x),2x^2+3x^3+4x^4) -> 4X^4 + 2X^2
```

7.10 SOLVE Operator

SOLVE is an operator for solving one or more simultaneous algebraic equations. It is used with the syntax:

```
SOLVE(EXPRN:algebraic[,VAR:kernel|,VARLIST:list of kernels])
       :list.
```

EXPRN is of the form <expression> or { <expression1>,<expression2>, ... }. Each expression is an algebraic equation, or is the difference of the two sides of the equation. The second argument is either a kernel or a list of kernels representing the unknowns in the system. This argument may be omitted if the number of distinct, non-constant, top-level kernels equals the number of unknowns, in which case these kernels are presumed to be the unknowns.

For one equation, SOLVE recursively uses factorization and decomposition, together with the known inverses of LOG, SIN, COS, ^, ACOS, ASIN, and linear, quadratic, cubic, quartic, or binomial factors. Solutions of equations built with exponentials or logarithms are often expressed in terms of Lambert's W function. This function is (partially) implemented in the special functions package.

Linear equations are solved by the multi-step elimination method due to Bareiss, unless the switch CRAMER is on, in which case Cramer's method is used. The Bareiss method is usually more efficient unless the system is large and dense.

Non-linear equations are solved using the Groebner basis package. Users should note that this can be quite a time consuming process.

Examples:

```
solve(log(sin(x+3))^5 = 8,x);
solve(a*log(sin(x+3))^5 - b, sin(x+3));
solve({a*x+y=3,y=-2},{x,y});
```

SOLVE returns a list of solutions. If there is one unknown, each solution is an equation for the unknown. If a complete solution was found, the unknown will appear by itself on the left-hand side of the equation. On the other hand, if the solve package could not find a solution, the "solution" will be an equation for the unknown in terms of the operator ROOT_OF. If there are several unknowns, each solution will be a list of equations for the unknowns. For example,

```
solve(x^2=1,x);              -> {X=-1,X=1}
```

```
solve(x^7-x^6+x^2=1,x)
                        6
       -> {X=ROOT_OF(X_  + X_ + 1,X_,TAG_1),X=1}

solve({x+3y=7,y-x=1},{x,y}) -> {{X=1,Y=2}}.
```

The TAG argument is used to uniquely identify those particular solutions. Solution multiplicities are stored in the global variable ROOT_MULTIPLICITIES rather than the solution list. The value of this variable is a list of the multiplicities of the solutions for the last call of SOLVE. For example,

```
solve(x^2=2x-1,x); root_multiplicities;
```

gives the results

```
{X=1}

{2}
```

If you want the multiplicities explicitly displayed, the switch MULTIPLICITIES can be turned on. For example

```
on multiplicities; solve(x^2=2x-1,x);
```

yields the result

```
{X=1,X=1}
```

7.10.1 Handling of Undetermined Solutions

When SOLVE cannot find a solution to an equation, it normally returns an equation for the relevant indeterminates in terms of the operator ROOT_OF. For example, the expression

```
solve(cos(x) + log(x),x);
```

returns the result

```
{X=ROOT_OF(COS(X_) + LOG(X_),X_,TAG_1)} .
```

An expression with a top-level ROOT_OF operator is implicitly a list with an unknown number of elements (since we don't always know how many solutions an equation has). If a substitution is made into such an expression, closed

form solutions can emerge. If this occurs, the ROOT_OF construct is replaced by an operator ONE_OF. At this point it is of course possible to transform the result of the original SOLVE operator expression into a standard SOLVE solution. To effect this, the operator EXPAND_CASES can be used.

The following example shows the use of these facilities:

```
solve(-a*x^3+a*x^2+x^4-x^3-4*x^2+4,x);
              2     3
{X=ROOT_OF(A*X_  - X_  + 4*X_ + 4,X_,TAG_2),X=1}

sub(a=-1,ws);

{X=ONE_OF({2,-1,-2},TAG_2),X=1}

expand_cases ws;

{X=2,X=-1,X=-2,X=1}
```

7.10.2 Solutions of Equations Involving Cubics and Quartics

Since roots of cubics and quartics can often be very messy, a switch FULLROOTS is available, that, when off (the default), will prevent the production of a result in closed form. The ROOT_OF construct will be used in this case instead.

In constructing the solutions of cubics and quartics, trigonometrical forms are used where appropriate. This option is under the control of a switch TRIGFORM, which is normally on.

The following example illustrates the use of these facilities:

```
let xx = solve(x^3+x+1,x);

xx;
             3
{X=ROOT_OF(X_  + X_ + 1,X_)}

on fullroots;

xx;

                                - SQRT(31)*I
                      ATAN(----------------)
                                 3*SQRT(3)
{X=(I*(SQRT(3)*SIN(-----------------------)
                                   3
```

```
                      - SQRT(31)*I
              ATAN(---------------)
                        3*SQRT(3)
        - COS(-----------------------)))/SQRT(3),
                         3

                                - SQRT(31)*I
                       ATAN(---------------)
                                 3*SQRT(3)
 X=( - I*(SQRT(3)*SIN(-----------------------)
                                   3

                           - SQRT(31)*I
                  ATAN(---------------)
                            3*SQRT(3)
          + COS(-----------------------)))/SQRT(
                             3

      3),

                  - SQRT(31)*I
           ATAN(---------------)
                    3*SQRT(3)
    2*COS(-----------------------)*I
                     3
 X=-----------------------------------}
                SQRT(3)

off trigform;

xx;
                                2/3
{X=( - (SQRT(31) - 3*SQRT(3))   *SQRT(3)*I

                                2/3    2/3
     - (SQRT(31) - 3*SQRT(3))    - 2   *SQRT(3)*I

        2/3                             1/3  1/3
     + 2   )/(2*(SQRT(31) - 3*SQRT(3))   *6

                1/6
              *3   ),

                           2/3
```

```
 X=((SQRT(31) - 3*SQRT(3))   *SQRT(3)*I

                              2/3    2/3
     - (SQRT(31) - 3*SQRT(3))    + 2   *SQRT(3)*I

        2/3                               1/3  1/3
     + 2   )/(2*(SQRT(31) - 3*SQRT(3))   *6

                 1/6
               *3   ),

                             2/3    2/3
     (SQRT(31) - 3*SQRT(3))    - 2
 X=-------------------------------------}
                            1/3  1/3  1/6
    (SQRT(31) - 3*SQRT(3))   *6   *3
```

7.10.3 Other Options

If SOLVESINGULAR is on (the default setting), degenerate systems such as x+y=0, 2x+2y=0 will be solved by introducing appropriate arbitrary constants. The consistent singular equation 0=0 or equations involving functions with multiple inverses may introduce unique new indeterminant kernels ARBCOMPLEX(j), or ARBINT(j), (j=1,2,...), representing arbitrary complex or integer numbers respectively. To automatically select the principal branches, do off allbranch; . To avoid the introduction of new indeterminant kernels do OFF ARBVARS - then no equations are generated for the free variables and their original names are used to express the solution forms. To suppress solutions of consistent singular equations do OFF SOLVESINGULAR.

To incorporate additional inverse functions do, for example:

```
put('sinh,'inverse,'asinh);
put('asinh,'inverse,'sinh);
```

together with any desired simplification rules such as

```
for all x let sinh(asinh(x))=x, asinh(sinh(x))=x;
```

For completeness, functions with non-unique inverses should be treated as ^, SIN, and COS are in the SOLVE module source.

Arguments of ASIN and ACOS are not checked to ensure that the absolute value of the real part does not exceed 1; and arguments of LOG are not checked to

ensure that the absolute value of the imaginary part does not exceed π; but checks (perhaps involving user response for non-numerical arguments) could be introduced using LET statements for these operators.

7.10.4 Parameters and Variable Dependency

The proper design of a variable sequence supplied as a second argument to SOLVE is important for the structure of the solution of an equation system. Any unknown in the system not in this list is considered totally free. E.g. the call

```
solve({x=2*z,z=2*y},{z});
```

produces an empty list as a result because there is no function $z = z(x, y)$ which fulfills both equations for arbitrary x and y values. In such a case the share variable requirements displays a set of restrictions for the parameters of the system:

```
requirements;

{x - 4*y}
```

The non-existence of a formal solution is caused by a contradiction which disappears only if the parameters of the initial system are set such that all members of the requirements list take the value zero. For a linear system the set is complete: a solution of the requirements list makes the initial system solvable. E.g. in the above case a substitution $x = 4y$ makes the equation set consistent. For a non-linear system only one inconsistency is detected. If such a system has more than one inconsistency, you must reduce them one after the other. [1] The set shows you also the dependency among the parameters: here one of x and y is free and a formal solution of the system can be computed by adding it to the variable list of solve. The requirement set is not unique - there may be other such sets.

A system with parameters may have a formal solution, e.g.

```
solve({x=a*z+1,0=b*z-y},{z,x});

     y      a*y + b
{{z=---,x=---------}}
```

[1] The difference between linear and non-linear inconsistent systems is based on the algorithms which produce this information as a side effect when attempting to find a formal solution; example: $solve(\{x = a, x = b, y = c, y = d\}, \{x, y\}$ gives a set $\{a - b, c - d\}$ while $solve(\{x^2 = a, x^2 = b, y^2 = c, y^2 = d\}, \{x, y\}$ leads to $\{a - b\}$.

```
b          b
```

which is not valid for all possible values of the parameters. The variable `assumptions` contains then a list of restrictions: the solutions are valid only as long as none of these expressions vanishes. Any zero of one of them represents a special case that is not covered by the formal solution. In the above case the value is

```
assumptions;

{b}
```

which excludes formally the case $b = 0$; obviously this special parameter value makes the system singular. The set of assumptions is complete for both, linear and non–linear systems.

SOLVE rearranges the variable sequence to reduce the (expected) computing time. This behavior is controlled by the switch VAROPT, which is on by default. If it is turned off, the supplied variable sequence is used or the system kernel ordering is taken if the variable list is omitted. The effect is demonstrated by an example:

```
s:= {y^3+3x=0,x^2+y^2=1};

solve(s,{y,x});

                6       2
{{y=root_of(y_  + 9*y_  - 9,y_),

          3
       - y
  x=-------}}
       3

off varopt; solve(s,{y,x});

                6       4        2
{{x=root_of(x_  - 3*x_  + 12*x_  - 1,x_),

             4      2
     x*( - x  + 2*x  - 10)
  y=----------------------}}
                3
```

In the first case, solve forms the solution as a set of pairs $(y_i, x(y_i))$ because the degree of x is higher - such a rearrangement makes the internal computation of the Gröbner basis generally faster. For the second case the explicitly given variable sequence is used such that the solution has now the form $(x_i, y(x_i))$. Controlling the variable sequence is especially important if the system has one or more free variables. As an alternative to turning off varopt, a partial dependency among the variables can be declared using the

`depend` statement: `solve` then rearranges the variable sequence but keeps any variable ahead of those on which it depends.

```
on varopt;
s:={a^3+b,b^2+c}$
solve(s,{a,b,c});

                          3       6
{{a=arbcomplex(1),b= - a ,c= - a }}

depend a,c; depend b,c; solve(s,{a,b,c});

{{c=arbcomplex(2),

              6
  a=root_of(a_  + c,a_),

        3
  b= - a }}
```

Here `solve` is forced to put c after a and after b, but there is no obstacle to interchanging a and b.

7.11 Even and Odd Operators

An operator can be declared to be *even* or *odd* in its first argument by the declarations EVEN and ODD respectively. Expressions involving an operator declared in this manner are transformed if the first argument contains a minus sign. Any other arguments are not affected. In addition, if say F is declared odd, then f(0) is replaced by zero unless F is also declared *non zero* by the declaration NONZERO. For example, the declarations

```
even f1; odd f2;
```

mean that

```
f1(-a)    ->    F1(A)
f2(-a)    ->   -F2(A)
f1(-a,-b) ->    F1(A,-B)
f2(0)     ->    0.
```

To inhibit the last transformation, say `nonzero f2;`.

7.12 Linear Operators

An operator can be declared to be linear in its first argument over powers of its second argument. If an operator F is so declared, F of any sum is broken up into sums of Fs, and any factors that are not powers of the variable are taken outside. This means that F must have (at least) two arguments. In addition, the second argument must be an identifier (or more generally a kernel), not an expression.

Example:

If F were declared linear, then

```
                                    5
f(a*x^5+b*x+c,x) ->  F(X ,X)*A + F(X,X)*B + F(1,X)*C
```

More precisely, not only will the variable and its powers remain within the scope of the F operator, but so will any variable and its powers that had been declared to DEPEND on the prescribed variable; and so would any expression that contains that variable or a dependent variable on any level, e.g. `cos(sin(x))`.

To declare operators F and G to be linear operators, use:

```
linear f,g;
```

The analysis is done of the first argument with respect to the second; any other arguments are ignored. It uses the following rules of evaluation:

```
f(0) -> 0
f(-y,x) -> -F(Y,X)
f(y+z,x) -> F(Y,X)+F(Z,X)
f(y*z,x) -> Z*F(Y,X)        if Z does not depend on X
f(y/z,x) -> F(Y,X)/Z        if Z does not depend on X
```

To summarize, `Y` "depends" on the indeterminate `X` in the above if either of the following hold:

1. `Y` is an expression that contains `X` at any level as a variable, e.g.: `cos(sin(x))`
2. Any variable in the expression `Y` has been declared dependent on `X` by use of the declaration `DEPEND`.

The use of such linear operators can be seen in the paper Fox, J.A. and A. C. Hearn, "Analytic Computation of Some Integrals in Fourth Order Quantum Electrodynamics" Journ. Comp. Phys. 14 (1974) 301-317, which contains a complete listing of a program for definite integration of some expressions that arise in fourth order quantum electrodynamics.

7.13 Non-Commuting Operators

An operator can be declared to be non-commutative under multiplication by the declaration `NONCOM`.

Example:

After the declaration
`noncom u,v;`
the expressions `u(x)*u(y)-u(y)*u(x)` and `u(x)*v(y)-v(y)*u(x)` will remain unchanged on simplification, and in particular will not simplify to zero.

Note that it is the operator (`U` and `V` in the above example) and not the variable that has the non-commutative property.

The `LET` statement may be used to introduce rules of evaluation for such operators. In particular, the boolean operator `ORDP` is useful for introducing an ordering on such expressions.

Example:

The rule

```
for all x,y such that x neq y and ordp(x,y)
   let u(x)*u(y)= u(y)*u(x)+comm(x,y);
```

would introduce the commutator of `u(x)` and `u(y)` for all `X` and `Y`. Note that since `ordp(x,x)` is *true*, the equality check is necessary in the degenerate case to avoid a circular loop in the rule.

7.14 Symmetric and Antisymmetric Operators

An operator can be declared to be symmetric with respect to its arguments by the declaration `SYMMETRIC`. For example

```
symmetric u,v;
```

means that any expression involving the top level operators `U` or `V` will have its arguments reordered to conform to the internal order used by REDUCE. The user can change this order for kernels by the command `KORDER`.

For example, `u(x,v(1,2))` would become `u(v(2,1),x)`, since numbers are ordered in decreasing order, and expressions are ordered in decreasing order of complexity.

Similarly the declaration `ANTISYMMETRIC` declares an operator antisymmetric. For example,

```
antisymmetric l,m;
```

means that any expression involving the top level operators `L` or `M` will have its arguments reordered to conform to the internal order of the system, and the sign of the expression changed if there are an odd number of argument interchanges necessary to bring about the new order.

For example, `l(x,m(1,2))` would become `-l(-m(2,1),x)` since one interchange occurs with each operator. An expression like `l(x,x)` would also be replaced by 0.

7.15 Declaring New Prefix Operators

The user may add new prefix operators to the system by using the declaration OPERATOR. For example:

```
operator h,g1,arctan;
```

adds the prefix operators H, G1 and ARCTAN to the system.

This allows symbols like `h(w), h(x,y,z), g1(p+q), arctan(u/v)` to be used in expressions, but no meaning or properties of the operator are implied. The same operator symbol can be used equally well as a 0-, 1-, 2-, 3-, etc.-place operator.

To give a meaning to an operator symbol, or express some of its properties, LET statements can be used, or the operator can be given a definition as a procedure.

If the user forgets to declare an identifier as an operator, the system will prompt the user to do so in interactive mode, or do it automatically in non-interactive mode. A diagnostic message will also be printed if an identifier is declared OPERATOR more than once.

Operators once declared are global in scope, and so can then be referenced anywhere in the program. In other words, a declaration within a block (or a procedure) does not limit the scope of the operator to that block, nor does the operator go away on exiting the block (use CLEAR instead for this purpose).

7.16 Declaring New Infix Operators

Users can add new infix operators by using the declarations INFIX and PRECEDENCE. For example,

```
infix mm;
precedence mm,-;
```

The declaration `infix mm;` would allow one to use the symbol MM as an infix operator:

```
a mm b      instead of      mm(a,b).
```

The declaration `precedence mm,-;` says that MM should be inserted into the infix operator precedence list just *after* the − operator. This gives it higher precedence than − and lower precedence than * . Thus

```
a - b mm c - d      means      a - (b mm c) - d,
```

while

```
a * b mm c * d      means      (a * b) mm (c * d).
```

Both infix and prefix operators have no transformation properties unless LET statements or procedure declarations are used to assign a meaning.

We should note here that infix operators so defined are always binary:

```
a mm b mm c      means      (a mm b) mm c.
```

7.17 Creating/Removing Variable Dependency

There are several facilities in REDUCE, such as the differentiation operator and the linear operator facility, that can utilize knowledge of the dependency between various variables, or kernels. Such dependency may be expressed by the command DEPEND. This takes an arbitrary number of arguments and sets up a dependency of the first argument on the remaining arguments. For example,

```
depend x,y,z;
```

says that X is dependent on both Y and Z.

```
depend z,cos(x),y;
```

says that Z is dependent on COS(X) and Y.

Dependencies introduced by DEPEND can be removed by NODEPEND. The arguments of this are the same as for DEPEND. For example, given the above dependencies,

```
nodepend z,cos(x);
```

says that Z is no longer dependent on COS(X), although it remains dependent on Y.

Chapter 8

Display and Structuring of Expressions

In this section, we consider a variety of commands and operators that permit the user to obtain various parts of algebraic expressions and also display their structure in a variety of forms. Also presented are some additional concepts in the REDUCE design that help the user gain a better understanding of the structure of the system.

8.1 Kernels

REDUCE is designed so that each operator in the system has an evaluation (or simplification) function associated with it that transforms the expression into an internal canonical form. This form, which bears little resemblance to the original expression, is described in detail in Hearn, A. C., "REDUCE 2: A System and Language for Algebraic Manipulation," Proc. of the Second Symposium on Symbolic and Algebraic Manipulation, ACM, New York (1971) 128-133.

The evaluation function may transform its arguments in one of two alternative ways. First, it may convert the expression into other operators in the system, leaving no functions of the original operator for further manipulation. This is in a sense true of the evaluation functions associated with the operators +, * and / , for example, because the canonical form does not include these operators explicitly. It is also true of an operator such as the determinant operator `DET` because the relevant evaluation function calculates the appropriate determinant, and the operator `DET` no longer appears. On the other hand, the evaluation process may leave some residual functions of the relevant operator. For example, with the operator `COS`, a residual

expression like `COS(X)` may remain after evaluation unless a rule for the reduction of cosines into exponentials, for example, were introduced. These residual functions of an operator are termed *kernels* and are stored uniquely like variables. Subsequently, the kernel is carried through the calculation as a variable unless transformations are introduced for the operator at a later stage.

In those cases where the evaluation process leaves an operator expression with non-trivial arguments, the form of the argument can vary depending on the state of the system at the point of evaluation. Such arguments are normally produced in expanded form with no terms factored or grouped in any way. For example, the expression `cos(2*x+2*y)` will normally be returned in the same form. If the argument `2*x+2*y` were evaluated at the top level, however, it would be printed as `2*(X+Y)`. If it is desirable to have the arguments themselves in a similar form, the switch `INTSTR` (for "internal structure"), if on, will cause this to happen.

In cases where the arguments of the kernel operators may be reordered, the system puts them in a canonical order, based on an internal intrinsic ordering of the variables. However, some commands allow arguments in the form of kernels, and the user has no way of telling what internal order the system will assign to these arguments. To resolve this difficulty, we introduce the notion of a *kernel form* as an expression that transforms to a kernel on evaluation.

Examples of kernel forms are:

```
a
cos(x*y)
log(sin(x))
```

whereas

```
a*b
(a+b)^4
```

are not.

We see that kernel forms can usually be used as generalized variables, and most algebraic properties associated with variables may also be associated with kernels.

8.2 The Expression Workspace

Several mechanisms are available for saving and retrieving previously evaluated expressions. The simplest of these refers to the last algebraic expression simplified. When an assignment of an algebraic expression is made, or an expression is evaluated at the top level, (i.e., not inside a compound statement or procedure) the results of the evaluation are automatically saved in a variable `WS` that we shall refer to as the workspace. (More precisely, the expression is assigned to the variable `WS` that is then available for further manipulation.)

Example:

If we evaluate the expression `(x+y)^2` at the top level and next wish to differentiate it with respect to `Y`, we can simply say

```
df(ws,y);
```

to get the desired answer.

If the user wishes to assign the workspace to a variable or expression for later use, the `SAVEAS` statement can be used. It has the syntax

```
SAVEAS <expression>
```

For example, after the differentiation in the last example, the workspace holds the expression `2*x+2*y`. If we wish to assign this to the variable `Z` we can now say

```
saveas z;
```

If the user wishes to save the expression in a form that allows him to use some of its variables as arbitrary parameters, the `FOR ALL` command can be used.

Example:

```
for all x saveas h(x);
```

with the above expression would mean that `h(z)` evaluates to `2*Y+2*Z`.

A further method for referencing more than the last expression is described in the section on interactive use of REDUCE.

8.3 Output of Expressions

A considerable degree of flexibility is available in REDUCE in the printing of expressions generated during calculations. No explicit format statements are supplied, as these are in most cases of little use in algebraic calculations, where the size of output or its composition is not generally known in advance. Instead, REDUCE provides a series of mode options to the user that should enable him to produce his output in a comprehensible and possibly pleasing form.

The most extreme option offered is to suppress the output entirely from any top level evaluation. This is accomplished by turning off the switch `OUTPUT` which is normally on. It is useful for limiting output when loading large files or producing "clean" output from the prettyprint programs.

In most circumstances, however, we wish to view the output, so we need to know how to format it appropriately. As we mentioned earlier, an algebraic expression is normally printed in an expanded form, filling the whole output line with terms. Certain output declarations, however, can be used to affect this format. To begin with, we look at an operator for changing the length of the output line.

8.3.1 LINELENGTH Operator

This operator is used with the syntax

```
LINELENGTH(NUM:integer):integer
```

and sets the output line length to the integer `NUM`. It returns the previous output line length (so that it can be stored for later resetting of the output line if needed).

8.3.2 Output Declarations

We now describe a number of switches and declarations that are available for controlling output formats. It should be noted, however, that the transformation of large expressions to produce these varied output formats can take a lot of computing time and space. If a user wishes to speed up the printing of the output in such cases, he can turn off the switch `PRI`. If this is done, then output is produced in one fixed format, which basically reflects the internal form of the expression, and none of the options below apply. `PRI` is normally on.

With `PRI` on, the output declarations and switches available are as follows:

ORDER Declaration

The declaration ORDER may be used to order variables on output. The syntax is:

```
order v1,...vn;
```

where the vi are kernels. Thus,

```
order x,y,z;
```

orders X ahead of Y, Y ahead of Z and all three ahead of other variables not given an order. `order nil;` resets the output order to the system default. The order of variables may be changed by further calls of ORDER, but then the reordered variables would have an order lower than those in earlier ORDER calls. Thus,

```
order x,y,z;
order y,x;
```

would order Z ahead of Y and X. The default ordering is usually alphabetic.

FACTOR Declaration

This declaration takes a list of identifiers or kernels as argument. FACTOR is not a factoring command (use FACTORIZE or the FACTOR switch for this purpose); rather it is a separation command. All terms involving fixed powers of the declared expressions are printed as a product of the fixed powers and a sum of the rest of the terms.

All expressions involving a given prefix operator may also be factored by putting the operator name in the list of factored identifiers. For example:

```
factor x,cos,sin(x);
```

causes all powers of X and SIN(X) and all functions of COS to be factored.

Note that FACTOR does not affect the order of its arguments. You should also use ORDER if this is important.

The declaration `remfac v1,...,vn;` removes the factoring flag from the expressions v1 through vn.

8.3.3 Output Control Switches

In addition to these declarations, the form of the output can be modified by switching various output control switches using the declarations `ON` and `OFF`. We shall illustrate the use of these switches by an example, namely the printing of the expression

```
x^2*(y^2+2*y)+x*(y^2+z)/(2*a) .
```

The relevant switches are as follows:

ALLFAC Switch

This switch will cause the system to search the whole expression, or any sub-expression enclosed in parentheses, for simple multiplicative factors and print them outside the parentheses. Thus our expression with `ALLFAC` off will print as

```
    2  2         2           2
(2*X *Y *A + 4*X *Y*A + X*Y  + X*Z)/(2*A)
```

and with `ALLFAC` on as

```
         2                 2
X*(2*X*Y *A + 4*X*Y*A + Y  + Z)/(2*A) .
```

`ALLFAC` is normally on, and is on in the following examples, except where otherwise stated.

DIV Switch

This switch makes the system search the denominator of an expression for simple factors that it divides into the numerator, so that rational fractions and negative powers appear in the output. With `DIV` on, our expression would print as

```
      2                  2  (-1)         (-1)
X*(X*Y  + 2*X*Y + 1/2*Y *A      + 1/2*A     *Z) .
```

`DIV` is normally off.

LIST Switch

This switch causes the system to print each term in any sum on a separate line. With `LIST` on, our expression prints as

```
          2
X*(2*X*Y *A

    + 4*X*Y*A

       2
    + Y

    + Z)/(2*A) .
```

`LIST` is normally off.

NOSPLIT Switch

Under normal circumstances, the printing routines try to break an expression across lines at a natural point. This is a fairly expensive process. If you are not overly concerned about where the end-of-line breaks come, you can speed up the printing of expressions by turning off the switch `NOSPLIT`. This switch is normally on.

RAT Switch

This switch is only useful with expressions in which variables are factored with `FACTOR`. With this mode, the overall denominator of the expression is printed with each factored sub-expression. We assume a prior declaration `factor x;` in the following output. We first print the expression with `RAT off`:

```
     2                    2
(2*X *Y*A*(Y + 2) + X*(Y  + Z))/(2*A) .
```

With `RAT` on the output becomes:

```
 2                  2
X *Y*(Y + 2) + X*(Y  + Z)/(2*A) .
```

RAT is normally off.

Next, if we leave X factored, and turn on both DIV and RAT, the result becomes

```
 2                      (-1)   2
X *Y*(Y + 2) + 1/2*X*A    *(Y  + Z) .
```

Finally, with X factored, RAT on and ALLFAC off we retrieve the original structure

```
 2   2              2
X *(Y  + 2*Y) + X*(Y  + Z)/(2*A) .
```

RATPRI Switch

If the numerator and denominator of an expression can each be printed in one line, the output routines will print them in a two dimensional notation, with numerator and denominator on separate lines and a line of dashes in between. For example, (a+b)/2 will print as

```
A + B
-----
  2
```

Turning this switch off causes such expressions to be output in a linear form.

REVPRI Switch

The normal ordering of terms in output is from highest to lowest power. In some situations (e.g., when a power series is output), the opposite ordering is more convenient. The switch REVPRI if on causes such a reverse ordering of terms. For example, the expression y*(x+1)^2+(y+3)^2 will normally print as

```
 2              2
X *Y + 2*X*Y + Y  + 7*Y + 9
```

whereas with REVPRI on, it will print as

```
                2             2
```

```
9 + 7*Y + Y  + 2*X*Y + X *Y.
```

8.3.4 WRITE Command

In simple cases no explicit output command is necessary in REDUCE, since the value of any expression is automatically printed if a semicolon is used as a delimiter. There are, however, several situations in which such a command is useful.

In a FOR, WHILE, or REPEAT statement it may be desired to output something each time the statement within the loop construct is repeated.

It may be desired for a procedure to output intermediate results or other information while it is running. It may be desired to have results labeled in special ways, especially if the output is directed to a file or device other than the terminal.

The WRITE command consists of the word WRITE followed by one or more items separated by commas, and followed by a terminator. There are three kinds of items that can be used:

1. Expressions (including variables and constants). The expression is evaluated, and the result is printed out.
2. Assignments. The expression on the right side of the := operator is evaluated, and is assigned to the variable on the left; then the symbol on the left is printed, followed by a ":=", followed by the value of the expression on the right – almost exactly the way an assignment followed by a semicolon prints out normally. (The difference is that if the WRITE is in a FOR statement and the left-hand side of the assignment is an array position or something similar containing the variable of the FOR iteration, then the value of that variable is inserted in the printout.)
3. Arbitrary strings of characters, preceded and followed by double-quote marks (e.g., "string").

The items specified by a single WRITE statement print side by side on one line. (The line is broken automatically if it is too long.) Strings print exactly as quoted. The WRITE command itself however does not return a value.

The print line is closed at the end of a WRITE command evaluation. Therefore the command WRITE ""; (specifying nothing to be printed except the empty string) causes a line to be skipped.

Examples:

1. If A is X+5, B is itself, C is 123, M is an array, and Q=3, then

```
write m(q):=a," ",b/c," THANK YOU";
```

will set M(3) to x+5 and print

```
M(Q) := X + 5 B/123 THANK YOU
```

The blanks between the 5 and B, and the 3 and T, come from the blanks in the quoted strings.

2. To print a table of the squares of the integers from 1 to 20:

```
for i:=1:20 do write i," ",i^2;
```

3. To print a table of the squares of the integers from 1 to 20, and at the same time store them in positions 1 to 20 of an array A:

```
for i:=1:20 do <<a(i):=i^2; write i," ",a(i)>>;
```

This will give us two columns of numbers. If we had used

```
for i:=1:20 do write i," ",a(i):=i^2;
```

we would also get A(i) := repeated on each line.

4. The following more complete example calculates the famous f and g series, first reported in Sconzo, P., LeSchack, A. R., and Tobey, R., "Symbolic Computation of f and g Series by Computer", Astronomical Journal 70 (May 1965).

```
x1:= -sig*(mu+2*eps)$
x2:= eps - 2*sig^2$
x3:= -3*mu*sig$
f:= 1$
g:= 0$
for i:= 1 step 1 until 10 do begin
   f1:= -mu*g+x1*df(f,eps)+x2*df(f,sig)+x3*df(f,mu);
   write "f(",i,") := ",f1;
   g1:= f+x1*df(g,eps)+x2*df(g,sig)+x3*df(g,mu);
   write "g(",i,") := ",g1;
   f:=f1$
   g:=g1$
  end;
```

A portion of the output, to illustrate the printout from the WRITE command, is as follows:

```
                ... <prior output> ...

                                2
F(4) := MU*(3*EPS - 15*SIG  + MU)

G(4) := 6*SIG*MU

                                        2
F(5) := 15*SIG*MU*( - 3*EPS + 7*SIG  - MU)

                                2
G(5) := MU*(9*EPS - 45*SIG  + MU)

                ... <more output> ...
```

8.3.5 Suppression of Zeros

It is sometimes annoying to have zero assignments (i.e. assignments of the form `<expression> := 0`) printed, especially in printing large arrays with many zero elements. The output from such assignments can be suppressed by turning on the switch `NERO`.

8.3.6 FORTRAN Style Output Of Expressions

It is naturally possible to evaluate expressions numerically in REDUCE by giving all variables and sub-expressions numerical values. However, as we pointed out elsewhere the user must declare real arithmetical operation by turning on the switch `ROUNDED`. However, it should be remembered that arithmetic in REDUCE is not particularly fast, since results are interpreted rather than evaluated in a compiled form. The user with a large amount of numerical computation after all necessary algebraic manipulations have been performed is therefore well advised to perform these calculations in a FORTRAN or similar system. For this purpose, REDUCE offers facilities for users to produce FORTRAN compatible files for numerical processing.

First, when the switch `FORT` is on, the system will print expressions in a FORTRAN notation. Expressions begin in column seven. If an expression extends over one line, a continuation mark (.) followed by a blank appears on subsequent cards. After a certain number of lines have been produced (according to the value of the variable `CARD_NO`), a new expression is started. If the expression printed arises from an assignment to a variable, the variable is printed as the name of the expression. Otherwise the expression is given

the default name ANS. An error occurs if identifiers or numbers are outside the bounds permitted by FORTRAN.

A second option is to use the WRITE command to produce other programs.

Example:

The following REDUCE statements

```
on fort;
out "forfil";
write "C     this is a fortran program";
write " 1    format(e13.5)";
write "      u=1.23";
write "      v=2.17";
write "      w=5.2";
x:=(u+v+w)^11;
write "C     it was foolish to expand this expression";
write "      print 1,x";
write "      end";
shut "forfil";
off fort;
```

will generate a file forfil that contains:

```
c this is a fortran program
 1    format(e13.5)
      u=1.23
      v=2.17
      w=5.2
      ans1=1320.*u**3*v*w**7+165.*u**3*w**8+55.*u**2*v**9+495.*u
     . **2*v**8*w+1980.*u**2*v**7*w**2+4620.*u**2*v**6*w**3+
     . 6930.*u**2*v**5*w**4+6930.*u**2*v**4*w**5+4620.*u**2*v**3*
     . w**6+1980.*u**2*v**2*w**7+495.*u**2*v*w**8+55.*u**2*w**9+
     . 11.*u*v**10+110.*u*v**9*w+495.*u*v**8*w**2+1320.*u*v**7*w
     . **3+2310.*u*v**6*w**4+2772.*u*v**5*w**5+2310.*u*v**4*w**6
     . +1320.*u*v**3*w**7+495.*u*v**2*w**8+110.*u*v*w**9+11.*u*w
     . **10+v**11+11.*v**10*w+55.*v**9*w**2+165.*v**8*w**3+330.*
     . v**7*w**4+462.*v**6*w**5+462.*v**5*w**6+330.*v**4*w**7+
     . 165.*v**3*w**8+55.*v**2*w**9+11.*v*w**10+w**11
      x=u**11+11.*u**10*v+11.*u**10*w+55.*u**9*v**2+110.*u**9*v*
     . w+55.*u**9*w**2+165.*u**8*v**3+495.*u**8*v**2*w+495.*u**8
     . *v*w**2+165.*u**8*w**3+330.*u**7*v**4+1320.*u**7*v**3*w+
     . 1980.*u**7*v**2*w**2+1320.*u**7*v*w**3+330.*u**7*w**4+462.
     . *u**6*v**5+2310.*u**6*v**4*w+4620.*u**6*v**3*w**2+4620.*u
     . **6*v**2*w**3+2310.*u**6*v*w**4+462.*u**6*w**5+462.*u**5*
     . v**6+2772.*u**5*v**5*w+6930.*u**5*v**4*w**2+9240.*u**5*v
     . **3*w**3+6930.*u**5*v**2*w**4+2772.*u**5*v*w**5+462.*u**5
```

```
     . *w**6+330.*u**4*v**7+2310.*u**4*v**6*w+6930.*u**4*v**5*w
     . **2+11550.*u**4*v**4*w**3+11550.*u**4*v**3*w**4+6930.*u**
     . 4*v**2*w**5+2310.*u**4*v*w**6+330.*u**4*w**7+165.*u**3*v
     . **8+1320.*u**3*v**7*w+4620.*u**3*v**6*w**2+9240.*u**3*v**
     . 5*w**3+11550.*u**3*v**4*w**4+9240.*u**3*v**3*w**5+4620.*u
     . **3*v**2*w**6+ans1
c     it was foolish to expand this expression
      print 1,x
      end
```

If the arguments of a WRITE statement include an expression that requires continuation records, the output will need editing, since the output routine prints the arguments of WRITE sequentially, and the continuation mechanism therefore generates its auxiliary variables after the preceding expression has been printed.

Finally, since there is no direct analog of *list* in FORTRAN, a comment line of the form

```
c ***** invalid fortran construct (list) not printed
```

will be printed if you try to print a list with FORT on.

FORTRAN Output Options

There are a number of methods available to change the default format of the FORTRAN output.

The breakup of the expression into subparts is such that the number of continuation lines produced is less than a given number. This number can be modified by the assignment

```
card_no := <number>;
```

where <number> is the *total* number of cards allowed in a statement. The default value of CARD_NO is 20.

The width of the output expression is also adjustable by the assignment

```
fort_width := <integer>;
```

which sets the total width of a given line to <integer>. The initial FORTRAN output width is 70.

REDUCE automatically inserts a decimal point after each isolated integer coefficient in a FORTRAN expression (so that, for example, 4 becomes 4.).

To prevent this, set the PERIOD mode switch to OFF.

FORTRAN output is normally produced in lower case. If upper case is desired, the switch FORTUPPER should be turned on.

Finally, the default name ANS assigned to an unnamed expression and its subparts can be changed by the operator VARNAME. This takes a single identifier as argument, which then replaces ANS as the expression name. The value of VARNAME is its argument.

Further facilities for the production of FORTRAN and other language output are provided by the SCOPE and GENTRAN packagesdescribed in chapters 15.24 and 15.51.

8.3.7 Saving Expressions for Later Use as Input

It is often useful to save an expression on an external file for use later as input in further calculations. The commands for opening and closing output files are explained elsewhere. However, we see in the examples on output of expressions that the standard "natural" method of printing expressions is not compatible with the input syntax. So to print the expression in an input compatible form we must inhibit this natural style by turning off the switch NAT. If this is done, a dollar sign will also be printed at the end of the expression.

Example:

The following sequence of commands

```
off nat; out "out"; x := (y+z)^2; write "end";
shut "out"; on nat;
```

will generate a file out that contains

```
X := Y**2 + 2*Y*Z + Z**2$
END$
```

8.3.8 Displaying Expression Structure

In those cases where the final result has a complicated form, it is often convenient to display the skeletal structure of the answer. The operator STRUCTR, that takes a single expression as argument, will do this for you. Its syntax is:

```
STRUCTR(EXPRN:algebraic[,ID1:identifier[,ID2:identifier]]);
```

The structure is printed effectively as a tree, in which the subparts are laid out with auxiliary names. If the optional `ID1` is absent, the auxiliary names are prefixed by the root `ANS`. This root may be changed by the operator `VARNAME`. If the optional `ID1` is present, and is an array name, the subparts are named as elements of that array, otherwise `ID1` is used as the root prefix. (The second optional argument `ID2` is explained later.)

The `EXPRN` can be either a scalar or a matrix expression. Use of any other will result in an error.

Example:

Let us suppose that the workspace contains `((A+B)^2+C)^3+D`. Then the input `STRUCTR WS;` will (with `EXP` off) result in the output:

```
ANS3

   where

                  3
      ANS3 := ANS2  + D

                  2
      ANS2 := ANS1  + C

      ANS1 := A + B
```

The workspace remains unchanged after this operation, since STRUCTR in the default situation returns no value (if STRUCTR is used as a sub-expression, its value is taken to be 0). In addition, the sub-expressions are normally only displayed and not retained. If you wish to access the sub-expressions with their displayed names, the switch SAVESTRUCTR should be turned on. In this case, STRUCTR returns a list whose first element is a representation for the expression, and subsequent elements are the sub-expression relations. Thus, with SAVESTRUCTR on, STRUCTR WS in the above example would return

```
                    3               2
{ANS3,ANS3=ANS2  + D,ANS2=ANS1  + C,ANS1=A + B}
```

The PART operator can be used to retrieve the required parts of the expression. For example, to get the value of ANS2 in the above, one could say:

```
part(ws,3,2);
```

If FORT is on, then the results are printed in the reverse order; the algorithm in fact guaranteeing that no sub-expression will be referenced before it is defined. The second optional argument ID2 may also be used in this case to name the actual expression (or expressions in the case of a matrix argument).

Example:

Let us suppose that M, a 2 by 1 matrix, contains the elements `((a+b)^2 + c)^3 + d` and `(a + b)*(c + d)` respectively, and that V has been declared to be an array. With EXP off and FORT on, the statement `structr(2*m,v,k);` will result in the output

```
V(1)=A+B
V(2)=V(1)**2+C
V(3)=V(2)**3+D
V(4)=C+D
```

```
K(1,1)=2.*V(3)
K(2,1)=2.*V(1)*V(4)
```

8.4 Changing the Internal Order of Variables

The internal ordering of variables (more specifically kernels) can have a significant effect on the space and time associated with a calculation. In its default state, REDUCE uses a specific order for this which may vary between sessions. However, it is possible for the user to change this internal order by means of the declaration KORDER. The syntax for this is:

```
korder v1,...,vn;
```

where the Vi are kernels. With this declaration, the Vi are ordered internally ahead of any other kernels in the system. V1 has the highest order, V2 the next highest, and so on. A further call of KORDER replaces a previous one. KORDER NIL; resets the internal order to the system default.

Unlike the ORDER declaration, that has a purely cosmetic effect on the way results are printed, the use of KORDER can have a significant effect on computation time. In critical cases then, the user can experiment with the ordering of the variables used to determine the optimum set for a given problem.

8.5 Obtaining Parts of Algebraic Expressions

There are many occasions where it is desirable to obtain a specific part of an expression, or even change such a part to another expression. A number of operators are available in REDUCE for this purpose, and will be described in this section. In addition, operators for obtaining specific parts of polynomials and rational functions (such as a denominator) are described in another section.

8.5.1 COEFF Operator

Syntax:

```
COEFF(EXPRN:polynomial,VAR:kernel)
```

COEFF is an operator that partitions EXPRN into its various coefficients with respect to VAR and returns them as a list, with the coefficient independent of VAR first.

Under normal circumstances, an error results if EXPRN is not a polynomial in VAR, although the coefficients themselves can be rational as long as they do not depend on VAR. However, if the switch RATARG is on, denominators are not checked for dependence on VAR, and are taken to be part of the coefficients.

Example:

```
coeff((y^2+z)^3/z,y);
```

returns the result

```
  2
{Z ,0,3*Z,0,3,0,1/Z}.
```

whereas

```
coeff((y^2+z)^3/y,y);
```

gives an error if RATARG is off, and the result

```
  3        2
{Z /Y,0,3*Z /Y,0,3*Z/Y,0,1/Y}
```

if RATARG is on.

The length of the result of COEFF is the highest power of VAR encountered plus 1. In the above examples it is 7. In addition, the variable HIGH_POW is set to the highest non-zero power found in EXPRN during the evaluation, and LOW_POW to the lowest non-zero power, or zero if there is a constant term. If EXPRN is a constant, then HIGH_POW and LOW_POW are both set to zero.

8.5.2 COEFFN Operator

The COEFFN operator is designed to give the user a particular coefficient of a variable in a polynomial, as opposed to COEFF that returns all coefficients. COEFFN is used with the syntax

```
COEFFN(EXPRN:polynomial,VAR:kernel,N:integer)
```

It returns the n^{th} coefficient of VAR in the polynomial EXPRN.

8.5.3 PART Operator

Syntax:

```
PART(EXPRN:algebraic[,INTEXP:integer])
```

This operator works on the form of the expression as printed *or as it would have been printed at that point in the calculation* bearing in mind all the relevant switch settings at that point. The reader therefore needs some familiarity with the way that expressions are represented in prefix form in REDUCE to use these operators effectively. Furthermore, it is assumed that PRI is ON at that point in the calculation. The reason for this is that with PRI off, an expression is printed by walking the tree representing the expression internally. To save space, it is never actually transformed into the equivalent prefix expression as occurs when PRI is on. However, the operations on polynomials described elsewhere can be equally well used in this case to obtain the relevant parts.

The evaluation proceeds recursively down the integer expression list. In other words,

```
PART(<expression>,<integer1>,<integer2>)
    ->  PART(PART(<expression>,<integer1>),<integer2>)
```

and so on, and

```
PART(<expression>) ->  <expression>.
```

INTEXP can be any expression that evaluates to an integer. If the integer is positive, then that term of the expression is found. If the integer is 0, the operator is returned. Finally, if the integer is negative, the counting is from the tail of the expression rather than the head.

For example, if the expression a+b is printed as A+B (i.e., the ordering of the variables is alphabetical), then

```
part(a+b,2)  ->   B
part(a+b,-1) ->   B
```

and

```
part(a+b,0)  ->  PLUS
```

An operator ARGLENGTH is available to determine the number of arguments of the top level operator in an expression. If the expression does not contain a top level operator, then -1 is returned. For example,

```
arglength(a+b+c) ->  3
arglength(f())   ->  0
arglength(a)     ->  -1
```

8.5.4 Substituting for Parts of Expressions

PART may also be used to substitute for a given part of an expression. In this case, the PART construct appears on the left-hand side of an assignment statement, and the expression to replace the given part on the right-hand side.

For example, with the normal settings of the REDUCE switches:

```
xx := a+b;
part(xx,2) := c;   ->  A+C
part(c+d,0) := -;   -> C-D
```

Note that xx in the above is not changed by this substitution. In addition, unlike expressions such as array and matrix elements that have an *instant evaluation* property, the values of part(xx,2) and part(c+d,0) are also not changed.

Chapter 9

Polynomials and Rationals

Many operations in computer algebra are concerned with polynomials and rational functions. In this section, we review some of the switches and operators available for this purpose. These are in addition to those that work on general expressions (such as DF and INT) described elsewhere. In the case of operators, the arguments are first simplified before the operations are applied. In addition, they operate only on arguments of prescribed types, and produce a type mismatch error if given arguments which cannot be interpreted in the required mode with the current switch settings. For example, if an argument is required to be a kernel and a/2 is used (with no other rules for A), an error

```
A/2 invalid as kernel
```

will result.

With the exception of those that select various parts of a polynomial or rational function, these operations have potentially significant effects on the space and time associated with a given calculation. The user should therefore experiment with their use in a given calculation in order to determine the optimum set for a given problem.

One such operation provided by the system is an operator LENGTH which returns the number of top level terms in the numerator of its argument. For example,

```
length ((a+b+c)^3/(c+d));
```

has the value 10. To get the number of terms in the denominator, one would first select the denominator by the operator DEN and then call LENGTH, as in

```
length den ((a+b+c)^3/(c+d));
```

Other operations currently supported, the relevant switches and operators, and the required argument and value modes of the latter, follow.

9.1 Controlling the Expansion of Expressions

The switch EXP controls the expansion of expressions. If it is off, no expansion of powers or products of expressions occurs. Users should note however that in this case results come out in a normal but not necessarily canonical form. This means that zero expressions simplify to zero, but that two equivalent expressions need not necessarily simplify to the same form.

Example: With EXP on, the two expressions

```
(a+b)*(a+2*b)
```

and

```
a^2+3*a*b+2*b^2
```

will both simplify to the latter form. With EXP off, they would remain unchanged, unless the complete factoring (ALLFAC) option were in force. EXP is normally on.

Several operators that expect a polynomial as an argument behave differently when EXP is off, since there is often only one term at the top level. For example, with EXP off

```
length((a+b+c)^3/(c+d));
```

returns the value 1.

9.2 Factorization of Polynomials

REDUCE is capable of factorizing univariate and multivariate polynomials that have integer coefficients, finding all factors that also have integer coefficients. The package for doing this was written by Dr. Arthur C. Norman and Ms. P. Mary Ann Moore at The University of Cambridge. It is described in P. M. A. Moore and A. C. Norman, "Implementing a Polynomial Factorization and GCD Package", Proc. SYMSAC '81, ACM (New York) (1981), 109-116.

The easiest way to use this facility is to turn on the switch FACTOR, which

causes all expressions to be output in a factored form. For example, with FACTOR on, the expression A^2-B^2 is returned as (A+B)*(A-B).

It is also possible to factorize a given expression explicitly. The operator FACTORIZE that invokes this facility is used with the syntax

```
FACTORIZE(EXPRN:polynomial[,INTEXP:prime integer]):list,
```

the optional argument of which will be described later. Thus to find and display all factors of the cyclotomic polynomial $x^{105} - 1$, one could write:

```
factorize(x^105-1);
```

The result is a list of factor,exponent pairs. In the above example, there is no overall numerical factor in the result, so the results will consist only of polynomials in x. The number of such polynomials can be found by using the operator LENGTH. If there is a numerical factor, as in factorizing $12x^2 - 12$, that factor will appear as the first member of the result. It will however not be factored further. Prime factors of such numbers can be found, using a probabilistic algorithm, by turning on the switch IFACTOR. For example,

```
on ifactor; factorize(12x^2-12);
```

would result in the output

```
{{2,2},{3,1},{X + 1,1},{X - 1,1}}.
```

If the first argument of FACTORIZE is an integer, it will be decomposed into its prime components, whether or not IFACTOR is on.

Note that the IFACTOR switch only affects the result of FACTORIZE. It has no effect if the FACTOR switch is also on.

The order in which the factors occur in the result (with the exception of a possible overall numerical coefficient which comes first) can be system dependent and should not be relied on. Similarly it should be noted that any pair of individual factors can be negated without altering their product, and that REDUCE may sometimes do that.

The factorizer works by first reducing multivariate problems to univariate ones and then solving the univariate ones modulo small primes. It normally selects both evaluation points and primes using a random number generator that should lead to different detailed behavior each time any particular problem is tackled. If, for some reason, it is known that a certain (probably univariate) factorization can be performed effectively with a known prime, P say, this value of P can be handed to FACTORIZE as a second argument. An

error will occur if a non-prime is provided to FACTORIZE in this manner. It is also an error to specify a prime that divides the discriminant of the polynomial being factored, but users should note that this condition is not checked by the program, so this capability should be used with care.

Factorization can be performed over a number of polynomial coefficient domains in addition to integers. The particular description of the relevant domain should be consulted to see if factorization is supported. For example, the following statements will factorize $x^4 + 1$ modulo 7:

```
setmod 7;
on modular;
factorize(x^4+1);
```

The factorization module is provided with a trace facility that may be useful as a way of monitoring progress on large problems, and of satisfying curiosity about the internal workings of the package. The most simple use of this is enabled by issuing the REDUCE command `on trfac;` . Following this, all calls to the factorizer will generate informative messages reporting on such things as the reduction of multivariate to univariate cases, the choice of a prime and the reconstruction of full factors from their images. Further levels of detail in the trace are intended mainly for system tuners and for the investigation of suspected bugs. For example, TRALLFAC gives tracing information at all levels of detail. The switch that can be set by `on timings;` makes it possible for one who is familiar with the algorithms used to determine what part of the factorization code is consuming the most resources. `on overview;` reduces the amount of detail presented in other forms of trace. Other forms of trace output are enabled by directives of the form

```
symbolic set!-trace!-factor(<number>,<filename>);
```

where useful numbers are 1, 2, 3 and 100, 101, ... This facility is intended to make it possible to discover in fairly great detail what just some small part of the code has been doing — the numbers refer mainly to depths of recursion when the factorizer calls itself, and to the split between its work forming and factorizing images and reconstructing full factors from these. If NIL is used in place of a filename the trace output requested is directed to the standard output stream. After use of this trace facility the generated trace files should be closed by calling

```
symbolic close!-trace!-files();
```

NOTE: Using the factorizer with MCD off will result in an error.

9.3 Cancellation of Common Factors

Facilities are available in REDUCE for cancelling common factors in the numerators and denominators of expressions, at the option of the user. The system will perform this greatest common divisor computation if the switch GCD is on. (GCD is normally off.)

A check is automatically made, however, for common variable and numerical products in the numerators and denominators of expressions, and the appropriate cancellations made.

When GCD is on, and EXP is off, a check is made for square free factors in an expression. This includes separating out and independently checking the content of a given polynomial where appropriate. (For an explanation of these terms, see Anthony C. Hearn, "Non-Modular Computation of Polynomial GCDs Using Trial Division", Proc. EUROSAM 79, published as Lecture Notes on Comp. Science, Springer-Verlag, Berlin, No 72 (1979) 227-239.)

Example: With EXP off and GCD on, the polynomial `a*c+a*d+b*c+b*d` would be returned as `(A+B)*(C+D)`.

Under normal circumstances, GCDs are computed using an algorithm described in the above paper. It is also possible in REDUCE to compute GCDs using an alternative algorithm, called the EZGCD Algorithm, which uses modular arithmetic. The switch EZGCD, if on in addition to GCD, makes this happen.

In non-trivial cases, the EZGCD algorithm is almost always better than the basic algorithm, often by orders of magnitude. We therefore *strongly* advise users to use the EZGCD switch where they have the resources available for supporting the package.

For a description of the EZGCD algorithm, see J. Moses and D.Y.Y. Yun, "The EZ GCD Algorithm", Proc. ACM 1973, ACM, New York (1973) 159-166.

NOTE: This package shares code with the factorizer, so a certain amount of trace information can be produced using the factorizer trace switches.

9.3.1 Determining the GCD of Two Polynomials

This operator, used with the syntax

```
GCD(EXPRN1:polynomial,EXPRN2:polynomial):polynomial,
```

returns the greatest common divisor of the two polynomials EXPRN1 and EXPRN2.

Examples:

```
gcd(x^2+2*x+1,x^2+3*x+2) ->  X+1
gcd(2*x^2-2*y^2,4*x+4*y) ->  2*X+2*Y
gcd(x^2+y^2,x-y)         ->  1.
```

9.4 Working with Least Common Multiples

Greatest common divisor calculations can often become expensive if extensive work with large rational expressions is required. However, in many cases, the only significant cancellations arise from the fact that there are often common factors in the various denominators which are combined when two rationals are added. Since these denominators tend to be smaller and more regular in structure than the numerators, considerable savings in both time and space can occur if a full GCD check is made when the denominators are combined and only a partial check when numerators are constructed. In other words, the true least common multiple of the denominators is computed at each step. The switch LCM is available for this purpose, and is normally on.

In addition, the operator LCM, used with the syntax

```
LCM(EXPRN1:polynomial,EXPRN2:polynomial):polynomial,
```

returns the least common multiple of the two polynomials EXPRN1 and EXPRN2.

Examples:

```
lcm(x^2+2*x+1,x^2+3*x+2) ->  X**3 + 4*X**2 + 5*X + 2
lcm(2*x^2-2*y^2,4*x+4*y) ->  4*(X**2 - Y**2)
```

9.5 Controlling Use of Common Denominators

When two rational functions are added, REDUCE normally produces an expression over a common denominator. However, if the user does not want denominators combined, he or she can turn off the switch MCD which controls this process. The latter switch is particularly useful if no greatest common divisor calculations are desired, or excessive differentiation of rational functions is required.

CAUTION: With MCD off, results are not guaranteed to come out in either normal or canonical form. In other words, an expression equivalent to zero may in fact not be simplified to zero. This option is therefore most useful for

avoiding expression swell during intermediate parts of a calculation.

MCD is normally on.

9.6 REMAINDER Operator

This operator is used with the syntax

```
REMAINDER(EXPRN1:polynomial,EXPRN2:polynomial):polynomial.
```

It returns the remainder when EXPRN1 is divided by EXPRN2. This is the true remainder based on the internal ordering of the variables, and not the pseudo-remainder. The pseudo-remainder and in general pseudo-division of polynomials can be calculated after loading the polydiv package. Please refer to the documentation of this package for details.

Examples:

```
remainder((x+y)*(x+2*y),x+3*y) ->  2*Y**2
remainder(2*x+y,2)             ->  Y.
```

CAUTION: In the default case, remainders are calculated over the integers. If you need the remainder with respect to another domain, it must be declared explicitly.

Example:

```
remainder(x^2-2,x+sqrt(2)); -> X^2 - 2
load_package arnum;
defpoly sqrt2**2-2;
remainder(x^2-2,x+sqrt2); -> 0
```

9.7 RESULTANT Operator

This is used with the syntax

```
RESULTANT(EXPRN1:polynomial,EXPRN2:polynomial,VAR:kernel):
   polynomial.
```

It computes the resultant of the two given polynomials with respect to the given variable, the coefficients of the polynomials can be taken from any domain. The result can be identified as the determinant of a Sylvester matrix, but can often also be thought of informally as the result obtained when the

given variable is eliminated between the two input polynomials. If the two input polynomials have a non-trivial GCD their resultant vanishes.

The switch BEZOUT controls the computation of the resultants. It is off by default. In this case a subresultant algorithm is used. If the switch Bezout is turned on, the resultant is computed via the Bezout Matrix. However, in the latter case, only polynomial coefficients are permitted.

The sign conventions used by the resultant function follow those in R. Loos, "Computing in Algebraic Extensions" in "Computer Algebra — Symbolic and Algebraic Computation", Second Ed., Edited by B. Buchberger, G.E. Collins and R. Loos, Springer-Verlag, 1983. Namely, with A and B not dependent on X:

```
                                deg(p)*deg(q)
resultant(p(x),q(x),x)= (-1)                *resultant(q,p,x)

                              deg(p)
resultant(a,p(x),x)   = a

resultant(a,b,x)      = 1
```

Examples:

```
                                      2
resultant(x/r*u+y,u*y,u)   ->  - y
```

calculation in an algebraic extension:

```
load arnum;
defpoly sqrt2**2 - 2;

resultant(x + sqrt2,sqrt2 * x +1,x) -> -1
```

or in a modular domain:

```
setmod 17;
on modular;

resultant(2x+1,3x+4,x)    -> 5
```

9.8 DECOMPOSE Operator

The DECOMPOSE operator takes a multivariate polynomial as argument, and returns an expression and a list of equations from which the original polynomial can be found by composition. Its syntax is:

```
DECOMPOSE(EXPRN:polynomial):list.
```

For example:

```
decompose(x^8-88*x^7+2924*x^6-43912*x^5+263431*x^4-
              218900*x^3+65690*x^2-7700*x+234)
                 2                  2            2
         -> {U  + 35*U + 234, U=V  + 10*V, V=X  - 22*X}
                                   2
decompose(u^2+v^2+2u*v+1)  -> {W  + 1, W=U + V}
```

Users should note however that, unlike factorization, this decomposition is not unique.

9.9 INTERPOL operator

Syntax:

```
INTERPOL(<values>,<variable>,<points>);
```

where <values> and <points> are lists of equal length and <variable> is an algebraic expression (preferably a kernel).

INTERPOL generates an interpolation polynomial f in the given variable of degree length(<values>)-1. The unique polynomial f is defined by the property that for corresponding elements v of <values> and p of <points> the relation $f(p) = v$ holds.

The Aitken-Neville interpolation algorithm is used which guarantees a stable result even with rounded numbers and an ill-conditioned problem.

9.10 Obtaining Parts of Polynomials and Rationals

These operators select various parts of a polynomial or rational function structure. Except for the cost of rearrangement of the structure, these operations take very little time to perform.

For those operators in this section that take a kernel VAR as their second argument, an error results if the first expression is not a polynomial in VAR, although the coefficients themselves can be rational as long as they do not depend on VAR. However, if the switch RATARG is on, denominators are not checked for dependence on VAR, and are taken to be part of the coefficients.

9.10.1 DEG Operator

This operator is used with the syntax

```
DEG(EXPRN:polynomial,VAR:kernel):integer.
```

It returns the leading degree of the polynomial EXPRN in the variable VAR. If VAR does not occur as a variable in EXPRN, 0 is returned.

Examples:

```
deg((a+b)*(c+2*d)^2,a) ->  1
deg((a+b)*(c+2*d)^2,d) ->  2
deg((a+b)*(c+2*d)^2,e) ->  0.
```

Note also that if RATARG is on,

```
deg((a+b)^3/a,a)       ->  3
```

since in this case, the denominator A is considered part of the coefficients of the numerator in A. With RATARG off, however, an error would result in this case.

9.10.2 DEN Operator

This is used with the syntax:

```
DEN(EXPRN:rational):polynomial.
```

It returns the denominator of the rational expression EXPRN. If EXPRN is a polynomial, 1 is returned.

Examples:

```
den(x/y^2)   ->  Y**2
den(100/6)   ->  3
        [since 100/6 is first simplified to 50/3]
den(a/4+b/6) ->  12
```

```
den(a+b)      ->  1
```

9.10.3 LCOF Operator

LCOF is used with the syntax

```
LCOF(EXPRN:polynomial,VAR:kernel):polynomial.
```

It returns the leading coefficient of the polynomial `EXPRN` in the variable `VAR`. If `VAR` does not occur as a variable in `EXPRN`, `EXPRN` is returned.

Examples:

```
lcof((a+b)*(c+2*d)^2,a) ->  C**2+4*C*D+4*D**2
lcof((a+b)*(c+2*d)^2,d) ->  4*(A+B)
lcof((a+b)*(c+2*d),e)   ->  A*C+2*A*D+B*C+2*B*D
```

9.10.4 LPOWER Operator

Syntax:

```
LPOWER(EXPRN:polynomial,VAR:kernel):polynomial.
```

LPOWER returns the leading power of EXPRN with respect to VAR. If EXPRN does not depend on VAR, 1 is returned.

Examples:

```
lpower((a+b)*(c+2*d)^2,a) ->  A
lpower((a+b)*(c+2*d)^2,d) ->  D**2
lpower((a+b)*(c+2*d),e)   ->  1
```

9.10.5 LTERM Operator

Syntax:

```
LTERM(EXPRN:polynomial,VAR:kernel):polynomial.
```

LTERM returns the leading term of EXPRN with respect to VAR. If EXPRN does not depend on VAR, EXPRN is returned.

Examples:

```
lterm((a+b)*(c+2*d)^2,a) ->  A*(C**2+4*C*D+4*D**2)
lterm((a+b)*(c+2*d)^2,d) ->  4*D**2*(A+B)
lterm((a+b)*(c+2*d),e)   ->  A*C+2*A*D+B*C+2*B*D
```

Compatibility Note: In some earlier versions of REDUCE, LTERM returned 0 if the EXPRN did not depend on VAR. In the present version, EXPRN is always equal to LTERM(EXPRN,VAR) + REDUCT(EXPRN,VAR).

9.10.6 MAINVAR Operator

Syntax:

```
MAINVAR(EXPRN:polynomial):expression.
```

Returns the main variable (based on the internal polynomial representation) of EXPRN. If EXPRN is a domain element, 0 is returned.

Examples:

Assuming A has higher kernel order than B, C, or D:

```
mainvar((a+b)*(c+2*d)^2) ->  A
mainvar(2)               ->  0
```

9.10.7 NUM Operator

Syntax:

```
NUM(EXPRN:rational):polynomial.
```

Returns the numerator of the rational expression EXPRN. If EXPRN is a polynomial, that polynomial is returned.

Examples:

```
num(x/y^2)  ->  X
num(100/6)   ->  50
num(a/4+b/6) ->  3*A+2*B
num(a+b)     ->  A+B
```

9.10.8 REDUCT Operator

Syntax:

```
REDUCT(EXPRN:polynomial,VAR:kernel):polynomial.
```

Returns the reductum of EXPRN with respect to VAR (i.e., the part of EXPRN left after the leading term is removed). If EXPRN does not depend on the variable VAR, 0 is returned.

Examples:

```
reduct((a+b)*(c+2*d),a) ->  B*(C + 2*D)
reduct((a+b)*(c+2*d),d) ->  C*(A + B)
reduct((a+b)*(c+2*d),e) ->  0
```

Compatibility Note: In some earlier versions of REDUCE, REDUCT returned EXPRN if it did not depend on VAR. In the present version, EXPRN is always equal to LTERM(EXPRN,VAR) + REDUCT(EXPRN,VAR).

9.11 Polynomial Coefficient Arithmetic

REDUCE allows for a variety of numerical domains for the numerical coefficients of polynomials used in calculations. The default mode is integer arithmetic, although the possibility of using real coefficients has been discussed elsewhere. Rational coefficients have also been available by using integer coefficients in both the numerator and denominator of an expression, using the ON DIV option to print the coefficients as rationals. However, REDUCE includes several other coefficient options in its basic version which we shall describe in this section. All such coefficient modes are supported in a table-driven manner so that it is straightforward to extend the range of possibilities. A description of how to do this is given in R.J. Bradford, A.C. Hearn, J.A. Padget and E. Schrüfer, "Enlarging the REDUCE Domain of Computation," Proc. of SYMSAC '86, ACM, New York (1986), 100–106.

9.11.1 Rational Coefficients in Polynomials

Instead of treating rational numbers as the numerator and denominator of a rational expression, it is also possible to use them as polynomial coefficients directly. This is accomplished by turning on the switch RATIONAL.

Example: With RATIONAL off, the input expression a/2 would be converted into a rational expression, whose numerator was A and denominator 2. With RATIONAL on, the same input would become a rational expression with numerator 1/2*A and denominator 1. Thus the latter can be used in operations that require polynomial input whereas the former could not.

9.11.2 Real Coefficients in Polynomials

The switch ROUNDED permits the use of arbitrary sized real coefficients in polynomial expressions. The actual precision of these coefficients can be set by the operator PRECISION. For example, precision 50; sets the precision to fifty decimal digits. The default precision is system dependent and can be found by precision 0;. In this mode, denominators are automatically made monic, and an appropriate adjustment is made to the numerator.

Example: With ROUNDED on, the input expression a/2 would be converted into a rational expression whose numerator is 0.5*A and denominator 1.

Internally, REDUCE uses floating point numbers up to the precision supported by the underlying machine hardware, and so-called *bigfloats* for higher precision or whenever necessary to represent numbers whose value cannot be represented in floating point. The internal precision is two decimal digits greater than the external precision to guard against roundoff inaccuracies. Bigfloats represent the fraction and exponent parts of a floating-point number by means of (arbitrary precision) integers, which is a more precise representation in many cases than the machine floating point arithmetic, but not as efficient. If a case arises where use of the machine arithmetic leads to problems, a user can force REDUCE to use the bigfloat representation at all precisions by turning on the switch ROUNDBF. In rare cases, this switch is turned on by the system, and the user informed by the message

```
ROUNDBF turned on to increase accuracy
```

Rounded numbers are normally printed to the specified precision. However, if the user wishes to print such numbers with less precision, the printing precision can be set by the command PRINT_PRECISION. For example, `print_precision 5;` will cause such numbers to be printed with five digits maximum.

Under normal circumstances when ROUNDED is on, REDUCE converts the number 1.0 to the integer 1. If this is not desired, the switch NOCONVERT can be turned on.

Numbers that are stored internally as bigfloats are normally printed with a space between every five digits to improve readability. If this feature is not required, it can be suppressed by turning off the switch BFSPACE.

Further information on the bigfloat arithmetic may be found in T. Sasaki, "Manual for Arbitrary Precision Real Arithmetic System in REDUCE", Department of Computer Science, University of Utah, Technical Note No. TR-8 (1979).

When a real number is input, it is normally truncated to the precision in effect at the time the number is read. If it is desired to keep the full precision of all numbers input, the switch ADJPREC (for *adjust precision*) can be turned on. While on, ADJPREC will automatically increase the precision, when necessary, to match that of any integer or real input, and a message printed to inform the user of the precision increase.

When ROUNDED is on, rational numbers are normally converted to rounded representation. However, if a user wishes to keep such numbers in a rational form until used in an operation that returns a real number, the switch ROUNDALL can be turned off. This switch is normally on.

Results from rounded calculations are returned in rounded form with two

exceptions: if the result is recognized as `0` or `1` to the current precision, the integer result is returned.

9.11.3 Modular Number Coefficients in Polynomials

REDUCE includes facilities for manipulating polynomials whose coefficients are computed modulo a given base. To use this option, two commands must be used; `SETMOD <integer>`, to set the prime modulus, and `ON MODULAR` to cause the actual modular calculations to occur. For example, with `setmod 3;` and `on modular;`, the polynomial `(a+2*b)^3` would become `A^3+2*B^3`.

The argument of `SETMOD` is evaluated algebraically, except that non-modular (integer) arithmetic is used. Thus the sequence

```
setmod 3; on modular; setmod 7;
```

will correctly set the modulus to 7.

Modular numbers are by default represented by integers in the interval [0,p-1] where p is the current modulus. Sometimes it is more convenient to use an equivalent symmetric representation in the interval [-p/2+1,p/2], or more precisely [-floor((p-1)/2), ceiling((p-1)/2)], especially if the modular numbers map objects that include negative quantities. The switch `BALANCED_MOD` allows you to select the symmetric representation for output.

Users should note that the modular calculations are on the polynomial coefficients only. It is not currently possible to reduce the exponents since no check for a prime modulus is made (which would allow x^{p-1} to be reduced to 1 mod p). Note also that any division by a number not co-prime with the modulus will result in the error "Invalid modular division".

9.11.4 Complex Number Coefficients in Polynomials

Although REDUCE routinely treats the square of the variable *i* as equivalent to -1, this is not sufficient to reduce expressions involving *i* to lowest terms, or to factor such expressions over the complex numbers. For example, in the default case,

```
factorize(a^2+1);
```

gives the result

```
{{A**2+1,1}}
```

and

```
(a^2+b^2)/(a+i*b)
```

is not reduced further. However, if the switch COMPLEX is turned on, full complex arithmetic is then carried out. In other words, the above factorization will give the result

```
{{A + I,1},{A - I,1}}
```

and the quotient will be reduced to A-I*B.

The switch COMPLEX may be combined with ROUNDED to give complex real numbers; the appropriate arithmetic is performed in this case.

Complex conjugation is used to remove complex numbers from denominators of expressions. To do this if COMPLEX is off, you must turn the switch RATIONALIZE on.

Chapter 10

Substitution Commands

An important class of commands in REDUCE define substitutions for variables and expressions to be made during the evaluation of expressions. Such substitutions use the prefix operator SUB, various forms of the command LET, and rule sets.

10.1 SUB Operator

Syntax:

```
SUB(<substitution_list>,EXPRN1:algebraic):algebraic
```

where <substitution_list> is a list of one or more equations of the form

```
VAR:kernel=EXPRN:algebraic
```

or a kernel that evaluates to such a list.

The SUB operator gives the algebraic result of replacing every occurrence of the variable VAR in the expression EXPRN1 by the expression EXPRN. Specifically, EXPRN1 is first evaluated using all available rules. Next the substitutions are made, and finally the substituted expression is reevaluated. When more than one variable occurs in the substitution list, the substitution is performed by recursively walking down the tree representing EXPRN1, and replacing every VAR found by the appropriate EXPRN. The EXPRN are not themselves searched for any occurrences of the various VARs. The trivial case SUB(EXPRN1) returns the algebraic value of EXPRN1.

Examples:

```
2              2
 sub({x=a+y,y=y+1},x^2+y^2) -> A  + 2*A*Y + 2*Y  + 2*Y + 1
```

and with `s := {x=a+y,y=y+1}`,

```
2              2
 sub(s,x^2+y^2)             -> A  + 2*A*Y + 2*Y  + 2*Y + 1
```

Note that the global assignments `x:=a+y`, etc., do not take place.

`EXPRN1` can be any valid algebraic expression whose type is such that a substitution process is defined for it (e.g., scalar expressions, lists and matrices). An error will occur if an expression of an invalid type for substitution occurs either in `EXPRN` or `EXPRN1`.

The braces around the substitution list may also be omitted, as in:

```
2              2
 sub(x=a+y,y=y+1,x^2+y^2)   -> A  + 2*A*Y + 2*Y  + 2*Y + 1
```

10.2 LET Rules

Unlike substitutions introduced via `SUB`, `LET` rules are global in scope and stay in effect until replaced or `CLEAR`ed.

The simplest use of the `LET` statement is in the form

`LET <substitution list>`

where `<substitution list>` is a list of rules separated by commas, each of the form:

`<variable> = <expression>`

or

`<prefix operator>(<argument>,...,<argument>) = <expression>`

or

`<argument> <infix operator>,..., <argument> = <expression>`

For example,

```
let {x => y^2,
     h(u,v) => u - v,
     cos(pi/3) => 1/2,
     a*b => c,
     l+m => n,
     w^3 => 2*z - 3,
     z^10 => 0}
```

The list brackets can be left out if preferred. The above rules could also have been entered as seven separate LET statements.

After such LET rules have been input, X will always be evaluated as the square of Y, and so on. This is so even if at the time the LET rule was input, the variable Y had a value other than Y. (In contrast, the assignment x:=y^2 will set X equal to the square of the current value of Y, which could be quite different.)

The rule let a*b=c means that whenever A and B are both factors in an expression their product will be replaced by C. For example, a^5*b^7*w would be replaced by c^5*b^2*w.

The rule for l+m will not only replace all occurrences of l+m by N, but will also normally replace L by n-m, but not M by n-l. A more complete description of this case is given in Section 10.2.5.

The rule pertaining to w^3 will apply to any power of W greater than or equal to the third.

Note especially the last example, let z^10=0. This declaration means, in effect: ignore the tenth or any higher power of Z. Such declarations, when appropriate, often speed up a computation to a considerable degree. (See Section 10.4 for more details.)

Any new operators occurring in such LET rules will be automatically declared OPERATOR by the system, if the rules are being read from a file. If they are being entered interactively, the system will ask DECLARE ... OPERATOR? . Answer Y or N and hit **Return**.

In each of these examples, substitutions are only made for the explicit expressions given; i.e., none of the variables may be considered arbitrary in any sense. For example, the command

```
let h(u,v) = u - v;
```

will cause h(u,v) to evaluate to U - V, but will not affect h(u,z) or H with any arguments other than precisely the symbols U,V.

These simple LET rules are on the same logical level as assignments made

with the := operator. An assignment `x := p+q` cancels a rule `let x = y^2` made earlier, and vice versa.

CAUTION: A recursive rule such as

```
let x = x + 1;
```

is erroneous, since any subsequent evaluation of `X` would lead to a non-terminating chain of substitutions:

```
x -> x + 1 -> (x + 1) + 1 -> ((x + 1) + 1) + 1 -> ...
```

Similarly, coupled substitutions such as

```
let l = m + n, n = l + r;
```

would lead to the same error. As a result, if you try to evaluate an `X`, `L` or `N` defined as above, you will get an error such as

```
X improperly defined in terms of itself
```

Array and matrix elements can appear on the left-hand side of a `LET` statement. However, because of their *instant evaluation* property, it is the value of the element that is substituted for, rather than the element itself. E.g.,

```
array a(5);
a(2) := b;
let a(2) = c;
```

results in `B` being substituted by `C`; the assignment for `a(2)` does not change.

Finally, if an error occurs in any equation in a `LET` statement (including generalized statements involving `FOR ALL` and `SUCH THAT`), the remaining rules are not evaluated.

10.2.1 FOR ALL . . . LET

If a substitution for all possible values of a given argument of an operator is required, the declaration `FOR ALL` may be used. The syntax of such a command is

```
FOR ALL <variable>,...,<variable>
        <LET statement> <terminator>
```

e.g.,

```
for all x,y let h(x,y) = x-y;
for all x let k(x,y) = x^y;
```

The first of these declarations would cause `h(a,b)` to be evaluated as `A-B`, `h(u+v,u+w)` to be `V-W`, etc. If the operator symbol `H` is used with more or fewer argument places, not two, the `LET` would have no effect, and no error would result.

The second declaration would cause `k(a,y)` to be evaluated as `a^y`, but would have no effect on `k(a,z)` since the rule didn't say `FOR ALL Y` ... .

Where we used `X` and `Y` in the examples, any variables could have been used. This use of a variable doesn't affect the value it may have outside the `LET` statement. However, you should remember what variables you actually used. If you want to delete the rule subsequently, you must use the same variables in the `CLEAR` command.

It is possible to use more complicated expressions as a template for a `LET` statement, as explained in the section on substitutions for general expressions. In nearly all cases, the rule will be accepted, and a consistent application made by the system. However, if there is a sole constant or a sole free variable on the left-hand side of a rule (e.g., `let 2=3` or `for all x let x=2`), then the system is unable to handle the rule, and the error message

```
Substitution for ... not allowed
```

will be issued. Any variable listed in the `FOR ALL` part will have its symbol preceded by an equal sign: `X` in the above example will appear as `=X`. An error will also occur if a variable in the `FOR ALL` part is not properly matched on both sides of the `LET` equation.

10.2.2 FOR ALL . . . SUCH THAT . . . LET

If a substitution is desired for more than a single value of a variable in an operator or other expression, but not all values, a conditional form of the `FOR ALL . . . LET` declaration can be used.

Example:

```
for all x such that numberp x and x<0 let h(x)=0;
```

will cause `h(-5)` to be evaluated as 0, but `H` of a positive integer, or of an argument that is not an integer at all, would not be affected. Any boolean expression can follow the `SUCH THAT` keywords.

10.2.3 Removing Assignments and Substitution Rules

The user may remove all assignments and substitution rules from any expression by the command CLEAR, in the form

```
CLEAR <expression>,...,<expression><terminator>
```

e.g.

```
clear x, h(x,y);
```

Because of their *instant evaluation* property, array and matrix elements cannot be cleared with CLEAR. For example, if A is an array, you must say

```
a(3) := 0;
```

rather than

```
clear a(3);
```

to "clear" element a(3).

On the other hand, a whole array (or matrix) A can be cleared by the command clear a; This means much more than resetting to 0 all the elements of A. The fact that A is an array, and what its dimensions are, are forgotten, so A can be redefined as another type of object, for example an operator.

The more general types of LET declarations can also be deleted by using CLEAR. Simply repeat the LET rule to be deleted, using CLEAR in place of LET, and omitting the equal sign and right-hand part. The same dummy variables must be used in the FOR ALL part, and the boolean expression in the SUCH THAT part must be written the same way. (The placing of blanks doesn't have to be identical.)

Example: The LET rule

```
for all x such that numberp x and x<0 let h(x)=0;
```

can be erased by the command

```
for all x such that numberp x and x<0 clear h(x);
```

10.2.4 Overlapping LET Rules

CLEAR is not the only way to delete a LET rule. A new LET rule identical to the first, but with a different expression after the equal sign, replaces the first. Replacements are also made in other cases where the existing rule would be in conflict with the new rule. For example, a rule for x^4 would replace a rule for x^5. The user should however be cautioned against having several LET rules in effect that relate to the same expression. No guarantee can be given as to which rules will be applied by REDUCE or in what order. It is best to CLEAR an old rule before entering a new related LET rule.

10.2.5 Substitutions for General Expressions

The examples of substitutions discussed in other sections have involved very simple rules. However, the substitution mechanism used in REDUCE is very general, and can handle arbitrarily complicated rules without difficulty.

The general substitution mechanism used in REDUCE is discussed in Hearn, A. C., "REDUCE, A User-Oriented Interactive System for Algebraic Simplification," Interactive Systems for Experimental Applied Mathematics, (edited by M. Klerer and J. Reinfelds), Academic Press, New York (1968), 79-90, and Hearn. A. C., "The Problem of Substitution," Proc. 1968 Summer Institute on Symbolic Mathematical Computation, IBM Programming Laboratory Report FSC 69-0312 (1969). For the reasons given in these references, REDUCE does not attempt to implement a general pattern matching algorithm. However, the present system uses far more sophisticated techniques than those discussed in the above papers. It is now possible for the rules appearing in arguments of LET to have the form

```
<substitution expression> = <expression>
```

where any rule to which a sensible meaning can be assigned is permitted. However, this meaning can vary according to the form of <substitution expression>. The semantic rules associated with the application of the substitution are completely consistent, but somewhat complicated by the pragmatic need to perform such substitutions as efficiently as possible. The following rules explain how the majority of the cases are handled.

To begin with, the <substitution expression> is first partly simplified by collecting like terms and putting identifiers (and kernels) in the system order. However, no substitutions are performed on any part of the expression with the exception of expressions with the *instant evaluation* property, such as array and matrix elements, whose actual values are used. It should also be noted that the system order used is not changeable by the user, even with the

KORDER command. Specific cases are then handled as follows:

1. If the resulting simplified rule has a left-hand side that is an identifier, an expression with a top-level algebraic operator or a power, then the rule is added without further change to the appropriate table.

2. If the operator * appears at the top level of the simplified left-hand side, then any constant arguments in that expression are moved to the right-hand side of the rule. The remaining left-hand side is then added to the appropriate table. For example,

```
let 2*x*y=3
```

   becomes

```
let x*y=3/2
```

   so that x*y is added to the product substitution table, and when this rule is applied, the expression x*y becomes 3/2, but X or Y by themselves are not replaced.

3. If the operators +, - or / appear at the top level of the simplified left-hand side, all but the first term is moved to the right-hand side of the rule. Thus the rules

```
let l+m=n, x/2=y, a-b=c
```

   become

```
let l=n-m, x=2*y, a=c+b.
```

One problem that can occur in this case is that if a quantified expression is moved to the right-hand side, a given free variable might no longer appear on the left-hand side, resulting in an error because of the unmatched free variable. E.g.,

```
for all x,y let f(x)+f(y)=x*y
```

would become

```
for all x,y let f(x)=x*y-f(y)
```

which no longer has Y on both sides.

The fact that array and matrix elements are evaluated in the left-hand side of rules can lead to confusion at times. Consider for example the statements

```
array a(5); let x+a(2)=3; let a(3)=4;
```

The left-hand side of the first rule will become X, and the second 0. Thus the first rule will be instantiated as a substitution for X, and the second will result in an error.

The order in which a list of rules is applied is not easily understandable without a detailed knowledge of the system simplification protocol. It is also possible for this order to change from release to release, as improved substitution techniques are implemented. Users should therefore assume that the order of application of rules is arbitrary, and program accordingly.

After a substitution has been made, the expression being evaluated is reexamined in case a new allowed substitution has been generated. This process is continued until no more substitutions can be made.

As mentioned elsewhere, when a substitution expression appears in a product, the substitution is made if that expression divides the product. For example, the rule

```
let a^2*c = 3*z;
```

would cause a^2*c*x to be replaced by 3*Z*X and a^2*c^2 by 3*Z*C. If the substitution is desired only when the substitution expression appears in a product with the explicit powers supplied in the rule, the command MATCH should be used instead.

For example,

```
match a^2*c = 3*z;
```

would cause a^2*c*x to be replaced by 3*Z*X, but a^2*c^2 would not be replaced. MATCH can also be used with the FOR ALL constructions described above.

To remove substitution rules of the type discussed in this section, the CLEAR command can be used, combined, if necessary, with the same FOR ALL clause with which the rule was defined, for example:

```
for all x clear log(e^x),e^log(x),cos(w*t+theta(x));
```

Note, however, that the arbitrary variable names in this case *must* be the same as those used in defining the substitution.

10.3 Rule Lists

Rule lists offer an alternative approach to defining substitutions that is different from either SUB or LET. In fact, they provide the best features of both, since they have all the capabilities of LET, but the rules can also be applied locally as is possible with SUB. In time, they will be used more and more in REDUCE. However, since they are relatively new, much of the REDUCE code you see uses the older constructs.

A rule list is a list of *rules* that have the syntax

```
<expression> => <expression> (WHEN <boolean expression>)
```

For example,

```
{cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2,
 cos(~n*pi)      => (-1)^n when remainder(n,2)=0}
```

The tilde preceding a variable marks that variable as *free* for that rule, much as a variable in a FOR ALL clause in a LET statement. The first occurrence of that variable in each relevant rule must be so marked on input, otherwise inconsistent results can occur. For example, the rule list

```
        {cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2,
cos(x)^2         => (1+cos(2x))/2}
```

designed to replace products of cosines, would not be correct, since the second rule would only apply to the explicit argument X. Later occurrences in the same rule may also be marked, but this is optional (internally, all such rules are stored with each relevant variable explicitly marked). The optional WHEN clause allows constraints to be placed on the application of the rule, much as the SUCH THAT clause in a LET statement.

A rule list may be named, for example

```
trig1 := {cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2,
          cos(~x)*sin(~y) => (sin(x+y)-sin(x-y))/2,
          sin(~x)*sin(~y) => (cos(x-y)-cos(x+y))/2,
          cos(~x)^2       => (1+cos(2*x))/2,
          sin(~x)^2       => (1-cos(2*x))/2};
```

Such named rule lists may be inspected as needed. E.g., the command `trig1;` would cause the above list to be printed.

Rule lists may be used in two ways. They can be globally instantiated by

means of the command LET. For example,

```
let trig1;
```

would cause the above list of rules to be globally active from then on until cancelled by the command CLEARRULES, as in

```
clearrules trig1;
```

CLEARRULES has the syntax

```
CLEARRULES <rule list>|<name of rule list>(,...) .
```

The second way to use rule lists is to invoke them locally by means of a WHERE clause. For example

```
cos(a)*cos(b+c)
where {cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2};
```

or

```
cos(a)*sin(b) where trigrules;
```

The syntax of an expression with a WHERE clause is:

```
<expression>
    WHERE <rule>|<rule list>(,<rule>|<rule list> ...)
```

so the first example above could also be written

```
cos(a)*cos(b+c)
   where cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2;
```

The effect of this construct is that the rule list(s) in the WHERE clause only apply to the expression on the left of WHERE. They have no effect outside the expression. In particular, they do not affect previously defined WHERE clauses or LET statements. For example, the sequence

```
let a=2;
a where a=>4;
a;
```

would result in the output

```
4
```

```
2
```

Although WHERE has a precedence less than any other infix operator, it still binds higher than keywords such as ELSE, THEN, DO, REPEAT and so on. Thus the expression

```
if a=2 then 3 else a+2 where a=3
```

will parse as

```
if a=2 then 3 else (a+2 where a=3)
```

WHERE may be used to introduce auxiliary variables in symbolic mode expressions, as described in Section 16.4. However, the symbolic mode use has different semantics, so expressions do not carry from one mode to the other.

Compatibility Note: In order to provide compatibility with older versions of rule lists released through the Network Library, it is currently possible to use an equal sign interchangeably with the replacement sign => in rules and LET statements. However, since this will change in future versions, the replacement sign is preferable in rules and the equal sign in non-rule-based LET statements.

Advanced Use of Rule Lists

Some advanced features of the rule list mechanism make it possible to write more complicated rules than those discussed so far, and in many cases to write more compact rule lists. These features are:

- Free operators
- Double slash operator
- Double tilde variables.

A free operator in the left hand side of a pattern will match any operator with the same number of arguments. The free operator is written in the same style as a variable. For example, the implementation of the product rule of differentiation can be written as:

```
operator diff, !~f, !~g;

prule := {diff(~f(~x) * ~g(~x),x) =>
```

```
     diff(f(x),x) * g(x) + diff(g(x),x) * f(x)};

let prule;

diff(sin(z)*cos(z),z);

         cos(z)*diff(sin(z),z) + diff(cos(z),z)*sin(z)
```

The double slash operator may be used as an alternative to a single slash (quotient) in order to match quotients properly. E.g., in the example of the Gamma function above, one can use:

```
gammarule :=
   {gamma(~z)//(~c*gamma(~zz))  => gamma(z)/(c*gamma(zz-1)*zz)
  when fixp(zz -z) and (zz -z) >0,
    gamma(~z)//gamma(~zz) => gamma(z)/(gamma(zz-1)*zz)
  when fixp(zz -z) and (zz -z) >0};

let gammarule;

gamma(z)/gamma(z+3);

          1
----------------------
  3      2
 z  + 6*z  + 11*z + 6
```

The above example suffers from the fact that two rules had to be written in order to perform the required operation. This can be simplified by the use of double tilde variables. E.g. the rule list

```
 GGrule :=  {
    gamma(~z)//(~~c*gamma(~zz))  => gamma(z)/(c*gamma(zz-1)*zz)
     when fixp(zz -z) and (zz -z) >0};
```

will implement the same operation in a much more compact way. In general, double tilde variables are bound to the neutral element with respect to the operation in which they are used.

Pattern given	Argument used	Binding
~z + ~~y	x	z=x; y=0
~z + ~~y	x+3	z=x; y=3 or z=3; y=x
~z * ~~y	x	z=x; y=1
~z * ~~y	x*3	z=x; y=3 or z=3; y=x
~z / ~~y	x	z=x; y=1
~z / ~~y	x/3	z=x; y=3

Remarks: A double tilde variable as the numerator of a pattern is not allowed. Also, using double tilde variables may lead to recursion errors when the zero case is not handled properly.

```
let f(~~a * ~x,x)  => a * f(x,x) when freeof (a,x);

f(z,z);

***** f(z,z) improperly defined in terms of itself

% BUT:

let ff(~~a * ~x,x)
       => a * ff(x,x) when freeof (a,x) and a neq 1;

ff(z,z);
                  ff(z,z)

ff(3*z,z);
                  3*ff(z,z)
```

Displaying Rules Associated with an Operator

The operator SHOWRULES takes a single identifier as argument, and returns in rule-list form the operator rules associated with that argument. For example:

```
showrules log;

{LOG(E) => 1,

 LOG(1) => 0,
```

```
      ~X
 LOG(E  ) => ~X,

    1
 DF(LOG(~X),~X) => ----}
    ~X
```

Such rules can then be manipulated further as with any list. For example `rhs first ws;` has the value `1`. Note that an operator may have other properties that cannot be displayed in such a form, such as the fact it is an odd function, or has a definition defined as a procedure.

Order of Application of Rules

If rules have overlapping domains, their order of application is important. In general, it is very difficult to specify this order precisely, so that it is best to assume that the order is arbitrary. However, if only one operator is involved, the order of application of the rules for this operator can be determined from the following:

1. Rules containing at least one free variable apply before all rules without free variables.
2. Rules activated in the most recent `LET` command are applied first.
3. `LET` with several entries generate the same order of application as a corresponding sequence of commands with one rule or rule set each.
4. Within a rule set, the rules containing at least one free variable are applied in their given order. In other words, the first member of the list is applied first.
5. Consistent with the first item, any rule in a rule list that contains no free variables is applied after all rules containing free variables.

Example: The following rule set enables the computation of exact values of the Gamma function:

```
operator gamma,gamma_error;
gamma_rules :=
{gamma(~x)=>sqrt(pi)/2 when x=1/2,
 gamma(~n)=>factorial(n-1) when fixp n and n>0,
 gamma(~n)=>gamma_error(n) when fixp n,
 gamma(~x)=>(x-1)*gamma(x-1) when fixp(2*x) and x>1,
 gamma(~x)=>gamma(x+1)/x when fixp(2*x)};
```

Here, rule by rule, cases of known or definitely uncomputable values are sorted out; e.g. the rule leading to the error expression will be applied for negative integers only, since the positive integers are caught by the preceding rule, and the last rule will apply for negative odd multiples of $1/2$ only. Alternatively the first rule could have been written as

```
gamma(1/2) => sqrt(pi)/2,
```

but then the case $x = 1/2$ should be excluded in the `WHEN` part of the last rule explicitly because a rule without free variables cannot take precedence over the other rules.

10.4 Asymptotic Commands

In expansions of polynomials involving variables that are known to be small, it is often desirable to throw away all powers of these variables beyond a certain point to avoid unnecessary computation. The command `LET` may be used to do this. For example, if only powers of `X` up to `x^7` are needed, the command

```
let x^8 = 0;
```

will cause the system to delete all powers of `X` higher than 7.

CAUTION: This particular simplification works differently from most substitution mechanisms in REDUCE in that it is applied during polynomial manipulation rather than to the whole evaluated expression. Thus, with the above rule in effect, `x^10/x^5` would give the result zero, since the numerator would simplify to zero. Similarly `x^20/x^10` would give a `Zero divisor` error message, since both numerator and denominator would first simplify to zero.

The method just described is not adequate when expressions involve several variables having different degrees of smallness. In this case, it is necessary to supply an asymptotic weight to each variable and count up the total weight of each product in an expanded expression before deciding whether to keep the term or not. There are two associated commands in the system to permit this type of asymptotic constraint. The command `WEIGHT` takes a list of equations of the form

```
<kernel form> = <number>
```

where `<number>` must be a positive integer (not just evaluate to a positive integer). This command assigns the weight `<number>` to the relevant kernel

form. A check is then made in all algebraic evaluations to see if the total weight of the term is greater than the weight level assigned to the calculation. If it is, the term is deleted. To compute the total weight of a product, the individual weights of each kernel form are multiplied by their corresponding powers and then added.

The weight level of the system is initially set to 1. The user may change this setting by the command

```
wtlevel <number>;
```

which sets `<number>` as the new weight level of the system. `<number>` must evaluate to a positive integer. WTLEVEL will also allow NIL as an argument, in which case the current weight level is returned.

Chapter 11

File Handling Commands

In many applications, it is desirable to load previously prepared REDUCE files into the system, or to write output on other files. REDUCE offers four commands for this purpose, namely, IN, OUT, SHUT, LOAD, and LOAD_PACKAGE. The first three operators are described here; LOAD and LOAD_PACKAGE are discussed in Section 18.2.

11.1 IN Command

This command takes a list of file names as argument and directs the system to input each file (that should contain REDUCE statements and commands) into the system. File names can either be an identifier or a string. The explicit format of these will be system dependent and, in many cases, site dependent. The explicit instructions for the implementation being used should therefore be consulted for further details. For example:

```
in f1,"ggg.rr.s";
```

will first load file F1, then ggg.rr.s. When a semicolon is used as the terminator of the IN statement, the statements in the file are echoed on the terminal or written on the current output file. If $ is used as the terminator, the input is not shown. Echoing of all or part of the input file can be prevented, even if a semicolon was used, by placing an `off echo;` command in the input file.

Files to be read using IN should end with ;END;. Note the two semicolons! First of all, this is protection against obscure difficulties the user will have if there are, by mistake, more BEGINs than ENDs on the file. Secondly, it triggers some file control book-keeping which may improve system efficiency. If END

is omitted, an error message "End-of-file read" will occur.

While a file is being loaded, the special identifier _LINE_ is replaced by the number of the current line in the file currently being read.

11.2 OUT Command

This command takes a single file name as argument, and directs output to that file from then on, until another OUT changes the output file, or SHUT closes it. Output can go to only one file at a time, although many can be open. If the file has previously been used for output during the current job, and not SHUT, the new output is appended to the end of the file. Any existing file is erased before its first use for output in a job, or if it had been SHUT before the new OUT.

To output on the terminal without closing the output file, the reserved file name T (for terminal) may be used. For example, out ofile; will direct output to the file OFILE and out t; will direct output to the user's terminal.

The output sent to the file will be in the same form that it would have on the terminal. In particular x^2 would appear on two lines, an X on the lower line and a 2 on the line above. If the purpose of the output file is to save results to be read in later, this is not an appropriate form. We first must turn off the NAT switch that specifies that output should be in standard mathematical notation.

Example: To create a file ABCD from which it will be possible to read - using IN - the value of the expression XYZ:

```
off echo$       % needed if your input is from a file.
off nat$        % output in IN-readable form. Each expression
                % printed will end with a $ .
out abcd$       % output to new file
linelength 72$ % for systems with fixed input line length.
xyz:=xyz;       % will output "XYZ := " followed by the value
                % of XYZ
write ";end"$  % standard for ending files for IN
shut abcd$      % save ABCD, return to terminal output
on nat$                 % restore usual output form
```

11.3 SHUT Command

This command takes a list of names of files that have been previously opened via an `OUT` statement and closes them. Most systems require this action by the user before he ends the REDUCE job (if not sooner), otherwise the output may be lost. If a file is shut and a further `OUT` command issued for the same file, the file is erased before the new output is written.

If it is the current output file that is shut, output will switch to the terminal. Attempts to shut files that have not been opened by `OUT`, or an input file, will lead to errors.

Chapter 12

Commands for Interactive Use

REDUCE is designed as an interactive system, but naturally it can also operate in a batch processing or background mode by taking its input command by command from the relevant input stream. There is a basic difference, however, between interactive and batch use of the system. In the former case, whenever the system discovers an ambiguity at some point in a calculation, such as a forgotten type assignment for instance, it asks the user for the correct interpretation. In batch operation, it is not practical to terminate the calculation at such points and require resubmission of the job, so the system makes the most obvious guess of the user's intentions and continues the calculation.

There is also a difference in the handling of errors. In the former case, the computation can continue since the user has the opportunity to correct the mistake. In batch mode, the error may lead to consequent erroneous (and possibly time consuming) computations. So in the default case, no further evaluation occurs, although the remainder of the input is checked for syntax errors. A message `"Continuing with parsing only"` informs the user that this is happening. On the other hand, the switch `ERRCONT`, if on, will cause the system to continue evaluating expressions after such errors occur.

When a syntactical error occurs, the place where the system detected the error is marked with three dollar signs ($$$). In interactive mode, the user can then use `ED` to correct the error, or retype the command. When a non-syntactical error occurs in interactive mode, the command being evaluated at the time the last error occurred is saved, and may later be reevaluated by the command `RETRY`.

12.1 Referencing Previous Results

It is often useful to be able to reference results of previous computations during a REDUCE session. For this purpose, REDUCE maintains a history of all interactive inputs and the results of all interactive computations during a given session. These results are referenced by the command number that REDUCE prints automatically in interactive mode. To use an input expression in a new computation, one writes `input(`n`)`, where n is the command number. To use an output expression, one writes `WS(`n`)`. `WS` references the previous command. E.g., if command number 1 was `INT(X-1,X)`; and the result of command number 7 was `X-1`, then

```
2*input(1)-ws(7)^2;
```

would give the result `-1`, whereas

```
2*ws(1)-ws(7)^2;
```

would yield the same result, but *without* a recomputation of the integral.

The operator `DISPLAY` is available to display previous inputs. If its argument is a positive integer, *n* say, then the previous n inputs are displayed. If its argument is `ALL` (or in fact any non-numerical expression), then all previous inputs are displayed.

12.2 Interactive Editing

It is possible when working interactively to edit any REDUCE input that comes from the user's terminal, and also some user-defined procedure definitions. At the top level, one can access any previous command string by the command `ed(`n`)`, where n is the desired command number as prompted by the system in interactive mode. `ED`; (i.e. no argument) accesses the previous command.

After `ED` has been called, you can now edit the displayed string using a string editor with the following commands:

B	move pointer to beginning
C<character>	replace next character by *character*
D	delete next character
E	end editing and reread text
F<character>	move pointer to next occurrence of *character*
I<string><escape>	insert *string* in front of pointer
K<character>	delete all characters until *character*
P	print string from current pointer
Q	give up with error exit
S<string><escape>	search for first occurrence of *string*, positioning pointer just before it
space or X	move pointer right one character.

The above table can be displayed online by typing a question mark followed by a carriage return to the editor. The editor prompts with an angle bracket. Commands can be combined on a single line, and all command sequences must be followed by a carriage return to become effective.

Thus, to change the command `x := a+1;` to `x := a+2;` and cause it to be executed, the following edit command sequence could be used:

```
f1c2e<return>.
```

The interactive editor may also be used to edit a user-defined procedure that has not been compiled. To do this, one says:

```
editdef <id>;
```

where `<id>` is the name of the procedure. The procedure definition will then be displayed in editing mode, and may then be edited and redefined on exiting from the editor.

Some versions of REDUCE now include input editing that uses the capabilities of modern window systems. Please consult your system dependent documentation to see if this is possible. Such editing techniques are usually much easier to use then `ED` or `EDITDEF`.

12.3 Interactive File Control

If input is coming from an external file, the system treats it as a batch processed calculation. If the user desires interactive response in this case, he can include the command `ON INT;` in the file. Likewise, he can issue the command `off int;` in the main program if he does not desire continual

questioning from the system. Regardless of the setting of INT, input commands from a file are not kept in the system, and so cannot be edited using ED. However, many implementations of REDUCE provide a link to an external system editor that can be used for such editing. The specific instructions for the particular implementation should be consulted for information on this.

Two commands are available in REDUCE for interactive use of files. PAUSE; may be inserted at any point in an input file. When this command is encountered on input, the system prints the message CONT? on the user's terminal and halts. If the user responds Y (for yes), the calculation continues from that point in the file. If the user responds N (for no), control is returned to the terminal, and the user can input further statements and commands. Later on he can use the command `cont;` to transfer control back to the point in the file following the last PAUSE encountered. A top-level `pause;` from the user's terminal has no effect.

Chapter 13

Matrix Calculations

A very powerful feature of REDUCE is the ease with which matrix calculations can be performed. To extend our syntax to this class of calculations we need to add another prefix operator, `MAT`, and a further variable and expression type as follows:

13.1 MAT Operator

This prefix operator is used to represent $n \times m$ matrices. `MAT` has *n* arguments interpreted as rows of the matrix, each of which is a list of *m* expressions representing elements in that row. For example, the matrix

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$$

would be written as `mat((a,b,c),(d,e,f))`.

Note that the single column matrix

$$\begin{pmatrix} x \\ y \end{pmatrix}$$

becomes `mat((x),(y))`. The inside parentheses are required to distinguish it from the single row matrix

$$\begin{pmatrix} x & y \end{pmatrix}$$

that would be written as `mat((x,y))`.

13.2 Matrix Variables

An identifier may be declared a matrix variable by the declaration `MATRIX`. The size of the matrix may be declared explicitly in the matrix declaration, or by default in assigning such a variable to a matrix expression. For example,

```
matrix x(2,1),y(3,4),z;
```

declares `X` to be a 2 x 1 (column) matrix, `Y` to be a 3 x 4 matrix and `Z` a matrix whose size is to be declared later.

Matrix declarations can appear anywhere in a program. Once a symbol is declared to name a matrix, it can not also be used to name an array, operator or a procedure, or used as an ordinary variable. It can however be redeclared to be a matrix, and its size may be changed at that time. Note however that matrices once declared are *global* in scope, and so can then be referenced anywhere in the program. In other words, a declaration within a block (or a procedure) does not limit the scope of the matrix to that block, nor does the matrix go away on exiting the block (use `CLEAR` instead for this purpose). An element of a matrix is referred to in the expected manner; thus `x(1,1)` gives the first element of the matrix `X` defined above. References to elements of a matrix whose size has not yet been declared leads to an error. All elements of a matrix whose size is declared are initialized to 0. As a result, a matrix element has an *instant evaluation* property and cannot stand for itself. If this is required, then an operator should be used to name the matrix elements as in:

```
matrix m; operator x;  m := mat((x(1,1),x(1,2));
```

13.3 Matrix Expressions

These follow the normal rules of matrix algebra as defined by the following syntax:

```
<matrix expression> ::=
          MAT<matrix description>|<matrix variable>|
          <scalar expression>*<matrix expression>|
          <matrix expression>*<matrix expression>
          <matrix expression>+<matrix expression>|
          <matrix expression>^<integer>|
          <matrix expression>/<matrix expression>
```

Sums and products of matrix expressions must be of compatible size; otherwise an error will result during their evaluation. Similarly, only square matrices may be raised to a power. A negative power is computed as the inverse of the matrix raised to the corresponding positive power. `a/b` is interpreted as `a*b^(-1)`.

Examples:

Assuming `X` and `Y` have been declared as matrices, the following are matrix expressions

```
y
y^2*x-3*y^(-2)*x
y + mat((1,a),(b,c))/2
```

The computation of the quotient of two matrices normally uses a two-step elimination method due to Bareiss. An alternative method using Cramer's method is also available. This is usually less efficient than the Bareiss method unless the matrices are large and dense, although we have no solid statistics on this as yet. To use Cramer's method instead, the switch `CRAMER` should be turned on.

13.4 Operators with Matrix Arguments

The operator `LENGTH` applied to a matrix returns a list of the number of rows and columns in the matrix. Other operators useful in matrix calculations are defined in the following subsections. Attention is also drawn to the LINALG (chapter 15.34) and NORMFORM (chapter 15.38) packages.

13.4.1 DET Operator

Syntax:

```
DET(EXPRN:matrix_expression):algebraic.
```

The operator `DET` is used to represent the determinant of a square matrix expression. E.g.,

```
det(y^2)
```

is a scalar expression whose value is the determinant of the square of the matrix `Y`, and

```
det mat((a,b,c),(d,e,f),(g,h,j));
```

is a scalar expression whose value is the determinant of the matrix

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & j \end{pmatrix}$$

Determinant expressions have the *instant evaluation* property. In other words, the statement

```
let det mat((a,b),(c,d)) = 2;
```

sets the *value* of the determinant to 2, and does not set up a rule for the determinant itself.

13.4.2 MATEIGEN Operator

Syntax:

```
MATEIGEN(EXPRN:matrix_expression,ID):list.
```

`MATEIGEN` calculates the eigenvalue equation and the corresponding eigenvectors of a matrix, using the variable `ID` to denote the eigenvalue. A square free decomposition of the characteristic polynomial is carried out. The result is a list of lists of 3 elements, where the first element is a square free factor of the characteristic polynomial, the second its multiplicity and the third the corresponding eigenvector (as an *n* by 1 matrix). If the square free decomposition was successful, the product of the first elements in the lists is the minimal polynomial. In the case of degeneracy, several eigenvectors can exist for the same eigenvalue, which manifests itself in the appearance of more than one arbitrary variable in the eigenvector. To extract the various parts of the result use the operations defined on lists.

Example: The command

```
mateigen(mat((2,-1,1),(0,1,1),(-1,1,1)),eta);
```

gives the output

```
{{ETA - 1,2,

  [ARBCOMPLEX(1)]
  [             ]
```

```
 [ARBCOMPLEX(1)]
 [              ]
 [      0       ]

 },

{ETA - 2,1,

 [      0       ]
 [              ]
 [ARBCOMPLEX(2)]
 [              ]
 [ARBCOMPLEX(2)]

 }}
```

13.4.3 TP Operator

Syntax:

```
TP(EXPRN:matrix_expression):matrix.
```

This operator takes a single matrix argument and returns its transpose.

13.4.4 Trace Operator

Syntax:

```
TRACE(EXPRN:matrix_expression):algebraic.
```

The operator TRACE is used to represent the trace of a square matrix.

13.4.5 Matrix Cofactors

Syntax:

```
COFACTOR(EXPRN:matrix_expression,ROW:integer,COLUMN:integer):
 algebraic
```

The operator COFACTOR returns the cofactor of the element in row ROW and column COLUMN of the matrix MATRIX. Errors occur if ROW or COLUMN do not simplify to integer expressions or if MATRIX is not square.

13.4.6 NULLSPACE Operator

Syntax:

```
NULLSPACE(EXPRN:matrix_expression):list
```

NULLSPACE calculates for a matrix A a list of linear independent vectors (a basis) whose linear combinations satisfy the equation $Ax = 0$. The basis is provided in a form such that as many upper components as possible are isolated.

Note that with `b := nullspace a` the expression `length b` is the *nullity* of A, and that `second length a - length b` calculates the *rank* of A. The rank of a matrix expression can also be found more directly by the RANK operator described below.

Example: The command

```
nullspace mat((1,2,3,4),(5,6,7,8));
```

gives the output

```
{
 [ 1  ]
 [    ]
 [ 0  ]
 [    ]
 [ - 3]
 [    ]
 [ 2  ]
 ,
 [ 0  ]
 [    ]
 [ 1  ]
 [    ]
 [ - 2]
 [    ]
 [ 1  ]
 }
```

In addition to the REDUCE matrix form, NULLSPACE accepts as input a matrix given as a list of lists, that is interpreted as a row matrix. If that form of input is chosen, the vectors in the result will be represented by lists as well. This additional input syntax facilitates the use of NULLSPACE in applications different from classical linear algebra.

13.4.7 RANK Operator

Syntax:

```
RANK(EXPRN:matrix_expression):integer
```

RANK calculates the rank of its argument, that, like NULLSPACE can either be a standard matrix expression, or a list of lists, that can be interpreted either as a row matrix or a set of equations.

Example:

```
rank mat((a,b,c),(d,e,f));
```

returns the value 2.

13.5 Matrix Assignments

Matrix expressions may appear in the right-hand side of assignment statements. If the left-hand side of the assignment, which must be a variable, has not already been declared a matrix, it is declared by default to the size of the right-hand side. The variable is then set to the value of the right-hand side.

Such an assignment may be used very conveniently to find the solution of a set of linear equations. For example, to find the solution of the following set of equations

```
a11*x(1) + a12*x(2) = y1
a21*x(1) + a22*x(2) = y2
```

we simply write

```
x := 1/mat((a11,a12),(a21,a22))*mat((y1),(y2));
```

13.6 Evaluating Matrix Elements

Once an element of a matrix has been assigned, it may be referred to in standard array element notation. Thus y(2,1) refers to the element in the second row and first column of the matrix Y.

Chapter 14

Procedures

It is often useful to name a statement for repeated use in calculations with varying parameters, or to define a complete evaluation procedure for an operator. REDUCE offers a procedural declaration for this purpose. Its general syntax is:

```
[<procedural type>] PROCEDURE <name>[<varlist>];<statement>;
```

where

```
<varlist> ::= (<variable>,...,<variable>)
```

This will be explained more fully in the following sections.

In the algebraic mode of REDUCE the `<procedure type>` can be omitted, since the default is `ALGEBRAIC`. Procedures of type `INTEGER` or `REAL` may also be used. In the former case, the system checks that the value of the procedure is an integer. At present, such checking is not done for a real procedure, although this will change in the future when a more complete type checking mechanism is installed. Users should therefore only use these types when appropriate. An empty variable list may also be omitted.

All user-defined procedures are automatically declared to be operators.

In order to allow users relatively easy access to the whole REDUCE source program, system procedures are not protected against user redefinition. If a procedure is redefined, a message

```
*** <procedure name> REDEFINED
```

is printed. If this occurs, and the user is not redefining his own procedure, he is well advised to rename it, and possibly start over (because he has *al-*

ready redefined some internal procedure whose correct functioning may be required for his job!)

All required procedures should be defined at the top level, since they have global scope throughout a program. In particular, an attempt to define a procedure within a procedure will cause an error to occur.

14.1 Procedure Heading

Each procedure has a heading consisting of the word `PROCEDURE` (optionally preceded by the word `ALGEBRAIC`), followed by the name of the procedure to be defined, and followed by its formal parameters – the symbols that will be used in the body of the definition to illustrate what is to be done. There are three cases:

1. No parameters. Simply follow the procedure name with a terminator (semicolon or dollar sign).

   ```
   procedure abc;
   ```

   When such a procedure is used in an expression or command, `abc()`, with empty parentheses, must be written.

2. One parameter. Enclose it in parentheses *or* just leave at least one space, then follow with a terminator.

   ```
   procedure abc(x);
   ```

   or

   ```
   procedure abc x;
   ```

3. More than one parameter. Enclose them in parentheses, separated by commas, then follow with a terminator.

   ```
   procedure abc(x,y,z);
   ```

Referring to the last example, if later in some expression being evaluated the symbols `abc(u,p*q,123)` appear, the operations of the procedure body will be carried out as if `X` had the same value as `U` does, `Y` the same value as `p*q` does, and `Z` the value 123. The values of `X`, `Y`, `Z`, after the procedure body operations are completed are unchanged. So, normally, are the values of `U`, `P`, `Q`, and (of course) 123. (This is technically referred to as call by value.)

The reader will have noted the word *normally* a few lines earlier. The call by value protections can be bypassed if necessary, as described elsewhere.

14.2 Procedure Body

Following the delimiter that ends the procedure heading must be a *single* statement defining the action to be performed or the value to be delivered. A terminator must follow the statement. If it is a semicolon, the name of the procedure just defined is printed. It is not printed if a dollar sign is used.

If the result wanted is given by a formula of some kind, the body is just that formula, using the variables in the procedure heading.

Simple Example:

If `f(x)` is to mean `(x+5)*(x+6)/(x+7)`, the entire procedure definition could read

```
procedure f x; (x+5)*(x+6)/(x+7);
```

Then `f(10)` would evaluate to 240/17, `f(a-6)` to `A*(A-1)/(A+1)`, and so on.

More Complicated Example:

Suppose we need a function `p(n,x)` that, for any positive integer `N`, is the Legendre polynomial of order *n*. We can define this operator using the textbook formula defining these functions:

$$p_n(x) = \frac{1}{n!}\,\frac{d^n}{dy^n}\,\frac{1}{(y^2-2xy+1)^{\frac{1}{2}}}\Big|_{y=0}$$

Put into words, the Legendre polynomial $p_n(x)$ is the result of substituting $y = 0$ in the n^{th} partial derivative with respect to y of a certain fraction involving x and y, then dividing that by $n!$.

This verbal formula can easily be written in REDUCE:

```
procedure p(n,x);
   sub(y=0,df(1/(y^2-2*x*y+1)^(1/2),y,n))
       /(for i:=1:n product i);
```

Having input this definition, the expression evaluation

```
2p(2,w);
```

would result in the output

```
   2
3*W  - 1 .
```

If the desired process is best described as a series of steps, then a group or

compound statement can be used.

Example:

The above Legendre polynomial example can be rewritten as a series of steps instead of a single formula as follows:

```
procedure p(n,x);
  begin scalar seed,deriv,top,fact;
       seed:=1/(y^2 - 2*x*y +1)^(1/2);
       deriv:=df(seed,y,n);
       top:=sub(y=0,deriv);
       fact:=for i:=1:n product i;
       return top/fact
  end;
```

Procedures may also be defined recursively. In other words, the procedure body can include references to the procedure name itself, or to other procedures that themselves reference the given procedure. As an example, we can define the Legendre polynomial through its standard recurrence relation:

```
procedure p(n,x);
   if n<0 then rederr "Invalid argument to P(N,X)"
    else if n=0 then 1
    else if n=1 then x
    else ((2*n-1)*x*p(n-1,x)-(n-1)*p(n-2,x))/n;
```

The operator REDERR in the above example provides for a simple error exit from an algebraic procedure (and also a block). It can take a string as argument.

It should be noted however that all the above definitions of p(n,x) are quite inefficient if extensive use is to be made of such polynomials, since each call effectively recomputes all lower order polynomials. It would be better to store these expressions in an array, and then use say the recurrence relation to compute only those polynomials that have not already been derived. We leave it as an exercise for the reader to write such a definition.

14.3 Using LET Inside Procedures

By using LET instead of an assignment in the procedure body it is possible to bypass the call-by-value protection. If X is a formal parameter or local variable of the procedure (i.e. is in the heading or in a local declaration), and LET is used instead of := to make an assignment to X, e.g.

```
let x = 123;
```

then it is the variable that is the value of X that is changed. This effect also occurs with local variables defined in a block. If the value of X is not a variable, but a more general expression, then it is that expression that is used on the left-hand side of the LET statement. For example, if X had the value p*q, it is as if `let p*q = 123` had been executed.

14.4 LET Rules as Procedures

The LET statement offers an alternative syntax and semantics for procedure definition.

In place of

```
procedure abc(x,y,z); <procedure body>;
```

one can write

```
for all x,y,z let abc(x,y,z) = <procedure body>;
```

There are several differences to note.

If the procedure body contains an assignment to one of the formal parameters, e.g.

```
x := 123;
```

in the PROCEDURE case it is a variable holding a copy of the first actual argument that is changed. The actual argument is not changed.

In the LET case, the actual argument is changed. Thus, if ABC is defined using LET, and `abc(u,v,w)` is evaluated, the value of U changes to 123. That is, the LET form of definition allows the user to bypass the protections that are enforced by the call by value conventions of standard PROCEDURE definitions.

Example: We take our earlier FACTORIAL procedure and write it as a LET statement.

```
for all n let factorial n =
            begin scalar m,s;
            m:=1; s:=n;
        l1: if s=0 then return m;
            m:=m*s;
            s:=s-1;
```

```
            go to l1
        end;
```

The reader will notice that we introduced a new local variable, S, and set it equal to N. The original form of the procedure contained the statement `n:=n-1;`. If the user asked for the value of `factorial(5)` then N would correspond to, not just have the value of, 5, and REDUCE would object to trying to execute the statement $5 := 5 - 1$.

If PQR is a procedure with no parameters,

```
procedure pqr;
   <procedure body>;
```

it can be written as a LET statement quite simply:

```
let pqr = <procedure body>;
```

To call *procedure* PQR, if defined in the latter form, the empty parentheses would not be used: use PQR not PQR() where a call on the procedure is needed.

The two notations for a procedure with no arguments can be combined. PQR can be defined in the standard PROCEDURE form. Then a LET statement

```
let pqr = pqr();
```

would allow a user to use PQR instead of PQR() in calling the procedure.

A feature available with LET-defined procedures and not with procedures defined in the standard way is the possibility of defining partial functions.

```
for all x such that numberp x let uvw(x)=<procedure body>;
```

Now UVW of an integer would be calculated as prescribed by the procedure body, while UVW of a general argument, such as Z or p+q (assuming these evaluate to themselves) would simply stay uvw(z) or uvw(p+q) as the case may be.

Chapter 15

User Contributed Packages

The complete REDUCE system includes a number of packages contributed by users that are provided as a service to the user community. Questions regarding these packages should be directed to their individual authors.

All such packages have been precompiled as part of the installation process. However, many must be specifically loaded before they can be used. (Those that are loaded automatically are so noted in their description.) You should also consult the user notes for your particular implementation for further information on whether this is necessary. If it is, the relevant command is LOAD_PACKAGE, which takes a list of one or more package names as argument, for example:

```
load_package algint;
```

although this syntax may vary from implementation to implementation.

Nearly all these packages come with separate documentation and test files (except those noted here that have no additional documentation), which is included, along with the source of the package, in the REDUCE system distribution. These items should be studied for any additional details on the use of a particular package.

The packages available in the current release of REDUCE are as follows:

15.1 ALGINT: Integration of square roots

This package, which is an extension of the basic integration package distributed with REDUCE, will analytically integrate a wide range of expressions involving square roots where the answer exists in that class of functions. It

is an implementation of the work described in J.H. Davenport, "On the Integration of Algebraic Functions", LNCS 102, Springer Verlag, 1981. Both this and the source code should be consulted for a more detailed description of this work.

Once the ALGINT package has been loaded, using LOAD_PACKAGE, one enters an expression for integration, as with the regular integrator, for example:

```
int(sqrt(x+sqrt(x**2+1))/x,x);
```

If one later wishes to integrate expressions without using the facilities of this package, the switch ALGINT should be turned off. This is turned on automatically when the package is loaded.

The switches supported by the standard integrator (e.g., TRINT) are also supported by this package. In addition, the switch TRA, if on, will give further tracing information about the specific functioning of the algebraic integrator.

There is no additional documentation for this package.

Author: James H. Davenport.

15.2 APPLYSYM: Infinitesimal symmetries of differential equations

This package provides programs APPLYSYM, QUASILINPDE and DETRAFO for applying infinitesimal symmetries of differential equations, the generalization of special solutions and the calculation of symmetry and similarity variables.

Author: Thomas Wolf.

15.3 ARNUM: An algebraic number package

This package provides facilities for handling algebraic numbers as polynomial coefficients in REDUCE calculations. It includes facilities for introducing indeterminates to represent algebraic numbers, for calculating splitting fields, and for factoring and finding greatest common divisors in such domains.

Author: Eberhard Schrüfer.

15.4 ASSERT: Dynamic Verification of Assertions on Function Types

ASSERT admits to add to symbolic mode RLISP code assertions (partly) specifying *types* of the arguments and results of RLISP expr procedures. These types can be associated with functions testing the validity of the respective arguments during runtime.

Author: Thomas Sturm.

15.5 ASSIST: Useful utilities for various applications

ASSIST contains a large number of additional general purpose functions that allow a user to better adapt REDUCE to various calculational strategies and to make the programming task more straightforward and more efficient.

Author: Hubert Caprasse.

15.6 AVECTOR: A vector algebra and calculus package

This package provides REDUCE with the ability to perform vector algebra using the same notation as scalar algebra. The basic algebraic operations are supported, as are differentiation and integration of vectors with respect to scalar variables, cross product and dot product, component manipulation and application of scalar functions (e.g. cosine) to a vector to yield a vector result.

Author: David Harper.

15.7 BIBASIS: A Package for Calculating Boolean Involutive Bases

Authors: Yuri A. Blinkov and Mikhail V. Zinin

15.8 BOOLEAN: A package for boolean algebra

This package supports the computation with boolean expressions in the propositional calculus. The data objects are composed from algebraic expressions connected by the infix boolean operators and, or, implies, equiv, and

the unary prefix operator not. Boolean allows you to simplify expressions built from these operators, and to test properties like equivalence, subset property etc.

Author: Herbert Melenk.

15.9 CDIFF: A package for the geometry of Differential Equations

Authors: P. Gragert, P.H.M. Kersten, G. Post and G. Roelofs, R. Vitolo.

15.10 CALI: A package for computational commutative algebra

This package contains algorithms for computations in commutative algebra closely related to the Gröbner algorithm for ideals and modules. Its heart is a new implementation of the Gröbner algorithm that also allows for the computation of syzygies. This implementation is also applicable to submodules of free modules with generators represented as rows of a matrix.

Author: Hans-Gert Gräbe.

15.11 CAMAL: Calculations in celestial mechanics

This packages implements in REDUCE the Fourier transform procedures of the CAMAL package for celestial mechanics.

Author: John P. Fitch.

15.12 CHANGEVR: Change of Independent Variable(s) in DEs

This package provides facilities for changing the independent variables in a differential equation. It is basically the application of the chain rule.

Author: G. Üçoluk.

15.13 COMPACT: Package for compacting expressions

COMPACT is a package of functions for the reduction of a polynomial in the presence of side relations. COMPACT applies the side relations to the polynomial so that an equivalent expression results with as few terms as possible. For example, the evaluation of

```
compact(s*(1-sin x^2)+c*(1-cos x^2)+sin x^2+cos x^2,
        {cos x^2+sin x^2=1});
```

yields the result

```
              2           2
SIN(X) *C + COS(X) *S + 1 .
```

Author: Anthony C. Hearn.

15.14 CRACK: Solving overdetermined systems of PDEs or ODEs

CRACK is a package for solving overdetermined systems of partial or ordinary differential equations (PDEs, ODEs). Examples of programs which make use of CRACK (finding symmetries of ODEs/PDEs, first integrals, an equivalent Lagrangian or a "differential factorization" of ODEs) are included. The application of symmetries is also possible by using the APPLYSYM package.

Authors: Andreas Brand, Thomas Wolf.

15.15 CVIT: Fast calculation of Dirac gamma matrix traces

This package provides an alternative method for computing traces of Dirac gamma matrices, based on an algorithm by Cvitanovich that treats gamma matrices as 3-j symbols.

Authors: V.Ilyin, A.Kryukov, A.Rodionov, A.Taranov.

15.16 DEFINT: A definite integration interface

This package finds the definite integral of an expression in a stated interval. It uses several techniques, including an innovative approach based on the Meijer G-function, and contour integration.

Authors: Kerry Gaskell, Stanley M. Kameny, Winfried Neun.

15.17 DESIR: Differential linear homogeneous equation solutions in the neighborhood of irregular and regular singular points

This package enables the basis of formal solutions to be computed for an ordinary homogeneous differential equation with polynomial coefficients over Q of any order, in the neighborhood of zero (regular or irregular singular point, or ordinary point).

Documentation for this package is in plain text.

Authors: C. Dicrescenzo, F. Richard-Jung, E. Tournier.

15.18 DFPART: Derivatives of generic functions

This package supports computations with total and partial derivatives of formal function objects. Such computations can be useful in the context of differential equations or power series expansions.

Author: Herbert Melenk.

15.19 DUMMY: Canonical form of expressions with dummy variables

This package allows a user to find the canonical form of expressions involving dummy variables. In that way, the simplification of polynomial expressions can be fully done. The indeterminates are general operator objects endowed with as few properties as possible. In that way the package may be used in a large spectrum of applications.

Author: Alain Dresse.

15.20 EXCALC: A differential geometry package

EXCALC is designed for easy use by all who are familiar with the calculus of Modern Differential Geometry. The program is currently able to handle scalar-valued exterior forms, vectors and operations between them, as well as non-scalar valued forms (indexed forms). It is thus an ideal tool for studying differential equations, doing calculations in general relativity and field theories, or doing simple things such as calculating the Laplacian of a tensor field for an arbitrary given frame.

Author: Eberhard Schrüfer.

15.21 FIDE: Finite difference method for partial differential equations

This package performs automation of the process of numerically solving partial differential equations systems (PDES) by means of computer algebra. For PDES solving, the finite difference method is applied. The computer algebra system REDUCE and the numerical programming language FORTRAN are used in the presented methodology. The main aim of this methodology is to speed up the process of preparing numerical programs for solving PDES. This process is quite often, especially for complicated systems, a tedious and time consuming task.

Documentation for this package is in plain text.

Author: Richard Liska.

15.22 FPS: Automatic calculation of formal power series

This package can expand a specific class of functions into their corresponding Laurent-Puiseux series.

Authors: Wolfram Koepf and Winfried Neun.

15.23 GCREF: A Graph Cross Referencer

This package reuses the code of the RCREF package to create a graph displaying the interdependency of procedures in a Reduce source code file.

Authors: A. Dolzmann, T. Sturm.

15.24 GENTRAN: A code generation package

GENTRAN is an automatic code GENerator and TRANslator. It constructs complete numerical programs based on sets of algorithmic specifications and symbolic expressions. Formatted FORTRAN, RATFOR, PASCAL or C code can be generated through a series of interactive commands or under the control of a template processing routine. Large expressions can be automatically segmented into subexpressions of manageable size, and a special file-handling mechanism maintains stacks of open I/O channels to allow output to be sent to any number of files simultaneously and to facilitate recursive invocation of the whole code generation process.

Author: Barbara L. Gates.

15.25 GNUPLOT: Display of functions and surfaces

This package is an interface to the popular GNUPLOT package. It allows you to display functions in 2D and surfaces in 3D on a variety of output devices including X terminals, PC monitors, and postscript and Latex printer files.

NOTE: The GNUPLOT package may not be included in all versions of REDUCE.

Author: Herbert Melenk.

15.26 GROEBNER: A Gröbner basis package

GROEBNER is a package for the computation of Gröbner Bases using the Buchberger algorithm and related methods for polynomial ideals and modules. It can be used over a variety of different coefficient domains, and for different variable and term orderings.

Gröbner Bases can be used for various purposes in commutative algebra, e.g. for elimination of variables, converting surd expressions to implicit polynomial form, computation of dimensions, solution of polynomial equation systems etc. The package is also used internally by the `SOLVE` operator.

Authors: Herbert Melenk, H.M. Möller and Winfried Neun.

15.27 GUARDIAN: Guarded Expressions in Practice

Computer algebra systems typically drop some degenerate cases when evaluating expressions, e.g., x/x becomes 1 dropping the case $x = 0$. We claim that it is feasible in practice to compute also the degenerate cases yielding *guarded expressions*. We work over real closed fields but our ideas about handling guarded expression can be easily transferred to other situations. Using formulas as guards provides a powerful tool for heuristically reducing the combinatorial explosion of cases: equivalent, redundant, tautological, and contradictive cases can be detected by simplification and quantifier elimination. Our approach allows to simplify the expressions on the basis of simplification knowledge on the logical side. The method described in this paper is implemented in the REDUCE package GUARDIAN.

Authors: Andreas Dolzmann and Thomas Sturm.

15.28 IDEALS: Arithmetic for polynomial ideals

This package implements the basic arithmetic for polynomial ideals by exploiting the Gröbner bases package of REDUCE. In order to save computing time all intermediate Gröbner bases are stored internally such that time consuming repetitions are inhibited.

Author: Herbert Melenk.

15.29 INEQ: Support for solving inequalities

This package supports the operator ineq_solve that tries to solves single inequalities and sets of coupled inequalities.

Author: Herbert Melenk.

15.30 INVBASE: A package for computing involutive bases

Involutive bases are a new tool for solving problems in connection with multivariate polynomials, such as solving systems of polynomial equations and analyzing polynomial ideals. An involutive basis of polynomial ideal is nothing but a special form of a redundant Gröbner basis. The construction of involutive bases reduces the problem of solving polynomial systems to simple linear algebra.

Authors: A.Yu. Zharkov and Yu.A. Blinkov.

15.31 LAPLACE: Laplace transforms

This package can calculate ordinary and inverse Laplace transforms of expressions. Documentation is in plain text.

Authors: C. Kazasov, M. Spiridonova, V. Tomov.

15.32 LIE: Functions for the classification of real n-dimensional Lie algebras

LIE is a package of functions for the classification of real n-dimensional Lie algebras. It consists of two modules: liendmc1 and lie1234. With the help of the functions in the liendmcl module, real n-dimensional Lie algebras L with a derived algebra $L^{(1)}$ of dimension 1 can be classified.

Authors: Carsten and Franziska Schöbel.

15.33 LIMITS: A package for finding limits

LIMITS is a fast limit package for REDUCE for functions which are continuous except for computable poles and singularities, based on some earlier work by Ian Cohen and John P. Fitch. The Truncated Power Series package is used for non-critical points, at which the value of the function is the constant term in the expansion around that point. L'Hôpital's rule is used in critical cases, with preprocessing of $\infty - \infty$ forms and reformatting of product forms in order to be able to apply l'Hôpital's rule. A limited amount of bounded arithmetic is also employed where applicable.

This package defines a `LIMIT` operator, called with the syntax:

```
LIMIT(EXPRN:algebraic,VAR:kernel,LIMPOINT:algebraic):
      algebraic.
```

For example:

```
limit(x*sin(1/x),x,infinity)   -> 1
limit(sin x/x^2,x,0)           -> INFINITY
```

Direction-dependent limit operators `LIMIT!+` and `LIMIT!-` are also defined.

This package loads automatically.

Author: Stanley L. Kameny.

15.34 LINALG: Linear algebra package

This package provides a selection of functions that are useful in the world of linear algebra.

Author: Matt Rebbeck.

15.35 LPDO: Linear Partial Differential Operators

Author: Thomas Sturm

15.36 MODSR: Modular solve and roots

This package supports solve (M_SOLVE) and roots (M_ROOTS) operators for modular polynomials and modular polynomial systems. The moduli need not be primes. M_SOLVE requires a modulus to be set. M_ROOTS takes the modulus as a second argument. For example:

```
on modular; setmod 8;
m_solve(2x=4);            ->  {{X=2},{X=6}}
m_solve({x^2-y^3=3});
    ->  {{X=0,Y=5}, {X=2,Y=1}, {X=4,Y=5}, {X=6,Y=1}}
m_solve({x=2,x^2-y^3=3}); ->  {{X=2,Y=1}}
off modular;
m_roots(x^2-1,8);         ->  {1,3,5,7}
m_roots(x^3-x,7);         ->  {0,1,6}
```

There is no further documentation for this package.

Author: Herbert Melenk.

15.37 NCPOLY: Non-commutative polynomial ideals

This package allows the user to set up automatically a consistent environment for computing in an algebra where the non-commutativity is defined by Lie-bracket commutators. The package uses the REDUCE noncom mechanism for

elementary polynomial arithmetic; the commutator rules are automatically computed from the Lie brackets.

Authors: Herbert Melenk and Joachim Apel.

15.38 NORMFORM: Computation of matrix normal forms

This package contains routines for computing the following normal forms of matrices:

- smithex_int
- smithex
- frobenius
- ratjordan
- jordansymbolic
- jordan.

Author: Matt Rebbeck.

15.39 NUMERIC: Solving numerical problems

This package implements basic algorithms of numerical analysis. These include:

- solution of algebraic equations by Newton's method

```
num_solve({sin x=cos y, x + y = 1},{x=1,y=2})
```

- solution of ordinary differential equations

```
num_odesolve(df(y,x)=y,y=1,x=(0 .. 1), iterations=5)
```

- bounds of a function over an interval

```
bounds(sin x+x,x=(1 .. 2));
```

- minimizing a function (Fletcher Reeves steepest descent)

```
num_min(sin(x)+x/5, x);
```

- Chebyshev curve fitting

```
chebyshev_fit(sin x/x,x=(1 .. 3),5);
```

- numerical quadrature

```
num_int(sin x,x=(0 .. pi));
```

Author: Herbert Melenk.

15.40 ODESOLVE: Ordinary differential equations solver

The ODESOLVE package is a solver for ordinary differential equations. At the present time it has very limited capabilities. It can handle only a single scalar equation presented as an algebraic expression or equation, and it can solve only first-order equations of simple types, linear equations with constant coefficients and Euler equations. These solvable types are exactly those for which Lie symmetry techniques give no useful information. For example, the evaluation of

```
depend(y,x);
odesolve(df(y,x)=x**2+e**x,y,x);
```

yields the result

```
        X                      3
    3*E  + 3*ARBCONST(1) + X
{Y=----------------------------}
                 3
```

Main Author: Malcolm A.H. MacCallum.

Other contributors: Francis Wright, Alan Barnes.

15.41 ORTHOVEC: Manipulation of scalars and vectors

ORTHOVEC is a collection of REDUCE procedures and operations which provide a simple-to-use environment for the manipulation of scalars and vectors. Operations include addition, subtraction, dot and cross products, division, modulus, div, grad, curl, laplacian, differentiation, integration, and Taylor expansion.

Author: James W. Eastwood.

15.42 PHYSOP: Operator calculus in quantum theory

This package has been designed to meet the requirements of theoretical physicists looking for a computer algebra tool to perform complicated calculations in quantum theory with expressions containing operators. These operations consist mainly of the calculation of commutators between operator expressions and in the evaluations of operator matrix elements in some abstract space.

Author: Mathias Warns.

15.43 PM: A REDUCE pattern matcher

PM is a general pattern matcher similar in style to those found in systems such as SMP and Mathematica, and is based on the pattern matcher described in Kevin McIsaac, "Pattern Matching Algebraic Identities", SIGSAM Bulletin, 19 (1985), 4-13.

Documentation for this package is in plain text.

Author: Kevin McIsaac.

15.44 RANDPOLY: A random polynomial generator

This package is based on a port of the Maple random polynomial generator together with some support facilities for the generation of random numbers and anonymous procedures.

Author: Francis J. Wright.

15.45 REACTEQN: Support for chemical reaction equation systems

This package allows a user to transform chemical reaction systems into ordinary differential equation systems (ODE) corresponding to the laws of pure mass action.

Documentation for this package is in plain text.

Author: Herbert Melenk.

15.46 RESET: Code to reset REDUCE to its initial state

This package defines a command RESETREDUCE that works through the history of previous commands, and clears any values which have been assigned, plus any rules, arrays and the like. It also sets the various switches to their initial values. It is not complete, but does work for most things that cause a gradual loss of space. It would be relatively easy to make it interactive, so allowing for selective resetting.

There is no further documentation on this package.

Author: John Fitch.

15.47 RESIDUE: A residue package

This package supports the calculation of residues of arbitrary expressions.

Author: Wolfram Koepf.

15.48 RLFI: REDUCE LaTeX formula interface

This package adds LaTeX syntax to REDUCE. Text generated by REDUCE in this mode can be directly used in LaTeX source documents. Various mathematical constructions are supported by the interface including subscripts, superscripts, font changing, Greek letters, divide-bars, integral and sum signs, derivatives, and so on.

Author: Richard Liska.

15.49 ROOTS: A REDUCE root finding package

This root finding package can be used to find some or all of the roots of a univariate polynomial with real or complex coefficients, to the accuracy specified by the user.

It is designed so that it can be used as an independent package, or it may be called from SOLVE if ROUNDED is on. For example, the evaluation of

```
on rounded,complex;
solve(x**3+x+5,x);
```

yields the result

```
{X= - 1.51598,X=0.75799 + 1.65035*I,X=0.75799 - 1.65035*I}
```

This package loads automatically.

Author: Stanley L. Kameny.

15.50 RSOLVE: Rational/integer polynomial solvers

This package provides operators that compute the exact rational zeros of a single univariate polynomial using fast modular methods. The algorithm used is that described by R. Loos (1983): Computing rational zeros of integral polynomials by p-adic expansion, *SIAM J. Computing*, 12, 286–293.

Author: Francis J. Wright.

15.51 SCOPE: REDUCE source code optimization package

SCOPE is a package for the production of an optimized form of a set of expressions. It applies an heuristic search for common (sub)expressions to almost any set of proper REDUCE assignment statements. The output is obtained as a sequence of assignment statements. GENTRAN is used to facilitate expression output.

Author: J.A. van Hulzen.

15.52 SETS: A basic set theory package

The SETS package provides algebraic-mode support for set operations on lists regarded as sets (or representing explicit sets) and on implicit sets represented by identifiers.

Author: Francis J. Wright.

15.53 SPDE: Finding symmetry groups of PDE's

The package SPDE provides a set of functions which may be used to determine the symmetry group of Lie- or point-symmetries of a given system of partial differential equations. In many cases the determining system is solved

completely automatically. In other cases the user has to provide additional input information for the solution algorithm to terminate.

Author: Fritz Schwarz.

15.54 SPECFN: Package for special functions

This special function package is separated into two portions to make it easier to handle. The packages are called SPECFN and SPECFN2. The first one is more general in nature, whereas the second is devoted to special special functions. Documentation for the first package can be found in the file specfn.tex in the "doc" directory, and examples in specfn.tst and specfmor.tst in the examples directory.

The package SPECFN is designed to provide algebraic and numerical manipulations of several common special functions, namely:

- Bernoulli Numbers and Euler Numbers;
- Stirling Numbers;
- Binomial Coefficients;
- Pochhammer notation;
- The Gamma function;
- The Psi function and its derivatives;
- The Riemann Zeta function;
- The Bessel functions J and Y of the first and second kind;
- The modified Bessel functions I and K;
- The Hankel functions H1 and H2;
- The Kummer hypergeometric functions M and U;
- The Beta function, and Struve, Lommel and Whittaker functions;
- The Airy functions;
- The Exponential Integral, the Sine and Cosine Integrals;
- The Hyperbolic Sine and Cosine Integrals;
- The Fresnel Integrals and the Error function;

- The Dilog function;
- Hermite Polynomials;
- Jacobi Polynomials;
- Legendre Polynomials;
- Spherical and Solid Harmonics;
- Laguerre Polynomials;
- Chebyshev Polynomials;
- Gegenbauer Polynomials;
- Euler Polynomials;
- Bernoulli Polynomials.
- Jacobi Elliptic Functions and Integrals;
- 3j symbols, 6j symbols and Clebsch Gordan coefficients;

Author: Chris Cannam, with contributions from Winfried Neun, Herbert Melenk, Victor Adamchik, Francis Wright and several others.

15.55 SPECFN2: Package for special special functions

This package provides algebraic manipulations of generalized hypergeometric functions and Meijer's G function. Generalized hypergeometric functions are simplified towards special functions and Meijer's G function is simplified towards special functions or generalized hypergeometric functions.

Author: Victor Adamchik, with major updates by Winfried Neun.

15.56 SUM: A package for series summation

This package implements the Gosper algorithm for the summation of series. It defines operators `SUM` and `PROD`. The operator `SUM` returns the indefinite or definite summation of a given expression, and `PROD` returns the product of the given expression.

This package loads automatically.

Author: Fujio Kako.

15.57 SYMMETRY: Operations on symmetric matrices

This package computes symmetry-adapted bases and block diagonal forms of matrices which have the symmetry of a group. The package is the implementation of the theory of linear representations for small finite groups such as the dihedral groups.

Author: Karin Gatermann.

15.58 TAYLOR: Manipulation of Taylor series

This package carries out the Taylor expansion of an expression in one or more variables and efficient manipulation of the resulting Taylor series. Capabilities include basic operations (addition, subtraction, multiplication and division) and also application of certain algebraic and transcendental functions.

Author: Rainer Schöpf.

15.59 TPS: A truncated power series package

This package implements formal Laurent series expansions in one variable using the domain mechanism of REDUCE. This means that power series objects can be added, multiplied, differentiated etc., like other first class objects in the system. A lazy evaluation scheme is used and thus terms of the series are not evaluated until they are required for printing or for use in calculating terms in other power series. The series are extendible giving the user the impression that the full infinite series is being manipulated. The errors that can sometimes occur using series that are truncated at some fixed depth (for example when a term in the required series depends on terms of an intermediate series beyond the truncation depth) are thus avoided.

Authors: Alan Barnes and Julian Padget.

15.60 TRI: TeX REDUCE interface

This package provides facilities written in REDUCE-Lisp for typesetting REDUCE formulas using TEX. The TEX-REDUCE-Interface incorporates three levels of TEXoutput: without line breaking, with line breaking, and with line breaking plus indentation.

Author: Werner Antweiler.

15.61 TRIGSIMP: Simplification and factorization of trigonometric and hyperbolic functions

TRIGSIMP is a useful tool for all kinds of trigonometric and hyperbolic simplification and factorization. There are three procedures included in TRIGSIMP: trigsimp, trigfactorize and triggcd. The first is for finding simplifications of trigonometric or hyperbolic expressions with many options, the second for factorizing them and the third for finding the greatest common divisor of two trigonometric or hyperbolic polynomials.

Author: Wolfram Koepf.

15.62 WU: Wu algorithm for polynomial systems

This is a simple implementation of the Wu algorithm implemented in REDUCE working directly from "A Zero Structure Theorem for Polynomial-Equations-Solving," Wu Wen-tsun, Institute of Systems Science, Academia Sinica, Beijing.

Author: Russell Bradford.

15.63 XCOLOR: Color factor in some field theories

This package calculates the color factor in non-abelian gauge field theories using an algorithm due to Cvitanovich.

Documentation for this package is in plain text.

Author: A. Kryukov.

15.64 XIDEAL: Gröbner Bases for exterior algebra

XIDEAL constructs Gröbner bases for solving the left ideal membership problem: Gröbner left ideal bases or GLIBs. For graded ideals, where each form is homogeneous in degree, the distinction between left and right ideals vanishes. Furthermore, if the generating forms are all homogeneous, then the Gröbner bases for the non-graded and graded ideals are identical. In this

case, XIDEAL is able to save time by truncating the Gröbner basis at some maximum degree if desired.

Author: David Hartley.

15.65 ZEILBERG: Indefinite and definite summation

This package is a careful implementation of the Gosper and Zeilberger algorithms for indefinite and definite summation of hypergeometric terms, respectively. Extensions of these algorithms are also included that are valid for ratios of products of powers, factorials, Γ function terms, binomial coefficients, and shifted factorials that are rational-linear in their arguments.

Authors: Gregor Stölting and Wolfram Koepf.

15.66 ZTRANS: Z-transform package

This package is an implementation of the Z-transform of a sequence. This is the discrete analogue of the Laplace Transform.

Authors: Wolfram Koepf and Lisa Temme.

Chapter 16

Symbolic Mode

At the system level, REDUCE is based on a version of the programming language Lisp known as *Standard Lisp* which is described in J. Marti, Hearn, A. C., Griss, M. L. and Griss, C., "Standard LISP Report" SIGPLAN Notices, ACM, New York, 14, No 10 (1979) 48-68. We shall assume in this section that the reader is familiar with the material in that paper. This also assumes implicitly that the reader has a reasonable knowledge about Lisp in general, say at the level of the LISP 1.5 Programmer's Manual (McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P. and Levin, M. I., "LISP 1.5 Programmer's Manual", M.I.T. Press, 1965) or any of the books mentioned at the end of this section. Persons unfamiliar with this material will have some difficulty understanding this section.

Although REDUCE is designed primarily for algebraic calculations, its source language is general enough to allow for a full range of Lisp-like symbolic calculations. To achieve this generality, however, it is necessary to provide the user with two modes of evaluation, namely an algebraic mode and a symbolic mode. To enter symbolic mode, the user types `symbolic;` (or `lisp;`) and to return to algebraic mode one types `algebraic;`. Evaluations proceed differently in each mode so the user is advised to check what mode he is in if a puzzling error arises. He can find his mode by typing

```
eval_mode;
```

The current mode will then be printed as `ALGEBRAIC` or `SYMBOLIC`.

Expression evaluation may proceed in either mode at any level of a calculation, provided the results are passed from mode to mode in a compatible manner. One simply prefixes the relevant expression by the appropriate mode. If the mode name prefixes an expression at the top level, it will then be handled as if the global system mode had been changed for the scope of

that particular calculation.

For example, if the current mode is ALGEBRAIC, then the commands

```
symbolic car '(a);
x+y;
```

will cause the first expression to be evaluated and printed in symbolic mode and the second in algebraic mode. Only the second evaluation will thus affect the expression workspace. On the other hand, the statement

```
x + symbolic car '(12);
```

will result in the algebraic value X+12.

The use of SYMBOLIC (and equivalently ALGEBRAIC) in this manner is the same as any operator. That means that parentheses could be omitted in the above examples since the meaning is obvious. In other cases, parentheses must be used, as in

```
symbolic(x := 'a);
```

Omitting the parentheses, as in

```
symbolic x := a;
```

would be wrong, since it would parse as

```
symbolic(x) := a;
```

For convenience, it is assumed that any operator whose *first* argument is quoted is being evaluated in symbolic mode, regardless of the mode in effect at that time. Thus, the first example above could be equally well written:

```
car '(a);
```

Except where explicit limitations have been made, most REDUCE algebraic constructions carry over into symbolic mode. However, there are some differences. First, expression evaluation now becomes Lisp evaluation. Secondly, assignment statements are handled differently, as we shall discuss shortly. Thirdly, local variables and array elements are initialized to NIL rather than 0. (In fact, any variables not explicitly declared INTEGER are also initialized to NIL in algebraic mode, but the algebraic evaluator recognizes NIL as 0.) Finally, function definitions follow the conventions of Standard Lisp.

To begin with, we mention a few extensions to our basic syntax which are

designed primarily if not exclusively for symbolic mode.

16.1 Symbolic Infix Operators

There are three binary infix operators in REDUCE intended for use in symbolic mode, namely `. (CONS), EQ and MEMQ`. The precedence of these operators was given in another section.

16.2 Symbolic Expressions

These consist of scalar variables and operators and follow the normal rules of the Lisp meta language.

Examples:

```
x
car u . reverse v
simp (u+v^2)
```

16.3 Quoted Expressions

Because symbolic evaluation requires that each variable or expression has a value, it is necessary to add to REDUCE the concept of a quoted expression by analogy with the Lisp `QUOTE` function. This is provided by the single quote mark `'`. For example,

```
'a        represents the Lisp S-expression (quote a)
'(a b c) represents the Lisp S-expression (quote (a b c))
```

Note, however, that strings are constants and therefore evaluate to themselves in symbolic mode. Thus, to print the string `"A String"`, one would write

```
prin2 "A String";
```

Within a quoted expression, identifier syntax rules are those of REDUCE. Thus `(A !. B)` is the list consisting of the three elements `A`, `.`, and `B`, whereas `(A . B)` is the dotted pair of `A` and `B`.

16.4 Lambda Expressions

LAMBDA expressions provide the means for constructing Lisp LAMBDA expressions in symbolic mode. They may not be used in algebraic mode.

Syntax:

```
<LAMBDA expression> ::=
        LAMBDA <varlist><terminator><statement>
```

where

```
<varlist> ::= (<variable>,...,<variable>)
```

e.g.,

```
lambda (x,y); car x . cdr y;
```

is equivalent to the Lisp LAMBDA expression

```
(lambda (x y) (cons (car x) (cdr y)))
```

The parentheses may be omitted in specifying the variable list if desired.

LAMBDA expressions may be used in symbolic mode in place of prefix operators, or as an argument of the reserved word FUNCTION.

In those cases where a LAMBDA expression is used to introduce local variables to avoid recomputation, a WHERE statement can also be used. For example, the expression

```
(lambda (x,y); list(car x,cdr x,car y,cdr y))
    (reverse u,reverse v)
```

can also be written

```
{car x,cdr x,car y,cdr y} where x=reverse u,y=reverse v
```

Where possible, WHERE syntax is preferred to LAMBDA syntax, since it is more natural.

16.5 Symbolic Assignment Statements

In symbolic mode, if the left side of an assignment statement is a variable, a SETQ of the right-hand side to that variable occurs. If the left-hand side is an expression, it must be of the form of an array element, otherwise an error will result. For example, `x:=y` translates into `(SETQ X Y)` whereas `a(3) := 3` will be valid if `A` has been previously declared a single dimensioned array of at least four elements.

16.6 FOR EACH Statement

The FOR EACH form of the FOR statement, designed for iteration down a list, is more general in symbolic mode. Its syntax is:

```
FOR EACH ID:identifier {IN|ON} LST:list
    {DO|COLLECT|JOIN|PRODUCT|SUM} EXPRN:S-expr
```

As in algebraic mode, if the keyword IN is used, iteration is on each element of the list. With ON, iteration is on the whole list remaining at each point in the iteration. As a result, we have the following equivalence between each form of FOR EACH and the various mapping functions in Lisp:

	DO	COLLECT	JOIN
IN	MAPC	MAPCAR	MAPCAN
ON	MAP	MAPLIST	MAPCON

Example: To list each element of the list `(a b c)`:

```
for each x in '(a b c) collect list x;
```

16.7 Symbolic Procedures

All the functions described in the Standard Lisp Report are available to users in symbolic mode. Additional functions may also be defined as symbolic procedures. For example, to define the Lisp function ASSOC, the following could be used:

```
symbolic procedure assoc(u,v);
   if null v then nil
    else if u = caar v then car v
    else assoc(u, cdr v);
```

If the default mode were symbolic, then SYMBOLIC could be omitted in the above definition. MACROs may be defined by prefixing the keyword PROCEDURE by the word MACRO. (In fact, ordinary functions may be defined with the keyword EXPR prefixing PROCEDURE as was used in the Standard Lisp Report.) For example, we could define a MACRO CONSCONS by

```
symbolic macro procedure conscons l;
   expand(cdr l,'cons);
```

Another form of macro, the SMACRO is also available. These are described in the Standard Lisp Report. The Report also defines a function type FEXPR. However, its use is discouraged since it is hard to implement efficiently, and most uses can be replaced by macros. At the present time, there are no FEXPRs in the core REDUCE system.

16.8 Standard Lisp Equivalent of Reduce Input

A user can obtain the Standard Lisp equivalent of his REDUCE input by turning on the switch DEFN (for definition). The system then prints the Lisp translation of his input but does not evaluate it. Normal operation is resumed when DEFN is turned off.

16.9 Communicating with Algebraic Mode

One of the principal motivations for a user of the algebraic facilities of REDUCE to learn about symbolic mode is that it gives one access to a wider range of techniques than is possible in algebraic mode alone. For example, if a user wishes to use parts of the system defined in the basic system source code, or refine their algebraic code definitions to make them more efficient, then it is necessary to understand the source language in fairly complete detail. Moreover, it is also necessary to know a little more about the way REDUCE operates internally. Basically, REDUCE considers expressions in two forms: prefix form, which follow the normal Lisp rules of function composition, and so-called canonical form, which uses a completely different syntax.

Once these details are understood, the most critical problem faced by a user is how to make expressions and procedures communicate between symbolic and algebraic mode. The purpose of this section is to teach a user the basic principles for this.

If one wants to evaluate an expression in algebraic mode, and then use that expression in symbolic mode calculations, or vice versa, the easiest way to do

this is to assign a variable to that expression whose value is easily obtainable in both modes. To facilitate this, a declaration SHARE is available. SHARE takes a list of identifiers as argument, and marks these variables as having recognizable values in both modes. The declaration may be used in either mode.

E.g.,

```
share x,y;
```

says that X and Y will receive values to be used in both modes.

If a SHARE declaration is made for a variable with a previously assigned algebraic value, that value is also made available in symbolic mode.

16.9.1 Passing Algebraic Mode Values to Symbolic Mode

If one wishes to work with parts of an algebraic mode expression in symbolic mode, one simply makes an assignment of a shared variable to the relevant expression in algebraic mode. For example, if one wishes to work with (a+b)^2, one would say, in algebraic mode:

```
x := (a+b)^2;
```

assuming that X was declared shared as above. If we now change to symbolic mode and say

```
x;
```

its value will be printed as a prefix form with the syntax:

```
(*SQ <standard quotient> T)
```

This particular format reflects the fact that the algebraic mode processor currently likes to transfer prefix forms from command to command, but doesn't like to reconvert standard forms (which represent polynomials) and standard quotients back to a true Lisp prefix form for the expression (which would result in excessive computation). So *SQ is used to tell the algebraic processor that it is dealing with a prefix form which is really a standard quotient and the second argument (T or NIL) tells it whether it needs further processing (essentially, an *already simplified* flag).

So to get the true standard quotient form in symbolic mode, one needs CADR of the variable. E.g.,

```
z := cadr x;
```

would store in `Z` the standard quotient form for `(a+b)^2`.

Once you have this expression, you can now manipulate it as you wish. To facilitate this, a standard set of selectors and constructors are available for getting at parts of the form. Those presently defined are as follows:

REDUCE Selectors

DENR	denominator of standard quotient
LC	leading coefficient of polynomial
LDEG	leading degree of polynomial
LPOW	leading power of polynomial
LT	leading term of polynomial
MVAR	main variable of polynomial
NUMR	numerator (of standard quotient)
PDEG	degree of a power
RED	reductum of polynomial
TC	coefficient of a term
TDEG	degree of a term
TPOW	power of a term

REDUCE Constructors

.+	add a term to a polynomial
./	divide (two polynomials to get quotient)
.*	multiply power by coefficient to produce term
.^	raise a variable to a power

For example, to find the numerator of the standard quotient above, one could say:

```
numr z;
```

or to find the leading term of the numerator:

```
lt numr z;
```

Conversion between various data structures is facilitated by the use of a set of functions defined for this purpose. Those currently implemented include:

`!*A2F` convert an algebraic expression to a standard form. If result is rational, an error results;

`!*A2K` converts an algebraic expression to a kernel. If this is not possible, an error results;

`!*F2A` converts a standard form to an algebraic expression;

`!*F2Q` convert a standard form to a standard quotient;

`!*K2F` convert a kernel to a standard form;

`!*K2Q` convert a kernel to a standard quotient;

`!*P2F` convert a standard power to a standard form;

`!*P2Q` convert a standard power to a standard quotient;

`!*Q2F` convert a standard quotient to a standard form. If the quotient denominator is not 1, an error results;

`!*Q2K` convert a standard quotient to a kernel. If this is not possible, an error results;

`!*T2F` convert a standard term to a standard form

`!*T2Q` convert a standard term to a standard quotient.

16.9.2 Passing Symbolic Mode Values to Algebraic Mode

In order to pass the value of a shared variable from symbolic mode to algebraic mode, the only thing to do is make sure that the value in symbolic mode is a prefix expression. E.g., one uses `(expt (plus a b) 2)` for `(a+b)^2`, or the format `(*sq <standard quotient> t)` as described above. However, if you have been working with parts of a standard form they will probably not be in this form. In that case, you can do the following:

1. If it is a standard quotient, call `PREPSQ` on it. This takes a standard quotient as argument, and returns a prefix expression. Alternatively, you can call `MK!*SQ` on it, which returns a prefix form like `(*SQ <standard quotient> T)` and avoids translation of the expression into a true prefix form.

2. If it is a standard form, call `PREPF` on it. This takes a standard form as argument, and returns the equivalent prefix expression. Alternatively, you can convert it to a standard quotient and then call `MK!*SQ`.

3. If it is a part of a standard form, you must usually first build up a standard form out of it, and then go to step 2. The conversion functions

described earlier may be used for this purpose. For example,

(a) If Z is an expression which is a term, `!*T2F Z` is a standard form.

(b) If Z is a standard power, `!*P2F Z` is a standard form.

(c) If Z is a variable, you can pass it direct to algebraic mode.

For example, to pass the leading term of `(a+b)^2` back to algebraic mode, one could say:

```
y:= mk!*sq !*t2q lt numr z;
```

where Y has been declared shared as above. If you now go back to algebraic mode, you can work with Y in the usual way.

16.9.3 Complete Example

The following is the complete code for doing the above steps. The end result will be that the square of the leading term of $(a+b)^2$ is calculated.

```
share x,y;                    % declare X and Y as shared
x := (a+b)^2;                 % store (a+b)^2 in X
symbolic;                     % transfer to symbolic mode
z := cadr x;                  % store a true standard quotient
                              % in Z
lt numr z;                    % print the leading term of the
                              % numerataor of Z
y := mk!*sq !*t2q lt numr z; % store the prefix form of this
                              % leading term in Y
algebraic;                    % return to algebraic mode
y^2;                          % evaluate square of the
                              % leading term of (a+b)^2
```

16.9.4 Defining Procedures for Intermode Communication

If one wishes to define a procedure in symbolic mode for use as an operator in algebraic mode, it is necessary to declare this fact to the system by using the declaration OPERATOR in symbolic mode. Thus

```
symbolic operator leadterm;
```

would declare the procedure LEADTERM as an algebraic operator. This declaration *must* be made in symbolic mode as the effect in algebraic mode is different. The value of such a procedure must be a prefix form.

The algebraic processor will pass arguments to such procedures in prefix form. Therefore if you want to work with the arguments as standard quotients you must first convert them to that form by using the function SIMP!*. This function takes a prefix form as argument and returns the evaluated standard quotient.

For example, if you want to define a procedure LEADTERM which gives the leading term of an algebraic expression, one could do this as follows:

```
symbolic operator leadterm; % Declare LEADTERM as a symbolic
                            % mode procedure to be used in
                            % algebraic mode.

symbolic procedure leadterm u; % Define LEADTERM.
   mk!*sq !*t2q lt numr simp!* u;
```

Note that this operator has a different effect than the operator LTERM . In the latter case, the calculation is done with respect to the second argument of the operator. In the example here, we simply extract the leading term with respect to the system's choice of main variable.

Finally, if you wish to use the algebraic evaluator on an argument in a symbolic mode definition, the function REVAL can be used. The one argument of REVAL must be the prefix form of an expression. REVAL returns the evaluated expression as a true Lisp prefix form.

16.10 Rlisp '88

Rlisp '88 is a superset of the Rlisp that has been traditionally used for the support of REDUCE. It is fully documented in the book Marti, J.B., "RLISP '88: An Evolutionary Approach to Program Design and Reuse", World Scientific, Singapore (1993). Rlisp '88 adds to the traditional Rlisp the following facilities:

1. more general versions of the looping constructs `for`, `repeat` and `while`;
2. support for a backquote construct;
3. support for active comments;
4. support for vectors of the form name[index];
5. support for simple structures;
6. support for records.

In addition, "-" is a letter in Rlisp '88. In other words, `A-B` is an identifier, not the difference of the identifiers `A` and `B`. If the latter construct is required, it is necessary to put spaces around the - character. For compatibility between the two versions of Rlisp, we recommend this convention be used in all symbolic mode programs.

To use Rlisp '88, type `on rlisp88;`. This switches to symbolic mode with the Rlisp '88 syntax and extensions. While in this environment, it is impossible to switch to algebraic mode, or prefix expressions by "algebraic". However, symbolic mode programs written in Rlisp '88 may be run in algebraic mode provided the rlisp88 package has been loaded. We also expect that many of the extensions defined in Rlisp '88 will migrate to the basic Rlisp over time. To return to traditional Rlisp or to switch to algebraic mode, say "`off rlisp88;`".

16.11 References

There are a number of useful books which can give you further information about LISP. Here is a selection:

Allen, J.R., "The Anatomy of LISP", McGraw Hill, New York, 1978.

McCarthy J., P.W. Abrahams, J. Edwards, T.P. Hart and M.I. Levin, "LISP 1.5 Programmer's Manual", M.I.T. Press, 1965.

Touretzky, D.S, "LISP: A Gentle Introduction to Symbolic Computation", Harper & Row, New York, 1984.

Winston, P.H. and Horn, B.K.P., "LISP", Addison-Wesley, 1981.

Chapter 17

Calculations in High Energy Physics

A set of REDUCE commands is provided for users interested in symbolic calculations in high energy physics. Several extensions to our basic syntax are necessary, however, to allow for the different data structures encountered.

17.1 High Energy Physics Operators

We begin by introducing three new operators required in these calculations.

17.1.1 . (Cons) Operator

Syntax:

```
(EXPRN1:vector_expression)
          . (EXPRN2:vector_expression):algebraic.
```

The binary . operator, which is normally used to denote the addition of an element to the front of a list, can also be used in algebraic mode to denote the scalar product of two Lorentz four-vectors. For this to happen, the second argument must be recognizable as a vector expression at the time of evaluation. With this meaning, this operator is often referred to as the *dot* operator. In the present system, the index handling routines all assume that Lorentz four-vectors are used, but these routines could be rewritten to handle other cases.

Components of vectors can be represented by including representations of unit vectors in the system. Thus if `E0` represents the unit vector `(1,0,0,0)`,

(p.eo) represents the zeroth component of the four-vector P. Our metric and notation follows Bjorken and Drell "Relativistic Quantum Mechanics" (McGraw-Hill, New York, 1965). Similarly, an arbitrary component P may be represented by (p.u). If contraction over components of vectors is required, then the declaration INDEX must be used. Thus

```
index u;
```

declares U as an index, and the simplification of

```
p.u * q.u
```

would result in

```
P.Q
```

The metric tensor $g^{\mu\nu}$ may be represented by (u.v). If contraction over U and V is required, then they should be declared as indices.

Errors occur if indices are not properly matched in expressions.

If a user later wishes to remove the index property from specific vectors, he can do it with the declaration REMIND. Thus remind v1...vn; removes the index flags from the variables V1 through Vn. However, these variables remain vectors in the system.

17.1.2 G Operator for Gamma Matrices

Syntax:

```
G(ID:identifier[,EXPRN:vector_expression])
        :gamma_matrix_expression.
```

G is an n-ary operator used to denote a product of γ matrices contracted with Lorentz four-vectors. Gamma matrices are associated with fermion lines in a Feynman diagram. If more than one such line occurs, then a different set of γ matrices (operating in independent spin spaces) is required to represent each line. To facilitate this, the first argument of G is a line identification identifier (not a number) used to distinguish different lines.

Thus

```
g(l1,p) * g(l2,q)
```

denotes the product of γ.p associated with a fermion line identified as L1, and γ.q associated with another line identified as L2 and where p and q are Lorentz four-vectors. A product of γ matrices associated with the same line may be written in a contracted form.

Thus

```
g(l1,p1,p2,...,p3) = g(l1,p1)*g(l1,p2)*...*g(l1,p3) .
```

The vector A is reserved in arguments of G to denote the special γ matrix γ^5. Thus

g(l,a)	=	γ^5	associated with the line L
g(l,p,a)	=	γ.p $\times\gamma^5$	associated with the line L.

γ^μ (associated with the line L) may be written as g(l,u), with U flagged as an index if contraction over U is required.

The notation of Bjorken and Drell is assumed in all operations involving γ matrices.

17.1.3 EPS Operator

Syntax:

```
EPS(EXPRN1:vector_expression,...,EXPRN4:vector_exp)
   :vector_exp.
```

The operator EPS has four arguments, and is used only to denote the completely antisymmetric tensor of order 4 and its contraction with Lorentz four-vectors. Thus

$$\epsilon_{ijkl} = \begin{cases} +1 & \text{if } i,j,k,l \text{ is an even permutation of 0,1,2,3} \\ -1 & \text{if an odd permutation} \\ 0 & \text{otherwise} \end{cases}$$

A contraction of the form $\epsilon_{ij\mu\nu}p_\mu q_\nu$ may be written as eps(i,j,p,q), with I and J flagged as indices, and so on.

17.2 Vector Variables

Apart from the line identification identifier in the G operator, all other arguments of the operators in this section are vectors. Variables used as such must be declared so by the type declaration VECTOR, for example:

```
vector  p1,p2;
```

declares P1 and P2 to be vectors. Variables declared as indices or given a mass are automatically declared vector by these declarations.

17.3 Additional Expression Types

Two additional expression types are necessary for high energy calculations, namely

17.3.1 Vector Expressions

These follow the normal rules of vector combination. Thus the product of a scalar or numerical expression and a vector expression is a vector, as are the sum and difference of vector expressions. If these rules are not followed, error messages are printed. Furthermore, if the system finds an undeclared variable where it expects a vector variable, it will ask the user in interactive mode whether to make that variable a vector or not. In batch mode, the declaration will be made automatically and the user informed of this by a message.

Examples:

Assuming P and Q have been declared vectors, the following are vector expressions

```
p
2*q/3
2*x*y*p - p.q*q/(3*q.q)
```

whereas p*q and p/q are not.

17.3.2 Dirac Expressions

These denote those expressions which involve γ matrices. A γ matrix is implicitly a 4×4 matrix, and so the product, sum and difference of such expressions, or the product of a scalar and Dirac expression is again a Dirac expression. There are no Dirac variables in the system, so whenever a scalar variable appears in a Dirac expression without an associated γ matrix expression, an implicit unit 4 by 4 matrix is assumed. For example, g(l,p) + m denotes g(l,p) + m*<unit 4 by 4 matrix>. Multiplication of Dirac expressions, as for matrix expressions, is of course non-commutative.

17.4 Trace Calculations

When a Dirac expression is evaluated, the system computes one quarter of the trace of each γ matrix product in the expansion of the expression. One quarter of each trace is taken in order to avoid confusion between the trace of the scalar M, say, and M representing M * <unit 4 by 4 matrix>. Contraction over indices occurring in such expressions is also performed. If an unmatched index is found in such an expression, an error occurs.

The algorithms used for trace calculations are the best available at the time this system was produced. For example, in addition to the algorithm developed by Chisholm for contracting indices in products of traces, REDUCE uses the elegant algorithm of Kahane for contracting indices in γ matrix products. These algorithms are described in Chisholm, J. S. R., Il Nuovo Cimento X, 30, 426 (1963) and Kahane, J., Journal Math. Phys. 9, 1732 (1968).

It is possible to prevent the trace calculation over any line identifier by the declaration NOSPUR. For example,

```
nospur l1,l2;
```

will mean that no traces are taken of γ matrix terms involving the line numbers L1 and L2. However, in some calculations involving more than one line, a catastrophic error

```
This NOSPUR option not implemented
```

can occur (for the reason stated!) If you encounter this error, please let us know!

A trace of a γ matrix expression involving a line identifier which has been declared NOSPUR may be later taken by making the declaration SPUR.

See also the CVIT package for an alternative mechanism (chapter 15.15).

17.5 Mass Declarations

It is often necessary to put a particle "on the mass shell" in a calculation. This can, of course, be accomplished with a LET command such as

```
let p.p= m^2;
```

but an alternative method is provided by two commands MASS and MSHELL. MASS takes a list of equations of the form:

```
<vector variable> = <scalar variable>
```

for example,

```
mass p1=m, q1=mu;
```

The only effect of this command is to associate the relevant scalar variable as a mass with the corresponding vector. If we now say

```
mshell <vector variable>,...,<vector variable>;
```

and a mass has been associated with these arguments, a substitution of the form

```
<vector variable>.<vector variable> = <mass>^2
```

is set up. An error results if the variable has no preassigned mass.

17.6 Example

We give here as an example of a simple calculation in high energy physics the computation of the Compton scattering cross-section as given in Bjorken and Drell Eqs. (7.72) through (7.74). We wish to compute the trace of

$$\frac{\alpha^2}{2}\left(\frac{k'}{k}\right)^2\left(\frac{\gamma.p_f+m}{2m}\right)\left(\frac{\gamma.e'\gamma.e\gamma.k_i}{2k.p_i}+\frac{\gamma.e\gamma.e'\gamma.k_f}{2k'.p_i}\right)\left(\frac{\gamma.p_i+m}{2m}\right)$$
$$\left(\frac{\gamma.k_i\gamma.e\gamma.e'}{2k.p_i}+\frac{\gamma.k_f\gamma.e'\gamma.e}{2k'.p_i}\right)$$

where k_i and k_f are the four-momenta of incoming and outgoing photons (with polarization vectors e and e' and laboratory energies k and k' respectively) and p_i, p_f are incident and final electron four-momenta.

Omitting therefore an overall factor $\frac{\alpha^2}{2m^2}\left(\frac{k'}{k}\right)^2$ we need to find one quarter of the trace of

$$(\gamma.p_f+m)\left(\frac{\gamma.e'\gamma.e\gamma.k_i}{2k.p_i}+\frac{\gamma.e\gamma.e'\gamma.k_f}{2k'.p_i}\right)(\gamma.p_i+m)$$
$$\left(\frac{\gamma.k_i\gamma.e\gamma.e'}{2k.p_i}+\frac{\gamma.k_f\gamma.e'\gamma.e}{2k'.p_i}\right)$$

A straightforward REDUCE program for this, with appropriate substitutions (using `P1` for p_i, `PF` for p_f, `KI` for k_i and `KF` for k_f) is

```
on div; % this gives output in same form as Bjorken and Drell.
mass ki= 0, kf= 0, p1= m, pf= m; vector e,ep;
% if e is used as a vector, it loses its scalar identity as
%      the base of natural logarithms.
mshell ki,kf,p1,pf;
let p1.e= 0, p1.ep= 0, p1.pf= m^2+ki.kf, p1.ki= m*k,p1.kf=
    m*kp, pf.e= -kf.e, pf.ep= ki.ep, pf.ki= m*kp, pf.kf=
    m*k, ki.e= 0, ki.kf= m*(k-kp), kf.ep= 0, e.e= -1,
    ep.ep=-1;
for all p let gp(p)= g(l,p)+m;
comment this is just to save us a lot of writing;
gp(pf)*(g(l,ep,e,ki)/(2*ki.p1) + g(l,e,ep,kf)/(2*kf.p1))
  * gp(p1)*(g(l,ki,e,ep)/(2*ki.p1) + g(l,kf,ep,e)/
    (2*kf.p1))$
write "The Compton cxn is",ws;
```

(We use `P1` instead of `PI` in the above to avoid confusion with the reserved variable `PI`).

This program will print the following result

```
                              (-1)        (-1)            2
The Compton cxn is 1/2*K*KP     + 1/2*K    *KP + 2*E.EP  - 1
```

17.7 Extensions to More Than Four Dimensions

In our discussion so far, we have assumed that we are working in the normal four dimensions of QED calculations. However, in most cases, the programs will also work in an arbitrary number of dimensions. The command

```
vecdim <expression>;
```

sets the appropriate dimension. The dimension can be symbolic as well as numerical. Users should note however, that the `EPS` operator and the γ_5 symbol (`A`) are not properly defined in other than four dimensions and will lead to an error if used.

Chapter 18

REDUCE and Rlisp Utilities

REDUCE and its associated support language system Rlisp include a number of utilities which have proved useful for program development over the years. The following are supported in most of the implementations of REDUCE currently available.

18.1 The Standard Lisp Compiler

Many versions of REDUCE include a Standard Lisp compiler that is automatically loaded on demand. You should check your system specific user guide to make sure you have such a compiler. To make the compiler active, the switch `COMP` should be turned on. Any further definitions input after this will be compiled automatically. If the compiler used is a derivative version of the original Griss-Hearn compiler, (M. L. Griss and A. C. Hearn, "A Portable LISP Compiler", SOFTWARE — Practice and Experience 11 (1981) 541-605), there are other switches that might also be used in this regard. However, these additional switches are not supported in all compilers. They are as follows:

PLAP If ON, causes the printing of the portable macros produced by the compiler;

PGWD If ON, causes the printing of the actual assembly language instructions generated from the macros;

PWRDS If ON, causes a statistic message of the form
`<function> COMPILED, <words> WORDS, <words> LEFT`
to be printed. The first number is the number of words of binary program space the compiled function took, and the second number the number of words left unused in binary program space.

18.2 Fast Loading Code Generation Program

In most versions of REDUCE, it is possible to take any set of Lisp, Rlisp or REDUCE commands and build a fast loading version of them. In Rlisp or REDUCE, one does the following:

```
faslout <filename>;
<commands or IN statements>
faslend;
```

To load such a file, one uses the command LOAD, e.g. `load foo;` or `load foo,bah;`

This process produces a fast-loading version of the original file. In some implementations, this means another file is created with the same name but a different extension. For example, in PSL-based systems, the extension is `b` (for binary). In CSL-based systems, however, this process adds the fast-loading code to a single file in which all such code is stored. Particular functions are provided by CSL for managing this file, and described in the CSL user documentation.

In doing this build, as with the production of a Standard Lisp form of such statements, it is important to remember that some of the commands must be instantiated during the building process. For example, macros must be expanded, and some property list operations must happen. The REDUCE sources should be consulted for further details on this.

To avoid excessive printout, input statements should be followed by a $ instead of the semicolon. With LOAD however, the input doesn't print out regardless of which terminator is used with the command.

If you subsequently change the source files used in producing a fast loading file, don't forget to repeat the above process in order to update the fast loading file correspondingly. Remember also that the text which is read in during the creation of the fast load file, in the compiling process described above, is *not* stored in your REDUCE environment, but only translated and output. If you want to use the file just created, you must then use `LOAD` to load the output of the fast-loading file generation program.

When the file to be loaded contains a complete package for a given application, `LOAD_PACKAGE` rather than `LOAD` should be used. The syntax is the same. However, `LOAD_PACKAGE` does some additional bookkeeping such as recording that this package has now been loaded, that is required for the correct operation of the system.

18.3 The Standard Lisp Cross Reference Program

`CREF` is a Standard Lisp program for processing a set of Standard LISP function definitions to produce:

1. A "summary" showing:
   (a) A list of files processed;
   (b) A list of "entry points" (functions which are not called or are only called by themselves);
   (c) A list of undefined functions (functions called but not defined in this set of functions);
   (d) A list of variables that were used non-locally but not declared `GLOBAL` or `FLUID` before their use;
   (e) A list of variables that were declared `GLOBAL` but not used as `FLUID`s, i.e., bound in a function;
   (f) A list of `FLUID` variables that were not bound in a function so that one might consider declaring them `GLOBAL`s;
   (g) A list of all `GLOBAL` variables present;
   (h) A list of all `FLUID` variables present;
   (i) A list of all functions present.
2. A "global variable usage" table, showing for each non-local variable:
   (a) Functions in which it is used as a declared `FLUID` or `GLOBAL`;
   (b) Functions in which it is used but not declared;
   (c) Functions in which it is bound;

(d) Functions in which it is changed by SETQ.

3. A "function usage" table showing for each function:

   (a) Where it is defined;
   (b) Functions which call this function;
   (c) Functions called by it;
   (d) Non-local variables used.

The program will also check that functions are called with the correct number of arguments, and print a diagnostic message otherwise.

The output is alphabetized on the first seven characters of each function name.

18.3.1 Restrictions

Algebraic procedures in REDUCE are treated as if they were symbolic, so that algebraic constructs will actually appear as calls to symbolic functions, such as AEVAL.

18.3.2 Usage

To invoke the cross reference program, the switch CREF is used. `on cref` causes the cref program to load and the cross-referencing process to begin. After all the required definitions are loaded, `off cref` will cause the cross-reference listing to be produced. For example, if you wish to cross-reference all functions in the file `tst.red`, and produce the cross-reference listing in the file `tst.crf`, the following sequence can be used:

```
out "tst.crf";
on cref;
in "tst.red"$
off cref;
shut "tst.crf";
```

To process more than one file, more IN statements may be added before the call of `off cref`, or the IN statement changed to include a list of files.

18.3.3 Options

Functions with the flag NOLIST will not be examined or output. Initially, all Standard Lisp functions are so flagged. (In fact, they are kept on a list

NOLIST!*, so if you wish to see references to *all* functions, then CREF should be first loaded with the command `load cref`, and this variable then set to NIL).

It should also be remembered that any macros with the property list flag EXPAND, or, if the switch FORCE is on, without the property list flag NOEXPAND, will be expanded before the definition is seen by the cross-reference program, so this flag can also be used to select those macros you require expanded and those you do not.

18.4 Prettyprinting Reduce Expressions

REDUCE includes a module for printing REDUCE syntax in a standard format. This module is activated by the switch PRET, which is normally off.

Since the system converts algebraic input into an equivalent symbolic form, the printing program tries to interpret this as an algebraic expression before printing it. In most cases, this can be done successfully. However, there will be occasional instances where results are printed in symbolic mode form that bears little resemblance to the original input, even though it is formally equivalent.

If you want to prettyprint a whole file, say `off output,msg;` and (hopefully) only clean output will result. Unlike DEFN, input is also evaluated with PRET on.

18.5 Prettyprinting Standard Lisp S-Expressions

REDUCE includes a module for printing S-expressions in a standard format. The Standard Lisp function for this purpose is PRETTYPRINT which takes a Lisp expression and prints the formatted equivalent.

Users can also have their REDUCE input printed in this form by use of the switch DEFN. This is in fact a convenient way to convert REDUCE (or Rlisp) syntax into Lisp. `off msg;` will prevent warning messages from being printed.

NOTE: When DEFN is on, input is not evaluated.

Chapter 19

Maintaining REDUCE

Since January 1, 2009 REDUCE is Open Source Software. It is hosted at

http://reduce-algebra.sourceforge.net/

We mention here three ways in which REDUCE is maintained. The first is the collection of queries, observations and bug-reports. All users are encouraged to subscribe to the mailing list that Sourceforge.net provides so that they will receive information about updates and concerns. Also on SourceForge there is a bug tracker and a forum. The expectation is that the maintainers and keen users of REDUCE will monitor those and try to respond to issues. However these resources are not there to seek answers to Maths homework problems - they are intended specifically for issues to do with the use and support of REDUCE.

The second level of support is provided by the fact that all the sources of REDUCE are available, so any user who is having difficulty either with a bug or understanding system behaviour can consult the code to see if (for instance) comments in it clarify something that was unclear from the regular documentation.

The source files for REDUCE are available on SourceForge in the Subversion repository. Check the "code/SVN" tab on the SourceForge page to find instructions for using a Subversion client to fetch the most up to date copy of everything. From time to time there may be one-file archives of a snapshot of the sources placed in the download area on SourceForge, and eventually some of these mat be marked as "stable" releases, but at present it is recommended that developers use a copy from the Subversion repository.

The files fetched there come with a directory called "trunk" that holds the main current REDUCE, and one called "branches" that is reserved for future experimental versions. All the files that we have for creating help files and

manuals should also be present in the files you fetch.

The packages that make up the source for the algebraic capabilities of REDUCE are in the "packages" sub-directory, and often there are test files for a package present there and especially for contributed packages there will be documentation in the form of a LaTeX file. Although REDUCE is coded in its own language many people in the past have found that it does not take too long to start to get used to it.

In various cases even fairly "ordinary end users" may wish to fetch the source version of REDUCE and compile it all for themselves. This may either be because they need the benefit of a bug-fix only recently checked into the subversion repository or because no pre-compiled binary is available for the particular computer and operating system they use. This latter is to some extent unavoidable since REDUCE can run on both 32 and 64-bit Windows, the various MacOSX options (eg Intel and Powerpc), many different distributions of Linux, some BSD variants and Solaris (at least). It is not practically feasible for us to provide a constant stream of up to date ready-built binaries for all these.

There are instructions for compiling REDUCE present at the top of the trunk source tree. Usually the hardest issue seems to be encuring that your computer has an adequate set of development tools and libraries installled before you start, but once that is sorted out the hope is that the compilation of REDUCE should proceed uneventfully if sometimes tediously.

In a typical Open Source way the hope is that some of those who build REDUCE from source or explore the source (out of general interest or to pursue an understanding of some bug or detail) will transform themselves into contributors or developers which moves on to the third level of support.

At this third level any user can contribute proposals for bug fixes or extensions to REDUCE or its documentation. It might be valuable to collect a library of additional user-contributed examples illustrating the use of the system too. To do this first ensure that you have a fully up to date copy of the sources from Subversion, and then depending on just what sort of change is being proposed provide the updates to the developers via the SourceForge bug tracker or other route. In time we may give more concrete guidance about the format of changes that will be easiest to handle. It is obviously important that proposed changes have been properly tested and that they are accompanied with a clear explanation of why they are of benefit. A specific concern here is that in the past fixes to a bug in one part of REDUCE have had bad effects on some other applications and packages, so some degree of caution is called for. Anybody who develops a significant whole new package for REDUCE is encouraged to make the developers aware so that it can be considered for inclusion.

So the short form explanation about Support and Maintenance is that it is mainly focussed around the SourceForge system. That if discussions about bugs, requirements or issues are conducted there then all users and potential users of REDUCE will be able to benefit from reviewing them, and the Sourceforge mailing lists, tracker, forums and wiki will grow to be both a static repository of answers to common questions, an active set of locations to to get new issues looked at and a focus for guiding future development.

Appendix A

Reserved Identifiers

We list here all identifiers that are normally reserved in REDUCE including names of commands, operators and switches initially in the system. Excluded are words that are reserved in specific implementations of the system.

Commands	ALGEBRAIC ANTISYMMETRIC ARRAY BYE CLEAR CLEARRULES COMMENT CONT DECOMPOSE DEFINE DEPEND DISPLAY ED EDITDEF END EVEN FACTOR FOR FORALL FOREACH GO GOTO IF IN INDEX INFIX INPUT INTEGER KORDER LET LINEAR LISP LISTARGP LOAD LOAD_PACKAGE MASS MATCH MATRIX MSHELL NODEPEND NONCOM NONZERO NOSPUR ODD OFF ON OPERATOR ORDER OUT PAUSE PRECEDENCE PRINT_PRECISION PROCEDURE QUIT REAL REMFAC REMIND RETRY RETURN SAVEAS SCALAR SETMOD SHARE SHOWTIME SHUT SPUR SYMBOLIC SYMMETRIC VECDIM VECTOR WEIGHT WRITE WTLEVEL
Boolean Operators	EVENP FIXP FREEOF NUMBERP ORDP PRIMEP
Infix Operators	:= = >= > <= < => + - * / ^ ** . WHERE SETQ OR AND MEMBER MEMQ EQUAL NEQ EQ GEQ GREATERP LEQ LESSP PLUS DIFFERENCE MINUS TIMES QUOTIENT EXPT CONS
Numerical Operators	ABS ACOS ACOSH ACOT ACOTH ACSC ACSCH ASEC ASECH ASIN ASINH ATAN ATANH ATAN2 COS COSH COT COTH CSC CSCH EXP FACTORIAL FIX FLOOR HYPOT LN LOG LOGB LOG10 NEXTPRIME ROUND SEC SECH SIN SINH SQRT TAN TANH

Prefix Operators	APPEND ARBCOMPLEX ARBINT ARGLENGTH CEILING CI COEFF COEFFN COFACTOR CONJ DEG DEN DET DF DILOG EI EPS ERF FACTORIZE FIRST GCD G IMPART INT INTERPOL LCM LCOF LENGTH LHS LINELENGTH LTERM MAINVAR MAT MATEIGEN MAX MIN MKID NULLSPACE NUM PART PF PRECISION PROD RANDOM RANDOM_NEW_SEED RANK REDERR REDUCT REMAINDER REPART REST RESULTANT REVERSE RHS SECOND SET SHOWRULES SI SIGN SOLVE STRUCTR SUB SUM THIRD TP TRACE VARNAME
Reserved Variables	_LINE_ CARD_NO E EVAL_MODE FORT_WIDTH HIGH_POW I INFINITY K!* LOW_POW NIL PI ROOT_MULTIPLICITIES T
Switches	ADJPREC ALGINT ALLBRANCH ALLFAC ALLOWDFINT BALANCE_MOD BFSPACE COMBINEEXPT COMBINELOGS COMMUTEDF COMP COMPLEX CRAMER CREF DEFN DEMO DFINT DIV ECHO ERRCONT EVALLHSEQP EXP EXPANDDF EXPANDLOGS EZGCD FACTOR FORT FULLROOTS GCD IFACTOR INT INTSTR LCM LIST LISTARGS MCD MODULAR MSG MULTIPLICITIES NAT NERO NOCOMMUTEDF NOSPLIT OUTPUT PERIOD PRECISE PRECISE_COMPLEX PRET PRI RAT RATARG RATIONAL RATIONALIZE RATPRI REVPRI RLISP88 ROUNDALL ROUNDBF ROUNDED SAVESTRUCTR SIMPNONCOMDF SOLVESINGULAR TIME TRA TRFAC TRIGFORM TRINT
Other Reserved Ids	BEGIN DO EXPR FEXPR INPUT LAMBDA LISP MACRO PRODUCT REPEAT SMACRO SUM UNTIL WHEN WHILE WS

Appendix B

Changes since Version 3.8

New packages assert bibasis breduce cdiff clprl gcref guardian lessons libreduce lpdo redfront reduce4 utf8

Package rlisp Support for namespaces (::)

Package alg New switch precise_complex

Improvements for switch combineexpt (exptchk.red)

Package poly Improvements for differentiation: new switches expanddf, allowdfint etc (from odesolve)

Package solve polyp

Package modsr legendre_symbol

Index