A Brief Overview of REDUCE
Highlighting Differences to R

Martin Gregory

November 3, 2026

Contents

1 Introduction **1**

2 Terminology **2**

3 Summary of Important Differences **2**

4 Language Elements **3**
4.1 Names, symbols and variables . . . 3
4.2 Numbers . . . 3
4.3 Strings . . . 4
4.4 Expressions . . . 4
4.5 Aggregate data types . . . 4
4.6 Comments . . . 5

5 Statements **5**
5.1 Assignment statements . . . 5
5.2 Expression as statement . . . 5
5.3 Group and Compound Statements . . . 5
5.4 Conditional Statement . . . 6
5.5 *FOR* statement . . . 6
5.6 *WHILE ... DO* . . . 7
5.7 *REPEAT ... UNTIL* . . . 7
5.8 *RETURN* . . . 7

6 Commands **7**

7 Declarations **8**

8 Options **8**

9 Operators and Substitution **9**

10 Procedures **10**

1 Introduction

The aim of this document is to provide a brief introduction to REDUCE for R users unfamiliar with REDUCE, highlighting differences between the two environments. This vignette is neither a reference to nor a tutorial for using REDUCE, for these purposes see the online REDUCE User's Manual [1] or the online Tutorials [2].

REDUCE is an open source, portable, general-purpose, computer algebra system. The first versions date from the early 1960's as a class of formal computations for problems in high energy physics which are difficult to do and verify manually. Today the system has expanded its scope significantly, is actively developed and has many packages contributed by users.

REDUCE is written in Standard LISP of which there are several implementations. It consist of two distinct but inter-related modes: algebraic and symbolic. The algebraic mode is written in symbolic mode and both modes support LISP features using a simpler, ALGOL-based syntax. In general users who are not developing REDUCE packages have no need for symbolic mode, but it can be used to improve performance of complicated calculations. Futher details are beyond the scope of this document but interested readers can find details in Chapter 21 of the REDUCE User Manual [4] and more in the REDUCE Symbolic Mode Primer [4].

Conventions used in this vignette:

- differences are highlighted by a ⋆ in the margin
- REDUCE keywords are rendered in italic upper case in the text and roman upper case in examples
- when examples show both commands and output, the commands are numbered. If only commands are shown, no numbers are used.

2 Terminology

Before going into details it is helpful to clarify some REDUCE terminology.

command refers to a certain type of statement which affect the environment, for example *LOAD_PACKAGE*, *QUIT*, *CLEAR*.

log is the REDUCE equivalent of the R transcript. In contrast to R, input is preceded by an input line number.

operator is used in four different contexts:

- a *prefix operator* is a function and takes one or more arguments and is written `f(a1, ... aN)`. In the case of functions with a single argument, the parenthesis may be omitted, e.g. sin(x) and sin x are identical. Most commands and declarations may be written as prefix operators, i.e. with parentheses; ⋆
- a *boolean operator* is a unary logical operator, e.g. *evenp* is true if its argument is even. Boolean operators may only be used in *IF*, *FOR*, *WHILE* or *UNTIL* statements; ⋆
- an *infix operator* is an operator which takes two arguments with the operator between the arguments. Arithmetic and comparison operators are examples;
- a variable may be declared to be an operator in which case it may take arguments. For an example of using this in combination with let to define mathematical functions with specific properties, see section 9.

procedure is equivalent to the R function *function*. Procedures are automatically declared as prefix operators.

switch is an option with only logical values.

kernel is, naively, an expression which cannot be further simplified. See Section 8.1 of the REDUCE User's Manual for details.

3 Summary of Important Differences

If using *redcas* without any programming in REDUCE, there are some differences to be aware of:

1. names consist of a letter followed by any combination of letters, decimal digits and underscore;
2. the variable names e, i and pi are reserved for the numbers e, $\sqrt{-1}$ and π, respectively;
3. names in REDUCE are case-insensitive by default;
4. the operator := is used for assignment, $=$ is the comparison operator;
5. statements require a terminator, either semi-colon or dollar sign.

4 Language Elements

4.1 Names, symbols and variables

1. the terms identifier and symbol are the equivalent of name and symbol in R;
2. variable names consist of a letter followed by any combination of letters, decimal digits and underscore. Other symbols may be used in variable names if they are escaped using the exclamation mark. Since REDUCE does not have a namespace concept, internally this feature is used extensively; ⋆
3. variables do not need to be declared but also do not need to have a value. A variable to which no value has been assigned is known as unbound and can represent a variable in the mathematical sense; ⋆
4. apart from literal strings and characters preceded by an exclamation mark, REDUCE is case-insensitive. In order to maintain mixed case in identifiers it is necessary to precede upper case letters with the exclamation mark;
5. REDUCE has a large number of reserved names covering commands, operators, variables and switches. The most relevant reserved names, in alphabetical order, are:

   e the base of natural logarithms. If the *ROUNDED* switch is on it has the value of e to the current degree of floating point precision;

   Golden_Ratio the number $(1 + \sqrt{5})/2$;

   i the square root of -1. In i^n, i^2 is replaced by -1. As a formal parameter to a procedure, as a local variable or as the index variable in a *for* statement i may be used as a variable although this prevents its use in complex numbers in those contexts;

   infinity represents ∞ in limit, power series and definite integration calculations. REDUCE does not handle arithmetic with *infinity*, treating it as any unbound variable, e.g. *infinity* - *infinity* returns 0;

   negative used in the *roots* package;

   nil in symbolic mode, logical false, in algebraic mode, a synonym for zero which is also logical false;

   pi represents π. If the *ROUNDED* switch is on it has the value of π to the current degree of floating point precision;

   positive used in the *roots* package;

   t logical true.

4.2 Numbers

integers are of arbitrary size; ⋆

rational numbers are stored as the quotient of two integers and are therefore also of arbitrary size; ⋆

floats are known as rounded numbers. The precision of the mantissa defaults to 50 but may be set to any value using the *precision* declaration. A value of 0 displays the current precision. The exponent, being ⋆

an integer, may be of arbitary size. By default, REDUCE does not use floats. Setting the switch *ROUNDED* to on enables floats;

complex numbers have the form $a + b * i$ where i is a reserved identifier with the property $i^2 = -1$. The real and imaginary parts may be numbers or variables.

scientific notation is supported both for input and output. Output is controlled via the declaration *SCIENTIFIC_NOTATION*;

modular arithmetic can be turned on using the declaration *SETMOD* to define the modulus and the *MODULAR* switch. When turned on, modular arithmetic also applies to the coefficients of polynomials. Note that a *SETMOD* declaration is required before turning the *MODULAR* switch on.

4.3 Strings

Strings are very sparingly used in REDUCE and so the features for manipulating them are limited to the *WRITE* command. Symbolic mode provides standard Lisp features for string handling. Details can be found in Chapter 21 of the REDUCE User Manual[1] or in the REDUCE Symbolic Mode Primer[4]

1. Literal strings are identified using double quotes;
2. double quotes are escaped within a string by doubling the double quote;
3. single quotes are generally only used in symbolic mode in the same way as in LISP, for example, 'a is the equivalent of (quote a).

4.4 Expressions

Mathematical expressions are similar to R, combining numbers, variables and functions using the operators +, -, *, / and ^ or ** and parentheses for precedence.

4.5 Aggregate data types

There are several aggregate data types:

array declares one or more arrays including their dimensions using the syntax

ARRAY a1(d1[,d2[,...[,dN]]]), a2(d1[,d2[,...[,dM]]]), ... ;

Unlike R, array indexing is **zero-based**, so a declared dimension size of n means an actual dimension size of $n + 1$. Furthermore, declaring an array causes all its element to be set to zero. Elements may be numbers, identifiers, expressions or strings and elements of different types are allowed in the same array. Arrays are global even when they are declared within a function. Each element of an array must be assigned separately, possibly via nested loops. Assigning an array to a variable or to the name of another declared array has no effect: the assigned variable is unbound. Unlike R, REDUCE arrays have no print method;

equation consists of two expressions separated by $=$. Functions *LHS* and *RHS* return the left and right hand side of the equations, respectively. Equations may be added, subtracted, simplified, solved, differentiated and integrated.

list as in R may contain any data type. Lists are displayed using braces and assigned using the *LIST* function or braces. The package *ASSIST*[1] provides some essential functions for working with lists. List elements cannot easily be named;

matrix elements may be numbers, identifiers or expressions but may not be strings. Similar to the *ARRAY* declaration, the *MATRIX* declaration declares one or more matrices with the dimensions and sets all elements to zero. Matrix indexing, in contrast to arrays, is **1-based**. In addition the *MAT* function can be used to assign all values to a matrix in a single statement and, if necessary, declare it. Elements

[1] *ASSIST* must be explicitly loaded. *redcas* does this during session start

may also be individually assigned. Functions to calculate the determinant, eigenvalue equation and eigenvectors, transpose, trace, cofactors, rank and linear basis of a matrix are available. Unlike arrays, matrices have a print method;

tensor the user-contributed packages *ATENSOR*, *CANTENS*, *DUMMY*, *EXCALC*, *FIDE* and *PHYSOP* provide support for tensor calculations.

avector the user-contributed packages *ASSIST*, *AVECTOR*, *EXCALC*, *ORTHOVEC*, *PHYSOP* and *SPARSE* provide support for vector calculations.

4.6 Comments

REDUCE provides three different ways to comment code:

- the *COMMENT* statement, terminated by semi-colon or dollar sign allows multi-line comments;
- anything following a % sign until the end of the line is ignored;
- The C-style multi-line comment (`/* ... */`) is supported.

5 Statements

Statements require a terminator which may be the semicolon or dollar sign. These differ in that a dollar terminator suppresses displaying the input even if the *ECHO* switch is on.

5.1 Assignment statements

Assignment of variables is done using the operator :=. The symbol = is either a comparison operator or part of an equation and is not used for assignment. As in R, assignment statements may be used as targets of assignments. For details see Section 5.1 of the REDUCE User's Manual.

5.2 Expression as statement

As in R, an expression can serve as a statement causing the result of evaluating the expression to be displayed. If the expression is an array only the array name is printed as there is no print method for arrays.

5.3 Group and Compound Statements

REDUCE has two different ways of using multiple statements where syntax requires a single statement:

GROUP statement uses the syntax `<<stmt1; ...; stmtN>>` and is equivalent to braces in R. For example:

```
IF x>1 THEN <<x:=x-1; WRITE "x is ", x>> ELSE WRITE x, " is not greater than 1";
```

The group statement returns the value of the last expression unless this expression is followed by a terminator prior to the closing >> in which case the value is zero.

compound statement, also know as block statement, has the form `BEGIN ... END` and allows for declaration of local variables and can use a *RETURN* statement to explicitly exit the block. As an example consider

```
a:=1 ; b:=2 ;
BEGIN SCALAR a ; % a is local to block, initialized to 0
   b:=a+b+25 ;
   IF a=1 THEN RETURN ;
   WRITE "In block a is ", a, ", b is ", b
END ;
WRITE "After block a is ", a, ", b is ", b ;
```

Because of the *SCALAR* declaration a is local to the block and is initialized to zero. b is accessible from the block. The result of execution is:

```
In block a is 0, b is 27
After block a is 1, b is 27
```

In addition to *SCALAR*, *INTEGER* and *REAL* may also be used.

Both types require a terminator after the block, but not on the last statement of the block. For group statement see Section 5.2, for compound statements Section 5.7 of the REDUCE User's Manual.

5.4 Conditional Statement

Syntax is similar R with two exceptions: a *THEN* keyword is required and using parentheses around the condition is optional:

IF *condition* THEN *statement* [ELSE *statement*] ;

The condition can be a boolean, numeric or scalar expression. In the non-boolean case, any value other than zero is true. Note that there is no terminator after the *THEN* statement if there is an *ELSE* statement. Group statements may be used. The *IF* statement returns the value of the last statement executed and may be part of an expression in the target of an assignment. For example:

```
a:=IF x<5 THEN 123 ELSE 456;
b:=u + v^(IF NUMBERP z THEN 10*z ELSE 1) + w;
```

If no action is to be performed when the condition is true, the *THEN* statement may be omitted. For example

```
IF x=1 THEN ELSE WRITE "not 1" ;
```

If the *THEN* keyword is omitted and an *ELSE* keyword is present, REDUCE responds with the following rather confusing error message

```
***** else invalid in if statement
```

If the *THEN* keyword is omitted and no *ELSE* keyword is present the message is:

```
***** ; invalid in if statement
```

For details see Section 5.3 of the REDUCE User's Manual.

5.5 *FOR* statement

The syntax differs significantly from that of R and is as follows:

FOR *interation-expression action expression*

In the case of indexed iteration *interation-expression* has two forms

var := *number* STEP *number* UNTIL *number*
var := *number* : *number*

with the latter case assuming the step is 1. *number* may also be an expression which resolves to a number. In the case of interating down a list the *interation-expression* has the form

EACH *var* IN|ON *list*

There are five different actions:

DO executes the expression for each iteration. This action does not return a value;

PRODUCT returns the product of all expressions;

SUM returns the sum of all expressions;

COLLECT returns a list of the expressions;

JOIN expects the result of each expression to be a list and returns a list of these lists.

For details see Section 5.4 of the REDUCE User's Manual.

5.6 *WHILE ... DO*

The *WHILE* statement differs from R in that it requires a *DO* keyword before the statement: ⋆

```
WHILE condition DO statement ;
```

The statement can be a group or compound statement. The condition is tested at the start of each loop. *WHILE* does not return a value and the only explicit statement for prematurely ending the loop is *GO TO* ⋆ or its alias *GOTO*. For details on *WHILE* see Section 5.5 of the REDUCE User's Manual and for *GO TO* see Section 5.7.2.

5.7 *REPEAT ... UNTIL*

The *REPEAT* statement differs from R in that it uses a condition. Essentially it is the same as *WHILE* with ⋆ the exception that the condition is tested after each loop execution.

```
REPEAT statement UNTIL condition ;
```

The statement can be a group or compound statement. Like *WHILE* no value is returned and prematurely ending the loop can only be done using *GO TO*. For details see Section 5.6 of the REDUCE User's Manual.

5.8 *RETURN*

By default the value returned from a compound statement is zero. The *RETURN* statement can be used to ⋆ return the result of an expression from a compound statement and control when the compound statement exits. It can be used

- directly within a compound statement,
- in a group statement within a compound statement, or
- in a conditional statement within a compound statement.

In all three cases it exits the compound statement. The syntax is

```
RETURN [expression] ;
```

Omitting the expression returns zero. For details see Section 5.7.3 of the REDUCE User's Manual.

6 Commands

Commands are statements which influence the REDUCE environment, for example, loading packages or determining what simplification methods to use.

CLEAR clears the value and any attributes of a variable, a component of an array, list or matrix or user-defined procedure.

RESETREDUCE removes all declarations, clears all variables and user-defined procedures, restores all switches to their default setting and resets the line number to one. Does not unload packages.

LOAD_PACKAGE loads one or more packages. If multiple packages are to be loaded package names are ⋆ separated by commas. Unlike R, this command is only needed for user-contributed packages – all ⋆ system packages are either loaded at session start or automatically when one of their functions are called.

ON / OFF turn a switch on or off. See section 8 below.

BYE / QUIT is used to terminate a session. For CSL, at least on UNIX-like operating systems, *control D* also terminates a session.

file handling commands There are three file handling commands:

IN is the equivalent of the R *source* function. It loads a file containing REDUCE commands. The file must end with the statement *;END;*. The only parameter is a string naming the file. If the file name is in a variable F, the syntax `<<F>>` may be used.

OUT is the equivalent of the R *sink* function. It takes a single parameter which is either the name of a file or `<<F>>` where F is a variable containing the name of the file. Further *OUT* commands change the file to be written to but do not close previously opened files. Note that output redirected by the *OUT* command is not available when using *redcas*. ⋆

SHUT closes an output file specified by the name of a file or `<<F>>` where F is a variable containing the name of the file.

File handling commands are documented in Section 12 of the REDUCEUser's Manual.

SHOWTIME writes the CPU time used since the last call of this command to the REDUCE log. On the first call it writes the CPU time used since the start of the session. The time is displayed to the nearest millisecond. See Section 6.5 of the REDUCE User's Manual for a description of using the Lisp *time* function which returns the number of milliseconds since the session started and may be assigned to a variable.

SHOW provides some limited information similar to R's *ls* function and is part of the *ASSIST* package. *SHOW* displays a list of objects specified by a (required) parameter which can take the values *ALL*, *SCALARS*, *ARRAYS*, *LISTS*, *MATRICES*, *VECTORS* or *FORMS*.

7 Declarations

Declarative statements assign a property or relationship to one or more variables. Important declarations are:

ARRAY described in section 4.5 above.

DEPEND declares a variable to be dependent on one or more other variables. This is relevant for functions such as differentiation: if an unbound variable is not declared dependent on the differentiating variable, it is treat as constant in this context and the result is zero. The declaration *nodepend* is used to remove dependencies. The operator *DEPATOM* provided by *ASSIST* returns a lists of the dependencies of its argument;

LET defines global substitution rules allowing one to influence how expressions are simplified. See section 9 below;

MATRIX declares one or more variables to be matrices. Unlike R a two dimensional array is not a matrix; ⋆

OPERATOR declares a variable to be a function or operator in the mathematical sense, i.e. it may take arguments; ⋆

Other declarative statements may be defined by user-contributed packages.

8 Options

While R provides a single feature (options) REDUCE uses three different ways to achieve the same effects. ⋆

The first feature is a *SWITCH* which, as the name suggests, is boolean. There are over two hundred switches affecting the behaviour of the system, some of which are defined in user-defined packages and only available if these are loaded. The commands *ON* and *OFF* are used to set switches[2]. See "Switch Summary" [3] or

[2]The *swset* procedure supplied with *redcas* can be used to toggle switches while recording the current state which can be restored using *swreset*

Appendix A.7 of the User's Manual [1] for a full list. For a list by context, see the vignette *Switches by Context*.

The second feature is to use declarations such as *PRECISION*, *KORDER* or *ORDER* with allow one to specify precision, the internal order of variables or the displayed order of variables in expressions, respectively.

The third feature uses global variables. For example, the *DF* operator (differentiation) uses the variable *FANCY_PRINT_DF* to determine how to display derivatives as in the following

$Setting$	$df(f,x,2,y)$
$PARTIAL$	$\frac{\partial^3 f}{\partial x^2 \partial y}$
$TOTAL$	$\frac{d^3 f}{dx^2 dy}$
$INDEXED$	$f_{x,x,y}$

9 Operators and Substitution

The operator declaration allows a variable to take arguments. For example

```
1: OPERATOR f, g ;
2: DF(f(x), x) ;
```

returns $df(x)/dx$ because $f(x)$ implies that f depends on x. REDUCE's substitution features allow one to define attributes of operators. There are two features providing substitution. The *SUB* operator provides once-off substitution while the *LET* declaration defines rules which apply whenever an expression in a rule is encountered. The syntax for *SUB* is

SUB(substitution_list, expression)

and the elements of the substitution list are of the form *kernel = expression*. If we want to define $df(x)/dx = g(x)$ in $d^2f(x)/dx^2 + df(x)/dx$ we could use

```
3: SUB(DF(f(x),x)=g(x), DF(f(x),x,2) + DF(f(x),x));

df(f(x),x,2) + g(x)
```

i.e. only the specific term is substituted. To have the substitution made in the second derivative we use a *LET* rule, the syntax of which is

LET substitution_list ;

where *substitution_list* is one or more rules of the following forms

var => expression
operator(arg,...,arg) => expression
arg infix-operator arg => expression

Using this we see

```
4: LET DF(f(x),x) => g(x) ;
5: DF(f(x),x,2) + DF(f(x),x));

df(g(x),x) + g(x)
```

If we used a variable other than x, no substitution will be made. If we define the rule as follows it applies to any variable:

```
6: LET DF(f(~x),x) => g(x) ;
7: DF(f(y),y,2) + DF(f(y),y));

df(g(y),y) + g(y)
```

10 Procedures

The *PROCEDURE* statement is the equivalent of *function()* in R. The syntax is

[type] *PROCEDURE* ([*arg, ..., arg*]); *statement* ;

where the procedure type is one of

ALGEBRAIC is the default type and may be omitted,

INTEGER an algebraic procedure which returns an integer,

REAL an algebraic procedure which returns a real number,

LISP creates a symbolic mode procedure, i.e. one which uses lisp code.

Some points to note:

- procedures are automatically declared as prefix operators;
- a procedure with exactly two argument can be declared an infix operator using the *INFIX* declaration;
- for a multi-statement procedure the *BEGIN . . . END* construct is used;
- although in some languages procedures don't return a value, in REDUCE they do. By default the value is zero (nil in symbolic mode). The *RETURN* statement allows the procedure to return a specified value and terminate.
- in the case of integer and real procedures, returning a value which does not match the type is an error;
- algebraic procedures can only return scalars. To return a list or matrix, instead of *PROCEDURE* use *LISTPROC* or *MATPROC*, respectively. No type is required. There is no corresponding procedure for returning arrays. These may be declared in the body of a procedure but will be global. Declaring an existing array will replace it;
- procedure arguments cannot be named;
- all arguments must be supplied even if they are empty. It is possible to bypass this restriction by using a list as the only argument;
- creating variables which have not been declared local are global. In this case REDUCE warns when the procedure is defined.

For further details see Section 15 of the REDUCE User's Manual.

References

[1] REDUCE Developers. *REDUCE User's Manual* [Online; accessed 2026-10-02]. Also part of the REDUCE installation.

[2] REDUCE Developers. *REDUCETutorials* [Online; accessed 2026-10-02]. Also partially available with the REDUCEinstallation.

[3] REDUCE Developers Switch Summary [Online; accessed 2026-10-02].

[4] Melenk, H. Symbolic Mode Primer [Online; accessed 2026-10-02], 2016. Also provided as part of the REDUCE installation.