REDUCE Symbolic Mode Primer

H. Melenk

Konrad–Zuse–Zentrum
für Informationstechnik Berlin
Takustrasse 7
14195 Berlin-Dahlem
Germany

Email: melenk@zib.de

1 Introduction

This document should explain some essential technical details for symbolic mode programming in REDUCE for those who have some experience in REDUCE algebraic programming and need to write a program in symbolic mode. For a general introduction to REDUCE please consult the REDUCE User's Manual or one of the available books, e.g. "Algebraic computing with REDUCE" by Malcolm MacCallum and Francis Wright (Oxford Press).

This text cannot be complete, as the set of facilities available in REDUCE is so rich that it would take years to describe all and months to read and understand such text. So a good policy for entering the business of symbolic mode programming is to study the source files - the liberal REDUCE distribution policy simplifies this - and to understand those parts which contribute to the field which one would like to use. This text tries to collect in one place some of the wide spread basic information in order to facilitate your walk through the REDUCE mountains.

When should you write a program in symbolic mode? Symbolic programs are not *a priori* better than algebraic programs - the essential thing is the mathematics which they implement. A common prejudice is that symbolic programs are more "efficient". This is true only if you can save in symbolic mode substantial algebraic evaluation steps. As long as most of the computing time is needed for a few big calculations steps (integrals, equation solving, polynomial gcd etc.) you will gain nothing when calling the same

procedures from symbolic mode. However, if you need structures which are not present in algebraic mode or if you can design an evaluation avoiding lots of conversions, the step to symbolic mode programming is justified.

As it is very difficult to design non trivial but short examples for symbolic mathematical programming no attempt has been made in that direction. The examples in this text all are silly - please look at the sources of REDUCE for meaningful material. The following pieces of the sources are good points for first reading as they provide a substantial functionality within a few pages of code:

1. module `polrep` in package `poly`: routines addf, addd and addm, the heart of standard form and standard quotient arithmetic,
2. module `det` of package `matrix`: internal arithmetic with symbolic entities (standard quotients) and clever local data structure,
3. module `rational` in package `poly`: implementation of a typical REDUCE domain,
4. module `maxmin` in package `alg`: a typical simplification routine for an operator,
5. module `algbool` in package `alg`: demonstrates how to supply "small" pieces of symbolic code for algebraic use.

For symbolic mode programming you will urgently need the *Standard LISP Report* which describes the basic LISP functions; these will be available under all REDUCE implementations and they guarantee an optimum of portability. However, in the course of the years REDUCE has imported some additional functions on the LISP level – they have been implemented on top of Standard LISP and live in the module `support` of the package `rlisp.red`. In order to prevent the reinvention of the wheel these functions are described in the appendix as an extension to the *Standard LISP Report*.

The description is based on the recent version of REDUCE. Some of the described features are not available in earlier versions.

2 Very short course on RLISP

2.1 What is RLISP

As you will know REDUCE is based on the programming language LISP, or to be more exact, on the LISP dialect `Standard LISP`. Fortunately you

need not learn the syntax of this language with its large number of brackets - the REDUCE language is used for algebraic programming as well as for symbolic programs and in that mode it allows you to use all of the rich LISP data structures. It is LISP semantics in high level REDUCE syntax. In the following it is expected that you are familiar with the REDUCE programming language as far as it is used for algebraic mode programs. You should know how to write a (recursive) procedure with parameters and local variables, how to build loops, while blocks, `if-then-else` clauses and `begin-end` resp $<< - >>$ blocks. If not, please study the REDUCE manual first or have a look at the source files of the REDUCE kernel - this is the best method for learning how to program in RLISP.

2.2 Modes, function classes

The symbols `symbolic` (or equivalent `lisp`) and `algebraic` play a double role: as a statement they switch the evaluation mode globally to *symbolic* or *algebraic mode* respectively, as a *mode prefix* they tell the REDUCE evaluator that one isolated evaluation should be done in the named mode. The scope of this prefix use can be rather narrow (e.g. one single expression) or cover a larger piece of code up to a complete procedure. Typically procedures in symbolic modules should be tagged "symbolic" explicitly although this might be redundant information in a totally symbolic context. If a procedure needs an evaluation mode different from the actual global one the *mode prefix* must be set.

In symbolic mode there are two additional procedure types available, `macro` and `smacro`. The discussion of `macro`s is beyond the scope of this document - their use requires extensive knowledge of LISP. On the other hand `smacro`s are frequently use in REDUCE and you will see lots of them in the sources. An `smacro` (an abbreviation for "substitution macro") is an ordinary procedure tagged with the symbol "smacro" and usually with a rather small body. At source read time (or better: at REDUCE translator time) any call for an `smacro` will be replaced literally by the body of the `smacro` procedure which saves execution time for the call protocol at run time. So the purpose of `smacro`s is twofold: encapsulation of frequently used pieces of code and an increased efficiency (avoiding function calls). Example:

```
smacro procedure my_diff(x,y); x-y;

symbolic procedure hugo(a,b); my_diff(a,b);
```

Here the formal function call *my_diff* in the procedure literally is replaced by the body of the `smacro` where x and y are replaced by a and b. Obviously this translation can be done only if the `smacro` declaration is entered in a REDUCE session before the first reference. And later changes of an `smacro` don't affect previously translated references.

Sometimes in older REDUCE sources you will find the symbol `expr` in front of a procedure declaration. This is more or less useless as the absence of `macro` or `smacro` symbols indicates that a procedure is of type `expr` - the default Standard LISP procedure type.

2.3 Evaluation model, symbols, and variables

The main difference between algebraic and symbolic mode lies in the `evaluation model`:

- In algebraic mode a symbol stands for itself as unknown as long as no value is assigned; after an assignment it plays the role of a representative for that value just like a variable in a standard programming language:

  ```
  1: x;
  X
  2: x:=y+1$
  3: x;
  Y + 1
  ```

  In symbolic mode there is a clear barrier between the role of a symbol as variable of the programming language RLISP, a named item which represents some variable value, and the role to be part of an algebraic expression. If you mean the *symbol* x you must write $'x$; without the quote tag x is considered as *variable* x and it will be asked for its value which is NOT $'x$ initially. Uninitialized variables cause bugs.

- Consequently all variables `must be declared`.

- In algebraic mode $u := (x + 1)\hat{}2$; [1] means it compute a formula by expanding (x+1)*(x+1); if a value had been assigned to x, substitute the value for x. In symbolic mode an algebraic expression is interpreted

[1] In this text the caret symbol "^" is used for exponentiation instead of the older FORTRAN like operator "**".

as statement to compute a numeric value, just like in *C* or *Pascal*. So $u := (x+1)\hat{}2$; in symbolic mode means: "take the value of variable x which is expected to be number, add 1, square and assign the resulting number to the variable u".

- If you want to refer to an `algebraic expression` as a data object, you must code it as an `algebraic form` (see below) and mark it as `constant` by a `quote` character. The only constants which don't need a `quote` are numbers and strings. Example:

```
u:='(expt (plus x 1) 2);
```

  assigns the (algebraic) expression $(x+1)\hat{}2$ to the variable u.

- algebraic mode implicitly supports standard arithmetic and algebraic evaluation for mathematical expressions of arbitrary complexity and for numbers from various domains. In symbolic mode, arithmetic with infix operators $+ \; - \; * \; \hat{} \; /$ is supported only for the basic LISP numbers (mainly integers). All arithmetic for formulas, even for domain elements such as rounded numbers, complex numbers etc. has to be performed by calling explicitly the functions which can do that job or by calling explicitly the REDUCE evaluator `reval` resp `aeval` (see below).

So symbolic mode programs are much more similar to programs in conventional programming languages such as Pascal or C. All algebraic functionality of REDUCE is available only as an explicit subroutine interface. In contrast to algebraic mode nothing is done implicitly for you.

2.4 Variable types

RLISP supports in symbolic mode the following classes of variables:

- `Local variables` in a procedure are those
  - declared in the parameter list,
  - declared in a `scalar` or `integer` statement in a begin-end block,
  - bound in the right-hand side of a `where statement`
  - implicitly introduced in a `for statement` .

These are valid only inside their local context. `scalar` variables are initialized to *nil*, `integer` to 0 unless they have a special initialization value (property `initvalue*`).

```
symbolic procedure my_adder(x,y);
   begin integer r;
      r:=x+y;
      return r:
   end;
```

In this routine x,y and r are local variables.

In algebraic mode the `where` statment is used to activate one more more rules locally. In symbolic mode mode only rules of the form `<name>=<value>` are allowed the the right-hand side of a `where` statement.

- `Global` variables have to be declared in a statement $global'(v_1\, v_2\, \cdots)$; These can be accessed from everywhere once they have been declared. Important: declare them before you use them for the first time in a session. They are initially set to *nil*.

  ```
  global '(my_support!* my_quotient!*);
  ```

  It is a common practice to use a trailing asterisk in names for global and fluid variables such that they easily can be distinguished from locals. Names of global variables may not be used as local variables.

- `Fluid` variables are similar to global variables as they can be accessed from everywhere. But in contrast to globals a `fluid` variable can occur as a local variable in a procedure. It then has temporarily the value assigned in the procedure (we say "it is rebound"); this value will be accessible globally by nested procedures as long as the rebinding procedure is not yet terminated. After termination the previous value is reassigned. Fluid variables are declared in a statement $fluid'(v_1\, v_2\, \cdots)$;. Like global variables they must be declared before use and they are initially set to *nil*.

  ```
  fluid '(my_var!*);
  my_var!*:='x;
  procedure compute(ex,my_var!*);
     do_some_work(ex);
  ```

In this case the variable my_var* has been initialized to the symbol x. If then a call $compute('a,'z)$ is performed, the variable my_var* will be set to z during the execution of the body of *compute*.

- A `switch` can be declared using the `switch` statement $switch\, s_1, s_2 \cdots;$ where $s_1, s_2, \cdots$ are symbols. REDUCE automatically connects a fluid variable with each switch by prefixing an asterisk to the symbol name. This variable represents the *value* of the switch, which is either *nil* or t and is set by the statements `on` and `off`. E.g. the switch *nat* has the associated global variable $*nat$.

- `share` variables are also global variables which are handled specially by the REDUCE evaluator such that their symbolic mode values are identical with their role as symbols in algebraic mode.

2.5 Symbols, numbers and strings

A `symbol` or `id` [2] in LISP is a unit which has a name, in the standard case beginning with a letter and followed by letters and digits. Special characters in a name or a leading digit have to be flagged by a preceding exclamation mark. Symbols are the same entities as in algebraic mode. Symbols are unique in the system. You may view a symbol as a data structure with four slots, the *name* with is always a string, the *value cell* to store assigned values, the *function cell* [3] to point to an assigned program structure and the *property cell* as anchor of the property list. Initially all cells except the name cell are marked empty.

In contrast to algebraic mode in LISP, only integers and floating point numbers (which don't exist in that form in algebraic mode) are considered as numbers. All other numeric quantities from algebraic mode such as rationals, rounded numbers, Gaussian integers are composite data structures encoded in the REDUCE domain structure (see below).

Just as in algebraic mode, a sequence of characters enclosed in string quotes is a string. Functions for manipulating symbols, numbers and strings are[4]:

[2] "id" is an abbreviation for "identifier".

[3] There are LISP systems which support only one cell for value and function – in such systems a symbol can represent only a variable or a function exclusively.

[4] Read the trailing *p* in names like "idp", "numberp" etc. as "predicate", e.g. "numberp" meaning "number predicate".

idp(q)	true if q is a symbol
numberp(q)	true if q is a number
stringp(q)	true if q is a string
liter(q)	true if q is a single alphabetic character symbol
digit(q)	true if a is a digit character symbol
intern(s)	make a symbol from a string
explode(a)	decompose a symbol/number/string into its characters
explode2(a)	explode, but without escape characters
compress(l)	make a symbol/number/string from a list of characters
gensym()	generate a new symbol

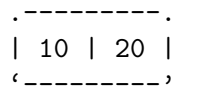
The functions `gensym` and `compress` make symbols which are not yet "internal": they are unique, but there may be other symbols with the same name. It makes debugging especially hard if you cannot access the symbol by its name of course. The function `intern` can be used to convert such a symbol into an internal one - then every reference to the symbol name guarantees access to the desired object.

2.6 Boolean values

There are two special symbols named *nil* and *t*. *nil* is used at the same time for the empty list and the Boolean value "false" . Any value other than *nil* is considered as "true" in conditional statements. In order to facilitate programming, the symbol *nil* has the preassigned constant value *nil* such that a quote here is superfluous. And for the same reason the symbol *t* evaluates to itself - this symbol can be used if the Boolean value "true" is needed explicitly, e.g. as a return value. But please keep in mind that any non-*nil* value can play this role. In contrast to algebraic mode the number zero does **not** represent the Boolean value "false".

2.6.1 Pairs, lists

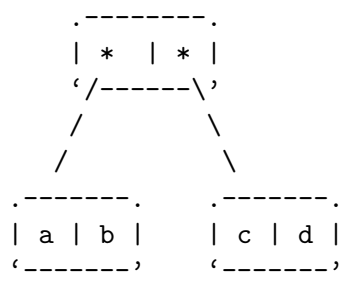
The central structure of LISP is the `pair`. The external representation of a pair of two objects is $(obj_1.obj_2)$. Here the infix dot is used as print symbol as well as infix constructor. It is often useful to use a double box for representing a pair graphically, e.g. for (10 . 20):



Functions:

o1.o2	(infix) construct pair ($o_1.o_2$)
cons(o1,o2)	same as 01.02
pairp(q)	true if q is a pair
car(q)	extract the first object from a pair
cdr(q)	extract the second object from a pair

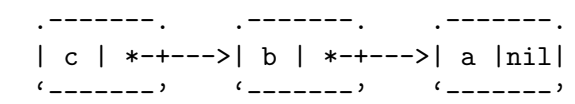
The pair is a very general construct - since its objects themselves can be pairs, arbitrary trees and data structures can be built. E.g. $((a.b).(c.d))$



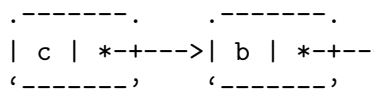
On the other hand, structures with many members lead to a very complicated representation if printed using the above notation. Therefore the `list` concept has been introduced which allows one to use a special class of pair trees in a simplified form:

- in a pair with second element *nil* the dot and the nil are omitted: $(A.nil) = (A)$.
- for a pair with another pair as second element one bracket pair and the dot are omitted: $(B.(A)) = (B\,A)$, $(C.(B\,A)) = (C\,B\,A)$.

So the `list` is a linear sequence of pairs, where the *car*s contain the "data" items, while the *cdr*s contain the reference to the successors. The last successor is *nil*, which signals the end of the linear list. For the graphical representation of linear lists horizontally aligned double boxes are useful, e.g. for $(C\,B\,A)$



or with omitting the final *nil*

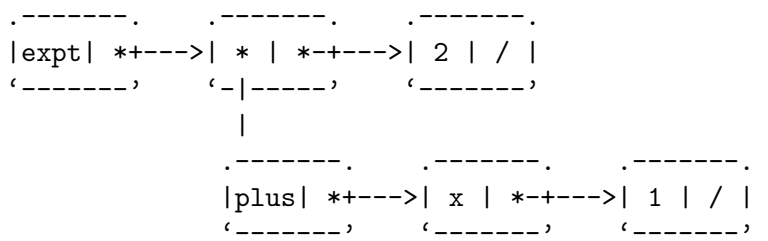


The `list notation` is much simpler than the `dot notation` - unfortunately the dots cannot be avoided completely because pairs with id's or numbers in the *cdr* part occur and they play an important role.

Lists can be nested; an example is the internal representation of algebraic forms (see below) which uses linear lists; e.g. $(x+1)^2$ would be represented by lists as

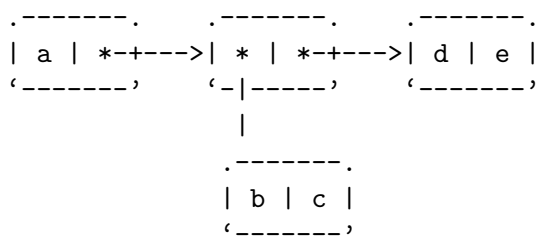
```
(expt (plus x 1) 2)
```

Here the top level list has three elements *expt*, $(plus\, x\, 1)$ and 2 where the second element itself is a linear list of length 3.



In this graph the `car`s contain the symbols or numbers or they point to substructures (downwards) while the `cdr`s point to the successors (to the right) until the end is reached.

As mentioned already there can be mixtures of both forms if one *cdr* is not *nil*: $(A(B.C)D.E)$ - here the second element of the list is a pair $(A.B)$, the third element is D and there is no fourth element - the list ends with a non *nil* symbol.



The empty list () is identical to the symbol `nil`. Note that the REDUCE `algebraic forms` are LISP lists where all lists are terminated by a *nil*.

Functions on lists:

{o1,o2,..on}	construct list $(o_1\ o_2 \cdots o_n)$
list(o1,o2,..on)	same as {o1,o2,..on}
o1.l1	(infix) put o_1 in front of list l_1
pairp(q)	true if q is a pair (or a list)
atom(q)	true if q is NOT a pair/list
null(q)	true if $q = nil$ (=empty list)
car(q)	extract the first object from a list
cdr(q)	extract the part of the list behind 1st element
nth(q,n)	extract the n-th element from list q
length(q)	count the number of elements of list q
reverse(q)	reverse a list q
append(a,b)	append b to the end of a copy of a
member(a,l)	test if a is equal to one member of l
memq(a,l)	test if a is identical one member of l
delete(a,l)	delete a from list l

Remarks:

- All of the above functions are non destructive: an input list remains as it has been before - the value of each access function is a reference to the corresponding part of the structure.

- The access functions can operate only if the desired elements are really there; e.g. if *nth* with 4 is applied to a 3 element list produces an error, and *car* or *cdr* applied to any symbol or number (including *nil*) will fail.

- Nested calls of *car* and *cdr* can be abbreviated by combining the a's and d's between c and r up to four inner a/d letters. E.g. $cadr(u) = car(cdr(u))$ - this is the direct access to the second element, *caddr* returns the third element and *cadddr* the fourth element.

- Although the functions *first*, *second*, *third*, *rest* known from algebraic mode operate in some REDUCE implementations in symbolic mode as synonyms of *car*, *cadr*, *caddr*, *cdr*, they should not be used here as this is the case in all REDUCE versions.

- The functions *member* and *memq* not only test the membership: if a is member of the list l, *member* (or *memq*) returns the (first) pair which contains a as its *car*. E.g. $memq('b,'(a\ b\ c))$ returns $(b\ c)$. This can be used either if you want to replace this item in the list, or if you

want to process the part after the search item. The function *memq* looks for a member of the list which is **eq** to the object, while *member* uses **equal** as test (see discussion of equality below). If applicable, *memq* is substantially faster than *member*.

- *Delete* produces a copy of the top level pair sequence leaving out the first occurrence of the search item: the original list is not changed, and a possible second occurence of the search item in the list is not removed. $delete('a,'(0\,a\,b\,a))$ will result in $(0\,b\,a)$ where the part $(b\,a)$ is the original tail of the input list while the pairs in front of the original first a are new.

Examples: let u be a variable with value $(A(B.C)NIL(D))$. Then

pairp(u)	=t
length(u)	=4
null(u)	=nil
car(u)	= A
cadr(u)	=(B.C)
caadr(u)	=B
cdadr(u)	=C
cdr(u)	=((B.C) NIL (D))
length cdr(u)	=3
cddr(u)	=(nil (D))
null cddr(u)	=nil
caddr(u)	=nil
null caddr(u)	=t
cdddr(u)	=((D))
cddddr(u)	=nil
null cddddr(u)	= t

All data which are not pairs (or lists) are classified as `atoms`: symbols, numbers, strings, vectors(see below) and, just as in real world, they are not very atomic – there are tools to crack them.

2.7 Programming with lists

There are several methods available to construct a list:

```
u := nil;
u :=1 . u;
u :=2 . u;
```

Starting with an empty list, elements are pushed on its front one after the other, here giving (2 1). Please note the blank between the number and the dot: this is necessary as otherwise the dot would have been parsed as part of the number representing a floating point value.

```
u := {9,10};
u := 1 .u;
u := 2 .u;
```

The same, here not starting with the empty list giving (2 1 9 10). The `for statement` has special forms to handle lists:

- `for each` x `in` $l \cdots$ performs an action elementwise for the elements in list l; the elements are accessible by the automatically declared local variable x one after the other.
- the iterator actions `collect` and `join` allow you to construct lists:
  - `collect` generates a list where each iteration step produces exactly one element,
  - `join` expects that each iteration step produces a complete list (or *nil*) and joins them to one long list.

Examples: let e.g. l be a list of numbers $(1\ -2\ 3\ -4\ 5\ -6)$:

```
m:=for each x in l sum x;
```

m will be the sum of the elements,

```
s:=for each x in l collect x*x;
```

s is computed as list with the numbers from x squared,

```
p:=for each x in l join
    if x>0 then {x} else nil;
```

in this loop the positive numbers produce one element lists, while the negative numbers are mapped to nil; joined together the result is a list with only the positive elements,

```
r:=for i:=1:10 collect -r;
```

here the result is the list with numbers $(-1\ -2\ -3 \cdots -10)$. The lists produced by *collect* and *join* are in "correct" order. **Warning**: In symbolic mode *join* links the lists **in place** for maximum speed [5]: The last *CDR* pointer of the first list is modified to point to the head of the second list , and so on. This is no problem if the joined objects are freshly built structures, or if there are no other references to them. If the in – place operation is dangerous for your application, you must copy the structure before, e.g. by a call to *append* with *NIL* as second argument:

```
r:=for i:=1:10 join append(my_extract(u,r),nil);
```

In this example, the top level chain of the results of *my_extract* is copied such that the original structure is not modified when joining.

Another very common style for list operations is to use iterative or recursive programs. The following programs each sum the elements of the list:

```
symbolic procedure sum_1(l);
 begin  integer n;
   while l do <<n:=n+car l; l:=cdr l>>;
   return n;
 end;
```

This program picks in each step of the the while loop one element from the list and sets the variable l to the successor position; the loop terminates as soon as the successor *nil* is found .

```
symbolic procedure sum_2(l);
  if null l then 0 else car l + sum_2(cdr l);
```

This program uses recursion for the same task. The program will go down the list until *nil* is found, then set the sum to 0 and when coming back from the recursion add the elements to the sum so far.

```
symbolic procedure sum_3(l); sum_31(l,0); % initializing
symbolic procedure sum_31(l,n);
  if null l then n else sum_31(cdr l,car l+n);
```

[5] In algebraic mode *join* operates different: the operatend are copied before linking them to one result list

The third example is a bit tricky: it initializes the sum to 0, and when stepping down in the recursion the sum is updated and passed to the next recursion level as a parameter.

It is also very common to produce lists in recursive style, e.g. inverting the signs in a list of numbers could be written as

```
symbolic procedure invertsgn(l);
   if null l then nil else -car l . invertsgn(cdr l);
```

and with most LISP compilers this program would as efficient as the corresponding loop

```
for each x in l collect -x;
```

but produce less code. Note that as with *for each - collect*, the elements in the resulting list will be in the original order. The recursive programming style is typical for LISP and it is used for hundreds of REDUCE routines.

2.8 In-place operations

All list construction described so far except **join** is of a copying nature: every construction by the dot or the curly brackets always creates fresh pairs and an old list is untouched. However, there are operations to modify elements or the structure of an existing list in place. These should be used only with greatest care: if there is another reference to the list, its structure will be modified too, a possibly unwanted side effect. The functions are[6]:

car l := u	replace first element of l by u
rplaca(l,u)	old form for car l := u
cdr l := u	replace cdr part of l by u
rplacd(l,u)	old form for cdr l := u
nth(l,j) := u	replace the j-th element of l by u
nconc(a,b)	insert b as cdr in the last pair of a
reversip(l)	invert the sequence of l using the same pairs again

2.9 Equal tests: Equality vs. Identity

There are two classes of equality in LISP, *eq* and *equal*, in REDUCE written as the infix operators `eq`, and equal sign = respectively. As symbols are unique, both are equivalent for them: if symbols are `equal`, they are `eq`.

[6]Read "reversip" as "reverse-in-place" and "nconc" as "concatenate"

$u :=' a; v :=' a;\ u\, eq\, v \Rightarrow t$

here the variables u and v represent the same symbol and consequently they are `eq`.

For lists, things are different. Here we must distinguish between references to pairs which contain the same contents ("equal") or references to the same identical pair instance("`eq`"). A pair is `eq` to itself only, but it is `equal` to a different pair if their *car*s and *cdr*s are `equal` - this is a recursive definition. Example:

$u := \{1,2\}; v := \{1,2\}; w := u;$

$u = v \Rightarrow t$,$u = w \Rightarrow t$

$u\ eq\ v \Rightarrow nil,\ u\ eq\ w \Rightarrow t$

Here u, v and w have as values pairs with same *car*s and `equal` *cdr*s, but only u and w refer to the identical pair because for v fresh pairs have been built. In some points REDUCE relies on identity of some structures, e.g. for composite kernels of `standard forms`.

2.10 Properties, Flags

Another important concept of LISP is the ability to assign `properties` and `flags` to each symbol. A `property` is a symbol (a "name") together with a value. Both, property name and property value are completely free. Properties are set by the function `put` and read by the function `get`.

```
put('hugo,'evaluator,'compute_hugo);
  . . .
get('hugo,'evaluator) => compute_hugo
get('otto,'evaluator) => nil
```

A property-value pair remains assigned for a symbol until the value is replaced by a different value or until the property is removed by `remprop`.

A `flag` is similar to a property, but it represents only the values "yes" or "no" (= flag is present or absent). The functions for setting and testing flags are `flag` and `flagp`, `remflag` for removal.

As most other LISP programs, REDUCE makes extensive use of properties. E.g. the algebraic properties of an operator are encoded in that way. So the symbol *plus* has among others the REDUCE properties $prtch = +$ (the print character), $infix = 26$[7] (the precedence among the infix operators), $simpfn = simpplus$ (the name of the routine responsible for its simplification) and the flags *symmetric*, *realvalued*, *nary* and some more. These

[7]This number may change if additional operators of lower precedence are introduced

allow a data driven programming model which has been used in the LISP world for decades already: if the simplifier operates on an expression, it asks the leading operator for its simplification function; if there is one (and there will be one for all operators), this function is invoked to do the job.

There are various fields where this type of programming is used inside REDUCE, e.g. the complete domain arithmetic has been implemented in that way: any non-integer domain element is encoded as a list with a leading symbol (the domain mode) which contains all knowledge about arithmetic, conversion, printing etc. for this domain as properties and flags.

If your REDUCE system is based on PSL, you can inspect a property list by calling the function *prop* which has one argument (the symbol to be inspected) and which returns the properties and flags as list. E.g.

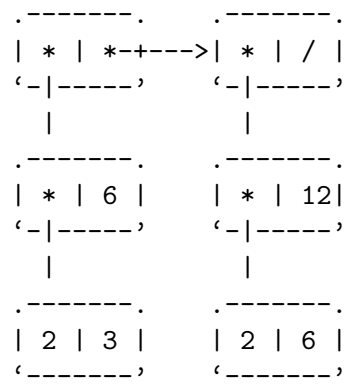
```
a := mat((1,2),(3,4));
prop 'a;
prop 'mat;
prop 'matrix;
```

2.11 Alists

Another frequently used LISP structure is the `association list` (`alist`), which is an associative memory: a list where each element is a pair with a search item and a value item. The function `assoc` picks the pair for a given search item out of the `alist`. For example constructing a multiplier with memory:

```
fluid '(product_database*);
symbolic procedure multiply(u,v);
 begin scalar w,r;
  w:=u.v;
  r:=assoc(w,product_database*);
  if r then return cdr r; % found
  r:=u*v;
  product_database*:=(w.r).product_database*;
  return r;
end;
```

here the pair $u.v$ is used as search item. If an entry is found in the database, its value part is returned. Otherwise the product is computed and the new entry is added to the database. After calling this function with arguments 2 and 3, and then with 2 and 6 the structure would be



Here each element in the top level list points to one generalized name-value pair, where the name itself is a structure (here a pair of two numbers). REDUCE uses association lists for many different purposes; e.g. if an expression is simplified, the expression and its simplified variant are stored in an association list such that for a second access to the same expression its simplified form is immediately accessible.

2.12 Vectors

Lists enable access of items of a linear collection by position number using the function `nth`; however, this access is inefficient if used frequently because for every access the sequence of pairs has to be traversed until the desired object is reached. An alternative structure for direct access by element number in symbolic mode is the `vector`. The first element in a vector has the index 0.

v:=mkvect(n)	create a vector of length n
putv(v,m,u)	put value u into mth slot of vector v
getv(v,m)	extract element m from vector v
upbv(v)	returns upper bound for v

Note that `putv` is a "destructive" operation.

3 Algebraic data formats and evaluation

3.1 Prefix forms

The basic format for representing mathematical formulae is the `algebraic form` : an expression is internally encoded as a list with `prefix` operators:

- the first element of a list is an operator,
- the following elements of a list are the arguments which can be symbols (unknowns), numbers or `algebraic forms` in prefix notation.

Examples:
$x + 1 \Rightarrow (PLUS\ X\ 1)$
$x + y + 1 \Rightarrow (PLUS\ X\ Y\ 1)$
$x + y * z + 1 \Rightarrow (PLUS\ X\ (TIMES\ Y\ Z)\ 1)$
$x\hat{}(y + 1) \Rightarrow (EXPT\ X\ (PLUS\ Y\ 1))$

Algebraic forms are used for many purposes, for data input, for transferring data between various system components and for output. To get a feel as to how algebraic forms look like, you can make REDUCE display them for you, e.g. by the following sequence:

```
u:=(x+y)^3/(log z-sqrt 2);
symbolic reval 'u;
```

The first statement assigns an algebraic expression to a variable as usual in algebraic mode, and the second statement forces REDUCE to print the algebraic form corresponding to this expression in symbolic mode.

3.2 *SQ

REDUCE uses internally a different representation for algebraic expressions, the `standard quotient`(`SQ`, see below). As conversion between algebraic forms and standard quotients is expensive, REDUCE tries to keep the data as long in `SQ` form as possible. For this purpose there is a special variant of the algebraic form $(*SQ\ u\ ind)$: here the internal operator $*SQ$ indicates that the parameter u is not an algebraic form but a standard quotient. If such expression is evaluated, REDUCE detects immediately that it will find the standard quotient already as the second element of the list and no more conversion is necessary.

However, if since creation of this form some values in the system have been changed, the form u might be no longer the "correct" standard quotient; e.g. one of its unknowns might have a value now. For this purpose the third element *ind* in the list has been introduced: As long as the form u is the correct one, it will be T. As soon as a significant value in the system changes which might affect the validity of u, REDUCE switches all *ind* elements in the $*SQ$ forms swimming around to *nil*. So the evaluator decides from the second parameter whether to use u unchanged or to re-evaluate.

By the way, the global replacement of the second parameter in the $*SQ$ forms is a good example of an in-place list operation with a wanted side effect: REDUCE maintains a unique list $*sqvar* = (t)$ and all $*SQ$ forms are built with this list as last pair. If the environment changes it is sufficient to replace this single t in $*sqvar*$ by nil and all $*SQ$ forms built with the actual $*sqvar*$-tail inherit this automatically switching from t to nil; REDUCE then creates a fresh list $*sqvar* = (t)$ to be used for the next generation of $*SQ$ forms.

3.3 Composite forms

3.3.1 Lists, equations

For algebraic mode, lists are represented by the prefix operator $LIST$. From the symbolic mode standpoint it may appear a bit curious that one has to insert the symbol $LIST$ into a list in order to make it a correct algebraic object, but this is necessary in the prefix algebraic context. E.g. try in algebraic mode

```
u:={a,b,{c,d},e};
lisp reval 'u;
```

you will see the result $(LIST\ A\ B\ (LIST\ C\ D)\ E)$.

Correspondingly `equations` are represented by the prefix operator $EQUAL$ followed by the right-hand and the left-hand sides as parameters; e.g. $x = sqrt(2)$ is represented as $(EQUAL\ X\ (SQRT\ 2))$;

The result of *solve (x∧ 2=2,x);* is an instructive mixture of all these forms:

```
(LIST (EQUAL X (!*SQ (((((EXPT 2 (QUOTIENT 1 2)). 1). -1)). 1) T))
      (EQUAL X (!*SQ (((((EXPT 2 (QUOTIENT 1 2)). 1).  1)). 1) T)))
```

this form is a list with two equations and the equations have $*SQ$ forms as right-hand sides.

3.3.2 Matrices

A `matrix` too is represented internally by a prefix form with operator MAT followed by the rows; each row is a list containing the elements as algebraic forms. The elements can be accessed easily using the access function nth. Example: the matrix $mat((1,2),(3,4))$ is encoded as $(MAT\ (1\ 2)\ (3\ 4))$.

The form $nth(nth(m,3),2)$ gives access to the element (2,2). Be careful if you use this form for an assignment: $nth(nth(m,3),2) := 17$; is an in-place operation. You can create a matrix easily by a symbolic program, e.g. a n*n unit matrix

```
symbolic procedure my_unit_matrix(n);
  'mat . for i:=1:n collect
    for j:=1:n collect if i=j then 1 else 0;
```

3.4 Algebraic evaluator: reval, aeval, aeval*

The main entries to the algebraic evaluator are the symbolic mode functions `reval` and `aeval`. Both have as input an algebraic expression and return the fully evaluated result. Reval always returns an algebraic form in full prefix operator form, while aeval tries to keep the result in $*SQ$-form whenever possible. Almost all algebraic functionality of REDUCE can be invoked by calling these procedures, e.g.

```
reval '(solve (plus (expt x 2) 3) x);
```

If you turn on the switch `defn` in algebraic mode you will see that REDUCE translates most statements into calls for `aeval`.

Reval and aeval are also adequate tools for accessing the values of algebraic quantities and for evaluation of unevaluated procedure parameters; they do the complete job for you.

However, there is one exception: the left-hand side of an equation in general is not evaluated by `reval` and `aeval`. E.g.

```
l:= a=b;
a:=17;b:=4;
symbolic reval 'l;
```

produces the output ($EQUAL\ A\ 4$). If you write a program for equations you either should evaluate the left-hand sides yourself by calling `reval` or rebind the internal switch `*evallhseqp` [8] locally(!) by t - then REDUCE will evaluate both sides of an equation.

REDUCE maintains an internal buffer `alglist*` where all evaluation results are stored such that a repeated request for the same evaluation can be

[8] Read "evallhseqp" as "evaluate left-hand side of equations predicate".

satisfied by a simple and fast table lookup[9]. As a side effect, `aeval` resets `alglist*` empty before starting a computation[10]. There is a second function `aeval*`: this function performs the same operation, but in contrast to `aeval` it does not reset `alglist*` and can profit from previous results. The REDUCE uses both, `aeval` and `aeval*` depending of the specific environment of a translated statement.

4 Standard Forms, Standard Quotients

4.1 Introduction

On of the most important tasks performed by a computer algebra system is the reduction of an algebraic expression to a canonical form. A reduction process is canonical if it guarantees that two expressions are transformed to the same form if and only if they are algebraically identical. Unfortunately canonical forms are not known for all algebraic objects, e.g. for surds. The canonical form of REDUCE is the `standard quotient` ("`SQ`") which is a quotient of two `standard forms` ("`SF`"). An `SF` is a polynomial in internal representation and an `SQ` is a quotient of two polynomials, a rational function. At the same time `SQ` and `SF` are the forms used internally for most arithmetic.

In this section the structures of `SF` and `SQ` are described, followed by a list of operations on these. For programming and debugging in symbolic mode it is often necessary to decipher an expression of this type. However, the knowledge about this part of the internal REDUCE structure should never be used to access `SF` of `SQ` directly by LISP primitives. Instead use only "official" primitives supplied by REDUCE.

4.2 Coefficients

The coefficients in `SF` and `SQ` are the "numbers" from various algebraic contexts. They are called `domain` elements and uniformly represented either as integer numbers or pairs where the `car` part contains an identifier describing

[9]The use of `alglist*` is controlled by an internal switch `*uncached`: if this variable has a non-nil value, the caching is suppressed.

[10]Of course, there are many more places where `alglist*` needs to be reset; e.g. any assignment causes `alglist*` to be reset because the assigned value might make one of the stored values invalid.

the domain and its arithmetic by its `properties`; the `cdr` part describes the value.

1. Integers: integer numbers $\in \mathbf{Z}$ represent integer coefficients directly,
2. Gaussian integers: complex numbers with integer real and imaginary part *(:gi: re . im)* with $re, im \in \mathbf{Z}$,
3. Rational numbers: quotients of integers *(:rn: nu . de)* with $nu, de \in \mathbf{Z}$,
4. Modular numbers: numbers modulo the current prime *(:mod: . j)* where $0 \leq j < currentmodulus*$ [11].
5. Rounded numbers: floating point numbers extended in precision and magnitude *(:rd: . fp)* with LISP floating point number fp or *(:rd: mant . exp)* with integers *mant* and *exp* representing an extended floating point value,
6. Rounded complex numbers: numbers with rounded real and imaginary parts *(:gr: fre . fim)* with LISP floating point numbers fre and fim or *(:gr: (rmant . rexp) . (imant . iexp))* with extended real and imaginary parts.

The neutral elements *zero* and *one* for addition and multiplication respectively are identical for all domains: *zero* is represented as *nil* and *one* by the integer number 1.

The list of domains is open ended: if a new coefficient domain is required for an application field, this can be created and linked to the system at any time. The necessary structure is described in the source file for the module *dmodeop*.

The tag of the currently active domain is stored in the value of the variable `dmode*`. *nil* represents the default domain (integers). This variable should not be modified directly - instead use the function `setdmode`:

```
setdmode(DOM:domain,MODE:bool);
```

where *domain* is a domain name (e.g. $'rational$) and the Boolean second parameter tells whether to activate (t) or to deactivate (nil) the domain.

[11] The upper half of the modular numbers are printed as negative numbers if the switch `balanced_mod` is on; hovever, that does not affect the internal representation which uses always integers ≥ 0

Note that *setdmode* does not set `on` the corresponding switches. You *must* do that yourself. E.g. for a proper activation and deactivation of complex rounded call

```
!*rounded := t;
setdmode('rounded,t);
!*complex := t;
setdmode('complex,t);
.... % do your computation
setdmode('complex,nil);
!*complex := nil;
setdmode('rounded,nil);
!*rounded := nil;
```

In order to convert the domain tag into the domain name use the property `dname`, e.g.

```
get(dmode!*,'dname);
```

will result in the name of the current domain.

4.3 Generic arithmetic with domain elements

There is a complete set of arithmetic functions for performing arithmetic operations with domain elements:

!:zerop(a)	test $a = 0$
!:plus(a,b)	$a + b$
!:difference(a,b)	$a - b$
!:minus(a)	$-a$
!:minusp(a)	test $a < 0$
!:times(a,b)	$a * b$
!:quotient(a,b)	a/b
!:recip(a)	$1/b$
!:expt(a,n)	a^n
!:gcd(a,b)	*greatest common divisor of* (a, b)

4.4 Kernels, kernel order

`Kernels` are the "unknowns" in a polynomial. A `kernel` can be an

- identifier: e.g. x, $alpha$, represented as LISP symbols X, $ALPHA$,

- operator form: e.g. $sin(alpha)$,$sqrt(x+1)$,$bessel(j,x)$, represented as algebraic expression $(SINALPHA)$, $(EXPT\ (PLUS\ X\ 1)\ (QUOTIENT\ 1\ 2))$, $(BESSEL\ J\ X)$.

- A standard form, e.g. $(((A\ .\ 1)\ .\ 1)\ .\ 1)$ This option is used only if the default mode of full expantions is turned off (see below: non-expanded standard form).

It is essential for the operation of REDUCE that kernels are unique. This is no problem for id's as these are unique by LISP definition. For composite kernels REDUCE ensures that these are made unique before they enter a `standard form`. *Never construct a composite kernel by hand.* Instead use the function `*a2k` for converting an algebraic expression into a kernel or use *simp* (see below). On the other hand, the uniqueness of kernels is of big advantage for comparisons: the equality can be tested with *eq* instead of = and faster *memq* and *atsoc* can be used instead of *member* and *assoc*.

The set of all kernels is totally ordered by the `kernel order`. REDUCE has by default some ordering which is roughly based on lexicographic comparison. The ordering can be changed internally by calling `setkorder` (corresponding to the algebraic statement `korder`) with a list of kernels as argument. These kernels precede all other kernels in the given sequence .`Setkorder` returns as result the previous order and it is a good policy to restore the old order after you have changed it:

```
begin scalar oldorder, ....;
  oldorder:=setkorder '(x y z);
   ....
  setkorder oldorder;
   ....
end;
```

The current kernel order list is the value of the variable `kord*`. The value *nil* signals that the default ordering is active.

To check the system order of two algebraic expressions use the function `ordp` with two arguments, which returns true if the first argument is lower in the ordering sequence as the second one. This check does not consider the actual `kord*`. For kernels there is an extended order check function `ordop` which does consider the actual `kord*`. However, you need to call these functions only rarely as the algebraic engine of REDUCE maintains a proper order in any context.

4.5 Standard form structure

A `standard form` is a polynomial in internal recursive representation where the kernel order defines the recursive structure. E.g. with kernel order (x y) the polynomial $x^2y + x^2 + 2xy + y^2 + x + 3$ is represented as a polynomial in x with coefficients which are polynomials in y and integer coefficients: $x^2 * (y * 1 + 1) + x * (y * 2 + 1) + (y^2 * 1 + 3)$; for better correspondence with the internal representation here the integer coefficients are in the trailing position and the trivial coefficients 1 are included. A standard form is

- <domain element>
- <mvar> .** <ldeg> .* <lc> .+ <red>, internally represented as *(((mvar.ldeg).lc).red)*

with the components

- *mvar*: the leading kernel,
- *ldeg*: an integer representing the power of *mvar* in the leading term,
- *lc*: the leading coefficient is itself a standard form, not containing *mvar* any more,
- *red*: the reductum too is a standard form, containing *mvar* at most in powers below *ldeg*.

Note that any standard form ends with a domain element which is *nil* if there is no constant term. E.g. the above polynomial will be represented internally by

```
(((X .2) ((Y. 1). 1). 1) ((X. 1) ((Y. 1). 2)) ((Y. 2). 1). 3)
```

with the components

- mvar: `X`
- ldeg: `2`
- lc: `((Y. 1). 1). 1)` $= (y + 1)$
- red: `(((X. 1) ((Y. 1). 2)) ((Y. 2). 1). 3)` $= xy^2 + 3$.

4.6 Operations on standard forms

Arithmetic with standard forms is performed by the functions[12]

null(f)	test $f = 0$
minusf(f)	test $f < 0$ (formally)
domainp(f)	test if f is a domain element (e.g. number)
addf(f1,f2)	f1 + f2
negf(f)	-f
addf(f1,negf f2)	f1-f2
multf(f1,f2)	f1*f2
mksp**(f1,n)	compute f1**n
quotf(f1,f2)	$f1/f2$ if divisible, else *nil*
quotfx(f1,f2)	$f1/f2$ divisiblity assumend, no check
qremf(f1,f2)	pair (quotient.remainder)

As standard forms are a superset of domain elements these functions also can be used for arithmetic with domain elements or for combining standard forms with integer numbers. They can even be used to compute with integers as long as these are regarded as "degenerate" polynomials.

For access of the components of a standard form there are functions

mvar(f)	extract leading kernel
ldeg(f)	extract leading degree
lc(f)	extract leading coefficient
red(f)	extract reductum f

These functions can be applied only if the form is not a domain element. The following example presents a typical function evaluating a standard form; it looks for the maximum power of a specified kernel y in a standard form f:

```
symbolic procedure my_maxpow(f,y);
  if domainp f then 0 else
  if mvar f eq y then ldeg f else
  max(my_maxpow(lc f,d),my_maxpow(red f,d));
```

[12]Be careful not to intermix the names "minusf" and "negf". And take into account that `minusf` will return a positive result only if the value is definitely known to be less that 0.

This function has a double recursion which is typically for functions traversing standard forms: if the top node is not trivial (`domainp`) or not a power of y the procedure branches to the coefficient form and to the reductum form - both can contain references to the kernel y. Kernels are unique in the system and therefore the equality test is performed by `eq`.

For the initial construction of a standard form a safe and efficient way is to build an algebraic expression first and convert it using *simp* (see below) or to use the arithmetic functions above too. For elementary conversions use

*k2f(k)	convert unique kernel k to SF
numr simp(a)	convert algebraic expression a to `SF`
prepf(f)	convert a `SF` f to an algebraic expression

Note that integers are already an `SF` and so need not be converted.

Although the are infix macros .** , .* , .+ to construct standard forms, they should be avoided unless you are completely sure what you are doing: they construct without any consistency checks, and wrong structures may lead to inconsistent results; such cases are difficult to debug.

4.7 Reordering

If you change the `kernel order` it is necessary to reorder `standard forms` if they are processed under the new order by calling

reorder(f)	reorders f to the current order

Please ensure that you never use a standard form which does not correspond to the actual order - otherwise very strange results can occur. Freshly created standard forms will always have the current order.

4.8 Standard Quotients

A `standard quotient` is a pair of `standard forms` (*numr ./ denr*), in LISP represented as (*numr . denr*). If the denominator is trivial, the standard form 1 is used. The constructor, selector and conversion functions are

numr(q)	select the numerator part
denr(q)	select the denominator part
f2q(f)	convert a standard form f to `SQ`
simp(a)	convert algebraic form a to `SQ`
prepsq(q)	convert `SQ` to algebraic form

Arithmetic with standard quotients is performed by the functions

addsq(q1,q2)	$q1 + q2$
negsq(q)	-q
subtrsq(q1,q2)	q1-q2
multsq(q1,q2)	q1*q2
quotsq(q1,q2)	$q1/q2$

Note that there is no zero test for standard quotients; instead use the zero test for the numerator $null(numr(q))$.

When should you use `standard quotients` and when `standard forms`? If you are sure that no denominator other than one can occur you should use `standard forms`, otherwise `standard quotients` are the only possible choice. You can of course mix arithmetic with both, however that strategy is not recommended unless you keep track carefully; debugging is hard. Arithmetic functions have been coded for maximum efficiency: none of them test the validity of their arguments.

4.9 Substitutions

For `substituting` kernels in `standard forms` or `standard quotients` by other kernels or arbitrary algebraic expressions the following functions are available :

subf(f,a)	substitute a in `SF` f
subsq(q,a)	substitute a in `SQ` q

where a is a list of pairs, each with a kernel to be replaced in the *car* and its replacement as algebraic form in the *cdr* [13], e.g.

```
uhu := simp('(expt(plus x y) 10));
otto:= subsq(uhu, '((x . 5) (y . (plus a b))));
```

Here x is replaced by 5 while y is replaced by $a + b$.

Note that

[13] The second argument of `subf` and `subsq` is a typical association list structure.

- both `subf` and `subsq` return a `standard quotient` (the substitution might have caused a non trivial denominator),
- `subf` and `subsq` also introduce substitutions which have been coded in rules - so it makes sense to call these routines with *nil* as the second argument if you have introduced a new rule and some standard forms or standard quotients need to be re simplified under these.

4.10 Useful facilities for standard forms

4.10.1 Kernel list

The function `kernels` extracts all kernels from a standard form and returns them as list.

```
kernels numr simp '(plus (expt x 3)(expt y 4));
```

will result in $(x\, y)$.

4.10.2 Greatest common divisor

The greatest common divisor of two polynomials is computed by the function `*gcdf`. It takes as arguments two standard forms and returns a standard form or a 1 if there is no non trivial common divisor.

4.10.3 Factorization

The function `fctrf` is the interface for calling the `factorizer`. It takes a standard form as parameter and returns a list which contains

- as first element the content of the polynomial - that is the gcd of the coefficients or the "common numeric factor",
- as second and following elements the factors in the form (`SF`.m) where `SF` is the factor as standard form and m is its multiplicity.

So calling $fctrf$ for $2x^2 - 2$ will result in

```
(2 ((((X . 1) . 1) . 1) . 1) ((((X . 1) . 1) . -1) . 1))
```

Here you see the common domain factor 2 and the two factors $x+1$ and $x-1$ as standard forms paired with their multiplicity 1.

4.11 Variant: non-expanded standard form

The standard form structure described so far is used by default for the fully expanded expressions with common denominator. By setting `on factor`, `off exp`, or `off mcd` the slots *mvar* and *ldeg* may be used in an exended style:

- *ldeg* can be a negative number to represent a reciprocal factor,

```
off mcd; (x*a+y*b)/(a*b);

 -1      -1
a  *y + b  *x

lisp ws;
(!*sq ((((a . -1) ((y . 1) . 1)) ((b . -1) ((x . 1) . 1))) . 1) t)
```

  Here the numerator form of the standard quotient is a sum where each term is a product of one kernel with a positive and one kernel with a neagative exponent.

- *mvar* may be a standard form to represent a factored polynomial,

```
on factor; (x^2+y)^2*(y+1);

  2     2
(x  + y) *(y + 1)

lisp ws;

(!*sq (((((((x . 2) . 1) ((y . 1) . 1)) . 2) (((((y . 1) . 1) . 1)
   . 1) . 1))) . 1) t)
```

  Here the numerator is a standard form in factored form where the inner polynomials $(x+y)$ and $(y+1)$ are used as *mvar*s, the first one with the exponent 2 and the second one with exponent 1.

Special functions to work with composite standard forms:

sfp(m)	test if m is a standard form (T) or a kernes(NIL)
expnd(f)	expand a standard form f

To distinguish between factored and expanded standard forms, use the predicated `sfp`: sfp applied to *mvar* of a standard form returns T if the main variable slot is occupied by a nested standard form. To expand a factored standard form you may use the funtion `expnd`; however, you need to turn the switch *exp* on during that call, e.g.

```
u:=expnd u where !*exp=t;
```

Don't build non–expanded standard forms yourself – otherwise you run the risk to produce objects with a bad structure (e.g. a wrong term order) resulting in wrong computational results. You better rely on the standard routines for manipulating these objects – as soon as $*exp$ is off and $*factor$ is on they produce the product forms anyway.

5 Communication between algebraic and symbolic modes

The main aim of programs written in symbolic mode is to implement the evaluation steps to be invoked from algebraic mode (top level REDUCE). This section describes how to link symbolic routines to algebraic mode and how to exchange data between the modes.

5.1 Algebraic variables

A variable which has been declared `share` by the REDUCE *share* command is accessible from both modes in the same style and it represents the same algebraic value. A share variable is the easiest way to transfer data between the modes. The only restriction is that a symbolic program should assign only a legal algebraic form to it - otherwise a later access in algebraic mode can lead to problems. Example:

```
share hugo;
hugo :=(x+y)**2$
hugo;
symbolic;
hugo := reval {'sqrt,hugo};
algebraic;
hugo;
```

Variables which have not been declared *share* have different values in symbolic and algebraic mode. Nevertheless a symbolic mode program has access to the algebraic value of a symbol:

- `reval(x)`: if the value of x is a symbol or if the parameter of *reval* is a directly quoted symbol the function returns the algebraic value associated with this symbol, e.g. $reval('y)$,
- `setk(x,y)` sets the algebraic value of the expression x which then must be a symbol (or more general: a `kernel`) to y, e.g. $setk('z, reval'(plus\, u\, 17))$ [14]

Of course a clever LISP programmer easily sees how REDUCE organizes the assignment and storage of values to algebraic variables in property lists, and he might be attempted to use this knowledge for "easier" access; please resist: otherwise your program might not run in a future version of REDUCE.

5.2 Calling symbolic routines from algebraic mode

REDUCE offers various ways to use symbolic routines in algebraic mode such that they appear on the algebraic level as genuine parts of REDUCE. These mainly differ in their strategy of evaluating parameters (and results).

5.2.1 Common protocol

Some general protocol rules apply for REDUCE symbolic mode routines:

- A routine should check its parameters carefully; in the case of illegal parameters the REDUCE error handlers should be invoked (see below).
- If a routine cannot process its parameters for mathematical reasons it best returns the expression which had caused the call unchanged.
- Evaluators should operate silently; events such as messages needed for testing and program development should be carried out in dependency of a switch.
- A routine should return its result as a value rather than printing it; for an isolated call in interactive mode a printed result would be sufficient, but for following evaluation steps the availability of an algebraic value is essential.

[14] Read "setk" as "set kernel value", the REDUCE equivalent to the Standard LISP assignment function "setq".

5.2.2 Symbolic operator

The simplest way to link a symbolic procedure to algebraic mode is the `symbolic operator` declaration. In that case the routine can be called by its proper name in algebraic mode with REDUCE first evaluating its parameters to fully simplified algebraic forms. Upon return the result should also be an algebraic form (or NIL). Example:

```
symbolic procedure my_plus(a,b);
  begin scalar r;
     print a; print b;
     r := reval{'plus,a,b};
     print r;
     return r;
  end;
symbolic operator my_plus;
```

This routine receives two algebraic expressions and computes their sum by calling the algebraic evaluator. The calls the the LISP function `print` have been inserted here and in the following examples to show you the forms passed to the routine.

5.2.3 Polyfn

If the symbolic routine is specialized for computations with pure polynomials it can be declared `polyfn`. REDUCE will evaluate the arguments for such a routine to `standard forms` and expects that the result also will have that form. Example:

```
symbolic procedure poly_plus(a,b);
  begin scalar r;
     print a; print b;
     r := addf(a,b);
     print r;
     return r;
  end;
put('poly_plus,'polyfn,'poly_plus);
```

This routine also adds its arguments but it is specialized for polynomials. If called with a non-polynomial form an error message will be generated. In

the `put` statement the first argument is the algebraic operator name and the third one is the name of the associated evaluator function. These may differ.

5.2.4 Psopfn

The most general type of function is that of a `psopfn`. REDUCE will not evaluate the parameters for such a routine, instead it passes the unevaluated list of parameters to the function. The routine has to evaluate its parameters itself (of course using the services of *reval* and friends). So a `psopfn` can handle variable numbers of parameters. The result must be an algebraic expression or *nil*. Example:

```
symbolic procedure multi_plus0 u;
  begin scalar r;
     r:=0;
     for each x in u do
     <<x:=reval x; print x;
       r:=reval{'plus,r,x}
     >>;
     print r;
     return r;
  end;

put('multi_plus,'psopfn,'multi_plus0);
```

This routine can be called with an arbitrary number of parameters; it will evaluate them one after the other, add them by calling *reval* and return the sum as result. Note that the name used in algebraic mode is *multi_plus* which is different from the name of the symbolic routine doing the work. This is necessary because otherwise the argument count check of REDUCE would complain as soon as the routine is called with more than one argument.

In the next example a `psopfn` implements an operator with a fixed number of arguments; it expects two numbers as arguments and performs an extensive checking as any such routine should do; again the name of the routine and the algebraic mode operator are different:

```
symbolic procedure bin_plus0 u;
 begin scalar p1,p2,r;
```

```
  if length u neq 2 then rederr "illegal number of arguments";
  p1:=reval car u; p2:=reval cadr u;
  if not numberp p1 then typerr(p1,"number");
  if not numberp p2 then typerr(p2,"number");
  r:=p1+p2;
  return r;
end;

put('bin_plus,'psopfn,'bin_plus0);
```

The functions `typerr` and `rederr` are explained in the section *error management*.

5.2.5 Simpfn

When you declare a new algebraic operator and want to assign a special evaluation mode for it which cannot be written in algebraic mode (e.g. as a rule set) you can create a simplifier which has the task to convert each operator expression to a `standard quotient`. A simplifier function is linked to the operator by the property `simpfn` and has one argument which is the unevaluated parameter list of the operator expression. It will be called every time the operator appears in an expression. It must return a standard quotient. Example:

```
algebraic operator op_plus;

symbolic procedure simp_op_plus u;
  begin scalar r;
     r:=simp 0;
     for each x in u do
     <<x:=simp x; print x;
       r:=addsq(r,x)
     >>;
     return print r;
  end;

put('op_plus,'simpfn,'simp_op_plus);
```

In many cases the simpfn is the method of choice as it is best integrated into the REDUCE evaluation process: its results can immediately be used

for subsequent calculations while algebraic form results need to be converted to standard quotients before combining them with other formulae.

Note that a simpfn which cannot simplify its argument must nevertheless return a `standard quotient`, then with the unevaluable form as kernel. E.g. a function for supporting the algebraic evaluation of the operator "<":

```
algebraic operator <;

symbolic procedure simp_lessp u;
  begin scalar a1,a2,d;
     a1:=simp car u; a2:=simp cadr u;
     d:=subtrsq(a1,a2);
     if domainp denr d and domainp numr d then
       return if !:minusp numr d then simp 1 else simp 0;
     return mksq({'lessp,prepsq a1,prepsq a2},1);
  end;

put('lessp,'simpfn,'simp_lessp);
```

Here all comparisons with numeric items are reduced to zero or one because of the equivalence of zero and "false" in algebraic mode, while in all other cases the non-evaluable expression is returned mapped by the function `mksq` which converts an algebraic expression into a standard quotient, ensuring the identity of composite kernels (see the section `standard forms`) .

5.3 Statements, forms

An additional way to link symbolic entries to algebraic mode is to invent a new syntactical unit. A REDUCE statement is introduced by assigning the property `stat` = `rlis` to the leading keyword of the new statement. The corresponding function will be called with one parameter which is a list of the (unevaluated) parameters of the statement. A statement normally is called for its side effect rather than for its value - so the result will be not interpreted as an algebraic quantity (`ws` not set, printed as symbolic list etc.). Example:

```
put('my_add,'stat,'rlis);

symbolic procedure my_add u;
```

```
begin scalar r;
   r:= 0;
   for each x in u do
   <<x:=reval x;
     r:=reval{'plus,x,r}
   >>;
   return r;
end;
```

to be called by a statement

```
my_add x,x^2,x^3;
```

A statement without parameters is introduced by the property - value pair `stat` = `endstat` because the statement "end" is the most prominent representative of this class.

For a more complicated syntax you can write a function which preprocesses a statement before evaluating it. Such preprocessors are linked to the leading keyword by the property `formfn`. However writing a formfn is rather difficult as you will have to organize the reading of the rest of the statement yourself. This approach is not recommended as the `formfn` interface is rather delicate and might be subject to future changes.

5.4 Printing and messages

For printing in symbolic mode you can use the Standard LISP functions

print(u)	print u with LISP syntax and following linefeed
prin1(u)	print u with LISP syntax but without linefeed
prin2(u)	print u without LISP syntax and without linefeed
prin2t(u)	print u without LISP syntax but with linefeed
terpri()	print a single linefeed

LISP syntax here means that a string is surrounded by string quotes and special characters in a symbol are tagged by an exclamation mark.

However if you want to print an algebraic expression in the same style as the REDUCE `write` statement you should use the function `writepri`. This function needs two arguments, the first one is the object to be printed which must be a string, a number or an algebraic expresssion tagged with a dynamic quote, e.g. using the function `mkquote`. The second argument is one of *only*, *first*, *nil*, and *last* and controls linefeeds. Example:

```
<<u:='(expt x 2);
  writepri(" u= ",'first);
  writepri(mkquote u,nil);
  writepri(" printed in algebraic style",'last);
>>;
```

For printing one single algebraic form you can also use the function `mathprint`:

```
u := '(expt x 3);
print u;
mathprint u;
```

6 Error management

6.1 Issue an error message

When programming an application in symbolic mode, it may be necessary to report an exception to the user, e.g. if an algebraic prerequisite for applying the algorithm is not fulfilled. There are several routines for signalling an error in symbolic mode:

- The easiest way to complain about a bad algebraic form is to call `typerr`. This function has two parameters $typerr(exp, str)$: the first one is an algebraic expression, the bad object, and the second one is a string which describes what the object should have been. When printed REDUCE will report *exp* in mathematical notation and insert a string "illegal as". The current program is terminated.

  ```
  u := '(plus x 1);
    . . .
  if not numberp u then typerr(u,"number");
  ```

- A general error can be reported using the function `rederr`; it prints some asterisks and then its argument in pure LISP style (no mathematical format); the argument can be a list - in that case the surrounding brackets are omitted in output. The current program is terminated.

  ```
  rederr "algorithm not applicable if denominator neq 1";
  rederr {"never call me with ",u," again"};
  ```

- Similar to *rederr* the routine `rerror` terminates the run with an error message; it has been designed for usage in a REDUCE package enabling the error handler to localize the error reason and to provide additional information to the end user (e.g. via a `help` system). It has 3 parameters: $rerror(pack, nbr, mess)$ where *pack* is the name of the package (usually a quoted identifier), *nbr* is an integer error number local to the package, beginning with 1 and *mess* is the error message just like *rederr*.

  ```
  rerror('asys,13,"remainder not zero");
  ```

  Wherever possible `rerror` should be used instead of `rederr`.

6.2 Catching errors

On the other hand a symbolic program might want to react to an exception which occurred in a calculation branch. REDUCE supports this by encapsulation with the routines `errorset*` and `errorset` . You have to construct the action to be encapsulated as a subroutine call in prefix LISP syntax which is identical to algebraic form syntax: a list where the first element is the name of the function (usually a quoted symbol) followed by the arguments; these must be tagged by quote, best using the function `mkquote`. The second parameter describes the error mode: if t, an eventual error during the evaluation will be reported to the user, if *nil* the error message will be suppressed. `errorset` additionally has a third parameter which tells the error manager whether to print a backtrace or not in an error case. The result of these functions will be

- in the case of an error an error code (its form depends on the underlying LISP system),
- otherwise the result of the computed value as `car` of a list.

You should use the function `errorp` for testing whether the call failed or not; in the positive case pick out the result by a *car* access. Example:

```
begin scalar u,v;
     .  .  .
 q := errorset!*({'quotsq,mkquote(u),mkquote(v)},nil);
 if errorp q then rederr "division failed" else
     q:=car q;
     .  .  .
```

Note that you cannot write the expression to be encapsulated in REDUCE syntax here. And also $'(quotsq\ u\ v)$ would be incorrect because then u and v were constants and not references to the local variables u and v. Only the *mkquote* expressions guarantee that at run time a list is generated where the values of u and v are inserted as parameters for the call.

7 Compiling, modules, packages

7.1 Program modules

It is a good practice to design a complex program group in separate modules each residing in a separate source file. You can use the REDUCE statement pair `module - endmodule` to reflect this structure. A module has a name which often will be identical with its source file (without extension, of course). This correspondence is not essential, but is very practical for the daily work. As a side effect `module` switches automatically to symbolic mode during input and `endmodule` switches back to the original mode. A typical module in a file would look like

```
module alggeo1; % Primitives for algebraic geometry.

 % Author: Hugo Nobody, Nirwanatown, Utopia.

 .......   the symbolic mode code

endmodule;
end;
```

If this module resides in a file *alggeo*1.*red* it can either be read into a REDUCE session by

```
in "alggeo1.red"$
```

or it can be compiled to a fast loadable binary by

```
faslout "alggeo1";
in "alggeo1.red"$
faslend;
```

You then can load this module by load alggeo.

Although REDUCE currently does not yet rely exclusively on declarations `exports` and `imports` , these should be used in order to be safe for the future. `exports` should name all entry points which the module offers for external access and `imports` is the list of modules or packages which the module needs at run time or during compilation.

7.2 Packages

If your application consists of several modules you can design it as a `package`. A package is a collection of modules which form a unit but allow you an individual management, e.g. updating and recompilation. A package is defined by a header module which should have the name of the package as the module name. This module then contains a declaration for the package by the function `create-package`, which expects as first parameter a list of the modules which belong to the package (this list must begin with the header package) and a second dummy parameter to be set to *nil* (this parameter is meaningful only for modules in the REDUCE kernel). Additionally the header module should contain all global declarations for the complete package, e.g. `fluid, global` declarations and `exports` and `imports` statements. Also, all macros, smacros used in more than one module in the package should be defined in the header module.

As long as you develop your application and frequently recompile single modules it would be practical to replace the function `create-package` by a loop calling `load_package` (please note the underscore: `load-package` is a different function not usable here). You then can load the complete package by one call to `load_package` for the header module. Example:

```
module alggeo; % Algebraic geometry support.

 % Author: Hugo Nobody, Nirwanatown, Utopia.

% create_package('(alggeo alggeo1 alggeo2 alggeo3));
for each x in '(alggeo1 alggeo2 alggeo3) do load x;

fluid '(alggeo_vars!* ......);
exports ...
     ... and so on

endmodule;
```

```
end;
```

Later you can replace the load loop `load` by a correct `create-package` expression - you will then have to compile the object as one big binary with the header module as the first entry.

For compiling you use the function `faslout` which creates a fast loading file - see the corresponding section in the User's manual.

8 Coding suggestions

In order to guarantee that your program is as portable and version independent as possible the following rules may be useful. Note that these are recommendations only - your program may do a good job for you even if you ignore all of them.

- Don't use functions from the underlying LISP which are not explicitly part of REDUCE or `standard lisp` - they will probably be missing in other support systems.
- Don't rely on character case. Some implementations support upper case output, others prefer lower case. As the tendency goes towards lower case, the best policy is to use lower case characters only, except in strings or comments.
- Use proper indentation. Take the REDUCE sources as example. If the indentation corresponds to the block structure your programs will be better readable.
- Don't produce lines with more than 72 characters. There are still computers and networks around which have problem if the size of the good old punched card is exceeded.
- Write comments. The major entry point routines of your symbolic code should have one comment line following the procedure declaration; this comment should be one or more complete English sentences, beginning with an upper case character and ending with a dot.
- Try to avoid name conflicts. It is a common policy to name global and fluid variables with a trailing asterisk and to use a common name prefix for the procedure names.

- If you produce a large piece of software, organize it as a REDUCE package.

9 Appendix: Standard LISP extensions

The following typical LISP functions are defined in the `support` module of the `rlisp` package.

`atsoc`(u:any,p:alist)
Same as `assoc`, but using the operator *eq* instead of *equal* for the search. *atsoc* can be applied only if the identity of the object to be searched for is guaranteed, e.g. for symbols. It is substantially faster than `assoc`.

`copyd`(u:id,v:id)
Copies the procedure definition of v to the symbol u. Afterwards the procedure can be called by using the name u. Frequently used for embedding a function, e.g. for implementing a private trace:

```
if null getd 'myprog_old then copyd('myprog_old,'myprog);
symbolic procedure myprog(u);
  <<print {"calling myprog with parameter ",u};
    myprog_old(u)>>
```

`eqcar`(u:any,v:any)
Implements the frequently needed test *pairp u and car u eq v*.

`listp`(u:any)
Returns t if u is a top level list ending with a *nil*.

`mkquote`(u:any)
tags u with a quote. This function is useful for cases where a LISP eval situation is given, e.g. for coding for an *errorset* or for *writepri*.

`reversip`(u:list)
Reverses the elements of l in place - faster than *reverse*, but desctroys the input list.

`smember`(u:any,v:any)

Similar to *member* this routine tests whether u occurs in the structure v, however in an arbitrarily deep nesting. While *member* tests only the top level list, *smember* descends down all branches of the tree.

`smemq`(u:any,v:any)
Same as *smember*, however using *eq* as test instead of *equal*.

The following routines handle lists as sets (every element at most once):
`union`(u:any,v:any)
`intersection`(u:any,v:any)
`setdiff`(u:any,v:any)

The Standard LISP `apply` function requires that the parameters for the function to be applied are encoded as a list. For standard cases of 1,2 or 3 parameters the following variants allow an easier encoding:
`apply1`(u:id,u:any)
`apply2`(u:id,u:any,v:any)
`apply3`(u:id,u:any,v:any,w:any)