# CLARIFICATIONS TO THE SYMBOLIC MODE IN REDUCE

by

Salim S. Abi-Ezzi

Mathematical Sciences Department
Rensselaer Polytechnic Institute
Troy, NY 12181

## 1. Introduction

I assume the reader's knowledge of REDUCE's syntax, see [1,2]. The symbolic mode in REDUCE is best thought of as LISP presented in PASCAL like form. This mode should be used for system building and major enhancements to the algebraic mode, while most application programs will be written in algebraic mode.

In the process of building a 'Computations By Homomorphic Images' (CBHI) package in REDUCE, I've used the symbolic mode intensively. I had to refer to the REDUCE source code for most of the information I've needed. To make the use of the symbolic mode easier I've decided to present this information in the following paper. First I start by describing the fundamental data structures used internally to represent symbolic expressions, then I discuss the basic functions that manipulate these data structures, and finally I address the problem of interfacing between the two modes.

## 2. Data Structures

REDUCE has two main formats for storing algebraic expressions. For input, output and similar top level operations it uses a prefix representation, very similar to ordinary LISP programs, e.g.:
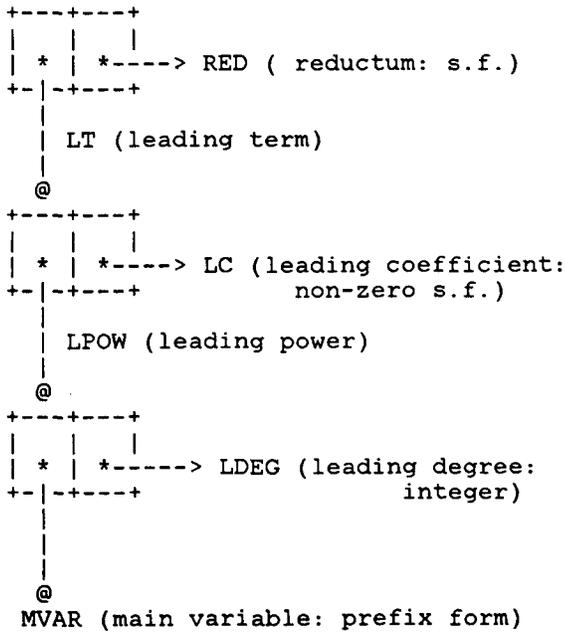
```
(PLUS 3 (TIMES 4 (EXPT X 2)))
```

Within the body of the algebraic system reduce uses a different representation, designed to be easy to compute with rather than easy to read. This type is called a standard quotient, and occasionally referred to as a canonical form. The standard quotient is a dotted pair of two standard forms. A standard form is the structure for representing a polynomial. The standard form of the above polynomial is the following S-expression:

```
(((X . 2) . 4) . 3)
```

in general, there are four types of standard forms. The first three are referred to as domain elements and satisfy the predicate DOMAINP.

A- NIL,       the zero polynomial

B- a number,   zero must be NIL not 0

C- p . q ,    where p and q are numbers. This represents the fraction p/q

D- a structure consisting of a leading term and a reductum (rest of polynomial):

```
+---+---+
|   |   |
| * |   *-----> RED ( reductum: s.f.)
+-|-+---+
  |
  | LT (leading term)
  |
  @
+---+---+
|   |   |
| * |   *-----> LC (leading coefficient:
+-|-+---+              non-zero s.f.)
  |
  | LPOW (leading power)
  |
  @ .
+---+---+
|   |   |
| * |   *------> LDEG (leading degree:
+-|-+---+                    integer)
  |
  |
  |
  @
  MVAR (main variable: prefix form)
```

It should be clear by now that a standard quotient is used to represent a rational function. The two standard forms of it's dotted pair are the numerator and denominator of that rational function. So the above expression in standard quotient form, is simply:

$$((((X . 2) . 4) . 3) . 1)$$

It is worth mentioning here that the integrity of these data structures should, by no means, be violated. A violation of this type will definitely lead to trouble when built-in functions are used. An example of offending REDUCE's data structure rules, is to use 0 instead of NIL, or to have a standard form that contains a term with a zero coefficient. In both cases 2 equal polynomials will be compared to be unequal!

3. Low level functions

Here I'll briefly list the basic functions for manipulating the above structures. For more details see [1].

- REDUCE selectors

. DENR    denominator of standard quotient
          CDR quotient
. LC      leading coefficient of polynomial
          CDAR form
. LDEG    leading degree of polynomial
          CDAAR form
. LPOW    leading power of polynomial
          CAAR form
. LT      leading term of polynomial
          CAR form
. MVAR    main variable of polynomial
          CAAAR form
. NUMR    numerator of standard quotient
          CAR quotient
. RED     reductum of polynomial
          CDR form
. TC      coefficient of term
          CDR term
. TDEG    degree of a term
          CDAR term
. TPOW    power of term
          CAR term

- REDUCE constructors
Note that .+, ./ and .* are all in fact implemented as CONS, CONS is also available in REDUCE as an infix operator (a dot '.').

    '.+'    add a term to a polynomial
    './'    divide (two polynomials to get quotient)
    '.*'    multiply a power by a coefficient to produce a term
    MKSP    raise a variable to a power

So, we can construct the above polynomial as a standard quotient as follows:

$$(((MKSP('X, 2) .* 4) .+ 3) ./ 1)$$

- REDUCE composite conversion functions

. !*F2Q converts a standard form to a standard quotient
. !*K2F convert a kernel to a standard form
. !*K2Q convert a kernel to a standard quotient
. !*P2F convert a standard power to a standard form
. !*P2Q convert a standard power to a standard quotient
. !*Q2F convert a standard quotient to a standard form
. !*Q2K convert a standard quotient to a kernel
. !*T2F convert a standard term to a standard form
. !*T2Q convert a standard term to a standard quotient
. SIMP!* convert from prefix to standard quotient
. PREPSQ convert a standard quotient to prefix form
. MK!*SQ package up standard quotient so as to be legal in prefix positions

As an example I'll show how the !*K2Q function could be written in terms of more primitive operations:

```
!*K2Q A = ((MKSP(A,1) .* 1) .+ NIL) ./ 1
```

So, first we construct a power (MKSP) then a term (.*) then a standard form (.+) and finally a standard quotient (./).

- Standard quotient functions

. ADDSQ addition
. MULTSQ multiplication
. NEGSQ negation
. INVSQ reciprocal

- Standard form functions

. ADDF addition
. NEGF negation
. MULTF multiplication

. GCDF returns The greatest common divisor
. QREMF returns a dotted pair of the quotient and remainder of division
. PP returns the primitive part of the standard form
. RANK returns the rank
. SQFRF returns a list of square free factors as standard forms
. DIFF returns the derivative
. SUBF takes a form and a list of substitutions, it returns the form resulting from applying these substitutions.

4. The Interface between the two modes of REDUCE

In the previous section I've listed some of the functions that will do the conversion between prefix form and standard quotients, and vice versa. These functions will play an essential role in the interface between the two modes. I think what I want to say, could best be described by an example. So, I will relate the process of making the GCDF function available to the algebraic mode users. GCDF takes two standard forms and returns their greatest common divisor as a standard form. First I will show how to obtain the GCD as a function (GCDAL1) available in algebraic mode.

```
SYMBOLIC PROCEDURE SGCDAL(L);
BEGIN
    SCALAR U,V,W;
    U := SIMP!* CAR L;
    V := SIMP!* CADR L;
    IF NOT AND(DENR U = 1,DENR V = 1)
        THEN REDERR "wrong argument" ;
    W := GCDF(NUMR U,NUMR V);
    RETURN W ./ 1
END;

SYMBOLIC
PUT('GCDAL1,'SIMPFN,'SGCDAL);

ALGEBRAIC W := GCDAL1(U,V);
```

4 5

The first thing to note is that to achieve the link between modes, we need to place the name of the symbolic function (SGCDAL), under the property SIMPFN, in the property list of the name of the algebraic mode function (GCDAL1). Second, the arguments are introduced to SGCDAL as a list (L); So, no matter how many arguments the function takes, the list of these arguments will be passed to the symbolic function. Third, the symbolic function transforms these arguments from prefix form to standard quotients, performs the operation and finally returns the result as a standard quotient. Luckily, after that, we don't have to worry about anything else, since REDUCE will handle the rest of the task. When the function is called from the algebraic mode, U and V are regular algebraic polynomials, W will also be of that type. One more thing to note is the REDERR function, which will be called in case of an error in an argument's value. This function takes a string as argument, writes it out and returns to top level. So, no matter how deep in recursion the function is, it will automatically unwind and empty the stack. Now I will show how to implement the GCD as a procedure:

```
SYMBOLIC OPERATOR GCDAL2;

SYMBOLIC PROCEDURE GCDAL2(V,U,W);
BEGIN
   SCALAR X;
   V:=SIMP!* V;
   U:=SIMP!* U;
   IF NOT AND(DENR U = 1,DENR V = 1)
      THEN REDERR "wrong argument" ;
   X:=GCDF(NUMR V,NUMR U);
   RETURN SETK(W,MK!*SQ (X ./ 1))
END;
```

Here I've used a completely different approach than in the function case. The key routine here is SETK, which takes its second argument, a prefix form, and assigns it to its first argument, a variable available in algebraic mode. The 'SYMBOLIC OPERATOR GCDAL2' command will make the procedure available in the algebraic mode.

A useful routine also, is SETCOEFF, which takes two arguments Z and W. Z is a list of form:

$$(( \text{integer} \ . \ \text{value}) \ ... \ )$$

and W is an array name. The procedure will assign the value part of each dotted pair to the array element whose subscript is the corresponding integer of that dotted pair. Another useful function for outputting expressions from symbolic mode, is MATHPRINT. It takes a standard quotient as an argument and prints it in customary 2 dimensional mathematical notation.

5. Hints for adding packages to REDUCE

The routines of the package should be written in symbolic mode and thus they have to deal with internal data structures, usually standard forms. The recursive definition of this data type gave a common general form to most of these recursive routines. Assume that we want to apply function FUN to a multivariate polynomial P in $Q[x,y,...]$, where FUN for instance is to return the homomorphic image of P in $Z_p[x,y,...]$. In the definition of FUN, the recursive call then will be as follows:

$$(((\text{FUN LC P}) \ . \ \text{LPOW P}) \ . \ (\text{FUN RED P}))$$

In the debugging phase of these routines, the users will be particularly annoyed by the difficulty of entering polynomials and comprehending outputted polynomials when these polynomials are in their standard forms. For this reason the following procedures are helpful.

```
SYMBOLIC PROCEDURE !*AL2SF(P);
    CAADAR GET (P,'AVALUE);
```

```
SYMBOLIC PROCEDURE !*SF2AL(F,R);
    REVAL SETK(R,AEVAL PREPSQ !*F2Q F);
```

When a variable is initialized in algebraic mode, REDUCE places the list of its stuffed prefix value, that has the form (!*SQ <standard quotient> T), under the property AVALUE in the variable's property list. So getting the standard form of that variable could be achieved by function !*AL2SF. Note that the argument to this function should be a quoted atom (variable's name). Function !*SF2AL makes the reverse job of !*AL2SF, it takes a standard form as its first argument and assigns the corresponding polynomial to its second argument, a variable which will be available in the algebraic mode and whose name should be passed as a quoted atom.

After completing the symbolic routines, the package still needs some interface routines. These routines will be available for the algebraic user, they will take care of the data type conversions and of the calls to appropriate symbolic routines. The invisibility of the symbolic part of the package will be achieved through these interface routines.

The routines of the CBHI package were a direct implementation of the algorithms presented in [3]. I've successfully used the package for dividing multivariate polynomials. Readers interested in more details are welcomed to contact me.

References

[1] R. Arthur Norman: Symbolic and Algebraic Modes In Reduce. University of Cambridge Tutorial, April-1979.

[2] C. Anthony Hearn: Reduce 2 User's manual. Second edition, March-1973.

[3] G.E. Collins, R. Loos, R. Albrecht. Computer Algebra Symbolic And Algebraic Computations. Austria Springer-Verlag/Wien 1982.