

Contents

1	Introduction	3
2	Command Language	5
2.1	identity command	5
2.2	remove command	5
2.3	generators command	6
2.4	field command	6
2.5	build command	7
2.6	display command	7
2.7	polynomial command	7
2.8	xpand command	8
2.9	type command	8
2.10	help command	9
2.11	view command	9
2.12	output command	10
2.13	save command	10
2.14	change command	10
2.15	quit command	11
2.16	Summary of commands	11
3	Basis Table and Multiplication Table	12
4	The .albert File	14
5	Invoking Albert	15
6	Installing and Obtaining Albert over Internet	16
7	Sample Scenario	17
8	Polynomial Expression Language	20

1 Introduction

Albert is an interactive research tool to assist the specialist in the study of nonassociative algebra. This document serves as a technical guide to Albert. We refer the reader to [1] for a more casual tutorial. The main problem addressed by Albert is the recognition of polynomial identities. Roughly, Albert works in the following way. Suppose a user wishes to study *alternative algebras*. These are algebras defined by the two polynomial identities $(yx)x - y(xx)$ and $(xx)y - x(xy)$, known respectively as the right and left alternative laws. In particular, the user wishes to know if, in the presence of the right and left alternative laws, $(a, b, c) \circ [a, b]$ is also an identity. Here (a, b, c) denotes $(ab)c - a(bc)$, $[a, b]$ denotes $ab - ba$, and $x \circ y$ denotes $xy + yx$. The user first supplies Albert with the right and left alternative laws, using the **identity** command. Next, the user supplies the *problem type*. This refers to the number and degree of letters in the target polynomial. For example in this problem, each term of the target polynomial has two a 's, two b 's, and one c , and so the problem type is $2a2b1c$. This is entered using the **generators** command. It may be that over certain rings of scalars the polynomial is an identity, but over others it is not an identity. Albert allows the user to supply the ring of scalars, but currently the user must select a Galois field Z_p in which p is a prime at most 251. This is done using the **field** command. If no field is entered, the default field Z_{251} is chosen.

In deciding whether a given polynomial is an identity or not, Albert internally constructs a certain homomorphic image of the free algebra. It is not necessary that the user understand the theory of free algebras, nor is it necessary to understand the algorithms Albert employs to create them. The user need only be aware that Albert builds a multiplication table for this free algebra. The user instructs Albert to begin the construction using the **build** command. Once this construction has been completed, the user can query whether the polynomial $(a, b, c) \circ [a, b]$ is an identity using the **polynomial** command. In fact, the user can ask Albert about any homogeneous polynomial $p(a, b, c)$ having most two a 's, two b 's and one c in each term. For example, the polynomial $(a, b, (a, b, c)) + [b, a](a, b, c)$ could also be tested. With Albert, polynomials and identities may be entered from the keyboard using associators, commutators, and much of the familiar notation used in nonassociative ring theory. Since the standard keyboard does not have the symbol \circ , we use $*$ to denote the Jordan product. See the section entitled Polynomial Expression Language for a complete description of how polynomials and identities can be entered in Albert.

Albert has a small set of commands, and meaningful research can be conducted using only the **identity**, **generators**, **build**, and **polynomial** commands. These commands, and others, are described in detail in this guide. The user who wishes to quickly learn the system is advised to first try these commands. Thus, the typical user supplies Albert with the following input:

- A set I of polynomials whose members are assumed to be identities.
- A problem type as described above.
- A field F of scalars.

In this document polynomial always means a nonassociative polynomial. These objects together implicitly define a fourth object, namely the free nonassociative algebra. In this document we refer to these four objects collectively as a *configuration*. Various commands may alter or delete certain objects of a configuration. Typically, Albert spends much of its time constructing a multiplication table. But once a table has been constructed, Albert can quickly decide if a polynomial or group of polynomials are identities.

After initiating a “build”, the user may wish to abort the construction without exiting Albert. As a convenience, entering the *control-c* character stops the build operation, and Albert will return to the command prompt.

All polynomials and defining identities must be homogeneous. That is, each must be expressible as a linear combination of words each having the same degree in each variable. This restriction is not very severe, since any nonhomogeneous polynomial can be replaced by its homogeneous parts. This replacement will not affect the results given a sufficiently high characteristic for the field. *Albert internally linearizes any defining identity that is not multilinear.* Thus the right alternative identity is always interpreted as $(yx)z - y(xz) + (yz)x - y(zx)$.

However, if a defining identity is not multilinear, the user is advised *not* to linearize it before entering it, but rather enter it in its non-multilinear form. In most cases, this allows Albert to treat it more efficiently, since the identity’s symmetry can be exploited. In general, the larger the degree of the problem type, the more memory Albert requires. Given two problems involving the same degree and the same defining identities, Albert will cope best with the one having fewer letters. The defining identities influence the problem, too. Defining identities such as the commutative law (as in the case of Jordan algebras) and the anticommutative law “drive down” the dimension of the free algebra, thus enabling larger problems to be solved than might otherwise be possible. If Albert is unable to complete a problem having degree n , adding additional identities of degree less than n may allow Albert to finish.

Finally, a word of caution. Like any program, the possibility is high for errors. Please report any suspected bugs or general comments to D.P. Jacobs at dpj@clemson.edu.

2 Command Language

The commands available with Albert are : **identity**, **remove**, **generators**, **field**, **build**, **display**, **polynomial**, **xpand**, **type**, **help**, **quit**, **view**, **output**, **save**, **change**.

Each command begins with a keyword, and the user can use any initial sub-string of the keyword. For example, the **identity** command can be used by typing **i**, **id**, etc. Every command is terminated by a carriage return. Some commands (e.g. **identity**, **polynomial**) require a polynomial as an argument. This polynomial may at times exceed the screen-width. In such cases, the user can continue on the next line by terminating the preceding line with a backslash (\).

2.1 identity command

identity [*polynomial*]

Examples:

```
identity (x,x,[y,x])  
i (x,(x,(x,y,x),x),x)
```

This command appends the polynomial to the current set of identities. Albert assigns a unique number to the entered identity for future use. Entering a new identity destroys any existing multiplication table in memory.

- **Arguments:** *polynomial* as described in the section Polynomial Expression Language. Names defined in the **.albert** file can also be used to enter a polynomial. The entered polynomial must be homogeneous.
- **Errors:** malformed or non-homogeneous polynomial.

2.2 remove command

remove [*number* | *]

This command removes one or more identities from the current set of identities. For example,

```
remove 2
```

would remove the identity whose number is 2. The remaining identities are renumbered after deletion of the identity. An asterisk (*) can be used in place of *number* to remove all existing identities:

r *

The **remove** command will destroy a resident multiplication table, if present.

- **Arguments:** The *number* of the identity, or “*”.
- **Errors:** Invalid *number*.

2.3 generators command

generators [*problem type*]

Before beginning the construction of an algebra, Albert must know what the generators will be, and the degrees of the generators. This information is referred to as the problem type, and the **generators** command is used to define it. This problem type is stored in the current configuration. Entering a new problem type destroys any existing problem type and any multiplication table, if present. For example,

generators aabcc
gen aabcc
g 2ab2c

- **Arguments:** *problem type* is a string of lower-case letters indicating the generators and degrees used in the problem. For example, *aabcc* indicates that the algebra to be built will be generated by *a*, *b*, and *c*, and will be spanned by words in these letters having at most two *a*'s, one *b*, and two *c*'s. This word must have degree at least two. This can also be entered in abbreviated form as *2a1b2c*, *2ab2c*, *b2a2c*, etc.

2.4 field command

field [*number*]

This command changes the field of scalars, and this information is stored as part of the current configuration. When the field is changed, any resident multiplication table is destroyed. For example,

field 17

will cause subsequent algebras to be constructed over the field Z_{17} . When Albert is first entered, the default field Z_{251} is selected. This field remains in effect until the field is changed.

- **Arguments:** The *number* must be prime and at most 251.
- **Errors:** *number* out of range or not prime.

2.5 build command

build

This command invokes Albert to begin construction of the algebra defined by the current configuration. Albert constructs the algebra using the current set of identities, problem type and field stored in the current configuration. Status information is printed during the construction. An old multiplication table is destroyed.

- **Arguments:** None.
- **Errors:** Problem type not defined, or memory overflow during the construction.

After initiating a “build”, the user may wish to abort the construction without exiting Albert altogether. If the user enters the *control-c* character during the build operation, Albert will return to the command prompt.

2.6 display command

display

Typing **display** causes Albert to display the current set of defining identities, field, problem type and information about the multiplication table, if present.

2.7 polynomial command

polynomial [*polynomial*]

After a multiplication table has either been constructed using **build**, this command may be used to test whether the given polynomial is zero in the resident algebra. For example, typing

p 2((ba)a)a +((aa)a)b -3((aa)b)a

might cause Albert to respond with:

Polynomial is not an identity.

- **Arguments:** *Polynomial* has to be homogeneous.
- **Errors:** Invalid or non-homogeneous polynomial, nonexistent table, polynomial incompatible with current problem type.

2.8 xexpand command

xexpand [*polynomial*]

This command is used to see the expanded form of a nonassociative polynomial. For example, typing

xexpand (x,y,z)

would cause Albert to respond with:

(xy)z - x(yz)

The command is used for information purposes only and does not effect the current configuration.

- **Arguments:** *Polynomial* has to be homogeneous.
- **Errors:** Invalid or non-homogeneous polynomial.

2.9 type command

type [*nonassociative word*]

Every *nonassociative word* has a particular *association type*. These *association types* are *numbered* by Albert. For example, the association type of the word $((ab)c)d$ is 1, while the association type of the word $(ab)(cd)$ is 3. Thus, typing

t (ab)(cd)

would cause Albert to respond with:

The association type of the word = 3.

This command prints the *number* of the *association* of the argument. This command can be used in conjunction with the W operator (see the section Polynomial Expression Language).

- **Arguments:** *nonassociative word* like $a((ac)b)$. The letters and their order are not important.
- **Errors:** Invalid *word*.

2.10 help command

help [*command*]

This command provides detailed information about the *command* named as the parameter. If no such command is given, a list of all commands is displayed.

- **Arguments:** An Albert command name or the abbreviated form of one.

2.11 view command

view [*b* | *m*]

This command prints the basis table or the multiplication table to the screen, provided they exist. The argument specifies the table to be output (b - basis table, m - multiplication table). For example, typing

view b

will output the current basis table to the screen.

- **Arguments:** b or m

2.12 output command

output [*b* | *m*]

This command outputs the basis table or the multiplication table to the printer, provided they exist. The argument specifies the table to be output (b - basis table, m - multiplication table). For example, typing

output m

will output the multiplication table to the printer. Output is sent to the user's default printer, which can be modified outside of Albert.

- **Arguments:** b or m

2.13 save command

save [*b* | *m*]

This command saves the basis table or the multiplication table to a file. The argument specifies the table to be output (b - basis table, m - multiplication table). For example, typing

save m

will cause the multiplication table to be written to a file. The user will be prompted for a file name.

- **Arguments:** b or m

2.14 change command

change

Most of the time Albert's computing time and memory are spent performing matrix computations. These matrices tend to be sparse (i.e. have relatively few nonzero entries). There are two ways to store matrices: As ordinary two-dimensional arrays, in which all matrix elements are stored, or alternately using a data structure in which only the nonzero entries

are stored. Of course in such a sparse implementation, there is more overhead associated with each entry. Consequently, for dense matrices the traditional two-dimensional array would use less memory, while a sparse implementation would be more efficient for sparse matrices. Albert can use either matrix implementation. With the sparse method, the overhead per entry is now about eight times that of the traditional method, and so the sparse implementation will be most beneficial for matrices whose density never exceeds 12%. The matrix densities that occur in Albert vary wildly. Thus, a sparse method will sometimes benefit memory utilization, and sometimes it will hinder memory utilization. For this reason, the change command allows the user to toggle between the two methods. This switch will not effect the results of the computation, only the time and memory needed by Albert to obtain the results. Since many interesting results often seem to be just beyond Albert's "reach", it is hoped that this fine tuning knob may enable more problems to be solved. A useful methodology is to begin a problem with the new (default) sparse setting. It is likely this will be the best method. But, if the problem is unable to finish due to lack of memory, there is a chance the "change" toggle will help, especially if the user feels that the matrix densities are exceeding 12%.

2.15 quit command

quit

This command exits the user from Albert.

2.16 Summary of commands

identity[polynomial]	enter a defining identity
remove[number *]	remove a defining identity
generators[word]	specify the problem type
field[number]	change the current ring of scalars
build	build a multiplication table
display	display the current configuration
polynomial[polynomial]	query if the polynomial is an identity
xpand[polynomial]	expand the polynomial
type[nonassociative word]	determine the association type
help[command]	help on albert
change	change matrix implementation
view[b m]	view basis or multiplication table
save[b m]	save basis or multiplication table
output[b m]	print basis or multiplication table
quit	quit from albert

3 Basis Table and Multiplication Table

There are two important structures created by the **build** command. These are the basis table and the multiplication table. These tables can be seen using **view**, printed using **output**, or stored using **save**. The basis table contains a list of elements that form a basis in the free algebra that was constructed. Under Albert's method, basis elements will always be *words* in the original generators. Shown below is the basis table for a 26 dimensional right alternative algebra using 2a's and 2b's. There are five columns. The first of these is simply the number by which the basis element is referred. The second and third columns indicate how the basis element factors into a product of two lower degree basis elements. The fourth columns indicates the type (degrees in each generator) of the element. The fifth column shows the basis element as a nonassociative word. For example, the table indicates that basis element #25 is the product of element #13 and element #2. It also indicates that this element has type 22 (2 a's and 2b's), and shows the element as (((ab)a)b). Obviously, the element's type is inherent in the last columns, however this column should make it easier to take a large table and pick out all elements of a certain type.

Basis Table:

```
1.  0 0 10 a
2.  0 0 01 b
3.  2 2 02 (bb)
4.  2 1 11 (ba)
5.  1 2 11 (ab)
6.  1 1 20 (aa)
7.  2 5 12 (b(ab))
8.  3 1 12 ((bb)a)
9.  4 2 12 ((ba)b)
10. 5 2 12 ((ab)b)
11. 1 5 21 (a(ab))
12. 4 1 21 ((ba)a)
13. 5 1 21 ((ab)a)
14. 6 2 21 ((aa)b)
15. 1 10 22 (a((ab)b))
16. 2 14 22 (b((aa)b))
17. 4 5 22 ((ba)(ab))
18. 5 5 22 ((ab)(ab))
19. 7 1 22 ((b(ab))a)
20. 8 1 22 (((bb)a)a)
21. 9 1 22 (((ba)b)a)
22. 10 1 22 (((ab)b)a)
23. 11 2 22 ((a(ab))b)
24. 12 2 22 (((ba)a)b)
25. 13 2 22 (((ab)a)b)
26. 14 2 22 (((aa)b)b)
```

A portion of the multiplication table for the same algebra is shown below. The table lists only the *nonzero* products of two basis elements. Coefficients are given in terms of the current field. Assuming the default field Z_{251} were in use, the table below can be interpreted as

$$\begin{aligned}
 b_1 b_1 &= b_6 \\
 b_1 b_2 &= b_5 \\
 b_1 b_3 &= b_{10} \\
 b_1 b_4 &= -b_{11} + b_{13} + b_{14} \\
 b_1 b_5 &= b_{11} \\
 b_1 b_7 &= -b_{15} + b_{18} + b_{23} \\
 b_1 b_8 &= -b_{15} + b_{22} + b_{26} \\
 b_1 b_9 &= b_{25}
 \end{aligned}$$

Had we shown the entire multiplication table, we would have seen that $b_{13}b_2 = b_{25}$. Hence b_{25} factors as b_1b_9 and $b_{13}b_2$. This merely says that $a((ba)b) = ((ab)a)b$ in a right alternative ring. Recall that when Albert constructs an algebra in, say, 2a's and 2b's, products involving more than 2a's or more than 2b's will be zero. Other kinds of products, too, can be zero.

Multiplication table:

(b1)*(b1)	1 b6
(b1)*(b2)	1 b5
(b1)*(b3)	1 b10
(b1)*(b4)	250 b11 + 1 b13 + 1 b14
(b1)*(b5)	1 b11
(b1)*(b7)	250 b15 + 1 b18 + 1 b23
(b1)*(b8)	250 b15 + 1 b22 + 1 b26
(b1)*(b9)	1 b25

4 The .albert File

The **.albert** file contains definitions for making it easier to enter identities. This file can be edited by the user outside of Albert. Arbitrarily many “defines” can be given. Long definitions can be continued on another line by placing a **backslash** (`\`) at the end of the line. The **.albert** file can include blank lines. Comments begin with `%` and extend to the end of a line. A typical **.albert** file might look like this.

```
rightalt      (x,y,y)          % Right alternative law.
jordan        ((xx)y)x - (xx)(yx)
com           [x,y]
doublecom     [[x,y],z]

% A strange new identity:
ident3        (x,[x,y],x)
```

When Albert is initialized, the **.albert** file is read into memory. *This file must reside in the user's current directory.* Definitions occurring within this file can be used in subsequent commands, by surrounding the defined entity with `$`'s. For example, one now could enter the command:

```
identity $jordan$ - $ident3$
```

The identity is interpreted as $((xx)y)x - (xx)(yx) - (x, [x, y], x)$. The **.albert** file can make use of definitions occurring elsewhere in the file. Moreover, a definition need not occur before its use. For example, one might have

```
RightAlt      (x,y,y)          % Right alternative law.
RightAltCom    [w, $RightAlt$] %Right alternators commute.
```

This last identity is interpreted as $[w, (x, y, y)]$. Circular definitions will cause great problems.

5 Invoking Albert

Albert is invoked on the command line by giving its name followed by two optional arguments.

```
albert -d dimlimit -a dirname
```

Here, `dirname` refers to the directory location where `albert` will get the `.albert` file. If this argument is not given, `albert` will look for it in the current directory. Here `dimlimit` refers to the dimension limit that `albert` will use. When the build command is given, and a construction is begun for an algebra whose dimension exceeds this limit, the construction will fail. This dimension limit will be in effect for the duration of the session. Thus the user should give an adequate setting. However, *setting this number too large will cause albert to run extraordinarily slow, so some care should be used.* Legitimate values for dimension limits are multiples of 500 up to 10,000. If no dimension limit is given, the default is 500.

6 Installing and Obtaining Albert over Internet

Albert is written in C and is intended to run on a UNIX based computer. Ideally, a workstation having at least 4 MB. of main memory is recommended. The amount of memory present will effect the success of the system. To obtain the latest albert system over internet, using an anonymous ftp, first create a new directory on your Unix system and from this directory,

1. ftp -i ftp.cs.clemson.edu
2. When it prompts for a name, type “anonymous”
3. When it prompts for a password, type “anonymous”
4. cd albert
5. mget *.c
6. mget *.h
7. get Makefile
8. quit

First-time users may also want a sample copy of the **.albert** file. After quitting from ftp, type “make”. After the make process, you should have an object program called “albert”. Execute the program, and hopefully you will see the Albert prompt -->. You may delete the *.o files that were generated.

7 Sample Scenario

The following example illustrates interaction with Albert. Text appearing after the --> prompt has been printed by the user; all other text has been typed by the Albert system.

```
indigo[25] albert
```

```
Albert, Version 3.0, 1993
Dept. of Computer Science, Clemson University
```

```
-->identity (x,x,y)
```

```
(xx)y - x(xy)
Entered as identity 1.
```

```
-->identity (x,y,y)
```

```
(xy)y - x(yy)
Entered as identity 2.
```

```
-->generators 2a2b1c
```

```
Problem type stored.
```

```
-->display
```

```
Defining identities are:
```

```
1. (x,x,y)
```

```
2. (x,y,y)
```

```
Field = 251.
```

```
Problem type = [2a,2b,c]; Total degree = 5.
```

```
Multiplication table not present.
```

```
-->build
```

Building the Multiplication Table.

Build begun at Fri Aug 31 13:51:34 1990

Degree	Current Dimension	Elapsed Time(s)
1	3	0
2	11	0
3	30	0
4	64	2
5	99	7

Build completed

-->polynomial (a,b,c)*[a,b]

Polynomial is an identity.

-->polynomial (a,b,(a,b,c)) + [b,a](a,b,c)

Polynomial is an identity.

-->polynomial [a,[b,(a,b,c)]]

Polynomial is not an identity.

-->remove 1

Identity 1 removed.
Destroyed the Multiplication Table.

-->generators 4a2b
Problem type changed.

-->display

Defining identities are:

1. (x,y,y)

Field = 251.

Problem type = [4a,2b]; Total degree = 6.

Multiplication table not present.

-->build

Building the Multiplication Table.

Build begun at Fri Aug 31 13:55:24 1990

Degree	Current Dimension	Elapsed Time(s)
1	2	0
2	6	0
3	15	0
4	35	1
5	77	2
6	146	5

Build completed

-->poly (a,a,b)^2

Polynomial is not an identity.

-->poly (a,a,b)^3

Polynomial type not a subtype of Target_type.

-->quit

8 Polynomial Expression Language

The **identity**, **polynomial**, and **xpand** commands require a nonassociative polynomial to be entered. This section describes the proper syntax of nonassociative polynomials. In Appendix A, a formal grammar is given. Nonassociative polynomials are described using the following operators.

addition: $x + y$.

subtraction: $x - y$.

unary minus: $-x$.

scalar product: $3x$. The scalar is interpreted to be from the ring of integers.

juxtaposed product: xy .

commutator: $[x, y] = xy - yx$.

associator: $(x, y, z) = (xy)z - x(yz)$.

Jordan product: $x * y = xy + yx$.

Jordan associator: $\langle x, y, z \rangle = (x * y) * z - x * (y * z)$.

Jacobi: $J(x, y, z) = (xy)z + (yz)x + (zx)y$.

left associated exponential: $x^{\wedge}3 = (xx)x$. Warning: $xy^{\wedge}3$ means $((xy)(xy))(xy)$ not $x((yy)y)$.

left/right multiplication: x^{\prime} denotes left multiplication by x , and x^{\backprime} denotes right multiplication. Here x can be more complicated expression. These operators are written to the right of an operand, and can be composed. Thus $\{xy^{\backprime}z^{\prime}u^{\prime}\}$ denotes $((yx)z)u$ and $\{xy^{\backprime}z^{\prime}u^{\prime}(w(ts))^{\prime}\}$ denotes $(w(ts))((yx)z)u$.

artificial word: $W\{n; a : b : a : c : d\}$: n is called an *association type*. Often it is cumbersome to type in long parenthesized expressions. To simplify the typing, the artificial word construct can be used. $W\{n;a:b:a:c:d\}$ represents the word having letters a, b, a, c, d and association type n . Albert places a well-ordering on associations. If two associations w_1 and w_2 have different degrees, then $w_1 < w_2$ if w_1 has smaller degree. Suppose w_1 and w_2 have the same degree. If this degree is 1 or 2 then $w_1 = w_2$. But if

$$\begin{aligned} w_1 &= (l_1)(r_1) \\ w_2 &= (l_2)(r_2) \end{aligned}$$

then $w_1 < w_2$ if either $r_1 < r_2$ or $r_1 = r_2$ and $l_1 < l_2$. Thus, the smallest degree n *association type* is

$$(\dots(((xx)x)x)\dots)x$$

and the largest degree in *association type* is

$$x(\dots(x(x(x(xx))))\dots).$$

The degree 4 association types, in order, are :

$$\begin{array}{l} \text{degree 4} \quad ((xx)x)x \\ \quad \quad \quad (x(xx))x \\ \quad \quad \quad (xx)(xx) \\ \quad \quad \quad x((xx)x) \\ \quad \quad \quad x(x(xx)) \end{array}$$

Thus $W\{4; a : a : c : b\}$ means $a((ac)b)$. Note that typing in a number n that exceeds the number of degree n *association types* is an error. The user can easily determine the number for an association using the **type** command described in the section Command Language.

The operators can be intermixed in any arbitrary fashion. For example the following constructs are allowed:

$$\begin{array}{l} (x, [x, y], x) \\ J(x, < x, y, z >, z) \\ [x \wedge 3, (x, x * y, y)] \end{array}$$

All variables must be entered lower case letters. The order in which operators are applied is controlled using parenthesis. Thus one may write $(x * y) * z$ or $x * (y * z)$. An ambiguous expression such as $x * y * z$ is illegal. Most expressions have the same common meaning as they do in mathematics. For example, scalar multiplication and unary minus(-) have higher precedence over addition and subtraction and therefore $3(a,b,c) + (c,b,a)$ means $(3(a,b,c)) + (c,b,a)$. However there are some caveats. When used in the presence of \wedge or $*$, juxtaposition has higher precedence : $xy \wedge 3$ means $((xy)(xy))(xy)$ not $x(y^2y)$ and $xy * x$ means $(xy) * y$.

Appendix A : Formal Grammar

<i>polynomial</i>	→	<i>term</i> <i>+ term</i> <i>- term</i> <i>polynomial + term</i> <i>polynomial - term</i>
<i>term</i>	→	<i>product</i> int <i>product</i>
<i>product</i>	→	<i>atom_or_double_atom</i> <i>atom_or_double_atom</i> ↑ int <i>atom_or_double_atom</i> * <i>atom_or_double_atom</i>
<i>atom_or_double_atom</i>	→	<i>atom</i> <i>double_atom</i>
<i>double_atom</i>	→	<i>atom atom</i>
<i>atom</i>	→	small_letter <i>commutator</i> <i>associator</i> <i>jacobi</i> <i>jordan_associator</i> <i>artificial_word</i> <i>operator_product</i> (<i>polynomial</i>)
<i>commutator</i>	→	[<i>polynomial, polynomial</i>]
<i>associator</i>	→	(<i>polynomial, polynomial, polynomial</i>)
<i>jacobi</i>	→	J (<i>polynomial, polynomial, polynomial</i>)
<i>jordan_associator</i>	→	<i>j polynomial, polynomial, polynomial i</i>
<i>artificial_word</i>	→	W { int ; <i>letter_list</i> }
<i>letter_list</i>	→	small_letter <i>letter_list</i> : small_letter

operator_product → { *atom operator_list* }

operator_list → *operator*
| *operator_list operator*

operator → *atom* ‘
| *atom* ’

int : Positive. Sequence of digits not beginning with 0.

References

- [1] D.P. Jacobs, S.V. Muddana, A.J. Offutt, "A Computer Algebra System for Nonassociative Identities, Hadronic Mechanics and Nonpotential Interactions", Proceedings of the Fifth International Conference, Cedar Falls, Myung, H.C. (Ed.), Nova Science Publishers, Inc., New York, 1993.
- [2] I.R. Hentzel and D. Pokrass Jacobs, "A Dynamic Programming Method for Building Free Algebras, Computers & Mathematics with Applications", 22 (1991), 12, 61-66.