kan/examples

Nobuki Takayama

January 7,1995 : Revised, August 15, 1996;
Revised December 17, 1998.

Contents

1 About this document

The system `kan/sm1` is a Gröbner engine specialized especially to the ring of differential operators with a subset of Postscript language and an extension for object oriented programming. It is designed to be a back-end engine for a heterotic distributed computing system. However, it is not difficult to control `kan/sm1` directly. This document is a collection of programs for `kan/sm1` Version 2.xxxx. Since the system is still evolving, there is no comprehensive manual for the libraries of kan and the Postscript-like language `sm1`. However, all operators in `kan/sm1` are shortly explained in `onlinehelp.tex` in this directory and it will be enough once one understands the fundamental design of the system. This document provides introductory examples and explains the fundamental design of the system. If there are questions, please send an E-mail to the author (`kan@math.kobe-u.ac.jp`).

There are two design goals of `kan/sm1` .

1. Providing a backend engine in a distributed computing system for computations in the ring of differential operators.
2. Providing a virtual machine based on stacks to teach intermediate level computer science especially for mathematics students.

2 Getting started

To start the system, type in `sm1`. To quit the system, type in `quit`. You can make a program run in `kan/sm1` by the operator

```
(filename) run  ;
```

or

```
$filename$ run  ;
```

The two expressions `$xyz$` and `(xyz)` have the same meaning; they means the string `xyz`. The pair of brackets generates a string object. The dollar sign is used for a compatibility to `kan/sm1` Version 1.x.

There are three groups of functions. The first group is those of primitive operators. They are functions written in C. The second group is those of macro operators. They are functions written in `sm1` language and automatically loaded when the system starts. The third group is those of macro operators defined in the library files in `lib/` directory. These operators provide a user friendly interfaces of computing characteristic ideal, holonomic rank, b-function, annihilating ideal, hypergeometric differential operators, restrictions, de Rham cohomology groups. You can get a list of primitive operators and macros by `?` and `??` respectively. To see the usage of a macro, type in `(macro name) usage ;` . Note that you need a space before `;`. All tokens should be separated by the space or special characters `( ) [ ] { } $ %` . The help message usually provides examples. For example, the line `(add) usage` present the example

`Example: 2 3 add :: ==> 5` You may try the input line `2 3 add ::` and will get the output `5`. All printable characters except the special characters `( ) [ ] { } $ %` can be a part of a name of a macro or primitive operator. For example, `::` is a name of macro which outputs the top of the stack and the prompt.

`kan/sm1` is a stack machine. Any object that has been input is put on the top of the stack. Any operator picks up objects from the stack, executes computations and puts results on the stack. For example, the primitive operator `print` picks up one object from the stack and print it to the screen. If you type in `(Hello World) print`, then the string "Hello World" is put on the stack and the operator `print` picks up the string and print it. The macro `message` works like `print` and outputs the newline. The macro `::` is similar to `message`, but it also outputs the newline and the prompt; it picks up one object from the stack, print the object to the screen and output the prompt `sm1>`. For example, when you type in

```
(Hello World) ::
```

you get

```
Hello World
sm1>
```

We introduce two more useful stack operators.

`pop` Remove the top element from the stack.

`pstack` Print the contents of the entire stack.

You can use `kan/sm1` as a reverse Polish calculator; try the following lines.

```
11 4 mul ::
3 4 add /a set
5 3 mul /b set
a b add ::
```

Mathematical expressions such as `x^2-1` are not parsed by the stackmachine. The parsing is done by the primitive operator `.` (dot) in the current ring. For example, type in the following line just after you started `kan/sm1`

```
( (x+2)^10 ). ::
```

then you will get the expansion of $(x+2)^{10}$. `( (x+2)^10 )` is a string and is pushed on the stack. Next, the operator `.` parses the expression and convert it to an internal expression of the polynomial. Note that the given string is parsed in the current ring. In order to see the current ring, use the operator `show_ring`. Note that the polynomials in `sm1` means polynomials with the coefficients in a given ring such as $\mathbf{Z}$. So, `(x/3+2).` is *not accepted*.

A variable is defined by placing the variable's name, value and `def` operator on the stack of `kan/sm1` as in the following line:

```
/abc  23  def
```

The macro `set` is an alternative way to define a variable and set a value.

```
23 /abc set
```

means to set the value `23` to the variable `abc`.

In order to output an expression to a file, the macro `output` is convinient. For example, the lines

```
( (x+2)^10 ). /a set
a output
```

output the expansion of $(x+2)^{10}$ to the file `sm1out.txt`.

If you need to run a start-up script, modify the shell script `Doc/startsm1` and write what you need in the file `Doc/Sm1rc`.

The system `kan/sm1` is not designed for a heavy interactive use. So, unless you are a stackmachine fan, it is recommended to write your input in a file, for example, in `abc.sm1`, and execute your input as

```
sm1 -q <abc.sm1
```

Here is an example of an input file `abc.sm1`:

```
(cohom.sm1) run
[(y^2-x^3-2) (x,y)] deRham ::
```

The option `-q` is for not outputting starting messages.

We close this section with introducing some useful references.

For the reader who are interested in writing a script for `kan/sm1`, it is strongly recommended to go through Chapters 2 and 4 (stack and arithmetic, procedures and variables) of the so called "postscript blue book" [2]. The control structure of `Kan/sm1` is based on a subset of Postscript.

The book [4] is a nice introduction to compute D-module invariants with Gröbner bases. The book [5] is the latest book on this subject. This book explains the notion of homogenized Weyl algebra, which is the main ring for computations in `kan/sm1` . and algorithms for D-modules. As to an introduction to mathematical aspect of D-modules, Chapter 5 of [3] is recommended.

The latest information on `kan/sm1` and related papers are put on the http address [6].

3 Package files in the Doc/ (lib/) directory

A set of user friendly packages are provided for people who are interested in D-modules (D is the ring of differential operators), but are not interested in the aspect of `sm1` as a part of distributed computing system. Here is a list of packages.

1. `bfunction.sm1` : Computing b-functions. This script is written by T.Oaku.

2. `factor-a.sm1` : A sample interface with `risa/asir` [1] to factor given polynomials.

3. `hol.sm1` : A basic package for holonomic systems (Gröbner basis and initial ideals, holonomic rank, characteristic variety, annihilating ideal of f^s).

4. `gkz.sm1` : Generate GKZ system for a given A and b.

5. `appell.sm1` : Generate Appell hypergeometric differential equations.

6. `cohom.sm1` : An experimental package for computing restrictions and de Rham cohomology groups mainly written by T.Oaku.

7. `kanlib1.c` : An example to explain an interface between kan and C-program. Type in "make kanlib1" to compile it.

8. `ox.sm1` : A package for communication based on the open XM protocol. The open sm1 server `ox_sm1` can be obtainable from the same ftp cite of `kan/sm1`. See `http://www.math.kobe-u.ac.jp/openxxx` for the protocol design.

9. `oxasir.sm1` : A package to use open asir server based on the open XM protocol. Open asir server `ox_asir` will be distributed from [1]. The package `cohom.sm1` (`deRham`) and `annfs` need this package to analyze the roots of b-functions. The built-in function to analyze the roots is slow. The open asir server and `oxasir.sm1` should be used for efficient analysis of the roots of b-functions. See the usage of `oxasir` for the latest information.

10. `intw.sm1` : Compute 0-th integration of a given D-module by using a generic weight vector.

See the section three of `onlinehelp.tex` for more informations.

3.1 Examples: `gb, rrank, gkz, bfunction, deRham`

Execute `Loadall` to load packages before executing examples. `Dx` means ∂_x.

Example 1 Compute a Gröbner basis and the initial ideal with respect to the weight vector $(0,0,1,1)$ of the D-ideal

$$D \cdot \{(x\partial_x)^2 + (y\partial_y)^2 - 1, xy\partial_x\partial_y - 1\}.$$

See [5] on the notion of Gröbner basis and the initial ideal with respect to a weight vector.

```
[ [( (x Dx)^2 + (y Dy)^2 -1) ( x y Dx Dy -1)] (x,y)
            [ [ (Dx) 1 (Dy) 1] ] ] gb pmat ;
```

Output:

```
[
 [ x^2*Dx^2+y^2*Dy^2+x*Dx+y*Dy-1 , x*y*Dx*Dy-1 , y^3*Dy^3+3*y^2*Dy^2+x*Dx ]
 [ x^2*Dx^2+y^2*Dy^2 , x*y*Dx*Dy , y^3*Dy^3 ]
]
```

The first line is the Gröbner basis and the second line is a set of generators of the initial ideal with respect to the weight vector $(0, 0, 1, 1)$. In order to get syzygies, use `syz`.

Example 2 Generate the GKZ system for $A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & 4 \end{pmatrix}$ and $\beta = (1, 2)$. Here, the GKZ system is a holonomic system of differential equations introduced by Gel'fand, Kapranov and Zelevinsky. The system is also called $\mathcal{A}$-hypergeometric system.

```
[ [[1 1 1 1] [0 1 3 4]] [1 2]] gkz  ::
```

Output:

```
[ x1*Dx1+x2*Dx2+x3*Dx3+x4*Dx4-1 , x2*Dx2+3*x3*Dx3+4*x4*Dx4-2 ,
  Dx2*Dx3-Dx1*Dx4 , -Dx1*Dx3^2+Dx2^2*Dx4 , Dx2^3-Dx1^2*Dx3 ,
  -Dx3^3+Dx2*Dx4^2 ]
```

Example 3 Evaluate the holonomic rank of the GKZ systems for $A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & 4 \end{pmatrix}$ and $\beta = (1, 2)$ and $\beta = (0, 0)$. Show also the time of the execution.

```
{ [ [[1 1 1 1] [0 1 3 4]] [1 2]] gkz  rrank ::} timer
{ [ [[1 1 1 1] [0 1 3 4]] [0 0]] gkz  rrank ::} timer
```

Output:

```
   5
User time: 1.000000 seconds, System time: 0.010000 seconds, Real time: 1 s
   4
User time: 1.320000 seconds, System time: 0.000000 seconds, Real time: 1 s
```

Example 4 Compute the b-function of $f = x^3 - y^2 z^2$ and the annihilating ideal of f^{r_0} where r_0 is the minimal integral root of the b-function.

```
(oxasir.sm1) run
[(x^3 - y^2 z^2) (x,y,z)] annfs /ff set
ff message
ff 1 get 1 get fctr ::
```

Output:

```
[  [ -y*Dy+z*Dz , 2*x*Dx+3*y*Dy+6 , -2*y*z^2*Dx-3*x^2*Dy ,
   -2*y^2*z*Dx-3*x^2*Dz , -2*z^3*Dx*Dz-3*x^2*Dy^2-2*z^2*Dx ]  ,
 [-1,-139968*s^7-1119744*s^6-3802464*s^5-7107264*s^4-7898796*s^3-5220720*s^2-1900500*s-294000]]
[[ -12 , 1 ] , [ s+1 , 1 ], [3*s+5 , 1], [ 3*s+4, 1], [6*s+7, 2], [6*s+5, 2]]
```

The first two rows of the output give generators of the annihilating ideal of $(x^3 - y^2 z^2)^{-1}$. The b-function is $(s + 1)(3s + 5)(3s + 4)(6s + 7)^2(6s + 5)^2$ and -1 is the minimal integral root.

Example 5 Compute the de Rham cohomology group of $X = \mathbf{C}^2 \setminus V(x^3 - y^2)$.

```
(cohom.sm1) run
[(x^3-y^2) (x,y)] deRham ;
```

Output:

```
 0-th cohomology:  [    0 , [   ]  ]
 -1-th cohomology:  [    1 , [   ]  ]
 -2-th cohomology:  [    1 , [   ]  ]
[1 , 1 , 0 ]
```

This means that $H^2(X, \mathbf{C}) = 0$, $H^1(X, \mathbf{C}) = \mathbf{C}^1$, $H^0(X, \mathbf{C}) = \mathbf{C}^1$.

Example 6 Compute the integral of $I = D \cdot \{\partial_t - (3t^2 - x), \partial_x + t\}$, which annihilates the function $e^{t^3 - xt}$, with respect to t.

```
(cohom.sm1) run
[ [(Dt - (3 t^2-x)) (Dx + t)] [(t)]
  [ [(t) (x)] [ ]] 0] integration
```

Output:

```
[    [    1 , [    3*Dx^2-x ]  ]  ]
```

4 Data types

Each object in `sm1` has a data type. Here is a list of main primitive data types, which are common to other languages except the type polynomial and the type ring.

null

integer(machine integer), 32 bit integer. Example: `152`

literal, literal. Example: `/abc`

string, string. Example: `(Hello)`

executableArray, program data. Example: { `add 2 mul` }

array, array, Example: `[(abc) 5 ]`

polynomial, polynomial, Example: `(x^2-1).`

ring, ring definition.

number(universalNumber), Big num. Example: `(123).. 456 power`

class, Class.

4.1 Array

Array is a collection of one dimensional objects surrounded by square brackets and indexed by integers (machine integers) $0, 1, 2, \ldots$. Elements of any array may be arrays again, so we can express list structures by using arrays. An array is constructed when the `sm1` encounters the right square bracket. Note that square brackets are also operators. Thus, the line

```
[(Hello) 2  50  add]
```

sets up an array

```
[(Hello)  52]
```

where the 0-th element of the array is the string `(Hello)` and the 1-th element is the integer `52`. The `put` and `get` operator store and fetch an element of an array. The `get` operator takes an array and an index from the stack and returned the object indexed by the second argument. The line

```
[(sm1)  12  [(is) (fun)] 15]   2    get
```

would return the array `[(is) (fun)]` on the stack. The `put` operator takes an array, an index `i`, an object from the stack and store the object at the i-th position of the array. That is,

```
/a [(sm1) (is) (fun)] def
a 2 (a stackmachine) put
```

would rewrite the contents of the variable `a` as

```
[(sm1) (is)  (a stackmachine)]
```

4.2 Ring

The ring object is generated by the operator `define_ring`. This operator has a side effect; it also changes the *current ring*. The line

```
[(x,y)  ring_of_differential_operators 0] define_ring /R set
```

would create the ring of differential operators

$$\mathbf{Z}\langle x, y, \partial_x, \partial_y \rangle,$$

store it in the variable `R` and changes the current ring to this Weyl algebra. ∂_x is denoted by `Dx` on `sm1`. The suffix `D` can be changed; for example, if you want to use `dx` instead of `Dx`, execute the command `/@@@.Dsymbol (d) def` The current ring can be obtained by `[(CurrentRingp)] system_variable` . The current ring is the ring of polynomials of two variables x, h when the system starts.

All polynomial except 0 belongs to a ring. For a non-zero polynomial `f`, the line

```
f (ring)  dc /rr set
```

put the associated ring object of `f` to the variable `rr`. As we have seen before, a given string is parsed as a polynomial in the current ring by the operator "`.`". To parse in a given ring, the operator "`__`" is used. That is,

```
[(x,y)  ring_of_differential_operators 0] define_ring /R set
(x^2-y) R  __  /f  set
```

means to parse the string `x^2-y` in the ring `R` and put the polynomial $x^2 - y$ in the variable `f`. Arithmetic operators for two polynomials can be performed only when the two polynomials belong to a same ring. If you want to map a polynomial to a different ring, the easiest way is to translate the polynomial into a string and parse it in the ring. That is,

```
[(x,y) ring_of_polynomials 0] define_ring /R1 set
(x-y). /f set
[(x,y,z) ring_of_differential_operators 0] define_ring /R2 set
(y+Dz). /g set
f toString . /f set
f g add ::
```

would output $(x - y) + (y + Dz) = Dz$.

It is convinient to have a class of numbers that is contained in any ring. The datatype number (universalNumber) is the class of bignum, which is allowed to be added and multiplied to any polynomials with characteristic 0.

4.3 Tag

Each object of `kan/sm1` has the tag expressed by an integer. The tag expresses the class to which the object belongs. You can see the tag of a given object by the operator `tag`. For example, if you type in

```
10 tag ::
```

then you get the number 1. If you type in

```
[ 1  2]  tag ::
```

then you get the number 6. The number 1 is the tag of the integer objects and the number 6 is the tag of the array objects. These tag numbers are stored in the variables `IntegerP` and `ArrayP`. In order to translate one object to that in a different class, there is the operator `data_conversion` or `dc`. For example,

```
(10). (integer) dc  ::
```

translates the polynomial 10 in the current ring into the integer 10 and

```
(10). (string) dc  ::
```

translates the polynomial 10 into the string 10.

5 Gröbner basis and Syzygy computation in `kan/sm1`

5.1 Computing Gröbner or standard basis in the ring of the polynomials

Example 7 *Obtain the Gröbner basis of the ideal generated by the polynomials* $x^2 + y^2 - 4$ *and* $xy - 1$ *in terms of the graded reverse lexicographic order :*

$$1 < x < y < x^2 < xy < y^2 < \cdots.$$

All inputs must be homogenized to get Gröbner basis by the command `groebner`. Usually, the variable h is used for the homogenization. In this example, Gröbner bases in $\mathbf{Q}[x, y, h]$ are computed, but rational coefficients in the input is not allowed. All coefficients must be integers.

The operator `groebner_sugar` is for non-homogenized computation of Gröbner basis.

The following code is a convinient template to obtain Gröbner bases.

```
% gbrev.sm1
[
 (y,x)  % Change here. Declare the variables that you use.
 ring_of_polynomials
 0  % Change here. Define the characteristic here.
] define_ring

/inputp
 [                    % Polynomials must be enclosed by ( and  ). or $ and $.
  ( x^2+y^2-4*h^2 ). % Change here. Write a polynomial for input.
  (x*y-h^2).          % Change here. Write a polynomial for input.
 ]
def

[inputp] groebner /ans set
ans dehomogenize ::
```

The letters after the symbol % are ignored by `kan/sm1` ; comments begin with the symbol %. If one needs to compute Gröbner basis of a given set of polynomials, one may only change the lines marked by the comment `% Change here`.

Grammer 1 *Any string of alphabets can be used as a name of a variable except* `h`*,* `E`*,* `H` *and* `e_`*. For q-difference operators,* `q` *is also reserved. Upper and lower case letters are distinct.*

Example 8 *Obtain the Gröbner basis of the ideal generated by the polynomials* $x^2 + y^2 - 4$ *and* $xy - 1$ *in terms of the lexicographic order :*

$$1 < x < x^2 < x^3 < \cdots < y < yx < yx^2 < \cdots.$$

```
% gblex.sm1
[
 (y,x)  % Change here. Declare the variables that you use.
 ring_of_polynomials
 [[(y) 1] [(x) 1]] weight_vector
 0  % Change here. Define the characteristic here.
] define_ring

/inputp
 [                    % Polynomials must be enclosed by ( and  ). or $ and $.
  ( x^2+y^2-4 ).      % Change here. Write a polynomial for input.
  ( x*y-1 ).          % Change here. Write a polynomial for input.
 ]
def

[inputp] groebner_sugar /ans set
ans ::
```

In this example, the order is specified by the weight vector. If the line `[vec1 vec2 ...] weight_vector` is given in the definition of the ring, monomials are compared by the weight vector `vec1`. If two monomials have the same weight, then they are compared by the weight vector `vec2`. This procedure will be repeated until all weight vectors are used.

The weigth vector is expressed in the form `[v1 w1 v2 w2 ... vp wp ]`, which means that the variable `v1` has the weight `w1`, the variable `v2` has the weight `w2`, ... For example, when the ring is defined by

```
  [(x,y,z) ring_of_polynomials
   [[(x) 1 (y) 3]  [(z) -5]] weight_vector 0]
  define_ring
```

two monomials $x^a y^b z^c \succ x^A y^B z^C$ if and only if $a + 3b > A + 3B$ or ($a + 3b = A + 3B$ and $-5c > -5C$) or ($a + 3b = A + 3B$ and $-5c = -5C$ and $(a, b, c) \succ_{grlex} (A, B, C)$) where $\succ_{grlex}$ denotes the graded reverse lexicographic order.

The Buchberger's criterion 1 is turned off by default, because it does not work in case of the ring of differential operators. To turn it on, input `[(UseCriterion1) 1] system_variable`

The operator `groebner` outputs the status of degree by degree computation of Gröbner basis. To turn off this message, input `[(KanGBmessage) 0] system_variable`

Example 9 *Obtain the Gröbner basis of the ideal generated by the polynomials*

$$x^2 + y^2 + z^2 - 1, xy + yz + zx - 1, xyz - 1$$

in terms of the elimination order $x, y > z$.

```
% gbelim.sm1
[
 (x,y,z)  % Change here. Declare the variables that you use.
 ring_of_polynomials
 [[(x) 1 (y) 1]] weight_vector
 0  % Change here. Define the characteristic here.
] define_ring

/inputp
 [                    % Polynomials must be enclosed by ( and  ). or $ and $.
  ( x^2+y^2+z^2-1 ). % Change here. Write a polynomial for input.
  ( x*y+y*z+z*x-1 ). % Change here. Write a polynomial for input.
  ( x*y*z-1 ).       % Change here. Write a polynomial for input.
 ]
def

[inputp] groebner_sugar /ans set
ans dehomogenize ::
```

5.2 Computing Gröbner basis in the ring of differential operators

Example 10 *Obtain the Gröbner basis of the ideal in the Weyl algebra*

$$\mathbf{Q}\langle x, y, \partial_x, \partial_y \rangle, \quad \textit{where } \partial_x = \frac{\partial}{\partial x}, \partial_y = \frac{\partial}{\partial y}$$

generated by the differential operators

$$x\partial_x + y\partial_y, \partial_x^2 + \partial_y^2$$

in terms of the elimination order $\partial_x, \partial_y > x, y$ by using the homogenized Weyl algebra.

```
%% gbdiff.sm1

[ (x,y) ring_of_differential_operators
  [[(Dx) 1 (Dy) 1]] weight_vector
 0
] define_ring

[ (x Dx + y Dy).
  (Dx^2 + Dy^2).
] /ff set

ff { [[(h). (1).]] replace homogenize} map /ff2 set

[ff2] groebner dehomogenize ::
```

5.3 Computing Gröbner basis in R^n

Example 11 *Let S be the ring of polynomials $Q[x, y]$. Obtain the Gröbner basis of the S-submodule of S^3 generated by the vectors*

$$(x-1, y-1, z-1), (xy-1, yz-2, zx-3).$$

```
%% gbvec.sm1

[ (x,y,z) ring_of_polynomials [[(x) 1 (y) 1 (z) 1]] weight_vector 0]
define_ring

[ [(x-1).        (y-1).         (z-1).] homogenize
  [(x y - 1).  (y z - 2). (z x - 3).]   homogenize ] /ff set

[ff] groebner {toVectors dehomogenize} map ::
```

5.4 Computing syzygies

Let R be a ring and $f_1, \ldots, f_m$ be elements of R. The left R-module

$$\{(s_1, \ldots, s_m \in R^m \mid \sum_{i=1}^{m} s_i f_i = 0\}$$

is called the syzygy among $f_1, \ldots, f_m$. The following script computes the generators of the syzygy among

$$x\partial_x + y\partial_y, \partial_x^2 + \partial_y^2$$

in the homogenized Weyl algebra.

```
%% syz.sm1

[ (x,y) ring_of_differential_operators
  [[(Dx) 1 (Dy) 1]] weight_vector
 0
] define_ring

[ (x Dx + y Dy).
  (Dx^2 + Dy^2).
] /ff set

ff { [[(h). (1).]] replace homogenize} map /ff2 set

[ff2 [(needBack) (needSyz)]] groebner /ans set ;
(Syzygies are ...) message
ans 2 get ::
```

The 0-th element of `ans` is the Gröbner basis. The 1st element of `ans` is the transformation matrix from the input to the Gröbner basis. The 2nd element of `ans` is a set of generators of the syzygies of the input.

6 Control Structures and programming

6.1 if

The conditional operator `if` requires three objects on the stack: an integer value and two executable arrays, which are program data. The first executable array will be executed if the integer value is not 0. The second executable array will be executed if the integer value is 0. For example, the program line

```
1 { op1 } {op2} ifelse
```

executes { `op1` } and the program line

```
0 { op1 } {op2} ifelse
```

executes { `op2` }.

Here is a list of comparison operators.

`eq`	=	Example: `[1 2] [1 3] eq`
`gt`	>	Example: `3 2 gt`
`lt`	<	Example: `3 2 lt`

`not` Example: `3 2 eq not`

`and` Example: `3 2 eq 5 6 lt and`

`or` Example: `3 2 eq 5 6 lt or`

6.2 for

The `for` operator implements a counting loop. This operator takes three integers and one executable array:

```
i0  d  i1  { ops } for
```

`i0` is the loop counter's starting value, `d` is the increment amount, `i1` is the final value. The `for` operator put the value of the counter on the stack before each execution of `ops`. For example, the program line

```
1 1 5  { /i set i message } for
```

outputs

```
1  2   3   4   5
```

6.3 `map` function

`map` function is used to apply an operator to each element of a given array. For example, the following line is used to translate each polynomial of the given array `aa` into the corresponding string

```
/aa [( (x-1)^2 ). (2^10).] def
aa { (string) dc } map /ff set ;
ff ::
```

It becomes easier to writing script for `kan/sm1` by using the `map` function.

6.4 Function definition

Programs are stored in executable arrays and the curly brackets generate executable arrays. For example, if you input the line

```
{  add 2 mul }
```

then the executable array object which represents the program "take two elements from the stack, add them, and multiply two and put the result on the stack" will be store on the top of the stack. You can bind the program to a name. That is,

```
/abc { add 2 mul } def
```

binds the executable array to the variable `abc`. The input `2 4 abc ::` outputs 12. When `sm1` encounters the name `abc`, it looks up the user dictionary and finds that the value of `abc` is the executable array `{ add 2 mul }` . The executable array is loaded to the stack machine and executed.

Funtions can be defined by using executable arrays. Here is a complete example of a function definition in `sm1` following standard conventions.

```
/foo {
  /arg1 set
  [/n /i /ans] pushVariables
  [
    /n arg1 def
    /ans 0 def
    1 1 n {
      /i set
      /ans ans i add def
    } for
    ans /arg1 set
  ] pop
  popVariables
  arg1
} def
```

The function returns the sum $1+2+\cdots+n$. For example, `100 foo ::` outputs 5050. The arguments of the function should firstly be stored in the variables `arg1`, `arg2`, ... It is a convension in `sm1` programming. The local variables are declared in the line

```
[/n /i /ans] pushVariables
```

The macro `pushVariables` stores the previous values of `n`, `i`, `ans` on the stack and the macro `popVariables` restores the previous values. So, you can use `n`, `i`, `ans` as a local variable of this function. The function body should be enclosed as

```
[

] pop
```

It is also a convension in `sm1` programming to avoid unmatched use of `pushVariables` and `popVariables`.

Example 12 `cv0.sm1` is a script to compute characteristic variety for D-submodules in D^n.

`cv2.sm1` is a script to compute the multiplicites of D-modules.

7 Dictionaries and contexts

The `def` or `set` operators associate a key with a value and that key-value pair is stored in the current dictionary. They key may starts with any printable character except `( ) [ ] { } $ %` and numbers and be followed by any printable characters except the special characters. For example,

```
foo  test   Test!   foo?59
```

are accepted as names for keys.

A key-value pair is stored in the current dictionary when you use the operator `def` or the operator `set`. For example, when you input the line

```
/foo  15   def
```

then the key-value pair (`foo`, 15) is stored in the current dictionary. We can generate several dictionaries in `sm1`. Each dictionary must have its parent dictionary. When you input a token (key) that is not a number or a string or a literal, `sm1` looks up the current dictionary to find the value of the key. If the value is an executable array, then it will be executed. If the value is not an executable array, then the value is put on the stack as an object. If the looking-up fails, then it tries to find the value in the parent dictionary. If it fails again, then it tries to find the value in the grandparent dictionary and so on. This mechanism enables us to write an object oriented system. When the system starts, there are two dictionaries: primitive dictionary and the standard user dictionary. For example, the input `?` makes `sm1` to look up the standard user dictionary and `sm1` finds the value of `?`, which is an executable array that displays all keys in the primitive dictionary.

A new dictionary can be created by the operator `newcontext`. Here is an example of creating a new dictionary.

```
/abc { (Bye) message } def
/aaa 20 def
abc  aaa ::
```

The key-value pairs (`abc`, `{ (Bye) message }` and (`aaa`, `20`) are stored in the current dictionary (`StandardContextp`). Here is the output from the system.

```
Bye
20
```

```
(mycontext) StandardContextp newcontext /nc set ;
nc setcontext ;
```

Create a new dictionary and change the current dictionary by `setcontext`.

```
/abc { (Hello) message } def ;
abc aaa ::
```

Store a new key-value pair in the new dictionary. Here is the output of the system.

```
Hello
20
```

The key `abc` was found in the current dictionary, so the system outputs `Hello`. The key `aaa` was not found in the current dictionary, so the system looked for it in the parent dictionary and outputs the value `20`.

It is sometimes preferable to protect the key-value pairs from unexpected rewriting. If you input the following lines, then all pairs in the current dictionary except `arg1`, `arg2`, `arg3`, `v1`, `v2`, `@.usages` will become readonly pairs.

```
[(Strict2) 1] system_variable  %% from var.sm1
[(chattrs) 2] extension
[(chattr) 0 /arg1] extension
[(chattr) 0 /arg2] extension
[(chattr) 0 /arg3] extension
[(chattr) 0 /v1] extension  %% used in join.
[(chattr) 0 /v2] extension

[(chattr) 0 /@.usages] extension
```

8 Using `sm1` to teach computer science for students in mathematics

There are two design goals in `sm1`. One goal is to provide a backend engine for the ring of differential operators in a heterotic distributed computing system. Another interesting design goal is to help to teach basics of intermediate level computer science quickly and invite students to mathematical programmers' world. It is a fun to learn computer science with `sm1`! Here are some topics that I tried in class rooms. These are intermediate level topics that should be learned after students have learned elementary programming by languages like Pascal, C, C++, Java, Basic, Mathematica, Maple, Macaulay 2, etc.

8.1 Recursive call and the stack

The notion of stack is one of the most important idea in computer science. The notion of recursive call of functions is usually taught in the first course of programming. I think it is important to understand how the stack is used to emulate recurisve calls. The idea is the use of the stack. Function arguments and local variables are stored in the stack. It enables the system to restore the values of the local variables and arguments after an execution of the function. However, it should be noted that, for each function call, the stack dynamically grows.

As an example that I used in a class room, let us evaluate the n-th Fibonacci number defined by

$$f_n = f_{n-1} + f_{n-2},\ f_1 = f_2 = 1$$

by using a recurisive call.

```
/fib {
  /arg1 set
  [/n /ans] pushVariables
  pstack
    /n arg1 def
    (n=) messagen n message
    (-----------------------------) message
    n 2 le {
      /ans  1  def
    }
    {
      n 1 sub fib  n 2 sub fib add /ans set
    } ifelse
    /arg1 ans def
  popVariables
  arg1
} def
```

The program would return the n-th Fibonacci number. That is, `4 fib ::` would return $f_4 = 3$. It also output the entire stack at each call, so you can observe how stack grows during the computation and how local variables `n`, `ans` are stored in the stack. You would also realize that this program is not efficient and exhausts huge memory space.

8.2 Implementing a Java-like language

One of the exciting topic in the course of computer science is mathematical theory of parsing. After learning the basics of the theory, it is a very good Exercise to design a small language and write a compiler or interpreter for the language. If you do not like to write a compiler for real CPU, the stackmachine `sm1` will be a good target machine. For example, the language may accept the input

```
12345678910111213*(256+2)
```

and the interpreter or the compiler generate the following code for `sm1`

```
(12345678910111213)..
(256)..
(2).. add
 mul message
```

One can easily write an arbitrary precision calculator by using `sm1` and also try algorithms in the number theory by one's own language.
Exercise 1: parse a set of linear equations like `2x+3y+z = 2; y-z =4;` , output the equation in the matrix form and find solutions.
Exercise 2: Modify the calculator `hoc` so that it can use `sm1` as the backend

engine. The calculator `hoc` is discussed in the book: Kerningham and Pike, Unix programming environment.

The stackmachine `sm1` provides a very strong virtual machine for object oriented system by the dictionary tree. We can easily implement a language, on which Java-like object oriented programming mechanism is installed, by using `sm1`. Here is a sample program of `kan/k0`, which is an object oriented language and works on `sm1`. I taught a course on writing mathematical softwares in a graduate school with `k0`.

```
class Complex extends Object {
  local re, /* real part */
        im; /* imaginary part*/
  def new2(a,b) {
    this = new(super.new0());
    re = a;
    im = b;
    return(this);
  }
  def real() { return(re); }
  def imaginary() { return(im); }
  def operator add(b) {
    return( new2(re+b.real(), im+b.imaginary()) );
  }
  def operator sub(b) {
    return( new2(re-b.real(), im-b.imaginary()) );
  }
  def operator mul(b) {
    return(new2( re*b.real()-im*b.imaginary(), re*b.imaginary()+im*b.real()));
  }
  def operator div(b) {
    local den,num1,num2;
    den = (b.real())^2 + (b.imaginary())^2 ;
    num1 = re*b.real() + im*b.imaginary();
    num2 = -re*b.imaginary()+im*b.real();
    return(new2(num1/den, num2/den));
  }

  def void show() {
    Print(re); Print(" +I["); Print(im); Print("]");
  }
  def void showln() {
    this.show(); Ln();
  }
}


 a = Complex.new2(1,3);
```

```
 a:
1 +I[3]
 a*a:
-8 +I[6]
```

8.3 Interactive distributed computing

The plugin modules file2, cmo, socket and the package file `ox.sm1` provide functions for interactive distributed computing. To install these plugin modules, compile `sm1` after modifying `kan/Makefile`. See `README` for details. These plugins are already installed in the binary distributions of `sm1`. The sm1 server `ox_sm1` and `ox` which are complient to the Open XM protocol (see [7]) is distributed from the same ftp cite with `sm1`. The sm1 server is also a stack machine. Here is an example input of server and client computation.
Server:

```
./ox -ox ox_sm1 -data 1300 -control 1200
```

Client:

```
(ox.sm1) run
[(localhost) 1300 1200] oxconnect /oxserver set
/f (123).. def ;
oxserver f oxsendcmo ;     %% send the data f to the server
oxserver f oxsendcmo ;     %% send the data f to the server
oxserver (power) oxexec ;  %% execute f f power
oxserver oxpopcmo ::       %% get data from the server.
```

The output is 123^{123} and equal to 114374367....9267.
Exercise: write a graphical interface for functions in packages of `sm1` by Java and call sm1 server to execute them.

8.4 More exercises

1. `kan/sm1` contains the GNU MP package for computations of bignumbers. You can call the functions in GNU MP by the operator `mpzext`. Write a program to find integral solution (x, y) of $ax + by = d$ for given integers a, b, d.

2. Write a program for RSA encryption.

References

[1] Risa/Asir — computer algebra system, `ftp://endaevor.fujitsu.co.jp/pub/isis/asir`.

[2] PostScript — Language Turorial and Cookbook, (1985), Addison-Wesley

[3] R.Hotta, Introduction to Algebra, Asakura-shoten, Tokyo (in Japanese).

[4] T.Oaku, Gröbner basis and systems of differential equations, (1994) Seminor note series of Sophia University. (in Japanese).

[5] M.Saito, B.Sturmfels, N.Takayama, Gröbner deformations of hypergeometric differential equations, to appear from Springer.

[6] `http://www.math.kobe-u.ac.jp/KAN` and
`http://www.math.kobe-u.ac.jp/~taka`

[7] `http://www.math.kobe-u.ac.jp/openxxx`