ПРИМЕРЫ ПРОСТЫХ ПРОГРАММ НА ЯЗЫКЕ VKACL.

Кон В. Г., http://kohnvict.narod.ru kohnvict@yandex.ru **30-06-2025**

Содержание

Пример #8. Запись картинок для разных вариантов карты цветов. Стр. 1
Пример #7. Создание новой карты цветов в виде природной радуги. Стр. 2
Пример #6. Анимация показа матрицы чисел по строкам и вычисление матрицы Стр. 3
Пример #5. Как использовать готовые программы в новой программе Стр. 5
Пример #4. Запуск команд операционной системы из программы Стр. 6
Пример #3. Картинка ждет реакции пользователя. Можно все заблокировать. Стр. 8
Пример #2. Проверка аппроксимации квадрата модуля интеграла. Стр. 8
Пример #1. Программа калькулятора на 8 переменных и с функциями. Стр. 11

Введение

В данном документе номера примеров идут в обратном порядке. Не трудно поменять номера, но они показывают хронологию написания. То есть первый номер был написан раньше второго и так далее. Но последние номера я ставлю в самое начало текста. Так как документ дополняется время от времени, то тем, кто уже читал про предыдущие примеры, удобнее сразу читать последний пример. Код примеров записывается сплошным текстом, то есть без разметки на строки и отступы. Но каждый может разметить его как ему нравится. Также полезно знать, что язык vkacl условно делится на три части.

Первая часть – это команды языка, то есть его основа. Справочник по всем командам можно посмотреть по этой ссылке https://kohnvict.ucoz.ru/acl/acl-info.htm .

Вторая часть – это стандартные процедуры, представляющие собой комбинации команд языка, которые часто используются и которые удобно записывать просто номером. Я их называю суперкомандами. Справочник по всем суперкомандам можно посмотреть по этой ссылке https://kohnvict.ucoz.ru/acl/acls-info.htm. Вторая ссылка отличается от первой только одной буквой.

Третья часть – это готовые программы на языке vkacl. Они содержат код, который часто тоже нет нужды переписывать и есть механизм удобно вставлять готовые программы в новые программы, которые еще создаются. Справочник по всем готовым программам можно посмотреть по этой ссылке https://kohnvict.ucoz.ru/acl/aclp-info.htm. Она тоже отличается от предыдущих одной буквой.

Пример #8.

Для изображения двумерных матриц можно использовать разные карты цветов. Сами карты на графиках изображаются разноцветными полосками как картинки матрицы линейно меняющейся по первой оси и постоянной по второй оси. Иногда бывает необходимо показать одну или несколько таких полосок в виде картинки. В этом примере показан код программы, которая это делает.

```
#f [op=fold; file=pro/1;]  # J=1; j=J; x=0; n=800; d=1/(n-1); #rep n # r(j)=x; j=j+1; x=x+d; #end
# j=J+n; #rep 3 #pas n r(J) r(j)  # j=j+n; #end  # k=0; m=8;
#rep m # &=k<4; #case 4 # NP=k-2; ##42 #end |  #d 6 A J n 4 -1100 19 1 #p [pa=k<3; em=0;] #pr cm0\I1 k;|-\E ##2
# k=k+1; #end  # NP=1; ##42 #end |
```

Здесь в 1-й строке определяется рабочая папка, затем массив из 800 значений с линейной функцией. Это можно было бы сделать одной командой без цикла, но цикл небольшой и выполняется быстро. Затем этот массив 3 раза копируется дальше по индексам и получается матрица (800,4), удобная для показа карты цветов. Затем переменная k определяется как начальный номер карты. Можно задавать значения от 0 до 8. А переменная m указывает сколько карт будет показано. Это входные данные для рассматриваемой программы, их можно менять. Так как сейчас есть всего 8 карт, то сумма k+m не может быть больше 8.

Затем делается цикл и в нем необходимо учесть, что для первых 4-х карт подмена файла colmap.txt не требуется. А вот для остальных карт ее нужно делать. Поэтому если k равно 4 или больше, то такую подмену надо делать. Она делается суперкомандой (СК) с номером 42. После этого записываются входные данные СК с номером 2 и запускается сама эта СК. После цикла надо восстановить стандартный файл палитры, который записан с номером 1 в коллекцию внешних карт цветов.

Пример #7.

Двумерные матрицы часто изображают в виде картинки из разных цветов. При этом картинка точно отображает значения аргументов двумерной функции или индексов матрицы, а значения матрицы очень приближенно указываются цветом. Соответствие цвета значению матрицы называется палитрой или картой цветов. Карта цветов, фактически тоже является картинкой, но для матрицы, которая линейно зависит от первого индекса и не зависит от второго индекса. Рядом с этой картинкой рисуется ось значений матрицы с числами и соответствие цвета числам как раз и определяет значение конкретного цвета.

Карты цветов могут быть самые разные. Самая примитивная карта – это просто градиент от черного к белому цвету и наоборот. Но могут быть и более сложные цветные карты. В программе vkacl есть три встроенные карты и одна карта внешняя, которая записана в файл с именем colmap.txt в главной папке программы. Но таких карт может быть много. Если необходимо использовать другую внешнюю карту, то временно необходимо скопировать ее в файл с указанным именем, а потом все вернуть назад.

В данном примере рассмотрено как организована карта цветов и как определить новую карту на примере карты rainbow (радуга). Имеется в виду реальная радуга, которую можно видеть на небе и не только там. В программе vkacl карта цветов задается тремя массивами размером 256 чисел для значений красной (R), зеленой (G) и синей (B) компонент цвета. Сами значения изменяются от 0 до 255 (то есть одним байтом). Но значения записываются как целые числа по 32 числа в строке, всего 24 строки.

В радуге выделяют 7 цветов. Условно они называются Красный (232, 20, 22), Оранжевый (255, 165, 0), Желтый (250, 235, 54), Зеленый (121, 195, 20), Синий (72, 125, 231), Индиго (75, 54, 157), Фиолетовый (112, 54, 157). Числовые компоненты цветов я просто нашел в интернете. Можно пробовать и другие числа, близкие к этим. Тогда получится другая карта цветов. Часто рассматривают и такие радуги, которые реально в природе не существуют.

Будем использовать следующий алгоритм. Определим матрицу (7,3) из указанных значений. Затем проинтерполируем эту матрицу на размер (256,3) и напечатаем в файл целыми числами по указанной выше схеме. Код такой программы написан ниже.

```
#f [op=fold; file=pro/hdata;]  # J=99; J1=J+21;
#d 21 r(J) 232 255 250 121 72 75 112   20 165 235 195 125 54 54   22 0 54 20 231 157 157  #d 8 r(1) 0 1 0 1 0 1 0 1
#ma [op=mmi; n=1; b=J; nx=7; ny=3; sav=J1; wid=256; hei=3; em=0; file=here; form=here;]  # j=J1; s(3)=1;
#rep 24 #rep 32 #pr \I4 r(j);\E # j=j+1; #end  #pr \n\E #end  #io [op=wt; fir=t(1); n=s(3)-1; file=cm3.txt;]
#io [op=rf; fir=r(J1); n=256*3; form=*0;]  #d 7 A J1 256 3 0 255 247 1100 #pr ../1/test|Figure\E ##1
```

Здесь в 1-й строке определяется рабочая папка и начала двух массивов. Во 2-й строке определяется первый массив из 21 значений. В 3-й строке проводится интерполяция этого массива как (7,3) на размер (256,3) и готовятся параметры для печати. В 4-й строке в двумерном цикле печатаются значения новой матрицы в 24 строки из 32 значений и полученный текст записывается в файл. В 5-й строке этот файл прочитывается и рисуются три кривые. Это делается исключительно для контроля проведенной операции. Сам результат уже получен в 4-й строке.

Пример #6.

Данный пример показывает как можно относительно просто вычислять массивы стандартных, функций, выполнять нестандартные расчеты и делать анимацию простым вызовом суперкоманды. Рассматривается следующая задача. Пусть у нас есть быстрые осцилляции на фоне медленных осцилляций. Например, квадраты синуса с разными периодами. Необходимо вычислить функцию, равную произведению квадратов двух синусов с разными периодами. Затем усреднить эту функцию, заменяя значение в каждой точке на среднее значение по некоторому числу k ближайших точек, причем число k будет постепенно возрастать, начиная от 1 и с шагом i .

В результате получается матрица, у которой по строкам число точек функции n, а по столбцам число m+1 разных значений k . Анимация должна показать как функция меняется при увеличении числа k . Я сразу покажу весь код этой программы, а потом раскажу как он работает.

```
# i=4; n=860; m=24; t=9; n1=6; n2=90;
# J1=99; J=J1+n; J2=J+n; C=0; D=1; #ma [op=via; b=J1; le=n;]  #pas n r(J1) r(J)
# C=1/n1; #ma [op=vmc;] #ma [op=vsi;] #ma [op=vvm; trx=0;]
# C=1/n2; #ma [op=vmc; b=J;] #ma [op=vsi;] #ma [op=vvm; trx=0;]  #ma [op=vvm; b=J1; trx=n;]
# x=C*n; x1=x+0.5; k=1; j=J2; #pas n r(J1) r(j)
#rep m  # k=k+i; #pas n r(J1) r(J) #ma [op=vsm; mo=2; b=J; le=k; to=0; nx=n; ny=1;]   # j=j+n; #pas n r(J) r(j)
#end  # m=m+1; #d 17 r(1) n m 0 x -0.5 x1 0 2 3   -0.05 1.05 0 0.5 4   1 J2 t  #pr 0|noise\E ##20
```

Разумно в самом начале любой программы задать входные данные, то есть те переменные, которые предполагается менять в будущем. В первой строке определяются значения параметров i, n, m, о которых уже написано выше. Далее t – это время показа каждого кадра анимации, n_1 – определяет период первого (быстрого) синуса, n_2 – определяет период второго (медленного) синуса.

Во второй строке определяются начала всех тех массивов, которые нам необходимы в расчетах. В ЯП vkacl (далее ЛЯП) есть только один большой массив. И все более мелкие массивы являются его частью. Они определяются началом и длиной. То есть начальный индекс является как бы аналогом названия массива в других ЯП. Начинать разумно не с 1, а чуть дальше, так как первые элементы массива часто используются суперкомандами. Начало первого массива имеет индекс 99, второй на n дальше первого, а третий на n дальше второго. Такая стандартная техника используется во многих программах на ЛЯПе.

Затем начинаем расчет. Надо определить массив аргументов первого синуса. Это арифметическая прогрессия. Начальное значение задается в переменной C, а шаг – в переменной D. Всю работу делает операция via (vector initialize ariphmetic) команды #ma (mathematics). Ей нужно указать параметры массива, то есть начальный индекс в параметре b (begin) и длину в пареметре le (length). Затем команда #pas копирует n значений из области с началом в J1 в область с началом J.

Для вычисления массива первого синуса массив значений J1 умножаем на константу C = 1/n1. Это делает операция vmc (vector multiply constant) команды #ma. Параметры массива те же самые, их определять не нужно. Затем значения массива заменяются на синус этих значений операцией vsi (vector sinus). Квадрат вычисляется поэлементным умножением

массива самого на себя. Вообще то операция vvm (vector vector multiply) сделана для умножения массивов. При этом первый массив определяется параметрами b и le, второй сдвинут по индексу на значение параметра trx (translate x).

А для квадрата, точнее любой степени, есть друга функция. Но она медленнее вычисляет и квадрат проще делать умножением. Итак, в массиве J1 получили первый синус. Массив имеет 860 точек и это не мало. Но циклов нет. Вычисления в циклах нужно делать вызовом готовых операций. Так программа будет намного быстрее работать. Если все писать как в компилируемых языках, то время работы программы увеличится в 150 или 200 раз. Как раз этим чисто интерпретируемые языки отличаются от компилируемых.

Чтобы они быстро работали, надо все расчеты с массивами делать как одну операцию. По этой причине в интерпетируемых языках намного больше операций, чем в компилируемых языках низкого уровня. В некоторых языках матрицы (массивы) чисел трактуются как переменные и операции с ним делаются как с переменными. Но в ЛЯПе не так. Тут для этого есть команда #ma и она явно выписывается, а для операций есть параметр op. Такая запись немного длиннее, но более понятна при чтении. Я мог бы переделать и по другому. Но так было сделано в самом начале, так и осталось.

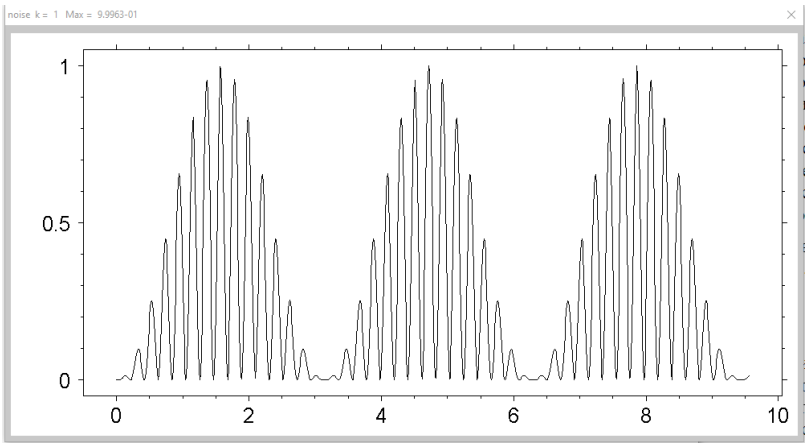
В следующей строке точно так же вычисляется квадрат второго синуса. И затем квадраты перемножаютя. Исходная функция готова в массиве J1. Затем определяются пределы на график, начальное значение параметра *k* и начальное значение массива, который будет усредняться в переменной j. Ее начальное значение равно началу матрицы J2, которая будет показана в анимации. Команда #rep (repeat) открывает цикл, в котором будет m повторений.

Всего в матрице будет m+1 строк, но первая строка содержит исходную функцию и она уже скопирована в первую строку матрицы. А в цикле сначала надо увеличить значение параметра *k* на шаг. Затем скопировать массив J1 в массив J. Это необходимо так как массив J будет испорчен, а массив J1 необходим для новой итерации и его нельзя портить. А сама процедура усреднения делается операцией vsm (vector special mathematics) команды #ma. Эта операция на самом деле выполняет много операций. Какую именно операцию она будет делать указывает параметр mo (modification). При mo=2 работа делается с матрицами, которые определяются параметрами b начало и nx, ny. Если у нас вектор, то ставим ny=1. Параметр le указывает длину усреднения, а параметр to в данном случае равен 0.

Для всех команд и их операций есть достаточно подробное описание. Фокус в том, что ЛЯП можно расширять бесконечно в рамках одного подхода добавляя новые команды и их операции в его интерпретатор. И это даже может делать любой пользователь, но для этого надо знать язык Java и иметь компилятор этого языка. Как это делается тоже подробно описано, а сам Java код интерпретатора доступен для скачивания. Пока только я сам могу это делать. И вот как раз данная операция была сделана совсем недавно, хотя ЛЯП существет более 20 лет даже в современном варианте.

Как только операция сделана и массив изменился нужно передвинуть начало строки матрицы и скопировать масив в новое место. Команда #end просто возвращает исполнение кода в начало цикла ровно столько раз, сколько написано в аргументе команды #rep. После того,, как цикл отработает матрица будет готова. А сам процесс анимационного ее показа делает суперкоманда номер 20. Для нее тоже есть описание, а ее входные данные надо записать в 17 первых элементов массива r(). Это делает команда #d , которой нет параметров, а аргументы – это число значений, указание типа массива и первого индекса и сами значения. Можно писать числа, или переменные, у которых используются их значения. Это как бы многократные присваивания, записанные более компактным способом.

Вот и все. То есть очень небольшой код делает на самом деле очень большую работу. Но все стандартные блоки этой работы уже сделаны и их заново делать не надо. Окно анимации показанона картинке ниже



Пример #5.

Данный пример, как и предыдущий, тоже одновременно является расширением описания языка *vkacl*, так как в описании не все написано. Речь идет о том, как использовать готовые программы при написании новых программ. Готовые программы не всегда имеют много кода. Часто это как раз были текстовые файлы малого размера. И чтобы не плодить много файлов, с которыми операционная система плохо справляется, я в новой версии программы создаю архивы, то есть записываю много программ в одном файле.

Это можно делать разными способами, но используется самый простой – каждая программа записывается в одну строку файла, то есть номер строки совпадает с номером программы. Но для готовых программ таких архивов два. Один для кода (pro00.txt) и один для входных данных (idc00.txt). Соответственно есть специальные программы, которые по номеру программы вынимают две строки в два временных файла. При этом специальный символ ~ (номер 126) в этих временных файлах заменяется на символ конца строки (номер 10). Это не обязательно, но так легче читать код, если в этом возникает необходимость.

И в готовых программах входные данные необходимо проверять на правильность, что делается специальной суперкомандой с номером 36, а сам код записывается как процедура с именем pr36, которую эта суперкоманда использует. Для непосредственного использования готовых программ в языке *vkacl* такой вариант не годится. По этой причине файл с кодом готовых программ (pro00.txt) копируется в другой файл (pri00.txt) и в этом файле код записывается без процедуры, а суперкоманда не используется. То есть это тот же самый код, но он сразу получает строку входных данных, точнее параметры строки, и работает.

Такой код как бы является почти суперкомандой, но суперкомандой старого типа. Они тоже возможны, но уже не используются. В новой версии программы код суперкоманды считывается из файла только один раз и потом запоминается в памяти для ускорения работы. Это похоже на кэш при показе сайтов в интернете. А у готовых программ такой механизм пока не используется. Итак, такой пример. Нам надо показать картинку из числового файла, то есть фактически представить числовой файл как картинку. Это делает готовая программа номер 70. Сначала можно организовать специальную процедуру запуска готовых программ. Она имеет такой код

```
#pro rrpn  #f [op=fold; file=null;]  # s(3)=KY-3500; #f [op=line; n=-ne; file=pri00.txt;] #f [op=fold; file=oldf;]
   #e [b=s(4); le=s(5);] _text @
```

Название процедуры от слов (Run Ready Program by Number). Перед вызовом этой процедуры надо напечатать строку входных данных (ВД) той программы, которую предполагается использовать, а также указать ее номер в переменной **ne**. И готовая программа будет исполнена.

Посмотрим что делает эта процедура. Она получает параметры строки входных данных в переменных I1 (начало) и S1 (длина). Все готовые программы используют эти переменные для указания строки ВД. В самом начале она обнуляет рабочую папку, так как файл (pri00.txt) находится в главной папке. И затем прочитывает нужную строку из файла в конец доступного текстового массива. Этот конец определяется переменной KY. На программу отводится не более 3500 символов. Как правило, они больше не бывают. Но если больше, то надо изменить число, которое вычитается. Затем надо вернуть ту рабочую папку, какая была, после чего программа выполняется из текстового массива.

Пусть, например, в папке (util/1), где находится ваша программа имеется файл с названием arr.dat, где записана матрица чисел с размерами 400*100, и вам надо нарисовать картинку этой матрицы в файл pic.png с помошью программы номер 70. Это сделает такой код

```
#f [op=fold; file= util/1;]  #pr arr pic 400 100 0 0 400 100 0 3 \E  # I1=s(4); S1=s(5); ne=70; #e _rrpn
```

Важно просто знать, что программа 70 реагирует на рабочую папку, а процедура rrpn ее не меняет. И там же появится файл картинки. При работе с файлами из другой папки имена файлов в программе печати надо записывать относительно той папки, которая установлена. Можно было бы написать код, который сразу эту картинку и покажет, но это уже другой пример. Важно, что готовые программы использую только числа во входных данных, а при печати можно использовать переменные и конвертировать их в числа с помощью какого-либо формата.

Пример #4.

Данный пример одновременно является и дополнением к описанию языка vkacl, так как в описании не все написано. Я даже сам не все использовал и только сейчас узнал о новых возможностях. Речь идет о том, как программа на языке vkacl может использовать команды операционной системы, в которой она работает. В данном примере я буду говорить об операционной системе Виндовс.

Первоначально я сделал только запуск внешних исполняемых файлов. Такими файлами в системе Виндовс являются файлы с раширениями exe и bat. Файлы с расширением exe – это просто программы, например программа менеджера файлов Тотал Коммандер. У меня есть несколько версий такой программы, в том числе и очень старая с регистрацией. Ее запуск можно сделать такой программой

```
#sys [op=rp; file=C:/0/totcmd-5/totalcmd.exe;] #pr OK\E ##33
```

Тут мы видим всего две команды и одну суперкоманду. Первая команда как раз и запускает внешнюю программу. Ее адрес надо писать как в Линукс, то есть разделитель такой (/), а не такой (\) как в Виндовс. Это общий принцип. Джава программа делалась для системы Юникс. Тут все просто. Команда sys (system), операция rp (run program). Адрес надо писать полный, либо начиная с буквы диска, либо относительно папки интерпретатора. Рабочая папка не используется.

А после запуска внешней программы основную программу разумно остановить. Проще всего напечатать какой-нибудь текст с кнопкой и кнопку не нажимать. Поработать с внешней программой, потом ее закрыть или так оставить и можно нажать кнопку. Все это легко и просто, но есть ограничения. Можно запускать только файлы, которые есть на компьютере. Это потому что программа проверяет наличие файла. Если его нет, то дается сообщение об ошибке и программа дальше не работает.

Но есть способ делать намного больше работы. Для этого надо записать bat файл, а потом его выполнить. Например, программа записывает файл (my.htm) в виде сайта пусть даже не на

сервер, а просто на компьютер в папку (pro) внутри папки интерпретатора. И надо предложить пользователю его посмотреть. Это можно сделать так.

```
# s(3)=1; #pr cd pro\nmy.htm\n\E  #f [op=fold; file=null;] #io [op=wt; fir=t(1); n=s(3)-1; file=runsys.bat;] #rob [mo=1; le=99;] #sys [op=rp;] #pr OK\E ##33
```

Тут сначала печатаются две строки текста. В первой (cd pro), то есть переходим в папку pro. Во второй – просто имя файла с расширением htm. Затем эти две строки записываются в файл runsys.bat и тут уже рабочая папка используется, так что ее надо обнулить. Затем надо сделать остановку программы на 99 миллисекунд. Фактически, время не важно, важно, чтобы программа остановилась, только тогда запустится процесс записи файла. И потом этот файл надо выполнить. Второй раз писать параметр файл не нужно. Он уже определен предыдущей командой.

Вот такой способ работает надежно и всегда. Особенно он хорош для файлов, записанных внутри папки интерпретатора, тогда можно писать короткие адреса. Но можно и длинные писать. В bat файлы можно писать все, что угодно. То есть использовать весь арсенал команд операционной системы. После использования файла runsys.bat его потом можно уничтожить, а можно и так оставить и много раз переписывать. При каждой новой записи он просто меняет свое содержание.

Так я работал много лет. Важно, что в bat файле можно просто указывать любые файлы. Они запускаются той программой, которая привязана к типу файла, тип определяется расширением, то есть текстом после последней точки. Просто так эти файлы не запускаются. И, тем не менее, есть возможность запускать сложные команды и без bat файла. Дело в том, что длина параметра file конечная, всего 42 символа. Так было сделано в самом начале.

Для относительных адресов вполне достаточно. Но для внешних адресов этого может не хватить. Позднее я это исправил следующим образом. Если написать (file=arg;), то адрес файла считывается из первого аргумента команды. И тут уже нет никаких ограничений. Более того, проверка на наличие файла с таким именем не делается. Значит можно писать не только имена файлов, а и любые команды. Вот например, так

```
# s(3)=1; #f [op=line; file=arg; n=-17;] C:/_vk/_ACLp/vkNP/par.txt\E # n=s(5); #te [op=repl; b=1; le=n; c=92; mo=47;] #sys [op=rp; file=arg;] \T1 n\u32; https://kohnvict.ucoz.ru/papersR.htm\E #pr OK\E ##33
```

Тут вот какая сложная ситуция. У меня на компьютере в 17-строке файла (par.txt) в папке (C:/_vk/_ACLp/vkNP/) записан адрес браузера Яндекса, который очень длинный и записан по правилам системы Виндовс. Я его прочитываю в текстовый массив. Затем меняю в нем символ (\) на такой (/). Затем я выполняю команду, в которой записан этот адрес браузера, пробел и адрес одной страницы моего сайта в интернете. Все адреса очень длинные, но это не помеха.

Можно и еще сложнее сделать. Адрес браузера переписывать не надо. Он один раз записан и на всю жизнь. А адрес сайта можно запрашивать у пользователя. Либо составить список адресов и указывать их по номеру. И так можно делать с любой программой, которая имеет аргументы. И при этом никаких bat файлов записывать не надо. Более того легко можно сделать программу простого менеджера файлов, который будет просматривать все файлы в папке и показывать их с помошью разных программ. Важно еще и то, что данная конструкция почти без изменений должна работать и в Линукс системе. Но я пока еще это не проверял. Среди готовых программ есть такая, которая делает что-то похожее. У нее есть таблица файлов и три операции –.редактировать, исполнять внешней программой, исполнять как программу на языке vkacl. Она как раз использует то, о чем написано выше.

Пример #3.

В некоторых программах, показывающих анимацию может понадобиться показать картинку на весь экран, которая никак не меняется, хотя программа все время запрашивает от пользователя данные о его действиях. То есть какие клавиши он нажимает и какие клики мышкой делает. То есть картинка стоит, а процесс идет. И только если пользователь нажимает клавишу [esc] картинка исчезает, а процес заканчивается. Я покажу код такой программы, а потом расскажу как он работает.

```
#d 4 s(14) -1*4 #w [op=im; file=pro/0/game.jpg; c=1; mo=1; tw=s(106); th=s(107); sc=1; sa=1; em=0; ]
#fr [op=o; mo=1; n=1; xs=0; ys=0;]   # s(2)=0; &=0;  &1=0;  #case 0  #ch [n=1; xs=-3; ys=-3;]  # &=s(2);
#case 27  # &1=1; #end |  # &=&1;   #end   #fr [op=c; mo=1;]
```

Итак, здесь сначала берется готовая картинка из файла game.jpg в папке pro/0/ и масштабируется точно на размер экрана компьютера. Эти размеры (ширина и высота) записаны в параметрах s(106) и s(107). Результат записывается в коллекцию готовых картинок с номером 1. Это делает команда w (window). Затем эта картинка выставляется на экран командой fr (frame). Команда w тоже умеет показывать картинки, но она это делает в раме и с дополнительной информацией. А команда fr показывает голую картинку без окна и это как раз то, что надо в данном случае.

Затем делается цикл командой case. Команда ch (choose) Снова показывает ту же картинку еще раз, но уже в цикле. И у нее другая функция. Она передает в программу действия, которые сделал пользователь после того как ее картинка была выставлена. То есть координаты клика мыши, модификатор (просто клик, или клик при нажатых клавишах Ctrl, Shift, Alt) и номер нажатой клавиши.

Как только пользователь проявил активность ее картинка закрывается, а сведения передаются в программу. Но реально картинка не исчезает, потому что на экране остается картинка от команды fr. Ее можно убрать только специальной операцией этой команды. И тут есть нюанс. Обе команды выставляют одну и ту же картинку по координатам. Но чтобы картинка не дергалась нужно в каманде ch выставлять координаты на 3 пиксела меньше, чем в команде fr. Почему так получилось, сложно сказать, но это правило легко выполнить, так что проблем никаких нет. Пользователь может нажимать разные клавиши и кликать мышкой в разных местах, но цикл все равно будет работать. А картинка будет стоять не двигаясь. И только если будет нажата клавиша [esc], то цикл закончится и картинка исчезнет. В конце как раз и стоит команда fr с операцией, которая снимает картинку с экрана.

У этого примера есть одна особенность. Можно так сделать, что картинка будет стоять и совсем не реагировать на клавиатуру и мышь. Мышь то будет ходить по всему экрану, но система отключена картинкой. И никаких кнопок нет. То есть компьютер потеряет способность работать. В этом случае в системе Виндовс спасают ситуацию горячие клавиши, такие как (Ctrl+Alt+Del) или (Win+X). Они имеют приоритет над всеми другими программами. Первая позволит закрыть задачу. Вторая даст меню, в котором есть функции выполнить другую программу.

Пример #2.

В опубликованных научных статьях было показано, что проблема сводится к расчету функции, представленной интегралом

$$F(u) = \int \mathrm{d}x \cos(ux) \exp(-gx^2)\, \theta(x[1-x]) \qquad (1)$$

Тут θ-функция задает пределы интеграла от 0 до 1. В частности интерес представляет полуширина кривой квадрата этой функции. Очевидно, что полуширина в 2 раза больше значения, при котором функция $F_1(u) = [F^2(u) - F^2(0)/2]$ равна нулю. То есть нужна

программа вычисления корня и процедура вычисления функции $F_1(u)$. Покажем как это можно сделать на ЯП (vkacl).

Прежде всего надо написать процедуру вычисления корня любой функции. Самый простой алгоритм – аппроксимировать функцию линейной на заданном интервале (Xa, Xb) и находить точку нуля, затем менять интервал до тех пор, пока в точке нуля функция не станет меньше значения в начальной точке интервала, умноженного на малое число (e). Однако этого может не случиться, поэтому надо задать еще максимальное число итераций (im). Если число итераций превысит это число, то заканчиваем расчет и пишем, что найти корень не удалось. Входные данные такой процедуры – указанные параметры и сама процедура func вычисления функции f(x). То есть на вход подается x, на выходе имеем f. Код такой процедуры выглядит так

```
#pro root  # X1=Xa; x=X1; #e _func; # F1=f;  F0=e*f; X2=Xb; x=X2; #e _func; # F2=f; i=0; &=1; #case 1
# A=(F2-F1)/(X2-X1); B=F1-A*X1; X3=-B/A; x=X3; #e _func; # F3=f; X1=X2; F1=F2; X2=X3; F2=F3; i=i+1;
&=abs(F3/F0)<1+int(i/im);  #end  # &=int(i/im); #case 1 #pr root fails !\E ##33 #end |  @
```

Эта процедура общая для многих задач и ее не надо писать каждый раз. Она может входить в список суперкоманд. Одна такая суперкоманда с номером 33 тут уже используется.

На втором этапе надо написать процедуру вычисления функции. Эта процедура частная, то есть только для нашей конкретной задачи. Она будет использоваться многократно и важно выделить в ней такие операции, которые не повторяются, то есть их можно вычислить предварительно один раз. Фактически, нам надо написать не одну, а две процедуры. Одна процедура выполняется один раз до поиска корня, вторая выполняется много раз при поиске корня. Назовем первую процедуру (befo) по началу слова before (перед). Там можно вычислить значение $F^2(0)/2$. Интеграл будем вычислять методом трапеций по числу точек (n). И еще у нас есть главный параметр (g) для зависимости от которого мы и делаем аппроксимацию. Код такой процедуры выглядит так

```
#pro befo # C=0; D=1/(n-1); J=199; J1=J+n; J2=J1+n; J3=J2+n; #ma [op=via; b=J; le=n;] #pas n r(J) r(J1)
#ma [op=vvm; b=J1; trx=0;] # C=-g; #ma [op=vmc;] #ma [op=vex; nx=n; ny=1;]  # x=0; S2=0; #e _func # S2=f/2; @
```

Необходимо объяснить алгоритм расчета. При вычислении интеграла используются массивы. Их нужно определить, то есть задать их начала. Это делается в переменных J, J1, J2, J3. Далее необходимо задать сразу все значения аргумента подинтегральной функции. Пределы интегрирования от 0, до 1. Первая точка записывается в переменную C, шаг до следующей точки – в переменную D. Операция via (vector ini arithmetic) команды ma (mathematica) делает то, что нужно, то есть определяет сразу весь массив значений переменной x. Затем этот массив надо продублировать. Это необходимо для расчета функции в цикле. Массив значений переменной x тоже достаточно вычислить 1 раз. Поэтому этот массив лучше не портить.

Следующая операция vvm (vector vector multiply) вычисляет во втором массиве x^2. Следующая операция vmc (vector multiply constant) вычисляет $-gx^2$. А следующая операция вычисляет экспоненту $\exp(-gx^2)$. Дело в том, что эта экспонента не зависит от параметра (u) и при поиске корня она никак не меняется. Ее достаточно вычислить один раз. что и сделано. А дальше делается такой трюк. Функция во второй процедуре func зависит от параметра S2, в котором как раз и записано значение $F^2(0)/2$. Но в самом начале если мы зададим (u) и (S2) равными нулю и запустим эту функцию, то мы получим $F^2(0)$ и потом можем задать истинное значение (S2). Тут же определяются параметры (nx) и (ny), хотя они никак не используются. Дело в том, что они тоже не меняются в цикле и их можно задать предварительно. На время расчета это не очень повлияет, но так более правильно.

Теперь осталось написать процедуру func. Ее аргумент x соответствует переменной (u) в формуле (1). Код такой процедуры выглядит так

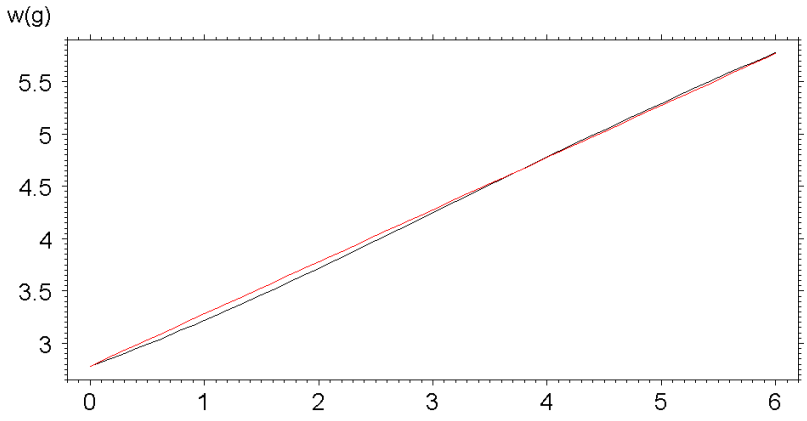
```
#pro func #pas n r(J) r(J2)  # C=x; #ma [op=vmc; b=J2;]  #ma [op=vco;]  #ma [op=vvm; trx=-n;] #ma [op=mss;]
# S=r(J3)-(r(J2)+r(J3-1))/2; f=S*S-S2; @
```

Тут уже все просто. Снова копируем массив аргумента, но массив J1 нельзя менять там сидит экпонента, которая нам нужна в расчете. Поэтому используем массив J2. Умножаем его на (x), вычисляем косинус, умножаем косинус на экспоненту и вычисляем сумму всех чисел. Вот для операции (mss), которая вычисляет сумму значений матрицы, и понадобилось определить параметры (nx) и (ny). А затем из суммы нужно вычесть половину значений в крайних точках. Так требует метод трапеций вычисления интеграла. И записываем ответ в переменную (f).

Полный ответ, то есть полуширина кривой w, равен 2*X2 после запуска процедуры root. А перед запуском надо определить интервал (Xa, Xb) и параметры (e), (im), (n) и самый главный (g). Нас интересует зависимость полуширины w от g. И эта зависимость приближенно аппроксимируется функцией w = 2.783 + 0.498 g. То еть надо вычислить зависимость w(g) и сравнить с этой линейной функцией на графике. А также напечатать значения в текстовом массиве. Это делается в процедуре depg. Код такой процедуры выглядит так

```
#pro depg  # s(3)=1; g=0; dg=0.1; j=1; m=61; n=200; e=0.001; im=19; Xa=0.7; Xb=3*Xa; #rep m  #e _befo  #e _root
# w=2*X2; r(j)=w; j=j+1; g=g+0.1; #pr \G w;\n\E  #end  # g=0; a=2.783; b=0.498; #rep m  # r(j)=a+b*g; j=j+1;
g=g+0.1; #end  # n=s(3)-1; #io [op=wt; fir=t(1); n=n; file=res1.txt;] #d 7 A 1 61 2 0 6 247 1100 #pr res1|w(g)\E ##1 @
```

Рассмотрим что тут делается. Параметр s(3) задает значение текстового массива для печати. Выглядит некрасиво, но так уж получилось и привыкнуть можно. Затем задается начальное значение параметра (g), шаг его изменения (dg), начальное значение индекса массива (j) для записи результатов расчета, число точек на кривой (m) и число точек расчета интеграла (n), и другие параметры, о которых выше говорилось. Затем открывается цикл из m повторений, определяется значение w, оно записывается в числовой массив и его значение записывается в текстовый массив командой (pr). Сдвигаются значения параметров на шаг и заканчивается цикл. После этого снова в цикле вычисляется линейная функция. Текстовый массив записывается в файл res1.txt, а график рисует суперкоманда с номером 1 в файл res1.png.



Собственно процедура depg – это и есть программа, которая полностью решает поставленную задачу, но она тоже оформлена как процедура потому что ее можно вставить в другую, более сложную программму, которая решает много разных задач. А задание входных параметров можно из этой процедуры выделить в другую процедуру и предложить пользователю самому определять эти значения. На рис. ниже показан график, который делает указанная программа для указанных значений параметров. Точный расчет показан черной кривой, а аппроксимация линейной функцией – красной кривой. Видно, что аппроксимация, озвученная уже в 2-х научных статьях, достаточно хорошо работает. Но в этих статьях расчет делался другим методом.

Пример #1.

Первый пример обычно делают самым простым. Но в моем случае это делать не интересно. И потому я рассмотрю относительно простой вариант по математике, но весьма сложный по возможности программирования. Речь пойдет о программе калькулятора. Калькуляторов в интернете много и часто они очень примитивные. Мы рассмотрим более сложный калькулятор, чтобы понять дополнительно как реализуется общение программы с пользователем. Этот калькулятор будет вычислять математические формулы, в которых могут быть не только числа, но и переменные от (a) до (h).

Математическая формула для каждой переменной записывается в одну строку. Вычисления проводятся в алфавитном порядке. Изначально все переменные равны 0, но если в первой строке переменная (a) определена и во второй строке она используется, то она принимает только что вычисленное значение. И такое же правило для всех остальных переменных.

Далее наша программа может запоминать набитые формулы в ящики шкафа, и в самом начале предлагается выбрать какой-либо из таких ящиков. При этом ящик номер 1 – это просто записная книжка. Рабочими являются ящики от 2 до 26. Всего кнопок в таблице 27. Так сделано в примере, но если необходимо, то число кнопок можно легко увеличить. После выбора ящика мы получаем уже готовые формулы, напечатанные ранее. Их можно переписать и они автоматически запомнятся в том же ящике для следущего раза.

Записная книжка записывается в файл calcul.inf, а ящики записываются в файл calcul.txt. И еще очень важный момент. Пользователь может записать математические выражения неправильно и тогда программа не сможет их вычислить. В этом случае она указывает, что сделана ошибка и просит ее исправить. Программа сделана из трех процедур, но это просто для удобства чтения кода. Реально можно было бы все записать в одну процедуру. Я буду рассматривать каждую процедуру отдельно. Так легче понять объяснения. Итак я показываю первую и главную процедуру

```
#pro main  # &=1; #case 1 # s(3)=1; j=1; n=27; #rep n #pr \I-2 j;|\E # j=j+1; #end  # n=s(3)-1;
#sel [nx=9; ny=3; mo=0; wid=30; hei=25; tf=3; tk=1; ts=16; col=255; em=1; xs=20; ys=70;] Select variant\E \T1 n\E
# S=&; &=S<2;  #case 1 #f [op=edit; mo=1; c=1; file=calcul.inf;] # Q=3; #end |  #case 2 # a=0; b=0; c=0; d=0; e=0;
f=0; g=0; h=0; #e _inpc # &=Q;  #case 1 #pr This|Other|No|Would you like to continue ?\E  ##33 # Q=&+1; #end |
# &=Q; #end # &=Q-2; #end  @
```

Теперь объяснение. Главная процедура реализует общую стратегию общения с пользователем. Программа начинает с того, что показывает таблицу кнопок. Это делает команда (sel) от слова select. Ей нужно указать число кнопок по горизонтали и вертикали, размер этих кнопок, параметры текста, который на этим кнопках написан, размещение таблицы на экране и сами тексты для каждой кнопки.

Эта команда может показывать не только тексты на кнопках, но и картинки. На то, что на кнопках будут тексты, указывает запись [mo=0;] У нее есть не только параметры, но и два аргумента. Первый – это заголовок окна, второй – тексты на всех кнопках, разделенные символом вертикаьной черты. Но второй аргумент просто печатается в цикле в текстовый массив и затем указывается часть этого массива. Команда печати имеет форматы и может записывать числа текстом.

Затем программа ждет реакции пользователя. Он выбирает нужную кнопку и ее значение передается в переменную &. Эта переменная очень часто меняется и ее сразу же надо запомнить в переменной S. А затем переопределить переменную & так, чтобы она имела всего два значения 1 и 2. Команда (case) сравнивает значение своего аргумента с значением переменной &. Если они совпадают, то все команды до команды #end выполняются, а если нет, то не выполняются. Видно, что если пользователь выбрал кнопку 1, то ему предлагается отредактировать содержимое файла calcul.inf . А если он выбрал любую кнопку больше 1, то обнуляются 8 переменных. Тут важно понимать, что переменные – это тоже элементы массива. И вместо явной записи можно было бы записать такой код (#d 8 a 0*8) и получился

бы такой же результат. А затем запускается вторая процедура, которая работает с пользователем и формирует новое окно, в котором показывает формулы для переменных и две кнопки [OK] и [Cancel]. Пользователь снова должен выбрать одну из двух. Если он выбирает первую, то производятся вычисления и показывается результат. А если вторую, то ничего не делается.

Значение выбора пользователя формирует значение переменной Q. Если оно равно 1, то есть расчет был, то программа просит пользователя снова выбрать один вариант из 3-х. Либо продолжить расчеты с выбранным вариантом, либо выбрать другой вариант, либо закончить работу. Суперкоманда 33 может показывать просто текст, а может показывать текст с кнопками. В данном случае это и делается. При этом текст надо разделять символом вертикальной черты. Выбор пользователя снова записывается в переменную Q. И дальше просто формируются разные выходы из команды case.

Теперь пора посмотреть что же делает процедура (inpc). Сначала я показываю ее код.

```
#pro inpc   # &=(S-1)<1; #case 1 # s(3)=1; #f [op=line; n=-S+1; file=calcul.txt;]  #p [err=1;]  #%  #case 10101
#pr \n\U32*5;Sorry, you have typed\n\U32*5;the invalid expssion.\n\U32*5;Please try again.\n\E ##33 #end | # s(3)=1;
#inp [n=8; le=80; mo=3; ysh=70;] Calculator\E a = |b = |c = |d = |e = |f = |g = |h = \E \T1 s(5)\E # Q=&; #case 2
# Q=Q+2; s(26)=0; #end |  #case 1 # z=s(5); s(3)=3999; #te [op=repl; b=1; le=z; c=10; mo=124;] #e _make #e _text
#p [err=0;] #f [op=line; n=S-1; b=1; le=z; em=0;] #pr \n a =\G a;\n b =\G b;\n c =\G c;\n d =\G d;\n e =\G e;\n\E
#pr f =\G f;\n g =\G g;\n h =\G h;\n\E ##33  #end | #end | @
```

Сразу видно, что процедура не будет работать, если (S=1). Это перестраховка. На самом деле при таком значении она и не вызывается. И первую команду case можно было и вообще не писать. Но иногда все же удобно перестраховаться. Внешний код может измениться, а тут все сразу ясно. Дальше тут из файла calcul.txt прочитывается строка с номером S-1. То есть при выборе второй кнопки считывается первая строка и так далее. Затем включается режим обработки ошибок путем присвоения параметру err значения 1. Стандартно он равен нулю и при возникновении ошибки интерпретатор прекращает работу. А в таком режиме интерпретатор присваивает переменной & значение 10101 и передает управление в то место, где стоит последняя команда #%, точнее сразу за ней. И тогда выполнится команда case с аргументом 10101. А если ошибки нет, то она не выполняется.

Затем команда (inp) от слова input создает окно, в котором указаны переменные и формулы для них. Эта команда имеет 3 аргумента. В первом – заголовок окна, во втором тексты в каждой строке, в третьем формулы в каждом окне ввода каждой строки. А параметры указывают, что таких строк 8, длину окон ввода и кое что еще. Описание этой команды есть в полном справочнике и можно все посмотреть. Я уже писал выше, что у этого окна есть две кнопки. Если выбирается вторая, то формируется значение переменной Q и больше ничего не делается. Точнее еще параметр err делается равным нулю. Его можно записывать по имени, а можно просто как элемент масива s(26).

А если выбирается первая кнопка то команда записывает новые значения текста в окнах ввода, которые разделяются символом конца строки. Начало текста задает параметр s(3), а сколько там символов указывает параметр s(5). Необходимо заменить все символы конца строки на символы вертикальной черты. Это делает операция (repl) команды (te). Далее нужно сформировать код языка vkacl. Это делает процедура (make) и выполнить этот код. Это делает процедура (text).

У процедуры (text) нет описания. Она выполняет код, записанный в текстовый мсссив. Эту запись процедура (make) как раз и делает. Далее осталось переопределить строки текста в файле calcul.txt и показать результат расчета. Это можно делать по разному. Сделано так, что значение каждой переменной показывается на новой строке. А можно было все показать на одной строке. Или для каждой переменной показать формулу и результат расчета. Все это очень легко и компактно делается. Но надо очень хорошо понимать функциональность

каждой команды. Осталось посмотреть что же конкретно делает процедура (make). Сначала я показываю ее код.

```
#pro make #te [op=find; b=1; le=z; n=1; cod=124;]  # A=1; B=i(1)-A; C=i(1)+1; D=i(2)-C; E=i(2)+1; F=i(3)-E;
G=i(3)+1; H=i(4)-G; I=i(4)+1; J=i(5)-I; K=i(5)+1; L=i(6)-K; M=i(6)+1; N=i(7)-M; O=i(7)+1; P=i(8)-O;
#pr \#\U32;a=\TA B ; b=\TC D ; c=\TE F ; d=\TG H ; e=\TI J ; f=\TK L ; g=\TM N ; h=\TO P ; \#p\U32;\[\]\U32; \E
#p [b=s(4); le=s(5);]  @
```

Тут в строке как части текстового массива с индексами от 1 до z определяются координаты, то есть индексы, всех символов вертикальной черты в этом массиве и записываются в целый массив, начиная с первого номера. Затем определются пары чисел для каждого текста между этими символами и командой печати формируется код в новую строку. В конце записываются параметры начала и длины этой новой строки. Вот и все. Хотя код всей программы достаточно компактный, для его объяснения пришлось записать много текста. На рисунке ниже показаны все окна, какие формирует программа.