JudoShiai SQL and Script Language Manual

For JudoShiai version 2.2

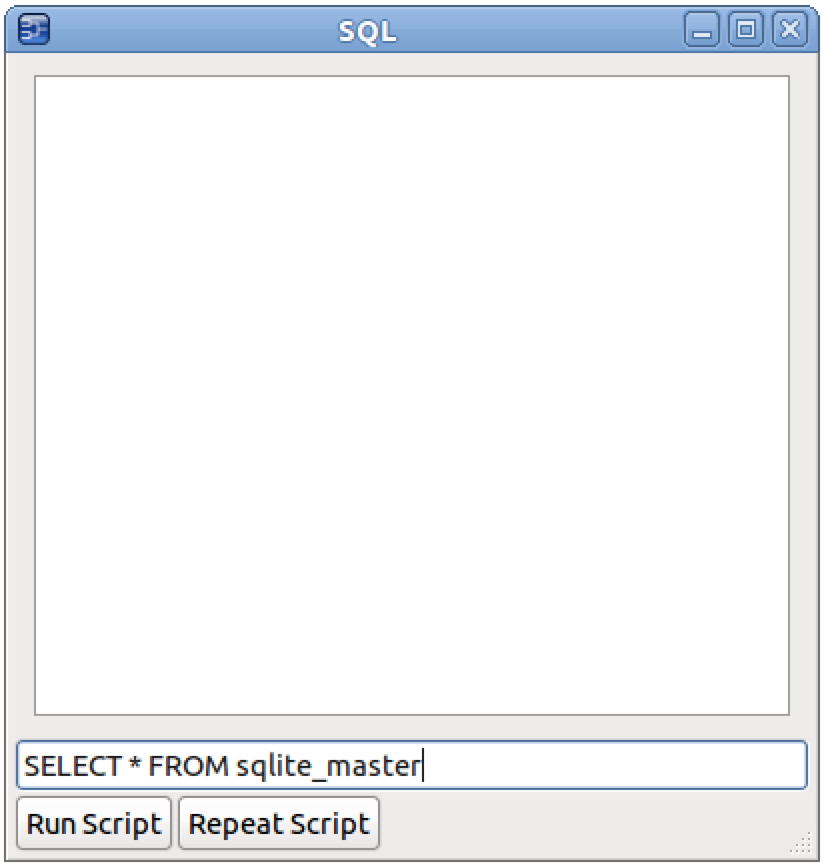
Introduction

You are running an international judo tournament. Suddenly you realize that all the French team's competitors have their country name misspelled as “FAR”. There are totally 58 French players and you should print out the sheets within one minute. What to do? You do not have enough time to correct the error manually, one by one. For a text file you could use a Find & Replace command, but JudoShiai database is not a text file. Fortunately there exists a more powerful method for database manipulation: Structured Query Language (SQL). The following SQL command would do the trick:

```
UPDATE competitors SET country='FRA' WHERE country='FAR'
```

This document will explain you how to use commonly used SQL commands. Only the simplest JudoShiai relevant commands are handled. Creation of tables is beyond the scope of this document since all the tables are created by the JudoShiai program. There are lots of learning material in the net if you want to try out some of the more complicated commands.

JudoShiai uses SQLite (http://www.sqlite.org/) as its database. There is free software available for database manipulation. However, in this document we are using the command line and scripting interface provided by the JudoShiai. Start JudoShiai, open a tournament and from the menu select Tournament → SQL Dialog.

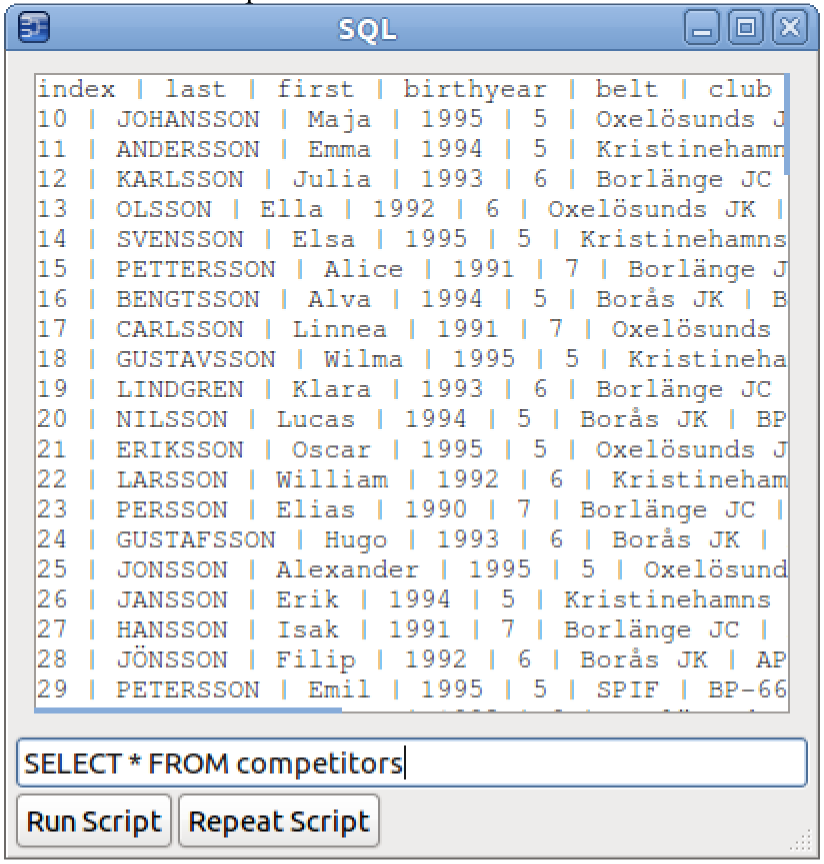


```
SELECT * FROM sqlite_master
```

A window will be opened. There is a big space for the results printing and a one line space for the SQL commands. The default command prints the basic structure of the database. Replace it with the command

```
SELECT * FROM competitors
```

and press Enter. You will have the competitors listed:



```
index | last | first | birthyear | belt | club
10 | JOHANSSON | Maja | 1995 | 5 | Oxelösunds J
11 | ANDERSSON | Emma | 1994 | 5 | Kristinehamn
12 | KARLSSON | Julia | 1993 | 6 | Borlänge JC
13 | OLSSON | Ella | 1992 | 6 | Oxelösunds JK |
14 | SVENSSON | Elsa | 1995 | 5 | Kristinehamns
15 | PETTERSSON | Alice | 1991 | 7 | Borlänge J
16 | BENGTSSON | Alva | 1994 | 5 | Borås JK | B
17 | CARLSSON | Linnea | 1991 | 7 | Oxelösunds
18 | GUSTAVSSON | Wilma | 1995 | 5 | Kristineha
19 | LINDGREN | Klara | 1993 | 6 | Borlänge JC
20 | NILSSON | Lucas | 1994 | 5 | Borås JK | BP
21 | ERIKSSON | Oscar | 1995 | 5 | Oxelösunds J
22 | LARSSON | William | 1992 | 6 | Kristineham
23 | PERSSON | Elias | 1990 | 7 | Borlänge JC |
24 | GUSTAFSSON | Hugo | 1993 | 6 | Borås JK |
25 | JONSSON | Alexander | 1995 | 5 | Oxelösund
26 | JANSSON | Erik | 1994 | 5 | Kristinehamns
27 | HANSSON | Isak | 1991 | 7 | Borlänge JC |
28 | JÖNSSON | Filip | 1992 | 6 | Borås JK | AP
29 | PETERSSON | Emil | 1995 | 5 | SPIF | BP-66
```

```
SELECT * FROM competitors
```

Not all the data fits in the window, but you can use scroll bars to move through the results.

Tables

JudoShiai database consists of tables, which store the database's data/information. Each table has its own unique name and consists of columns and rows.

The database table columns (called also table fields) have their own unique names and have a pre-defined data types. While table columns describe the data types, the table rows contain the actual data for the columns.

Here is an example of a simple database table, containing data. The first row, listed in bold, contains the names of the table columns:

Table: **Customers**

FirstName	**LastName**	**Phone**
John	Smith	626 123 6789
Mary	Jones	375 456 7754
William	Brown	234 765 7783

JudoShiai consists of the following tables:

- "competitors" lists all the competitors

- “categories” lists all the categories
- “matches” contains information about the matches
- “info” is used to save miscellaneous data
- “catdef” lists all the category criteria data

Table “competitors”

Column name	Column type	Description
index	Integer	Internal index. Visible in weigh-in notes. Do not change.
last	Text	Competitor's family name.
first	Text	Competitor's first name.
birthyear	Integer	Year of birth.
belt	Integer	Belt as a numeric value. 0 = unknown, 1 = 6. kyu, 2 = 5. kyu, etc.
club	Text	Name of the club.
regcategory	Text	Category name the competitor has registered.
weight	Integer	Weight in grams.
category	Text	The real category the competitor attends.
deleted	Integer	1 = competitor has been deleted.
country	Text	Name of the competitor's country.
id	Text	ID field for free use.

Table “categories”

Column name	Column type	Description
index	Integer	Internal index. Do not change.
category	Text	Name of the category in text format.
tatami	Integer	Number of the assigned mat
deleted	Integer	Not used.
group	Integer	Number of the group the category belongs to.
system	Integer	Match system: 1 = pool 2 = double pool 3 = double repechage for 8 competitors 4 = double repechage for 16 competitors 5 = double repechage for 32 competitors 6 = double repechage for 64 competitors 9 = quadruple pool
numcomp	Integer	Number of competitors in the category.

Column name	Column type	Description
table	Integer	Detailed description of the double repechage: 0 = double repechage (commonly used) 1 = Swedish dubbelt återkval 2 = Swedish direkt återkval 3 = Estonian system for D-klass 4 = no repechage 5 = Swedish enkelt återkval 6 = Spanish doble perdida 7 = Spanish repesca doble 8 = Spanish repesca simple 9 = American modified double elimination 10 = double repechage, one bronze only
wishsys	Integer	Preferred system, if possible: 0 = default for the country 1 = pool 2 = double pool 3 = double repechage 4 = Swedish dubbelt återkval 5 = Swedish direkt återkval 6 = Estonian D-klass 7 = no repechage 8 = Swedish enkelt återkval 9 = quadruple pool 10 = Spanish doble perdida 11 = Spanish repesca doble 12 = Spanish repesca simple 13 = American modified double elimination 14 = double repechage, one bronze only
pos1	Integer	Winner's index number (table “competitors”)
pos2	Integer	Silver medalist's index number
pos3	Integer	1st bronze
pos4	Integer	2nd bronze or fourth
pos5	Integer	1st fifth
pos6	Integer	2nd fifth or sixth
pos7	Integer	1st seventh
pos8	Integer	2nd seventh

Table “matches”

Column name	Column type	Description
category	Integer	Internal index for the category.
number	Integer	Number of the match.

Column name	Column type	Description
blue	Integer	Index of the blue competitor (table “competitors”)
white	Integer	Index of the white competitor (table “competitors”)
blue_score	Integer	Scoring for the blue (IWYKS): = shidos + 16*kokas + 256*yukos + 4096*wazaris
white_score	Integer	Scoring for the white (IWYKS).
blue_points	Integer	Winning points for the blue (0, 1, 3, 5, 7, 10).
white_points	Integer	Winning points for the white (0, 1, 3, 5, 7, 10).
time	Integer	Length of the match in seconds.
comment	Integer	Comment: 0 = no comment 1 = next match 2 = preparing match 3 = wait
deleted	Integer	Not used.
forcedtatami	Integer	Number of the mat this match has been moved to.
forcednumber	Integer	Number in the match queue.

Table “info”

Column name	Column type	Description
item	Text	Name of the data • Competition • Date • Place • three_matches_for_two • Time • NumTatamis
value	Text	Value of the data

Table “catdef”

Column name	Column type	Description
age	Integer	Competitor's maximum age for this category (e.g. 16)
agetext	Text	Age part of the category (e.g. Men-U17)
flags	Integer	1 = male 2 = female
weight	Integer	Maximum weight in grams
weighttext	Text	Weight part of the category (e.g. -66).

Column name	Column type	Description
matchtime	Integer	Match time in seconds (e.g. 300).
pintimekoka	Integer	Osaekomi time for koka in seconds.
pintimeyuko	Integer	Osaekomi time for yuko in seconds.
pitimewazaari	Integer	Osaekomi time for waza-ari in seconds.
pintimeippon	Integer	Osaekomi time for ippon in seconds.
resttime	Integer	Resttime in seconds (e.g. 600).
gstime	Integer	Golden score time in seconds.

SQL

SQL is a standard language for accessing databases. Most of the actions you need to perform on a database are done with SQL statements. The following SQL statement will select all the records in the "info" table:

```
SELECT * FROM info
```

Keep in mind that SQL is not case sensitive. It is easier to write

```
select * from info
```

but to emphasize the keywords they are written in upper case. Some database systems require a semicolon at the end of each SQL statement, but JudoShiai doesn't require or allow that practises.

SQL can be divided into two parts: The Data Manipulation Language (DML) and the Data Definition Language (DDL). The query and update commands form the DML part of SQL:

- SELECT - extracts data from a database
- UPDATE - updates data in a database
- DELETE - deletes data from a database
- INSERT INTO - inserts new data into a database

The DDL part of SQL permits database tables to be created or deleted. It also define indexes (keys), specify links between tables, and impose constraints between tables. The most important DDL statements in SQL are:

- CREATE DATABASE - creates a new database
- ALTER DATABASE - modifies a database
- CREATE TABLE - creates a new table
- ALTER TABLE - modifies a table
- DROP TABLE - deletes a table
- CREATE INDEX - creates an index (search key)
- DROP INDEX - deletes an index

You are not going to modify the data definition, so we are concentrating to the DDL part only.

SELECT

The SQL SELECT statement is used to select data from a SQL database table. Please have a look at the general SQL SELECT syntax:

```
SELECT Column1, Column2, Column3,
FROM Table1
```

The list of column names after the SQL SELECT command determines which columns you want to be returned in your result set. If you want to select all columns from a database table, you can use the following SQL statement:

```
SELECT *
FROM Table1
```

When the list of columns following the SELECT SQL command is replaced with asterisk (*) all table columns are returned. The table name following the SQL FROM keyword (in our case Table1) tells the SQL interpreter which table to use to retrieve the data.

Now we want to select the content of the columns named "last" and "first" from the table “competitors”:

```
SELECT last,first FROM competitors
```

SELECT DISTINCT

In a table, some of the columns may contain duplicate values. This is not a problem, however, sometimes you will want to list only the different (distinct) values in a table. The DISTINCT keyword can be used to return only distinct (different) values.

Suppose we want to select only the distinct values from the column named "country" from the table “competitors” to find out which countries the competitors are from. We use the following SELECT statement:

```
SELECT DISTINCT country FROM competitors
```

We are going to have a list countries, each listed only once.

WHERE

The SQL WHERE clause is used to select data conditionally, by adding it to already existing SQL SELECT query. Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator value
```

If we want to select all the French competitors from the competitors table we need to use the following SQL syntax:

```
SELECT * FROM competitors WHERE country='FRA'
```

SQL uses single quotes around text values (JudoShiai accepts double quotes, too). Numeric values should not be enclosed in quotes. For numeric values:

```
SELECT * FROM competitors WHERE birthyear=1995
```

With the WHERE clause, the following operators can be used:

Operator	**Description**
=	Equal
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range

Operator	Description
LIKE	Search for a pattern
IN	If you know the exact value you want to return for at least one of the columns

AND & OR

The AND operator displays a record if both the first condition and the second condition is true. The OR operator displays a record if either the first condition or the second condition is true. Examples:

```
SELECT * FROM competitors WHERE first='John' AND last='Smith'
SELECT * FROM competitors WHERE birthyear=1995 OR birthyear=1996
```

Now we want to select only the persons with the last name equal to “Smith” and the year of birth equal to 1995 or 1996:

```
SELECT * FROM competitors WHERE last='Smith' AND (birthyear=1995 OR birthyear=1996)
```

ORDER BY

The ORDER BY keyword is used to sort the result-set by a specified column. The ORDER BY keyword sort the records in ascending order by default. If you want to sort the records in a descending order, you can use the DESC keyword.

```
SELECT column_name(s)
FROM table_name
ORDER BY column_name(s) ASC|DESC
```

Example:

```
SELECT * FROM competitors ORDER BY country,last,first
```

INSERT INTO

The INSERT INTO statement is used to insert a new row in a table. It is possible to write the INSERT INTO statement in two forms. The first form doesn't specify the column names where the data will be inserted, only their values:

```
INSERT INTO table_name
VALUES (value1, value2, value3,...)
```

The second form specifies both the column names and the values to be inserted:

```
INSERT INTO table_name (column1, column2, column3,...)
VALUES (value1, value2, value3,...)
```

In the next example we insert size of the mat to the table “info”:

```
INSERT INTO info VALUES ('Size_of_tatami', '7x7')
```

UPDATE

The UPDATE statement is used to update existing records in a table. SQL UPDATE Syntax:

```
UPDATE table_name
SET column1=value, column2=value2,...
WHERE some_column=some_value
```

Example: We have a club mostly called “Jukara” although its full name is “Vantaan Jukara”. It is very important to use only one spelling style. The following SQL statement replaces all the names containing the string “jukara” (case insensitive) to “Vantaan Jukara”:

```
UPDATE competitors SET club='Vantaan Jukara', country='FIN'
WHERE club LIKE %jukara%
```

Notice the operator LIKE and the search pattern %jukara%. That search pattern matches all the strings containing the word “jukara” (case insensitive).

DELETE

The DELETE statement is used to delete rows in a table. SQL DELETE Syntax:

```
DELETE FROM table_name
WHERE some_column=some_value
```

Example: French team has canceled their participation. Remove them:

```
DELETE FROM competitors WHERE country='FRA'
```

Delete everything from a table (you cannot undo, be very careful!):

```
DELETE FROM table
```

Script language

SQL is handy for the database manipulation. However, it is not good enough for more complicated tasks. Consider the following scenario: You have a JudoShiai database for a domestic tournament. Some of the club names are misspelled and there are no country information. You wish to correct the club names and add an area information. For example in Great Britain the areas could be England, Wales, and Scotland. Some other country may be divided just to Northern, Eastern, Southern, and Western areas. Correct club name spelling and the country (area) information ensure that competitors from the same or closely located clubs wouldn't have the first matches against each other.

The following SQL statements could be used:

```
UPDATE competitors SET club='Judo Club Ippon', country='North'
WHERE club LIKE %ippon%
UPDATE competitors SET club='Helsinki Judo', country='South'
WHERE club LIKE %helsinki%
```

and so forth for each club. However, this not too handy if you have tens of clubs. It would be better to have a separate text file that has relevant information about the clubs and a script that reads in the club data and runs the SQL statements.

JudoShiai has a built in script language interpreter that is based on the MOLE BASIC. A script is a plain text file created using for example Notepad (Windows) or Gedit (Linux). Script file name suffix is “.bas”, for example “my-script.bas”. To run a script

- Start JudoShiai and open a tournament
- Select from the menu Tournament → SQL Dialog
- Click button Run Script and select your script file
- Next time you don't need to select the same file if you click the Repeat Script button

Introduction

To illustrate the nature of script language, we first give a program that multiplies two numbers and prints the result:

```
! mass in kilograms
LET m = 2
! acceleration in mks units
LET a = 4
! force in Newtons
LET force = m*a
PRINT force
END
```

The features of the script language included in the above program include:

- Comment lines begin with ! and continue to the end of line.
- LET, PRINT, and END are keywords (words that are part of the language and cannot be redefined) and are here given in upper case. The case is insignificant.
- The LET statement causes the expression to the right of the = sign to be evaluated and then causes the result to be assigned to the left of the = sign. The LET statement can be omitted.
- Script language recognizes only two types of data: floating point numbers and strings (characters). The first character of a variable must be a letter.
- The PRINT statement displays output on the screen.
- The last statement of the program must be END. It is optional if it is the last command.

Loop structures

Script language uses a FOR/NEXT or WHILE/WEND or DO/UNTIL construct to execute the same statements more than once. An example of a FOR loop follows:

```
! add the first 100 terms of a simple series
! Variables are automatically initialized to zero
sum = 0
FOR n = 1 to 100
    sum = sum + 1/(n*n)
    PRINT n,sum
NEXT
```

- The use of the FOR loop structure allows a set of statements to be executed a predetermined number of times. The index or control variable (n) monitors the number of times the loop has been executed. The FOR statement specifies the first and last value of the index and the amount that the index is incremented each time the NEXT statement is reached. Unless otherwise specified, the index is increased by one until the index is greater than its last value in which case the program goes to the statement after the NEXT statement. In the example the index n assumes the values 1 through 100.
- The block of statements inside the loop is indented for clarity.
- The order of evaluation follows the mathematical conventions shared by all computer languages. Multiplications and divisions are performed first from left to right. Parentheses should be used whenever the result might be ambiguous to the reader. The parentheses in the statement, sum = sum + 1/(n*n), are included for clarity. Note that the keyword LET has been omitted.
- All unassigned variables are automatically initialized to zero.

In many cases the number of repetitions is not known in advance. An example of a WHILE/WEND loop follows:

```
! illustrate use of WHILE LOOP structure
sum = 0
n = 0
relChg = 1
WHILE relChg > 0.0001
   n = n + 1
   newterm = 1/(n*n)
   sum = sum + newterm
   relChg = newterm/sum
   PRINT n,relChg,sum
WEND
```

Note the use of the WHILE loop structure to repeat the sum until the specified condition is no longer satisfied. Example of DO/UNTIL loop:

```
n = 0
DO
    PRINT n,n*n
UNTIL n >= 10
```

Example will print squares for values 0 – 9. Looping stops when n reaches value 10.

Conditional statements

The IF statement lets a program branch to different statements depending on the outcome of previous computations. An example of the use of the IF statement follows:

```
x = 0
WHILE x < 20
   x = x + 1
   IF x <= 10 THEN f = 1/x ELSE f = 1/(x*x)
   PRINT x,f
WEND
```

General format for IF statement is

```
IF condition THEN command/number [ELSE command/number]
```

ELSE branch is optional. After THEN and ELSE you may have a command or a line number to jump to. Whole IF statement must be at the same line. Examples:

```
IF a > 7 THEN x = 17 : c = a + 1
```

You can have many colon separated commands at the same line if there is no ELSE branch.

```
IF a > 7 THEN x = 17 ELSE c = a + 1 : b = a*8
```

After the ELSE branch you may have several colon separated commands.

```
    IF a > 7 THEN 110 ELSE 120
110 print "A is greater than 7. Stop the script."
    end
120 x = 17
    c = a + 1
```

If condition is true program execution will jump to line 110 otherwise it will jump to line 120. All the lines can be preceded by a line number but usually it is optional. Line number can be any unique number and they don't need to be in order. Line numbering is a historical way to edit Basic files using a teletypewriter.

The decisions of an IF structure are based on (logical or Boolean) expressions which are either true or false. A logical expression is formed by comparing two numerical or two string expressions by a relational operator. These operators are given in the next table:

Operator	Relation
=	Equal
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal

Subroutines

It is convenient to divide a program into smaller units consisting of a main program and subroutines. Subroutines are called from the main program or other subroutines.

General format for a subroutine call is

GOSUB number
where number is a line number. A subroutine ends to RETURN statement. Example:

```
FOR i = 1 to 10
    GOSUB 100
NEXT
i = 313
GOSUB 100
END

! Subroutine to make calculations
! and printing
100
x = 0.78*i
IF i > 100 THEN x = x + i*i + 3.234
PRINT i,x
RETURN
```

Subroutine takes variable “i” as its input and calculates a value for variable “x” and prints both. The subroutine is called from the FOR loop for values 1 – 10 and later for value 313. Line number “100” is at its own line but it could be in front of the “x = 0.78*i” statement. Note that all the variables are global: if subroutine changes the variable “i” the change is visible everywhere.

Functions

Functions are subprograms that take arguments and return a value. You cannot create functions yourself, but you can use some predefined functions. Example:

```
i = 0
WHILE i <= 3.14/2
    PRINT i, SIN(i)
    i = i + 0.2
```

```
WEND
```
Example prints sine values for angles 0 – PI/2 radians at 0.2 rad steps. Function SIN accepts one argument, angle in radians and it returns sine of given angle. Note that FOR/NEXT statement can STEP only integer values.

String variables

As mentioned, script language recognizes only two types of variables, numeric and strings. A string variable may be any combination of characters. String variables end in a dollar sign ($). A string constant is any list of characters enclosed in quotation marks. An example of an assignment of a string variable is

```
fileName$ = "config.dat"
```

A program illustrating the most common operations on string variables follows:

```
a$ = " "
b$ = "good"
PRINT b$
b$ = b$ + a$ + "morning"
PRINT b$
```
Example will first print “good” and then “good morning”. “+” operator concatenates two strings. There are many useful string handling functions. Examples:

```
! VAL returns value of string representing number
b$ = "18"
c = 5 + VAL(b$)
! print date (no arguments for the function)
PRINT DATE$
! print substring "morning"
b$ = "good morning"
PRINT MID$(b$, 6)
```

Arrays

An array variable is a data structure consisting of an ordered set of elements of the same data type. One advantage of arrays is that they allow for the logical grouping of data of the same type, for example the x and y coordinates of a particle. The dimension of an array is illustrated in the next example:

```
! array for three numbers
DIM age(3)
! two dimensional array for the first and last names
DIM name$(3,2)
! ages
age(1) = 23
age(2) = 45
age(3) = 16
! first and last names
name$(1,1) = "Frank"
name$(1,2) = "Jones"
name$(2,1) = "Bob"
name$(2,2) = "Smith"
name$(3,1) = "Jane"
name$(3,2) = "Brown"
! print person data
PRINT "Name     ", "Age"
FOR i = 1 TO 3
```

```
    PRINT name$(i,1);" ";name$(i,2), age(i)
NEXT
```

- Arrays are defined in a DIM statement and the total number of elements of an array is given in parentheses. The array variable "age" is an example of one-dimensional array while variable "name$" is an example of two-dimensional array. You can have at most three dimensions.
- Subscript in an array starts from 1 (age(1) – age(3), name$(1,1) – name$(3,2)).
- An element of an array is specified by its subscript value.

Input/output

The PRINT statement displays output on the screen. Some simple extensions of the PRINT statement include

```
PRINT "x","y","z"
PRINT x,y,z
PRINT                      ! skip line
PRINT "time = ";hour;":";min;
PRINT ", date = "; DATE$  ! time and date at the same line
```

Script language prints at the current cursor position. Comma (",") moves cursor to the next tab, semicolon (";") continues printing to the current cursor location. Semicolon at the end of line prevents printing a newline.

The cursor may be moved by the LOCATE statement

```
LOCATE 0,0  ! upper left corner (column, row; both starting from 0)
```

The following program illustrates how to open a text file, write to the file, close the file, and read the file.

```
! save data in a single column
! channel number #1 is associated with the file
! various options may be specified in OPEN statement
OPEN "testfile.txt" FOR OUTPUT AS #1
FOR i = 1 TO 4
    x = i*i
    PRINT #1 x                  ! print column of data
NEXT
! close the file
CLOSE #1

! read data back
OPEN "testfile.txt" FOR INPUT AS #2
FOR i = 1 TO 4
    INPUT #2 y                  ! print column of data
    PRINT y
NEXT
CLOSE #2
END
```

You can save data in several columns separated by a tab:

```
PRINT #1 var1, var2, table(2)
```

You can read more that one variable at the same line, if they are

- all numeric
- separated by a tab

```
INPUT #1 var1, table(i), var2
```

It is possible to print mixed numeric and string values to a file, but you can read the whole line as one string variable only:

```
num = 4
vehicle = "car"
PRINT #1 vehicle$; " has "; num ; " doors"
... later ...
INPUT #1 line$
! line$ has the value "car has 4 doors"
```

A string may contain what ever characters so it is not clear where one data field ends and another starts.

READ/DATA

One way to incorporate data into a program from a file. Another way to store information within a program is by using the DATA and READ statements as illustrated below:

```
DIM x(6)
DATA 4.48,3.06,0.20,2.08,3.88,3.36
FOR i = 1 to 6
    READ x(i)                ! reads input from DATA statement
NEXT i
```

SQL

SQL queries start with a keyword “SQL”. Example to list all the competitors:

```
SQL "SELECT * FROM competitors"
```

Here “SELECT * FORM competitors” is the familiar SQL query. You can compose the SQL statement from constants and variables:

```
table$ = "competitors"
lookfor$ = "club,last,first"
weight = 50000
SQL "SELECT "+lookfor$+" FROM "+table$+" WHERE weight<"+weight
```

A SQL query “SELECT club,last,first FROM competitors WHERE weight<50000” will be done. All the competitors whose weight is less than 50 kg will be listed. Note that although SQL statement accepts a string parameter only you can use numeric variable after the “+” since script language automatically changes the result to a string.

Example

In the beginning of the script language introduction we wanted to correct the misspelled club names and add area information. Let's start by writing a data file “clubs.txt”. It has three columns:

- Clubs name, for example “Cambridge Judo”
- A string for SQL operator LIKE, for example “%cambr%”
- An area, for example “England”

The columns are separated by a comma, so the example file looks like this:

```
Cambridge Judo,%cambr%,England
Barnet Judo,%barnet%,England
Walderslade Judo Club,%walder%,England
Stonehaven Judo Club,%stoneh%,Scotland
Peebles Judo Club,%peebles%,Scotland
Llantwit Major Judo Club,%lantwit%,Wales
```

The script is listed below:

```
OPEN "clubs.txt" for INPUT as #1

WHILE NOT EOF #1             ! read while lines left (not End Of File)
    GOSUB 100                ! call subroutine at line 100
    cmd$ = "UPDATE competitors SET club='"+club$+"',country='"+area$+\
           "' WHERE club LIKE '"+like$+"'"
    PRINT cmd$               ! print command first
    SQL cmd$                 ! execute the sql command
WEND
GOTO 999

100                          ! subroutine starts
INPUT #1 line$               ! read a line
c1 = INSTR(line$, ",")       ! find the first comma
IF c1 = 0 THEN 999           ! no comma found
c2 = INSTR(c1+1, line$, ",")! find the second comma
IF c2 = 0 THEN 999           ! no comma found
club$ = MID$(line$, 1, c1-1)! extract the three words
like$ = MID$(line$, c1+1, c2-c1-1)
area$ = MID$(line$, c2+1)
RETURN

999 CLOSE #1
END
```

Script lines explained:

- OPEN line opens file “clubs.txt” for reading. Later on it is addressed as #1.
- WHILE loops as long as there is something to read from the file.
- GOSUB calls a subroutine that reads a line and extracts the variables club$, like$, and area$.
- cmd$ holds the SQL command. It is handy to make a variable first which is easy to print to check for correctness. Split long lines by writing a backslash (“\”) at the end of the first line.
- After printing it is time to call the SQL command. If everything went well you will see “OK” printed.
- WEND closes the loop.

- GOTO 999 jumps execution to the end.
- Subroutine starts with line number 100.
- INPUT reads one whole line from the file.
- INSTR function finds the first occurrence of the comma.
- If the found position is zero there were no comma at the line and it is better to go to the end.
- Second INSTR finds the second comma. Search starts one position after the first finding.
- MID function extracts substrings. Three calls are needed to do the job.
- RETURN jumps script execution back to the next line where this subroutine was called.
- Finally at line 999 the file is closed and execution stops at the END.

Command reference

ABS(number)	Returns absolute value of number
ACOS(number)	Calculates the arc cosine number/condition
AND number/condition	Logical AND used in conditions or numerical expressions
ASC(string)	ASCII code of first letter of string
ASIN(number)	Calculates the arc sine
ATN(number)	Calculates the arc tan
CHR$(number)	Returns char with value number
CHDIR stringexpression	Change to directory string
CINT(number)	Truncated number (NOTE: differs from INT !)
CLS	Clears screan (if you're screen supports VT100/VT102/ANSI codes
CLOSE #number	Close file number (number must be 1 or higher)
COS(number)	Cosine of number
DATE$	Date in form of "yyyy-mm-dd"
DIM variable(dim, [dim,] [dim,]..)	Dimension variable.

DO [commands] UNTIL condition	Execute everything between DO and UNTIL until condition is true.
ENVIRON stringvariable = string	Sets environment var "stringvariable" using value
ENVIRON$(string)	Returns string matching environment var "string"
END	End program
EOF #filenumber	Returns TRUE if last byte is reached from file
EXP(number)	Exponential value of number
FOR variable = beginexpression TO endexpression STEP stepxpression ...... NEXT	Start FOR..NEXT loop with variable as counter, increased by stepexpression till endexpression is reached
FREEFILE	Returns first free filenumber
GET #filenumber, recordnumber , variable	Retrieve record number "recordnumber" from "filenumber" and place it in "variable"
GOSUB linunumber.. RETURN	Goto linenumber and preceed when found RETURN
GOTO	Goto linenumber
HEX$(number)	Returns number in hexformat

IF condition
THEN command/number
[ELSE command/number]
	if condition is TRUE run command/jump to line , if it is FALSE execute command following ELSE (or jump to line).

INPUT [#filenumber] ["Comment";] var [,var [,var...]]	Reads variables from STDIN or file
INSTR([starting,] searchstring, keyword)	Returns position (counting from 1) of keyword in searchstring starting at offset 0 or "starting". Returns zero if isn't found.
INT(number)	Rounds to biggest integer.
KILL filename	Remove file
LET	Sets variable. Could be left out.
LEFT$(string , total)	Returns total chars of left side of string
LEN(string)	Returns size of string
LOC #filenumber	Returns value of file position indicator of filenumber

LOCATE column,row	Place cursor on column and row (counting from 0,0 upper left)
LOF	Returns length of file
LOG(number)	Returns logarithm of number
LOWER$(string)	Returns string in lowercase
LSET stringvar = string	Place string on left side of stringvar
LTRIM$(string)	Returns string without spaces on left side of string
MID$(string, from [,total])	Returns string from position "from" and maximum of total chars
number MOD number	Modulo
NOT expression	Logical not
NAME filename AS newname	Rename file
OCT$(number)	Return string of number in octal format
OR	Logical OR
ON variable GOSUB line [,line [,line ..]]	Depending on value of variable, do GOSUB to first mentioned line if value=1, second if value=2 and so on
ON variable GOTO line [,line [,line ..]]	Depending on value of variable, do GOTO to first mentioned line if value=1, second if value=2 and so on
OPEN filename FOR INPUT|OUTPUT|APPEND AS #filenumber [LEN recordsize]	Open filename with recordlen "recordsize" (used by GET and PUT)
PRINT [#filenumber] "text"| variable[,;]..[;]	prints variables or text. Semicolon on end prevents printing of newline, comma prints tab
PUT #filenumber, recordnumber	Gets record number "recordnumber" from file
RANDOMIZE [seed]	Sets seed for random generator, normally TIMER is used for this
REM [remarks]	Remark, ignored. Same as line starting with “!”.
READ variable [, variable].. DATA value [, value] ..	Read puts value from DATA into variable, next READ will cause next value from DATA to be read and so on.
RESTORE	Restore READ pointer. DATA values are read from beginning
RIGHT$(string , total)	Returns total chars of right side of string
RTRIM$(string)	Returns string without spaces on right side of string

RND(number)	Returns random value between zero and number.
RSET stringvar = string	Place string on right side of stringvar
SEEK #filenumber, position	Place filepointer on new position (NOTE: this is byte oriented, *NOT* recordlen oriented)
SPACE$(number)	Returns number of spaces
SWAP variable1 , variable2	Swaps two variables
SGN(value)	Returns -1 if value is negative, 0 if zero and 1 if positive
SIN(angle)	Returns sinus of angle in rad
SQR(value)	Returns square root of value
STR$(number)	Returns string representation of number
STRING$(total, charvalue|string)	Returns string filled with total times charvalue of first char of string.
SYSTEM(command)	Execute command
TAN(angle)	Returns tangent of angle
TIMER	Returns elapsed seconds since last midnight
TIME$	Returns timestring in format "hh:mm:ss"
UPPER$(string)	Returns string in uppercase
VAL(string)	Returns value of string representing number
WHILE condition .... WEND	Execute commands between while/wend as long as condition is true
XOR	Logical XOR