Lecture 3

Programming structures

1 Introduction

In this lecture we introduce the basic programming tools that you will use to build programs. In particular we will be introduced to the `if`, `for` and `while` loops.

2 Program files

2.1 Directories

The `pwd` command (print working directory) tells you what directory you are currently working in.

```
-->pwd
 ans  =
 C:\Users\Hudson
```

Use the `cd` (change directory) command followed by a valid directory name to move to another directory.

```
-->cd Desktop
 ans  =
 C:\Users\Hudson\Desktop
```

In Scilab the `cd` command with no argument moves you to your "home" directory.

```
-->cd
 ans  =
 C:\Users\Hudson
```

Entering '..' for the directory name moves you up one directory level.

```
-->cd '..'
 ans  =
 C:\Users

-->pwd
 ans  =
 C:\Users
```

The `mkdir` and `rmdir` commands make/remove subdirectories.

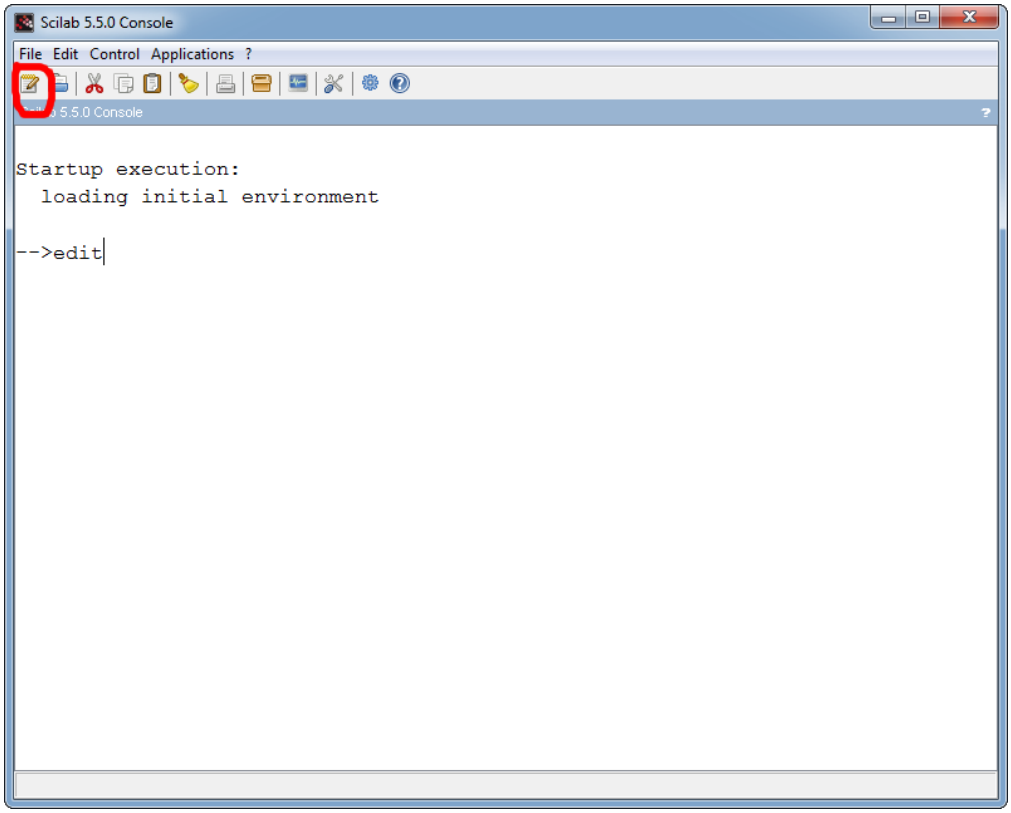
```
-->mkdir newone;

-->cd newone
 ans  =
 C:\Users\Hudson\newone

-->cd '..';

-->rmdir newone
```

The `dir` and `ls` commands list the contents of the current directory. If you want to list specific files you can enter a name. You can use the '*' character as a wild card. For example




```
-->ls *menu
 ans  =
 Start Menu
```

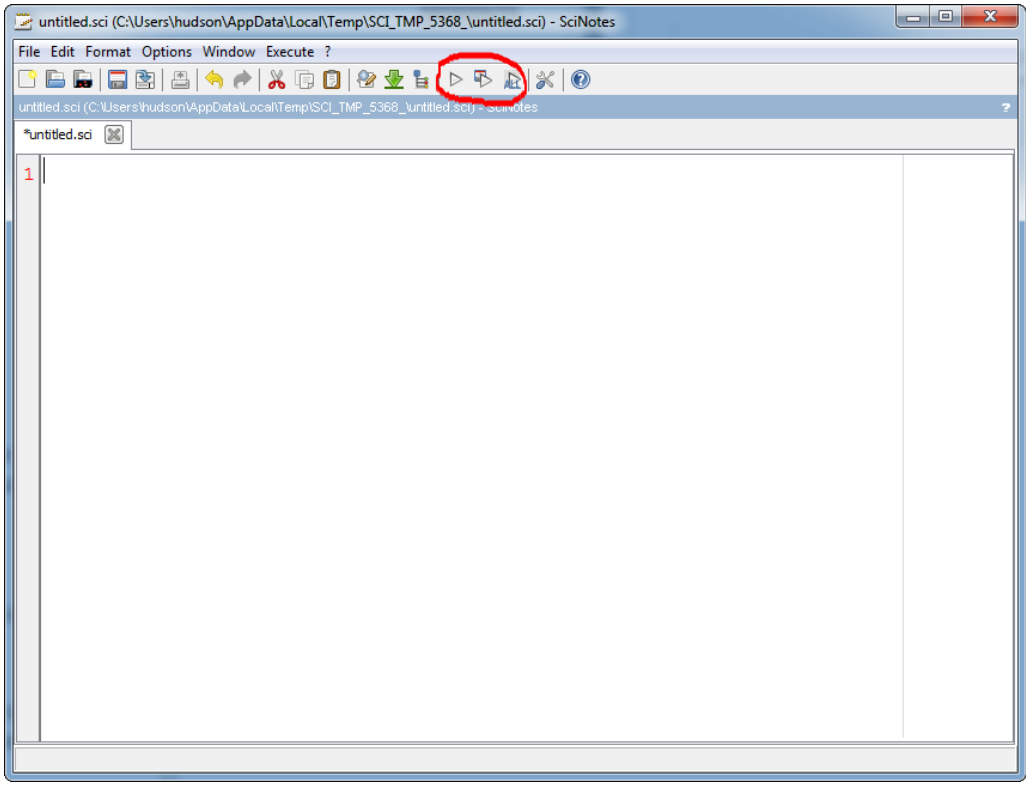
lists all file names that end in 'menu' (note this is not case sensitive).

2.2 Editing and running program files

From the command line, type `edit` to open up an editor window. Alternately there is an edit icon in the upper-left corner of the command-line environment.

The Scilab editor is called SciNotes and is shown below. The circled arrow icons execute whatever program you've entered into SciNotes. The first just executes the program as saved on disk. The second first saves your code and then executes. The third saves all open programs (you might have more than one open in SciNotes) before execution. To save without execution either use the File menu or the disk icon.

Scilab program files have the extension *.sce. In Windows Scilab file icons look something like

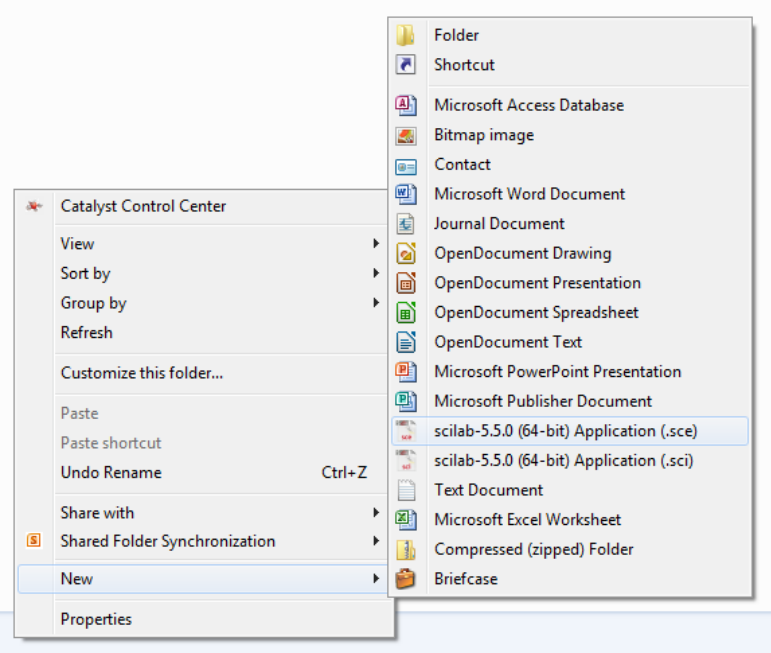


this

In Windows you can right-click on the desktop, or in a directory, and you should see a menu similar to the following. This will create an empty *.sce file named something like

```
New scilab-5.5.0 (64-bit) Application (.sce)
```

In either case double click on a Scilab file icon to open it in SciNotes. From there you can edit and execute it.



2.3 Comments

A program is just a file containing one or more Scilab commands. In principle you could have entered these one at a time on the command line, but when structured as a program they are much more convenient to develop, save, modify and rerun. This brings up a key point about programming. Generally the goal when writing a program is to create a resource that can be reused and modified in the future, either by yourself or by others. When you are writing a program you (hopefully) are clear about the purpose of the various commands and how they fit together to implement an algorithm that solves the problem of interest. However, if you come back in a month and look at the list of commands in your program it is quite common to have forgotten all this. And, obviously, if some else opens your program they probably will have little to no idea of what it's supposed to do. For this reason, programming languages allow programs to include *comments* designed to explain in words what the program commands are doing. A well-written program will contain comments that clearly explain what the program as a whole and its various components do. This allows you or another programmer to easily understand how to use and/or modify the program.

Scilab follows the C++ syntax for comments. Any text from the double back-slash symbol (//) until the end of a line is treated as a comment. Matlab uses the percent sign (%). For example the rather wordy code

```
//tryit.sce, Scott Hudson, 2011-06-03
//This program is an example, blah blah blah
x = 2; //this is an end-of-line comment
disp(sqrt(x)); //here's another
```

does exactly the same thing as the “bare” program

```
x = 2;
disp(sqrt(x));
```

The comment text is there only for the benefit of the human trying to understand the program.

2.4 Coding standards

In the spirit of using comments for understanding, adherence to a *coding standard* is an effective way to ensure that all programs have the same "look and feel" and correctly interact with one another. This is the software equivalent of the *International Mechanical Code*, or the *National Electrical Code*. Work that does not "meet code" is unacceptable and must be corrected. In this course we will not use a coding standard, but you should strive to adopt a consistent "look and feel" for the way you format your code. In particular, you should, at a minimum, perform the following command when you are done with a program.

"Edit" "Select All" , "Format" "Correct Indentation"

With this as background, we are now ready to learn about basic components with which we can build a computer program. The details will be specific to Scilab/Matlab, but the concepts are universal across almost all programing languages.

3 Flow control

Flow control refers to program statements that determine which parts of the program get executed and under what conditions based on logical (true/false) conditions. The main flow control statements we will use are: `if`, `select`, `for`, and `while`.

3.1 Logical Expressions

The most common logical expressions consist of relational statements combined with the basic logical operators: and, or, not.

```
<  less than
<= less than or equal to
>  greater than
>= greater than or equal to
== equal to
~= not equal to
&  logical and
|  logical or
~  logical not
```

The statement 1<2 asks Scilab/Matlab to answer the question, Is one less than 2? The answer is yes, so Scilab returns T for true.

```
-->1<2
 ans  =
  T
```

We use the equal sign = to assign values to variables. To test the equality of two expressions we use the double-equal symbol ==. Here we have Scilab tell us that 1 is not equal to 2 (F stands for false).

```
-->1==2
 ans  =
  F
```

On the other hand the statement "1 is not equal to 2" is true.

```
-->1~=2
 ans  =
  T
```

We can combine multiple T/F statements using the and/or/not logical operators.

```
-->(1<2)&(3>=4)
 ans  =
  F
```

This statement is false because the "and" operator "&" requires both expressions to be true. (1<2) is true but (3>=4) is false. Therefore the entire statement is false. On the other hand

```
-->(1<2)|(3>=4)
 ans  =
  T
```

since the "or" operator "|" only requires one of the statements to be true.

It is good programming practice to use parentheses to clarify the order in which the elements of an expression are evaluated. Blank space can help readability also. For example

```
( ((x>1)&(y<0)) | (z>=0) )
```

An important point is that an interval test, such as $a \le x \le b$, has to be performed as the "and" of two inequalities, such as

```
((a<=x)&(x<=b))
```

> **Exercise** 1: Using the console, "ask" Scilab/Matlab if the following statement is true: 1 is less than or equal to 2 and 4 is greater than 3.

3.2 if statement

The if statement is one of the most commonly encountered in programming. It allows you to do something in response to a logical condition. In Scilab/Matlab its syntax is

```
if expression
  statements
end
```

Note that Scilab allows an optional `then` after `expression` as in

```
if expression then
  statements
end
```

Since Matlab does *not* use the `then` syntax we will avoid it in this class. This will make our Scilab code more compatible with Matlab.

The way an if statement works is that Scilab/Matlab evaluates `expression`. If true, all statements before the `end` keyword are executed. If false the statements are skipped and execution continues at the statement immediate following the end keyword.

```
if (1<2)
  disp('that is true');
end
```

This code will write “that is true” to the command environment since 1 is less than 2. On the other hand

```
if (1>2)
  disp('that is true');
end
```

will not write anything since the expression `(1>2)` is false. The statements can be of any number and kind.

```
if (x>1)
  y = x;
  x = x^2;
end
```

An optional "default" else statement can be included

```
if expression
  statements1
else
  statements2
end
```

If `expression` is true then `statements1` are executed, and if it is false then `statements2` are executed. Additionally, any number of `elseif` statements can be included:

```
if expression1
  statements1
elseif expression2
  statements2
elseif expression3
  statements3
...
else
  statments
end
```

Here `expression1` is evaluated. If it's true then `statements1` are executed and the program continues after the `end` statement. If `expression1` is false then `expression2` is evaluated. If it is true then `statements2` are executed and the program continues after the `end` statement. This can be repeated for as many expressions as you want. If the optional `else` is present, and if none of the statements are true, the else `statements` will execute. It's important to note that no *more than one set of statements will be executed*, even if more than one of the expressions are true. Only the statements corresponding to the first true expression are executed, or failing that, the `else` statements if present.

As an example

```
if (x<1)
  y = 0;
elseif (x>1)
  y = 1;
else
  y = x;
```

```
end
```

Here is an important error to watch out for. *You should not use arrays in logical expressions*. For example, consider the following

```
x = [0,1];
disp(x<0.5);
```

This produces the output

```
T F
```

Scilab goes through the array `x` and evaluates the statement (x<0.5) for each element, producing an array out binary T/F values. Now consider

```
x = [0,1];
if (x<0.5)
  disp('x<0.5');
end
```

This produces no output, even though 0<0.5. Scilab can't both execute the disp statement and not execute it. It will only execute the statement (and only once at that) if (x<0.5) is true *for all* elements of *x*. If your intention is to run the if statement for each element of *x* independently, then you need to add a for loop (see below), as in

```
x = [0,1];
for i=1:2
  if (x(i)<0.5)
    disp('x<0.5');
  end
end
```

Exercise 2: Write a program in which you first assign a value to the variable `x`. Then the program uses an if statement to see if `x` is positive. If it is the program executes the statement `disp('x is positive');`, otherwise it executes the statement `disp('x is zero or negative');`.

3.3 select statement

Sometimes you want to perform certain actions based on the value of a variable. You can implement this using the `if … elseif … end` structure, but the `select` statement in Scilab (`switch` statement in Matlab) can be more convenient. The syntax is

```
select expression
  case v1
    statements
  case v2
    statements
  else
    statements
end
```

If `expression` has the value `v1` then the statements listed under case `v1` are executed. If it has value `v2` then the `v2 statements` are executed, and so on. If it has none of specified values then the optional `else statements` are executed. For example,

```
select x
  case 1
    msg = "x is 1";
  case 2
    msg = "x is 2";
  else
    msg = "x is something else";
end
```

3.4 for loop

The for loop is used to execute a set of statements multiple times as determined by an *index variable* or a list of values. It has the syntax

```
for x=expression
  statements
end
```

As an example

```
x = zeros(1,5);
for i=1:5
  x(i) = i;
end

-->x
 x  =
       1.    2.    3.    4.    5.
```

The for loop evaluates `1:5` as the list of numbers `[1,2,3,4,5]`. It initializes the index variable `i` to the first value (`i=1`) then runs the statement `x(i) = i` causing `x(1)` to be assigned the value 1. When the end statement is reached the for loop assigns to `i` the second value in the list (2) and once again executes the statements causing `x(2)` to be assigned the value 2. It continues until it exhausts the list.

The `expression` can be a sequence defined with colon notation, such as 1:5, or 2:0.1:3, or it can be an array. An example of the latter case would be

```
y = [1,5,-3];
for x=y
  x^3
end
```

The output is

```
 ans  =
    1.
 ans  =
    125.
 ans  =
```

```
 - 27.
```

Exercise 3: Write a program that uses a for loop to display the squares of the first six positive integers.

3.5 while loop

The while loop allows you to continue executing a set of statements as long as some logical statement is true. The syntax is

```
while expression
  statements
end
```

The loop evaluates `expression`. If it's false then execution skips to the statement following `end`. If it's true then `statements` are executed and `expression` is evaluated again. If it's still true then `statements` are once again executed. This continues until `expression` is false. At that point execution continue with the first statement after the `end` keyword. As an example, consider

```
x = 1;
while (x<10)
  x = 2*x;
end
```

This produces

```
-->x
 x  =

    16.
```

as follows

```
1. (x=1)<10 so x=2*x=2
2. (x=2)<10 so x=2*x=4
3. (x=4)<10 so x=2*x=8
4. (x=8)<10 so x=2*x=16
5. (x=0) is not <10 so the loop terminates with x=16
```

Exercise 4: Write a program that assigns a value to the variable `x`. Then use a while loop to subtract 1 from `x` as long as `x>1`. After the while loop ends, execute `disp(x);`

4 Functions

Functions allow us to break large programs into conceptual blocks that can be reused and reorganized. One form of "top down" programming is to take a large problem and break it into manageable parts. If those parts are themselves large they can be broken up into subparts and so on. A programing solution to some problem might then have the structure

```
Big problem
```

```
  Part A
    Subpart A1
    Subpart A2
  Part B
    Subpart B1
    Subpart B2
...
```

In this approach, each of the subparts might be a separate *function* that is called by its "parent" part. Those parent parts might also be functions which are called by the main program. These functions are logically separate and often exist in separate files. We can even build up "libraries" of useful functions which can be used repeatedly in different programs.

There are a few important differences between the ways Scilab and Matlab treat functions.

4.1 Simple function definition (Scilab)

In Scilab you can define a relatively simple function using the `deff` (define function) command. For example

```
deff('y=f(x)','z=x^2,y=z+1');
```

This defines `f(x)` to return the value `x^2+1` by first calculating `z=x^2` followed by `y=z+1`. Multiple statements can separated by commas as shown.

> **Exercise** 5: In the console, use `deff` to define a function `y = sinc(x)` where `sinc(x) = sin(x)/x`.

4.2 General function definition (Scilab)

For more complicated functions we use the `function ... endfunction` construct. As an example

```
function y = myabs(x)
  if (x>=0)
    y = x;
  else
    y = -x;
  end
endfunction
```

implements the absolute value operation. This function is now defined and can be called from the command line

```
-->myabs(-3)
 ans  =
    3.
```

or from within a program.

Arguments and returned variables can be arrays. It is also possible to return more than one variable. In this case an example of the syntax is

```
function [m,s] = moms(x)
  m = mean(x);
  s = mean(x.^2);
```

```
endfunction
```

This takes a vector as input and outputs two scalars

```
-->x = 1:5
 x  =
    1.    2.    3.    4.    5.

-->[a,b] = moms(x)
 b  =
    11.
 a  =
    3.
```

In Scilab multiple functions can reside in a single program file. The filename does not have to be related to any of the function names.

> **Exercise** 6: Use the `function ... endfunction` construct to implement the `sinc(x)` function.

4.3 General function definition (Matlab)

The Matlab function syntax is the same as in Scilab with the exception that `end` is used in place of `endfunction`. In fact `end` is optional. Additionally (from Matlab documentation)

> The commands and functions that comprise the new function must be put in a file whose name defines the name of the new function, with a filename extension of '.m'.

In other words if you write a function `y=myfunc(x)` it must be saved in a file named myfunc.m (an "m-file"). You can then call `myfunc(x)` from the command line or in other functions. The Scilab approach is closer to languages such as C and Fortran.

4.4 Recursive functions

A function which calls itself is said to be *recursive*. It may seem bizarre to have a function call itself, but it can be very useful. This simple example implements the factorial function.

```
function m = myfact(n)
  if (n==1)
    m = 1;
  else
    m = n*myfact(n-1);
  end
endfunction
```

If $n=1$ then $n!=1$ is the returned value. For $n>1$ we use the fact that $n!=n\cdot(n-1)!$. Suppose we call `myfact(3)`. The function wants to return `m = 3*myfact(2).` But `myfact(2)` needs to be evaluated first. Scilab opens a new instance of the function and passes the argument 2. This second function call wants to return `m = 2*myfact(1).` But `myfact(1)` needs to

be evaluated. So Scilab opens a third instance of the function and passes the argument 1. In this case the condition `(n==1)` is true and the value `m = 1` is returned to the second function call. The second function call now has the value of `m = 2*1` and returns this to the first function call. Finally the first function call can now evaluate `m = 3*2` and return this to the user.

Obviously a recursive function will work only if eventually it stops calling itself and returns a specific value.

4.5 Nested functions

It is possible to have a function defined within another function. For example, in Scilab (for Matlab simply replace `endfunction` with `end`)

```
function y = f(x)
  function v = g(u)
    v = cos(u);
  endfunction
  y = g(x)*sin(x);
endfunction

-->f(2)
 ans  =
  - 0.3784012
-->g(2)
     !--error 4
Undefined variable: g
```

Function `g(x)` is defined inside function `f(x)`. When we evaluate `f(x)` at a value of 2, that function sets `y = g(2)*sin(2)` where `g(2)=cos(2)` resulting in a returned value of

$$\cos(2)\sin(2) = -0.3784012$$

When we try to call `g(u)` outside of `f(x)`, however, we get an error. Since `g(u)` is defined inside of `f(x)` it is not visible outside of `g(u)`. We say that the *scope* of the function `g(u)` is limited to inside of `f(x)`. Now consider the following

```
function v = g(u)
  v = sin(u);
endfunction

function y = f(x)
  function v = g(u)
    v = cos(u);
  endfunction
  y = g(x)*sin(x);
endfunction

-->f(2)
 ans  =
  - 0.3784012

-->g(2)
 ans  =
    0.9092974
```

We have two functions named `g(u)`. When `g(u)` is called inside `f(x)` it clearly returns the value `g(2)=cos(2)` since we get the same output for `f(2)` as before. But now calling `g(2)` outside of `f(x)` does not produce an error. Instead it returns

```
g(2)=cos(2)=0.9092974
```

which is the definition of `g(u)` given *outside* of `f(x)`. Generally speaking, variables and functions are only "visible" inside the function in which they are defined, and inside all nested functions. Thus variables in the "main program" are globally visible. If a variable or function is defined a second time inside a function, that definition overrides the previous definition *within the scope of that function* (and nested functions).

4.6 Variable scope and global variables

Now consider the behavior of the following five similar programs. In case 1, $a=2$ is defined in the main program. Function $f(x)$, being nested within the main program, can "see" the value of *a*, so it calculates $f(2)=4$. In case 2, $f(x)$ changes the value of *a* to $a=4$, and calculates $f(2)=8$, but notice that outside of the function definition we still have $a=2$. The scope of the change $a=4$ is only within the function declaration, not outside it. In case 3 we initially have $a=2$, but before calling $f(x)$ we change the value to $a=4$. This change is visible inside the function, so $f(2)=8$. In case 4 we declare the variable *a* to be *global* and assign it the value $a=2$. Inside the function we also declare *a* to be global and assign it the value $a=4$. As in case 2, this results in $f(2)=8$ but it also changes the value of *a* outside the function (globally).

```
a = 2; //case 1

function y = f(x)
  y = a*x;
endfunction

disp('f(2)='+string(f(2))+', a='+string(a));

 f(2)=4, a=2


a = 2; //case 2

function y = f(x)
  a = 4;
  y = a*x;
endfunction

disp('f(2)='+string(f(2))+', a='+string(a));

 f(2)=8, a=2


a = 2; //case 3

function y = f(x)
  y = a*x;
endfunction

a = 4;
disp('f(2)='+string(f(2))+', a='+string(a));
```

```
 f(2)=8, a=4
```

```
global a //case 4
a = 2;

function y = f(x)
  global a
  a = 4;
  y = a*x;
endfunction

disp('f(2)='+string(f(2))+', a='+string(a));
```

```
 f(2)=8, a=4
```

4.7 Variable number of arguments

It can be very useful for functions to have a variable number of input and/or output arguments. The following function has two input and two output arguments.

```
function [y,ySize] = f(x,z)
  [nargout,nargin] = argn(); //Scilab needs this, Matlab doesn't
  if (nargin==1)
    y = x^2;
  else
    y = z*x^2;
  end
  if (nargout==2)
    if (abs(y)>=10)
      ySize = 'big';
    else
      ySize = 'small';
    end
  end
endfunction
```

However we can call it with a single input and assign its output to a single variable

```
u = f(3);
disp('u='+string(u));
```

```
 u=9
```

Or supply both input variables and assign its output to two variables

```
[u,v] = f(3,2);
disp('u='+string(u)+' and v='+v);
```

```
 u=18 and v=big
```

The first line of the function assigns values to `nargin`, the number of input arguments, and `nargout`, the number of output arguments. (This assignment is needed in Scilab; in Matlab these variables are automatically assigned.) If `nargin==1` we know that only an *x* value was

passed to the function. Otherwise we know that both *x* and *z* values are available. Likewise, if `nargout==1` we know that only *y* needs to be calculated. Otherwise we also assign a string value to `ySize`.