Introduction to IDL

Week 1
Rachel McCrary
3-34-2009
# Introduction to Programming: IDL Short Course

Scheduling: Tuesdays 3:00-5:00 in ATS 101  
Instructor: Rachel McCrary  
ATS West 127 (the new building)  
[rachel@atmos.colostate.edu](mailto:rachel@atmos.colostate.edu)  
(970) 491-3334  
Class Website: [http://www.atmos.colostate.edu/gradprog/programming/](http://www.atmos.colostate.edu/gradprog/programming/)

<table>
<thead>
<tr>
<th>Date</th>
<th>Topics</th>
</tr>
</thead>
</table>
| 24 - March | • Why choose IDL?  
• Getting started using IDL  
• interactive vs. compiled modes in IDL  
• variables, types, conversions  
• arithmetic operations  
• arrays and array operations (basics)  
• Control Statements (if, for, while etc.)  
• procedures and functions |
| 31 - March | • reading and writing data  
• Introduction to plotting in IDL  
• line plots, bar plots, scatter plots  
• modifying axes  
• multiple plots on a page  
• adding color to a plot  
• adding a legend  
• creating PostScripts and other file types  
• changing the font  
• special characters |
| 7 - April  | • contour plotting and mapping  
• interesting problems with global mapping  
• missing data  
• color bars  
• creating your own color tables  
• vector plots, streamlines  
• plotting multiple variables |
| 14 - April | • statistical tools in IDL  
• EOFs + matrix operations  
• power spectra and FFT  
• optimizing your code  
• Give me a problem, I'll help you find a solution :) |
Why use IDL?

• IDL offers interactive and compiled programming modes. (Fortran offers compiled mode only)

• IDL is optimized to perform array operations. (Fortran is optimized to perform scalar operations)

• IDL variables may be created or redefined with a new type, size, or value at any time. (Fortran variables may be defined at runtime with a new size or value only)

• IDL includes built-in routines for visualization, numerical analysis and graphical user interface development. (Fortran requires external libraries)
Goals for today!

1. Learn how to open and start using IDL
2. Learn how to declare variables, and convert variables to different types
3. learn how to declare arrays and types
4. understand the basics of array operations
5. learn how to develop your own programs and functions and how to call programs and functions
The real answer is **VISUALIZATION!**

**b) CM 2.0**

- **Precipitation**
- **Evapotranspiration**
- **Solar Radiation Flux**
- **Soil Moisture Variations**

---

**CM 2.0**

- Observations

**HadISST**

- Period (years)

---

**Correlations**

- Precipitation and Evapotranspiration:
  - $r = 0.507$
  - $a = 0.239$
Background

• If you are acquainted with C or Fortran some of the concepts for using IDL will be familiar.

• Unfortunately (or well maybe fortunately) IDL differs from these languages in significant ways.

• It is a mistake to think of IDL as a variant of Fortran.

• It pays to think of IDL in terms of C (rather than Fortran) particularly when it comes to dynamic variable types and efficiency of array processing ... of course this only helps if you know C.
Getting Started

1. Get IDL installed on your computer (talk to your IT/Computer person about this).

2. Figure out which version of IDL you are using ... hopefully it is more recent than IDL 5.6 (current version is IDL 7.0).

3. IDL can be started in two different modes (start both from a terminal window). *note* Mac users should use x-term
   - IDL Development Environment (type -idlde).
   - IDL Command Line (type -idl) only available on unix/linux machine.

4. To exit IDL just type exit at the IDL prompt.
idl

Terminal — idl — 80×24

Last login: Fri Mar 13 15:21:41 on tty2?
oasis> idl
IDL Version 7.8, Mac OS X (darwin ppc m32). (c) 2007, ITT Visual Information Solutions
Installation number: 208110-14.
Licensed for use by: Colorado State University

IDL>
Getting Help

• To open the IDL online help type “?” in a terminal.

• Websites

  http://www.dfanning.com/

  http://astro.berkeley.edu/~jbloom/IDL/

  http://idlastro.gsfc.nasa.gov/homepage.html

  http://groups.google.com/group/comp.lang.idl-pvwave/topics?pli=1

• Books


  Liam E. Gumley, practical IDL programming
## Interactive vs Compiled Modes

<table>
<thead>
<tr>
<th>Interactive mode</th>
<th>Compiled mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commands are typed at the IDL prompt and executed when you press the enter key</td>
<td>Programs consisting of sequences of IDL commands are created and executed.</td>
</tr>
<tr>
<td>• Good for prototyping and interactive analysis</td>
<td>• There are two types of program units - procedures and functions.</td>
</tr>
<tr>
<td>• provides immediate feedback in numerical and visual form</td>
<td>• Can be reused by you and your colleagues</td>
</tr>
<tr>
<td></td>
<td>• portable between different IDL platforms.</td>
</tr>
</tbody>
</table>

example: ‘hello_world.pro’
the print command

• In the previous examples, the print command for IDL was used.

• print is used to print information to the screen

• print is good for debugging, writing error statements, and prompting the user

print, [Expression1, Expression2 etc]

eexample: example_print_command.pro
Variables

• IDL offers more flexibility for variable typing than either C or Fortran because it must allow new variables to be created in both the interactive and compiled modes.

• IDL allows you to create new variables at any time and also allows you to redefine existing variables at any time.

• The ability to create new variables dynamically in IDL provides great flexibility. However it means that you must diligently keep track of the types of all variables in your IDL programs.
Examples

create a 32-bit floating-point scalar variable with a value 1.0, and then verify its type using `help`

```
IDL> var = 1.0
IDL> help, var
VAR       FLOAT  =   1.00000
```

redefine the variable as a 16-bit signed integer with a value of 1,024, and then verify its type.

```
IDL> var = 1024
IDL> help, var
VAR       INT     =   1024
```
Keeping Track

• The `help` command is a good way to keep track of variable types and sizes

• For scalar variables, `help` prints the name, type and value of the argument

```idl
IDL> value = 10.34
IDL> help, value
VALUE    FLOAT    =   10.3400
```

```idl
IDL> char = 'Hello World'
IDL> help, char
CHAR     STRING    = 'Hello World'
```
## Numeric Data Types

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Explanation</th>
<th>Bits</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>Unsigned integer</td>
<td>8</td>
<td>0 to 255</td>
</tr>
<tr>
<td>int</td>
<td>Signed integer</td>
<td>16</td>
<td>-32,768 to 32,768</td>
</tr>
<tr>
<td>uint</td>
<td>Unsigned integer</td>
<td>32</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>long</td>
<td>Signed integer</td>
<td>32</td>
<td>-2^{31} to 2^{31} - 1</td>
</tr>
<tr>
<td>ulong</td>
<td>Unsigned integer</td>
<td>32</td>
<td>0 to 2^{32} - 1</td>
</tr>
<tr>
<td>long64</td>
<td>Signed integer</td>
<td>64</td>
<td>-2^{63} to 2^{63} - 1</td>
</tr>
<tr>
<td>ulong64</td>
<td>Unsigned integer</td>
<td>64</td>
<td>0 to 2^{64} - 1</td>
</tr>
<tr>
<td>float</td>
<td>IEEE floating-point</td>
<td>32</td>
<td>-10^{38} to 10^{38} - 1</td>
</tr>
<tr>
<td>double</td>
<td>IEEE floating-point</td>
<td>64</td>
<td>-10^{308} to 10^{308} - 1</td>
</tr>
<tr>
<td>complex</td>
<td>Real-imaginary pair</td>
<td>64</td>
<td>see float</td>
</tr>
<tr>
<td>dcomplex</td>
<td>Real-imaginary pair</td>
<td>128</td>
<td>see double</td>
</tr>
</tbody>
</table>
Nonnumeric data types

- In addition to numeric data types, IDL includes several nonnumeric data types that add programming flexibility

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Character string (0-32,767 characters)</td>
</tr>
<tr>
<td>struct</td>
<td>Container for one or more variables</td>
</tr>
<tr>
<td>pointer</td>
<td>References to a dynamically allocated variable</td>
</tr>
<tr>
<td>objref</td>
<td>Reference to an object structure</td>
</tr>
</tbody>
</table>

* object-oriented programming can be done in IDL, but requires a different approach and will not be discussed in this class
Variable Names

• must begin with a letter
• may be up to 128 characters in length
• may include letters a-z, digits 0-9, and ‘_’
• idl reserved words cannot be used (see table)
• be careful not to use IDL built in commands/functions as variable names (to check try typing ? ‘variable name’ at the IDL prompt)
<table>
<thead>
<tr>
<th>Reserved words</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>case</td>
</tr>
<tr>
<td>common</td>
</tr>
<tr>
<td>do</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>endcase</td>
</tr>
<tr>
<td>endelse</td>
</tr>
<tr>
<td>endfor</td>
</tr>
<tr>
<td>endif</td>
</tr>
<tr>
<td>endrep</td>
</tr>
<tr>
<td>endwhile</td>
</tr>
<tr>
<td>function</td>
</tr>
<tr>
<td>ge</td>
</tr>
<tr>
<td>goto</td>
</tr>
<tr>
<td>gt</td>
</tr>
<tr>
<td>if</td>
</tr>
<tr>
<td>le</td>
</tr>
<tr>
<td>lt</td>
</tr>
<tr>
<td>mod</td>
</tr>
<tr>
<td>ne</td>
</tr>
<tr>
<td>not</td>
</tr>
<tr>
<td>of</td>
</tr>
<tr>
<td>or</td>
</tr>
<tr>
<td>pro</td>
</tr>
<tr>
<td>repeat</td>
</tr>
<tr>
<td>then</td>
</tr>
<tr>
<td>until</td>
</tr>
<tr>
<td>while</td>
</tr>
<tr>
<td>xor</td>
</tr>
<tr>
<td>on_ioerror</td>
</tr>
<tr>
<td>Data Type</td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>byte</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>uint</td>
</tr>
<tr>
<td>long</td>
</tr>
<tr>
<td>ulong</td>
</tr>
<tr>
<td>long64</td>
</tr>
<tr>
<td>ulong64</td>
</tr>
<tr>
<td>float</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>complex</td>
</tr>
<tr>
<td>dcomplex</td>
</tr>
<tr>
<td>string</td>
</tr>
</tbody>
</table>

example: variable_types.pro
Converting Types

- IDL provides functions that convert variables from one type to another.
- example: Convert a variable to long type, the using the long() function:

```idl
IDL> x = 3.14159
IDL> help, x
X FLOAT = 3.14159
IDL> x = long(x)
IDL> help, x
X LONG = 3
```

example: convert_variables.pro
“short” integer problems

• A signed integer in IDL is only 16 bits

• This means “short” integers can only be set to values between -32,768 to 32,768.

• Why is this a problem?

    IDL> x = 425600L
    IDL> help, x
    X         LONG      =     425600
    IDL> x = fix(x)
    IDL> help, x
    X         INT       =   32384
Convert to String

• use `string()` to convert numbers to strings
• use `strtrim()` to remove leading/trailing blanks
• use `strcompress()` to all white space compressed to a single space or completely removed
• use `strmid()` to extract a substring from a string expression

example: string_conversions.pr
Arithmetic operations

- Like all other computer languages, IDL follows the algebraic order of operations:
  1. Parenthesis ( )
  2. Exponentiation ^
  3. Multiplication and division *,/
  4. Addition and subtraction +,-

- Please Excuse My Dear Aunt Sally
Arithmetic Operations

• Try the following, what do you get?
  12-3+4*6
  (12-3+4)*6
  4^2+3
  4^(2+3)
  4^2*0.5
  4^(2*0.5)

Example: order_operations.pro
Arrays

- Arrays can have up to 8 dimensions
- Arrays can be formed from any IDL data type (int, float, double, string)
- The compact syntax allows arrays to be processed without the use of loops (very different from Fortran where all data processing is done in loops)
- Array operations are optimized for speed. Processing the entire array operation is much faster than processing each array element in turn using a for loop.
- One of the secrets to becoming an effective IDL programmer is learning how to use arrays effectively.
  - Array operations are much faster than loops
  - Array syntax is more compact than the corresponding loop construct
Creating Arrays

• Arrays are created and referenced in IDL by using square bracket characters [ ]. This is to help distinguish them from a call to a function command.

IDL> x = [0, 1, 2, 3, 4, 5]
IDL> help, x
X               INT       = Array[6]
IDL> print, x
  0       1       2       3       4       5

• Here we have created a 6 element integer array.
Multidimensional arrays

- Multidimensional arrays can be created by using nested square brackets

```
IDL> x = [[0,1,2],[3,4,5],[6,7,8]]
IDL> help, x
X               INT       = Array[3, 3]
IDL> print, x
0   1   2
3   4   5
6   7   8
```

- Here we have created a two dimensional integer array, with three elements contained in each dimension
Using Array Subscripts

• Array subscripts start at 0, not 1

• to print the values in the first and third elements in array x do the following:

```
IDL> x = [0,1,2,3,4,5]
IDL> print, x[0]
  0
IDL> print, x[2]
  2
```

• Array subscripts have their upper and lower bounds separated by a colon

```
IDL> print, x[0:2]
  0  1  2
```
Array Storage

- Arrays are stored in memory using the [column, row] format. This is different from fortran where arrays are stored as [row, column].

- This will be important when reading in data and plotting contour maps.

- Print the all values in the first column of array x
  
  IDL> x = [[1,2,3],[4,5,6],[7,8,9]]
  IDL> print, x[*,0]
  
  1 2 3

- Print all values in the first row of array x
  
  IDL> print, x[0,*]
  
  1
  4
  7
Append to an existing Array

• Start with a 1-D, 4 element integer array

```
IDL> x = [0,1,2,3]
IDL> help, x
X               INT       = Array[4]
IDL> print, x
  0      1      2      3
```

• Now add a 5th element to array x

```
IDL> x = [x,5]
IDL> help, x
X               INT       = Array[5]
IDL> print, x
  0      1      2      3      5
```
Functions to create arrays

- create arrays of given type with every element initialized to zero, or with every element set to its array index value

- To generate a zeroed floating point array use `fltarr()`

```idl
IDL> x = FLTARR(5)
IDL> help, x
X               FLOAT     = Array[5]
IDL> print, x
0.00000    0.00000    0.00000    0.00000   0.00000
```

- To generate an indexed integer array use `indgen()`

```idl
IDL> y = INDGEN(5)
IDL> help, y
Y               INT      = Array[5]
IDL> print, y
0    1    2    3    4
```
### Functions to create arrays

<table>
<thead>
<tr>
<th>Data type</th>
<th>Zeroed array</th>
<th>Index array</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>bytarr()</td>
<td>bindgen()</td>
</tr>
<tr>
<td>int</td>
<td>intarr()</td>
<td>indgen()</td>
</tr>
<tr>
<td>uint</td>
<td>uintarr()</td>
<td>uindgen()</td>
</tr>
<tr>
<td>long</td>
<td>lonarr()</td>
<td>lindgen()</td>
</tr>
<tr>
<td>ulong</td>
<td>ulonarr()</td>
<td>ulindgen()</td>
</tr>
<tr>
<td>long64</td>
<td>lon64arr()</td>
<td>l64indgen()</td>
</tr>
<tr>
<td>ulong64</td>
<td>ulon64arr()</td>
<td>ul64indgen()</td>
</tr>
<tr>
<td>float</td>
<td>fltarr()</td>
<td>findgen()</td>
</tr>
<tr>
<td>double</td>
<td>dblarr()</td>
<td>dindgen()</td>
</tr>
<tr>
<td>complex</td>
<td>complexarr()</td>
<td>cindgen()</td>
</tr>
<tr>
<td>dcomplex</td>
<td>dcomplexarr()</td>
<td>dcindgen()</td>
</tr>
<tr>
<td>string</td>
<td>strarr()</td>
<td>sindgen()</td>
</tr>
</tbody>
</table>

*example: create_lat_lon.pro*
Totaling an array

• IDL is optimized to perform array operations.

• Processing the entire array operation is much faster than processing each array element in turn using a for loop.

• In fortran77, how would we sum over all elements in an array? - think DO loops!

• In IDL, you can just use the total() function!

example: loop_comparison.pro
Array Properties

• IDL provides a suite of functions that are designed to return information about the properties of an array.
  
  - n_elements() - number of array elements
  - size() - Array size and type info
  - min() and max() - Min and Max array values
  - mean() - Mean of array values
  - variance() - variance of array values
  - stddev() - standard deviation of array values
  - moment() - mean, variance, skew, kurtosis, standard deviation
  - total() - sum of array values
  - median() - median of array values

examples: array_character & where_example.pro
Locating values within an array

• One of the most common tasks when working with arrays is locating elements that meet a certain condition, such as values that are greater than a threshold.

• The `where` function is used for this purpose and it is one of the most useful array-related functions in IDL.

• `where` returns the indices of the nonzero elements in an array or expression.

example: `where_example.pro`
Locating values within an array

IDL> arr = indgen(9) * 10
IDL> print, arr
0      10      20      30      40      50      60
  70      80
IDL> index = where(arr gt 35)
IDL> print, arr[index]
  40      50      60      70      80
IDL> print, index
  4       5       6       7       8
Using \textit{where} in multidimensional arrays

- When the argument to \textit{where} is a 2-D or higher dimension array, the returned array of index values always appears as though the input array were one-dimensional.

- The command \texttt{array_indices} converts one-dimensional subscripts of an array into corresponding multi-dimensional subscripts.

\texttt{example: max\_location.pro}
Arithmetic Operations on Arrays

- Loops are not required to process arrays in IDL.
- It is generally more efficient to perform an arithmetic operation on an array than it is to perform the same operation on each element of the array in a loop.
- You should always strive to find ways to use array operations in IDL because it will result in programs that are more efficient (fast) and contain fewer lines of code.
Arithmetic Operations on Arrays

• Operators can be applied to arrays just as easily as they can be applied to scalars.

• If any one variable in an expression is an array, the result will be an array.

• If an expression contains arrays of different sizes, the results will have as many elements as the smallest array in the expression.

• If an expression contains arrays with different numbers of elements and dimensions, the result array will have as many elements and as many dimensions as the smallest array.

• If an expression contains arrays with the same number of elements but different dimensions, the result array will have as many dimensions as the leftmost array in the expression.

example: array_arith.pro
Array & Matrix Multiplication

- arrays are stored in memory in [column, row] format
- this differs from the normal mathematical concept of a matrix which is [row, column]
- IDL includes two operators that provide matrix multiplication functionality
  - # - array multiplication operator multiplies the columns of the first array by the rows of the second array
  - ## - matrix multiplication operator multiplies the rows of the first array by the columns of the second array

example: arr_matrix_mulip.pro
Array Multiplication (#)

\[
a[0,0] \times b[0,0] + a[0,1] \times b[1,0]
\]

\[
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6
\end{array}
\]

\[
\begin{array}{cc}
-1 & 0 \\
1 & 2 \\
3 & 4
\end{array}
\]

\[
\begin{array}{ccc}
-1 & -2 & -3 \\
9 & 12 & 15 \\
19 & 26 & 33
\end{array}
\]

transpose ( )

\[
\begin{array}{ccc}
-1 & -2 & -3 \\
9 & 12 & 15 \\
19 & 26 & 33
\end{array}
\]

\[
\begin{array}{ccc}
-1 & 9 & 19 \\
-2 & 12 & 26 \\
-3 & 15 & 33
\end{array}
\]
Matrix Multiplication (##)

\[ a_{0,0} \times b_{0,0} + a_{1,0} \times b_{0,1} \]

\[
\begin{array}{cccc}
-2 & -1 & 2 & 3 \\
0 & 1 & 4 & 5 \\
2 & 3 & 6 & 7 \\
4 & 5 & 8 & 9 \\
\end{array}
\]

\[
\begin{array}{cccc}
-10 & -13 & -16 & -19 \\
6 & 7 & 8 & 9 \\
22 & 27 & 32 & 37 \\
38 & 47 & 56 & 65 \\
\end{array}
\]
Procedures & Functions

• IDL programs fall into two classes: procedures and functions.

• A procedure normally encapsulates several related operations into one program unit.

• A function normally encapsulates one operation into a program unit and returns a single result variable.
Procedures

- A procedure begins with a `pro` statement and ends with an `end` statement.
- The `pro` statement must contain at least a procedure name and may contain an argument list and/or a keyword list.
- A procedure with no arguments:
  
  ```
  PRO hello
      print, 'hello world'
  END
  ```

- A procedure with three parameters:
  
  ```
  PRO read_image, image, date, time
      ; statements ...
  END
  ```

- A procedure with three parameters and two optional keywords:
  
  ```
  PRO print_image, image, date, time, landscape=landscape,$
      color = color
      ; statements ...
  END
  ```
Functions

- A function begins with a FUNCTION statement, includes a RETURN statement and ends with an END statement.
- Functions must include at least a function name and may contain an argument list and/or keyword list.
- A function with one parameter:
  ```
  PRO f_to_c, deg_f
  print, 'hello world'
  END
  ```
- A function with three parameters and two optional keywords:
  ```
  PRO get_bounds, image, lat, lon,$
  polar=polar, normal=normal
  ; statements
  END
  ```
Arguments and Keywords

• Using the predefine “contour” command for IDL as an example:

```
contour, peak, lon, lat,Xstyle=1,Ystyle=1,$
   Levels = vals,/follow, C_labels = $[1,0,1,0,0,1,1,0]
```

• `peak`, `lon`, `lat` are arguments or variables. They are used to pass information into or out of the command or program. They are also called positional parameters and must be passed in the correct order (from left to right).

• `Xstyle`, `Ystyle`, `Follow`, `Levels`, and `C_labels` are keyword parameters. Unlike positional parameters, they can come in any order. By convention keyword parameters are always optional.

• Keywords can be set to a particular value (`Xstyle = 1`) or to a vector of values, or set with a slash character (`/follow`)
Creating Functions/Procedures

• Examples of functions:
  • surface_area_of_sphere.pro
  • volume_of_sphere.pro
  • sphere_info.pro

• Example of procedures:
  • test_if.pro
  • spheres.pro
Control Statements

- Control statements allow the programmer to specify the order in which computations will be carried out.

- If statements are very important for checking to see if procedures/functions have been called by the user correctly.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>If a condition is true, execute statement(s)</td>
</tr>
<tr>
<td>case</td>
<td>Select one case to execute from a list of cases</td>
</tr>
<tr>
<td>for</td>
<td>For a specified number of times, execute a statement loop</td>
</tr>
<tr>
<td>while</td>
<td>while a condition is true, execute a statement loop</td>
</tr>
<tr>
<td>repeat</td>
<td>Repeat a statement loop until a condition is true</td>
</tr>
<tr>
<td>return</td>
<td>Return control to the calling function or procedure</td>
</tr>
<tr>
<td>goto</td>
<td>Go to a label</td>
</tr>
<tr>
<td>switch</td>
<td>Branch to a case in a list of cases</td>
</tr>
<tr>
<td>break</td>
<td>Break out of a loop, case statement, or switch statement</td>
</tr>
<tr>
<td>continue</td>
<td>Continue execution on the next iteration of a loop</td>
</tr>
</tbody>
</table>
if Statements

if *condition* the *statement*

if *condition* then begin

  *statement(s)*

endif

if *condition* then *statement* else *statement*

if *condition* then begin

  *statement(s)*

endif else begin

  *statement*

endelse

example:test_if.pro
nested if statements

if \textit{condition1} then \textit{statement1} else $ \$
if \textit{condition2} then \textit{statement2} else 4
...
if \textit{conditionn} then \textit{statementn} else \textit{statementx}

• If none of the statements are true then statement x will be executed
for loops

for i = v1, v2 do statement
for i = v1, inc do statement
for i = v1, v2, inc do begin
    statement(s)
endfor

example: test_for.pro
Questions???