IDL Primer - Week I

John Rausch

3 December 2009

A distillation of a CSU class
What is IDL?

• Interactive Data Language

• Like MATLAB, IDL is a high level computing language and visualization tool.

• It allows for quick manipulation and visualization of data. *Try that with FORTRAN or C!*

• Used heavily in medical imaging (radiographs), physics/astronomy and of course, atmospheric science
Why use IDL?

• IDL can execute statements interactively or in “compiled” mode.

• IDL is designed for array operations (fewer loops needed)

• Variables are dynamic: Can be created or redefined at runtime

• Visualization routines are built-in
Starting IDL’s two different flavors

• Comand Line Environment: type `idl` in a terminal window

• IDL Development Environment (graphical): type `idlde` in a terminal window
IDLDE - IDL Development Environment

```idl
FOR i=0,68 DO BEGIN
  ind = WHERE(h_map EQ h[i] AND x_map EQ x[i])
  rcs_map[ind] = data_res[i]
ENDFOR

STOP

END
```

`INDGEN`

Syntax

```
Result = INDGEN(D1[, ..., Dn [, /BYTE | | /COMPLEX | | /DWORD | | /FLOAT | | /LP64 | | /LONG | | /STRING | | /UL64 | | /ULONG | | /TYPE=value )]
```

Return Value

Each element of the returned integer array is set to the value of its one-dimensional subscript.

Arguments

- `D1`

Either an array or a series of scalar expressions specifying the result. If a single argument is given, it is repeated for each index.

Go To

- All Topics
- Search
- Related Topics
- Bookmarks
- Index
IDL (Old School)

[4:18pm] John 11> idl
IDL Version 7.0.4, Mac OS X (darwin i386 m32). (c) 2008, ITT Visual Information Solutions
Installation number: 17991.
Licensed for use by: Univ Wisconsin-Madison

IDL> PRINT, 2 + 2
   4
IDL> PRINT, 2 + 2.0
   4.00000
IDL>
## Interactive & “Compiled” Modes

### Interactive mode

Commands are typed at the IDL prompt and executed when you press the enter key

- Good for prototyping and interactive analysis
- Provides immediate feedback in numerical and visual form

### Compiled mode

Programs consisting of sequences of IDL commands are created and executed.

- There are two types of program units - procedures and functions.
- Can be reused by you and others
- Portable between different IDL platforms.
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]
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> value = 10.34
IDL> help, value
VALUE    FLOAT  =  10.3400
```

```
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.
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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>endfor</td>
</tr>
<tr>
<td>begin</td>
<td>endif</td>
</tr>
<tr>
<td>case</td>
<td>endrep</td>
</tr>
<tr>
<td>common</td>
<td>endwhile</td>
</tr>
<tr>
<td>do</td>
<td>eq</td>
</tr>
<tr>
<td>else</td>
<td>for</td>
</tr>
<tr>
<td>end</td>
<td>function</td>
</tr>
<tr>
<td>endcase</td>
<td>ge</td>
</tr>
<tr>
<td>endelse</td>
<td>goto</td>
</tr>
<tr>
<td></td>
<td>gt</td>
</tr>
<tr>
<td></td>
<td>if</td>
</tr>
<tr>
<td></td>
<td>le</td>
</tr>
<tr>
<td></td>
<td>lt</td>
</tr>
<tr>
<td></td>
<td>mod</td>
</tr>
<tr>
<td></td>
<td>ne</td>
</tr>
<tr>
<td></td>
<td>not</td>
</tr>
<tr>
<td></td>
<td>of</td>
</tr>
<tr>
<td></td>
<td>on_ioerror</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>pro</td>
</tr>
<tr>
<td></td>
<td>repeat</td>
</tr>
<tr>
<td></td>
<td>then</td>
</tr>
<tr>
<td></td>
<td>until</td>
</tr>
<tr>
<td></td>
<td>while</td>
</tr>
<tr>
<td></td>
<td>xor</td>
</tr>
</tbody>
</table>
# Type Declarations & Conversions

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Syntax example</th>
<th>Conversion Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>1B</td>
<td>byte()</td>
</tr>
<tr>
<td>int</td>
<td>1, 1S</td>
<td>fix()</td>
</tr>
<tr>
<td>uint</td>
<td>IU, IUS,</td>
<td>uint()</td>
</tr>
<tr>
<td>long</td>
<td>IL</td>
<td>long()</td>
</tr>
<tr>
<td>ulong</td>
<td>IUL</td>
<td>ulong()</td>
</tr>
<tr>
<td>long64</td>
<td>ILL</td>
<td>long64()</td>
</tr>
<tr>
<td>ulong64</td>
<td>IULL</td>
<td>ulong64()</td>
</tr>
<tr>
<td>float</td>
<td>1.0, 1.0E+0, 1.</td>
<td>float()</td>
</tr>
<tr>
<td>double</td>
<td>1.0D, 1.0D+0</td>
<td>double()</td>
</tr>
<tr>
<td>complex</td>
<td>use complex()</td>
<td>complex()</td>
</tr>
<tr>
<td>dcomplex</td>
<td>use dcomplex()</td>
<td>dcomplex()</td>
</tr>
<tr>
<td>string</td>
<td>‘this is a string’</td>
<td>string()</td>
</tr>
</tbody>
</table>
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> x = 3.14159
IDL> help, x
X FLOAT = 3.14159
IDL> x = long(x)
IDL> help, x
X LONG = 3
“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
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
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
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 = [5,6,7,8,9,10]
IDL> print, x[0]
5
IDL> print, x[2]
7
```

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

```
IDL> print, x[0:2]
5       6       7
```
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
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
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 \texttt{fltarr()}

\begin{verbatim}
IDL> x = FLTARR(5)
IDL> help, x
X               FLOAT    = Array[5]
IDL> print, x
0.00000    0.00000    0.00000    0.00000   0.00000
\end{verbatim}

• To generate an indexed integer array use \texttt{indgen()}

\begin{verbatim}
IDL> y = INDGEN(5)
IDL> help, y
Y               INT     = Array[5]
IDL> print, y
0      1    2    3    4
\end{verbatim}
## 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>ulongarr()</td>
<td>ulindgen()</td>
</tr>
<tr>
<td>long64</td>
<td>lon64arr()</td>
<td>l64indgen()</td>
</tr>
<tr>
<td>ulong64</td>
<td>ulong64arr()</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>
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!
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
Taking a closer look at the size function

IDL> x = FLTARR(10,20)
IDL> PRINT, SIZE(x)

<table>
<thead>
<tr>
<th># Dims</th>
<th># Cols</th>
<th># Rows</th>
<th>Var Type</th>
<th>#Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>20</td>
<td>4</td>
<td>200</td>
</tr>
</tbody>
</table>
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.
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, index
    4    5    6    7    8
IDL> print, arr[index]
    40    50    60    70    80
Using `where` in multidimensional arrays

- When the argument to `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 `array_indices` converts one-dimensional subscripts of an array into corresponding multidimensional subscripts.
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.
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]  Watch out!!!
• 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
Array Multiplication (#)

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>-1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>-1</th>
<th>-2</th>
<th>-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-2</td>
<td>-3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>19</th>
<th>26</th>
<th>33</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>26</td>
<td>33</td>
<td>15</td>
</tr>
</tbody>
</table>
Matrix Multiplication (###)

\[ \begin{array}{ccc}
-2 & -1 & \\
0 & 1 & \\
2 & 3 & \\
4 & 5 & \\
\end{array} \] \times \begin{array}{cccc}
2 & 3 & 4 & 5 \\
6 & 7 & 8 & 9 \\
\end{array} = \begin{array}{cccc}
\text{[a\text{[0,0]} \times b\text{[0,0] + a\text{[1,0]} \times b\text{[0,1]}]} & \text{[a\text{[0,0]} \times b\text{[1,0] + a\text{[1,0]} \times b\text{[0,1]}]} & \text{[a\text{[0,0]} \times b\text{[2,0] + a\text{[1,0]} \times b\text{[1,1]}]} & \text{[a\text{[0,0]} \times b\text{[3,0] + a\text{[1,0]} \times b\text{[2,1]}]} \\
\text{[a\text{[0,0]} \times b\text{[0,1]} + a\text{[1,0]} \times b\text{[1,1]}]} & \text{[a\text{[0,0]} \times b\text{[1,1]} + a\text{[1,0]} \times b\text{[2,2]}]} & \text{[a\text{[0,0]} \times b\text{[2,1]} + a\text{[1,0]} \times b\text{[3,2]}]} & \text{[a\text{[0,0]} \times b\text{[3,1]} + a\text{[1,0]} \times b\text{[4,2]}]} \\
\text{[a\text{[0,0]} \times b\text{[0,2]} + a\text{[1,0]} \times b\text{[1,2]}]} & \text{[a\text{[0,0]} \times b\text{[1,2]} + a\text{[1,0]} \times b\text{[2,3]}]} & \text{[a\text{[0,0]} \times b\text{[2,2]} + a\text{[1,0]} \times b\text{[3,3]}]} & \text{[a\text{[0,0]} \times b\text{[3,2]} + a\text{[1,0]} \times b\text{[4,3]}]} \\
\text{[a\text{[0,0]} \times b\text{[0,3]} + a\text{[1,0]} \times b\text{[1,3]}]} & \text{[a\text{[0,0]} \times b\text{[1,3]} + a\text{[1,0]} \times b\text{[2,4]}]} & \text{[a\text{[0,0]} \times b\text{[2,3]} + a\text{[1,0]} \times b\text{[3,4]}]} & \text{[a\text{[0,0]} \times b\text{[3,3]} + a\text{[1,0]} \times b\text{[4,4]}]} \\
\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:

```plaintext
FUNCTION SIND, deg
    result = SIN(deg * !dtor)
RETURN, result
END
```
Arguments and Keywords

- Using the predefine “contour” command for IDL as an example
  ```idl
  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)
FUNCTION f_to_c, temp_in

   temp_out = (temp_in - 32.) * 5. / 9. ; Radix points force floating point division

RETURN, temp_out
END

PRO c_to_f, temp_in, temp_out

   temp_out = temp_in * 9. / 5. + 32.

RETURN
END

PRO temp_example

   ; Create 5 random temperatures between 32 and 132 deg F
   temp_far = RANDOMU(1,5) * 100. + 32.

   PRINT, 'Fahrenheit temperatures:'
   PRINT, temp_far

   ; Calling a function to do the work
   temp_cent = f_to_c(temp_far)

   PRINT, 'Centigrade temperatures:'
   PRINT, temp_cent
   PRINT, 'Centigrade temperatures:'
   PRINT, temp_cent
   PRINT, 'Trying a new case...'

   ; Create 5 random temps between -40 and 40 deg C
   temp_cent2 = RANDOMU(1,5) * 80. - 40.

   PRINT, 'Centigrade temperatures:'
   PRINT, temp_cent2

   ; Calling a procedure to perform the conversion
   c_to_f, temp_cent2, temp_far2

   PRINT, 'Fahrenheit temperatures:'
   PRINT, temp_far2

STOP
END
Executive Commands

- Executive commands sent to IDL control the execution of procedures

- In IDLDE many executive statements are under the RUN menu

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>.compile</td>
<td>Compiles text from files or keyboard without executing.</td>
</tr>
<tr>
<td>.continue</td>
<td>Continues execution of a stopped program</td>
</tr>
<tr>
<td>.reset_session</td>
<td>Resets much of the state of an IDL session without requiring the user to exit and restart the IDL session.</td>
</tr>
<tr>
<td>.return</td>
<td>Continues execution until encountering a RETURN statement.</td>
</tr>
<tr>
<td>.rnew</td>
<td>Erases main program variables and then executes .RUN</td>
</tr>
<tr>
<td>.run</td>
<td>Compiles and possibly executes text from files or keyboard</td>
</tr>
<tr>
<td>.skip</td>
<td>Skips over the next statement and then single steps</td>
</tr>
<tr>
<td>.step</td>
<td>Executes a single statement (abbreviated as .S)</td>
</tr>
<tr>
<td>.stepover</td>
<td>Executes a single statement if the statement does not call a routine (abbreviated as .SO)</td>
</tr>
<tr>
<td>.trace</td>
<td>Similar to .CONTINUE, but displays each line of code before execution</td>
</tr>
</tbody>
</table>
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 THEN statement

IF condition THEN BEGIN

statement(s)

ENDIF

IF condition THEN statement ELSE statement

IF condition THEN BEGIN

statement(s)

ENDIF ELSE BEGIN

statement

ENDELS
nested if statements

IF condition1 THEN statement1 ELSE $
IF condition2 THEN statement2 ELSE $
...
IF conditionn THEN statementn ELSE 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, v2, inc DO statement
FOR i = v1, v2, inc DO BEGIN
    statement(s)
ENDFOR
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