# Command Language Syntax and Functionality

This is an informal description of the syntax and semantics that the aips++ Command Language (CL) will follow. The philosophy is a hybrid of **AIPS, IRAF, APL/IDL** and **C/C++**.

W.Jaffe, 28–August–1992

## Some Syntax

A session consists of a series of commands, terminated by the command **exit**

A command is a string of symbols terminated by a semicolon ; or a newline (**nl**). If the newline is preceeded by a backslash (\) (and optional whitespace) it is ignored. This allow continuation of multiline statements. In general all commands on a line (i.e. up to a newline) are read in one gulp, then parsed and executed one by one. An exception is compound (multiple) statements which are read and parsed one by one, but only executed when the compound statement is completed. On encountering a syntax error in a single statement the parser prints a message and aborts. In the middle of a compound statement only the current statement is aborted the user can complete the compound statement. Special keystrokes (e.g. control sequences) will exist to retrieve and edit earlier lines, as well as operations like aborting an entire compound statement.

A single statement consists of a string of control symbols and expressions. Expressions consist of operators, operands and separators. All such symbols must be known (i.e. predefined or defined earlier in the session) except possibly the leftmost symbol if this is immediately followed by the = operator. In this case, an operand with this name will be created dynamically, and is thereafter known to the system. until it is destroyed, explicitly or otherwise. If a previously known operand is placed before an = operator, it is destroyed and a new one is created.

Most symbols, including all user–defined symbols, must start with an alphabetic character, and consist of (case sensitive) alphabetic characters, digits, and the underscore character.

## Control symbols

Normally statements are executed syncronously in the order encountered, but certain control symbols alter this flow:

**COMPOUND:**

  **{statement; statement; statement;}**
  is considered to be one statement

**IF:**

  **if (expression) then statement**
  **if (expression) then statement else statement**

**FOR:**

  **for (initial-statement; test-expression; increment-statement) statement**

**WHILE:**

  **while (expression) do statement**

**DO:**

  **do variable = initial:final:increment statement**

**BACKGROUND:**

  **statement &**

In the last case the statement is executed asyncronously in the background. Certain special conditions then apply on the statement to assure the integrity of the environment (i.e. all currently known symbols). More on this below.

## Operators

All operators are either predefined, or defined in the session itself. They may take none or more operands, and the number of operands can be variable, but the application programmer or user who creates an operator must specify which operands are optional. Maximally one operand can be prefixed (occur before the operand); the rest must be postfixed. The most common use of prefixed operands are things like **+,*,-...** where **A + B** produces the result of the binary operator **+** operating on the operands **A** and **B**. Postfix operands can be surrounded by parenthesis, but this is unnecessary if the syntax is unambiguous.

Operands should be separated from operators by white space, and from each other by white space or commas. Optional ommited non–final operands must be indicated by a pair of commas; optional ommited final operands need not be indicated unless they could be confused with a prefixed operand of a following operator. Then parentheses should be used to close the operand list.

All operators return a result operand which may, however, be **void**. A hanging operand left over when a statement is completely executed is discarded. Thus the statement **sin(x)** will calculate the sine of x, and then discard it. Output to the screen is arranged by defining stream variables.

Probably all arguments to operators should be passed by value, not by reference. This means that input parameters cannot be directly altered by the operators; they operators only alter the output result. Input parameters can cause indirect changes however: *e.g.* a file name in an input parameter list might indicate they name of a file to be created by a function. This is similar to the action of a *pointer* in **C**. We do not include explicit pointers in the **CL** because we think they are too confusing for general users.

A very special operator is the assignment operator **=** which has one prefix and one postfix operand. The prefix operand, an *lside* has some specialized behaviour. If it is specified as a single symbol, then it is created dynamically, with type given by what is on the right side of the = sign, and filled in upon evaluation of the expression. Any previous definition of the symbol is deleted. If given with subset syntax e.g.:

$$\mathbf{A[1:20,2:5] = SIN(B)}$$

it will be assumed (and checked) that **A** already exists, that the subset syntax makes sense on **A** , and that the output of the right-hand-side can be converted to the type of **A** . In this case the subset syntax might include the whole of **A** . e.g. **A[*]**. The specified elements of **A** will be updated.

## Special Symbols

We name here a number of symbols with special meaning:
**&** indicates asyncronous processing.
**$** is the first character is a number of special system operands or operators.
() specify subscripts of arrays, precedence when not default, and optionall eenclose groups of operands.
The period **.** separates elements of a structure.
[ ] enclose subscripts of an array.
The vertical bar | is for temporary, **IRAF**–like substitution
of parameters in task calls.

## Arrays, Structures etc.

Operands can be single cases of various *atomic* things like (int, float, double, complex, double-complex, char, etc.) or more complicated composites called Structures. The form of a Structure is defined with the Structdef operator:

$$B \quad = \quad Structdef(Q)$$

where B, the result, is of type Structdef, and Q is a string with things like

$$Q \quad = \quad \text{“}sizes\ int[5], temp\ float, position\ double[2], name\ char[20]\text{”}$$

and such. Don't take this suggested syntax too seriously, but typically at least a name, type, and size will be specified for each sub-element. Possibly other attributes can be specified such as "Readonly" or "Interactive". The elements in a Structure may be atomic elements, or previously defined Structures, so that if the above statement has already been executed, the statement

$$C \quad = \quad Structdef(\text{“}type\ int, instances\ B[3]\text{”})$$

defines a new type of structure, containing one integer and an array of three substructures of type $B$. In the predefined symbols **int, float** etc. are in fact Operands of type Structdef. An empty structure is created with:

$$C \quad = \quad Makestruct(B)$$

This creates an empty operand, $C$ with form defined by $B$. The definition of an existing structure can be obtained with the *Getstruct* operator:

$$D \quad = \quad Getstruct(C)$$

will create a Structdef operand $D$ identical with $B$.

Generally Structure operands can only be used by operators expecting exactly that type of structure as input, but there are some exceptions: for instance the = operator creates an identical copy of any structure.

Arrays are collections of identical elements or structures. They can be created and manipulated with some special operators. For instance

$$B \quad = \quad Array(Size, Def)$$

where $Size$ is a number like 27 and $Def$ is a Strucdef, creates an operand called $B$ consisting of 27 empty elements of type Def. Size may also be an array such as [4 2 5], in which case a multidimensional array is created. As indicated above the values **int** or **double** may be used as Definitions.

The square brackets [ ] can create a one–dimensional array on the **CL** line. Thus $A \quad = \quad [2\ 3.1\ 0.]$ creates a 3 element float array. Symbols as well as numbers may be included, but they must all be of the same type.

The **Concat** operator abutts two arrays together in their last dimension. The must have the same number of dimensions, and all dimensions but the last must be equal.

The **Reform** operator reshapes an existing array to other specifications. $C \quad = \quad Reform(B, [3\ 3\ 3])$, where $B$ is defined as a 27 element vectors, converts $B$ to a $3 \times 3 \times 3$ 3-dimensional matrix.

The **Dimension** operator returns an integer vector with the dimensions of its arguements.

The **Indarray** operator generates a series of integers starting from 0. $A \quad = \quad Indarray(6)$ creates the 6 element array [0 1 2 3 4 5].

The common built–in Unary operators like **sin, cos, -, ...** can be applied to an array (of numbers of course) and they produce a similarly dimensioned output array.

The same is true of built–in binary operators (+,*,**,...) with the proviso that both operands have the same dimensions, or that all the non-equal dimensions be trailing dimensions equal to 1. This last exception allow combinations between scalars and vectors etc. by replication of the trailing dimensions. Thus an array of size [6] can be added to an array of size [1] (or a scalar); the scalar is replicated 6 times. An array of size [2 3 5 4] can be multiplied by an array of size [2 3]; twenty copies of the second array are made before multipling. The output array is of the larger dimensions. User supplied operators that are for general use should follow this convention.

**Substructures**

The period **.** is used to specify substructure elements by name. In the above definition of $Q$, $Q.sizes(2)$ selects the third (indices are zero based) element of the 5 element integer array specified above.

Subarrays are specified with parenthesis. Thus as just used, $sizes(2)$ specifies a single element in an array. Vector subscripts may also be used so that $sizes([2\ 0\ 2\ 1\ 3\ 4])$ specifies a 6 element integer array consisting of the 3rd, 1st, 3rd, 1nd, 4th and 5th elements of $sizes$. Multiple dimensions are separated by commas. If $Matrix \quad = \quad Array([4\ 3], float)$ creates a $4 \times 3$ array, $Matrix([0\ 3],[0\ 2])$ creates a $2 \times 2$ subarray. The asterisk * represents all elements in a specific dimension, and the colon syntax represents ranges of elements, i.e. [3:6] is equal to [3 4 5 6] and [3:22:3] is equal to [3 6 9 12 15 18 21]. Elements and ranges may be combined within the [ ] notation.

**Tasking**

Tasks should appear as similar as possible to functions, either system functions or user defined procedures. The essential difference in tasks is that they have some complicated characteristics defined by their designer (summarized by **HELP, DEFAULT, RANGE** structures), they are compiled externally to the **CL** , and they should be able to operate asynchonously. This last means that their access to shell objects must be tightly controlled.

Input arguements to complicated operators, be they functions or tasks, are typically complicated structures. For example the **AIPS** ***task*** **MX** would in the new **CL** be an operator with a syntax like:

$$\mathbf{A} \quad = \quad \mathbf{MX(B)} \qquad \text{or} \quad \mathbf{A} \quad = \quad \mathbf{MX\ B}$$

where **A** is a **CL** variable of type *MXOUTPUT* and **B** is of type *MXINPUT*. This last is a large structure with elements such as String MXINPUT.INNAME or float[2] MXINPUT.CELLSIZE. The output **A** would probably not contain the output image; this would clutter up the **CL** with very large objects. It probably contains at most an output identifier (e.g. file name), some general information (e.g. number of iterations actually used in clean, rms residual at end of clean) and **status** and **done** members. (see below about Asynchronous operations)

There is a system operator **EPARAM** such that $C \quad = \quad EPARAM('MX', B)$ invokes a full screen editor that reads the *MXINPUT* structure $B$; allows you to alter it contents, and writes the updated version into the *MXINPUT* structure $C$. Of course $B \quad = \quad EPARAM('MX', B)$ causes an in place update. Also $Outstream \quad = \quad B$ causes formatted output of the contents of $B$, similar to the current use of **INPUTS** in **AIPS**. At any time you may have any number of input structures to MX. You can also have an array

$$MANYMXRUNS \quad = \quad Array(MXINPUT, 20)$$

This essentially replaces the **TPUT** and **TGET** functions in **AIPS** and allows the user to maintain his/her own "database" of parameters. The same is true of output parameters.

We suggest additional **IRAF**–like syntax constructions to make small changes in input structures simple. For example

$$A \quad = \quad MX(B|IMSIZE = [128\ 256])$$

means use the current value of $B$ as input to MX, except change the value of the **IMSIZE** vector to 128, 256 temporarily (i.e. do not change $B$). Perhaps $MX(B||IMSIZE = [128\ 256)]$ causes a permanent change to $B$ i.e. is the equivalent of

$$B.IMSIZE \quad = \quad [128\ 256];\ MX(B)$$

We also suggest that for major *tasks* there be a current default **$MX** and a system default **$0MX**. If **MX** is called without inputs **A = MX** or **A = MX()**, then **$MX** is used as input. **$MX** can be changed by the user, as can any input structure, but it cannot be destroyed or its structure redefined. **$0MX**, the system default, cannot be altered. Thus **$MX = $0MX** resets the current defaults to the system defaults. Although it can be confusing, a task may have multiple arguements: **CLEAN(A, B)** whose defaults are **$CLEAN1** and **$CLEAN2** and whose system defaults are **$0CLEAN1** etc.

Since *Tasks* are defined and compiled externally to the **CL** it is necessary to inform the **CL** that they exist. Otherwise if you type **A = BLABLA(C)** and BLABLA is not in the current symbol table, the **CL** can't determine whether **BLABLA** is a typographical error or a Task. To resolve this it would have to search all possible Task libraries. This ambiguity is (one) justification for the **GO** construct in **AIPS**. We propose to remove this ambiguity with an **INCLUDETASK (Taskname, LibraryPath)** command. This has the effect of:
Including Taskname in the **CL** symbol table
Including structure definitions for TaskINPUT and TaskOUTPUT (predefined
by the Task Designer and available in LibraryPath) in the **CL**
Creating Links to HELP, RANGE, and DEFAULT equipment created by the designer

**Aspects of Asynchronous (background) operations**

Some special conditions are necessary for putting commands, especially functions returning values, into the background. These are necessary to avoid unpredictable behavior if the user alters input shell variables while the command is executing or if the command alters shell variables itself.

The special conditions are essentially: all shell variables (objects) used in the command are passed to it by value and cannot be modified by it in the parent shell. The output of the command goes into a structure that is *locked* i.e. cannot be modified or deleted until the command returns. Hopefully,given the possibility of using structures as inputs, it is never necessary to access *global* variables from within a function or procedure.

The output **lside** has in fact two implicit members, **lside.done** and **lside.status**. **lside.done** can be accessed at any time, and is *true* when the background operation is complete. The system function **wait(lside)** waits until **lside.done** is *true*. **lside.status** returns the status of the operation.

**Metavariables**

In some case the values filled in task structures are not actual values but special system operators. These would be indicated with a special sytax, for instance the % sign.

For example, to combine the advantages of global and local variables for complicated task calls, we suggest the concept of deferred evaluation. That is; it is acceptable to specify a value in an input structure as a **CL** expression that is only evaluated at execution time. For example, supposing that we use signs to indicate deferred execution, we could write **MX.INNAME = %PROJECT%** and **UVSRT.INNAME = %PROJECT%** With this syntax, the values of **MX.INNAME** and **UVSRT.INNAME** are not the current value of **PROJECT**, but its value when they are executed. Thus if the user changes **PROJECT** from "3C129" to "VirgoA", this change will apply to both **UVSRT** and **MX**. On the other hand, this coupling can be removed at any time simply by reassigning **MX.INNAME** to a non–deferred variable. Expressions as well as objects can be deferred. In essence, we are allowing the user to write a "mini–procedure" as an arguement to a function, without invoking a cumbersome procedure writing step.

Other metavariables are also useful. We can define the symbols **%IP** and **%DP** to mean **Immediate Prompt** and **Deferred Prompt**. Using these symbols as inputs to a task would force the task input processing facility to prompt the user for input at execution time (form **%IP**) or when the parameter is requested by the task (form **%DP** ). For this last feature to be useful there must be some kind of agreement (perhaps using an "interactive" attribute) as to which parameters can be so deferred. Normally all task parameters are read in at one time at task startup. In batch modes these prompts would revert back to the task defaults.

**Graphics** objects can be defined and used as the right hand side of assignments (and possibly as left hand sides, resulting in leaving markers on a graphics display). Assigning a task input element to one of these objects causes a cursor appear together with a request for input to the task. The graphics object must be predefined by the task writer, or by the **CL** system writer, or possibly in a procedure by the user from elementary objects, so that it produces an output of type that matches that expected by the task input structure.

Other Metavariables include File Managers or Catalogues. When the **CL** detects these symbols it actives a new window containing the Manager or Catalogue, allows the user to click on the requested item, and this is entered as the value of the requested variable.

## Input/Output

These are handled by creating variables of type **Stream**. Some of these are built–in variables and refer to stdin, stout, and stderr, but others can be created and destroyed with **open** and **close** operators which connect them with devices (files, terminals...) A **Tee** operator can connect two output streams together to form a new one. Streams should probably be opened as either **ASCII** or binary. Assigning a value to an ASCII stream results in a default formatting of the value (which may be an array or structure) into the stream. Various output formatting operations should exist, similar to those in C. Simple input formatting could follow the C model, or also the **Mongo** model, where a stream could be defined to be a single column in an **ASCII** table.

## Packages

The **Package(Packagename, Symbol)** operator can be used to group symbols, either operands or operators, into packages which can be manipulated as groups in certain operations such as **Save** or **Default**. The Symbols added to a package are *frozen*, that is they cannot be redefined or deleted, unless the package is deleted first.

## Some Fancy Extensions

Possibly, although this complicates the syntax, a general *reduction* operator can be defined. This could have the syntax $\mathbf{A} = \mathbf{REDUCE(bop, B)}$, where **bop** is any binary operator defined for **A** and **B**. This is the equivalent of $\mathbf{A} = \mathbf{B[1]bopB[2]bopB[3]}$... Thus the dot product of two vectors is simply $\mathbf{A} = \mathbf{REDUCE(+, B * C)}$.▌ The notation in **APL** is actually simpler: $\mathbf{A} = +/\mathbf{B} * \mathbf{C}$. The sum of the elements in **A** is $+/\mathbf{A}$. If **A** is the index vector **[1 2 3...]** then $*/\mathbf{A}$ is **A!**.

## External Classes

We would like some of the objects and methods developed by system and application programmers to be available directly to users. This cannot be done at arbitrary times, since the classes are defined in **C++** and only available after compiling and linking to the **CL** . We suggest that the **CL** be generated, at least in part from a *meta***CL** definition. This, an analog to the old **POPSDEF**, is a list of classes to be included, which objects and methods should be available to the **CL** user, and the aliases (or possibly more complicated syntax description) of how **CL** symbol sytnax maps into the **C++** calling sequence of the methods. The **CL** is compiled by reading this *meta***CL** file, which informs the **CL** parser that the necessary object types are defined, and tells it which methods to call when the appropriate commands are typed in.

While all this is probably fairly simple to implement, it can easily get out of hand and generate an unmanageable shell. In principle the kernel **AIPS++** group would distribute a vanilla version of the *meta***CL** definition, that would generate a small and simple system. Local site managers could then follow a recipie for adding new classes to the kernel.

## Some More System Functions

**SAVE (Workspace, Variable)** saves the current value of **Variable** in the named **Workspace**. Any existing variable in the same workspace with the same name is destroyed. **RESTORE (Workspace, Variable)** restores it, overwriting any current variable with the same name. **SAVEALL(Workspace)**and **RESTOREALL(Workspace)** save and restore the entire working environment. **RESTORE0** restores the default environment. On **EXIT, STOREALL** is executed to a specific workspace that is RESTORED as next login.

**DELETE(Variable)** destroys reference to that variable and releases dynamic storage assigned to it.

**D=DESCRIBE(Structure)** generates an ASCII description of the structure of that object. **D** can then be used (edited??) as the input to a new **Structdef** command. **A = B**, if **A** and **B** are structures, copies **B** into **A** . **A = NEW(B)** creates a new object with the same structure as **B**, but empty.

**DRYRUN (Task, Taskinput)** verifies whether Taskinput is acceptable to Task, and prints various diagnostics.

**RESET (Packagename)** resets the current default input structures for all tasks in a package to the system defaults.

**WAIT(TaskOutputStucture)** waits for completion of that task.