DATE: OCTOBER 30, 1985

TO: NRAO TCUS COMMITTEE

FROM: ALLEN FARRIS

SUBJECT: FORMAL DEFINITION OF THE COMMAND LANGUAGE

The following document is a first attempt to formally define the grammar of the command language we have discussed in past meetings. I have tried to remain faithful to the intent of the discussions and decisions we have reached. I have also attempted to define the language without giving very much thought to details of implementation. At some points I have given remarks in the comments to clarify certain issues but these are pretty brief.

I am aware that this is a long document and it is a tedious process to wade through it. However, I know of no other mechanism to precisely pin these issues down. I won't guarantee that this is the simplest or the shortest form the definition can take, but I have tried to make it intelligible.

Please carefully review this language definition and search for any errors and points of ambiguity. This formal definition is only the first step in implementation. If this definition adequately captures the intent of the language, then we can proceed to further implementation details. At this stage, the grammar has not been verified using any compiler development tools, such as YACC under UNIX. This will be one of the next tasks to be accomplished. Finally, this document only defines the language. It discusses neither its interactive nature nor its implementation.

```
/*
**********************************************************
**                                                      *
**              SYNTAX DEFINITION OF                    *
**                                                      *
**    NRAO COMMAND LANGUAGE FOR TELESCOPE CONTROL       *
**                                                      *
**            DRAFT WORKING VERSION 2.0                 *
**                                                      *
**********************************************************
*/
```

METALANGUAGE DESCRIPTION

:=	means 'is defined to be'
|	means 'alternatives' (one and only one choice can be made
[ ]	means 'optional'
...	means previous syntax unit can be repeated one or more times
< >	means groupings of syntax units
' '	represents a character string -- if the character ' is in the string it is represented by \'
'x'..'y'	means one of the ascii characters between the ascii characters x and y
/* */	means a comment and does not indicate the syntax of the language

Each statement defining a syntax unit begins:

name :=

where name is the name of the syntax unit. Such a statement may be continued onto subsequent lines and is terminated by a blank line.

It is frequently desirable to state rules which have syntatical and semantic importance in English sentences rather than a formal notation. Such rules are written as

RULE rule_name --> English language sentences

These sentences may be continued onto subsequent lines and are terminated be a blank line.

END OF METALANGUAGE DESCRIPTION

```
/*
** DEFINITION OF BASIC LANGUAGE ENTITIES
*/


RULE case_insensitivity -->
        In any context upper and lower case characters for
        the letters 'a'..'z' have the same meaning.  For
        example 'a' and 'A' mean the same thing.

digit :=
        '0'..'9'

hexadecimal_digit :=
        '0'..'9' | 'a'..'f'

octal_digit :=
        '0'..'7'

blank :=
        /* ASCII code for space (32 in decimal) */

tab :=
        /* ASCII code for horizontal tab (9 in decimal) */

form_feed :=
        /* ASCII code for form feed (12 in decimal) */

line_feed :=
        /* ASCII code for line feed (10 in decimal) */

carriage_return :=
        /* ASCII code for carriage return (13 in decimal) */

ascii :=
        '!'..'~' | blank | tab | form_feed | line_feed |
         carriage_return
/*
** ascii is the subset of printable ASCII characters
** (33 - 126 in decimal) plus blank, tab, form_feed,
** line_feed, and carriage return
*/

/*
** end of text
*/
eot :=
        /* The text is assumed to be read from a system file.
          The end of text is represented by the end of file
          condition. */

/*
** end of line
*/
```

```
eol :=
        line_feed carriage_return |
        line_feed |
        carriage_return line_feed |
        carriage_return

/*
** end of statement
*/
eos :=
        eol | ';' | end_of_line_comment

end_of_line_comment :=
        <'//' [ascii...] eol

RULE comment_restriction_1 -->
        The sequence of ascii characters making up an end of
        line comment may not contain the syntax unit eol.

/*
** white space
*/
ws :=
        < blank | tab | form_feed | <'\' eol> | comment >...

comment :=
        '/*' [ascii...] '*/'

RULE comment_restriction_2 -->
        The sequence of ascii characters making up a comment
        may not contain the combination '*/'.

name :=
        first_letter [ <subsequent_letter>... ]

first_letter :=
        'a'..'z' | 'A'..'Z' | '#' | '$' | '@' | '_' | '?'

subsequent_letter :=
        first_letter | '0'..'9'

list_of_names :=
        name [ [ws] <list_separator [ws] name>...]

list_separator :=
        < ',' [ws] eol > | ','

/*
** This definition of list separator allows a list to be
** continued onto a subsequent line by placing a comma after
** the last item on a given line.  Thus,
**              name1, name2, name3,
**                      name4, name5
** would comprise one list, while
**              name1, name2, name3
**                      name4, name5
```

```
** would comprise two lists.
*/


constant :=
        character_string | numeric_constant

character_string :=
        '\'' [ ascii... ] '\''

RULE character_string_formation -->
        The beginning and ending single quote mark is not
        part of the string.  If the sequence of ascii
        characters must contain the ascii character for
        single quote then that single quote character must
        be preceded by the ascii character '\'.  The
        character '\' is not part of the string.  A
        character_string is continued onto another line by
        the sequence <'\' eol>.  The sequence <'\' eol> is
        not part of the string.

numeric_constant :=
        basic_constant | base16_constant | base8_constant |
        converted_constant | angle_constant | date_constant

basic_constant :=
        integer_constant | real_constant

integer_constant :=
        digit...

base16_constant :=
        '0' <'X' | 'x'> hexadecimal_digit...

base8_constant :=
        '0' <'O' | 'o'> octal_digit...

real_constant :=
        basic_real |
        <basic_real [ws] real_exponent> |
        <integer_constant [ws] real_exponent>

basic_real :=
        <integer_constant '.' [integer_constant]> |
        < '.' integer_constant>

real_exponent :=
         <'E' | 'e'> [ws] [<'+' | '-'>] [ws]
        integer_constant

/* This definition of real constants differs slightly from
** the FORTRAN definition.  The definition here allows no
** spaces between the integer part and the fractional part
** of a basic_real.  It also does away with the distinction
** between single and double precision constants and allows
** either upper or lower case for the exponent indicator.
```

```
** Real constants are stored as double precision.
*/

converted_constant :=
        basic_constant converted_constant_identifier

converted_constant_identifier :=
        'cm' | 'day' | 'dB' | 'dBm' | 'C' | 'K' | 'GHz' |
        'Hz' | 'Jy' | 'kHz' | 'km' | 'km/s' | 'MHz' |'m' |
        'um' | 'uW' | 'mm' | 'mmHg' | 'ms' | 'mV' | 'mW' |
        'nm' | 'ns' | 'nW' | 'r' | 't' | 'V' | 'W' | 'yr'

/*
** These are the unit identifiers taken from
** TCUS Memo No. 18.
*/

angle_constant :=
        < [ [integer_constant degrees_delimiter_2]
                integer_constant minutes_of_arc_delimiter_2]
                basic_constant seconds_of_arc_delimiter > |
        < [integer_constant degrees_delimiter_2]
                basic_constant minutes_of_arc_delimiter_1 > |
         < basic_constant degrees_delimiter_1 >

degrees_delimiter_1 :=
        [ws] 'd' [ws]

degrees_delimiter_2 :=
        <[ws] <'d' | ':'> [ws]> | <ws>

minutes_of_arc_delimiter_1 :=
        [ws] '\'' [ws]

minutes_of_arc_delimiter_2 :=
        <[ws] <'\'' | ':'> [ws]> | <ws>

seconds_of_arc_delimiter :=
        [ws] '"' [ws]

RULE angle_validity -->
        Angle constants must conform to the general rules for
        forming valid angle expressions.

time_constant :=
        < [ [integer_constant hours_delimiter_2]
                integer_constant minutes_of_time_delimiter_2]
                basic_constant seconds_of_time_delimiter > |
        < [integer_constant hours_delimiter_2]
                basic_constant minutes_of_time_delimiter_1 > |
        < basic_constant hours_delimiter_1 >

hours_delimiter_1 :=
        [ws] 'h' [ws]

hours_delimiter_2 :=
```

```
        <[ws] <'h' | ':'> [ws]> | <ws>

minutes_of_time_delimiter_1 :=
        [ws] 'm' [ws]

minutes_of_time_delimiter_2 :=
        <[ws] <'m' | ':'> [ws]> | <ws>

seconds_of_time_delimiter :=
        [ws] 's' [ws]

RULE time_validity -->
        Time constants must conform to the general rules for
        forming valid time expressions.

date_constant :=
        year_constant [ws] month_constant
        [ws] day_constant

year_constant :=
        integer_constant

month_constant :=
        'jan' | 'feb' | 'mar' | 'apr' | 'may' | 'jun' |
        'jul' | 'aug' | 'sep' | 'oct' | 'nov' | 'dec'

day_constant :=
        integer_constant

RULE date_validity -->
        Date constants must conform to the general rules for
        forming valid dates.

RULE year_convention -->
        If the year_constant is a two digit number x, the
        following conversions are applied:
                if 50 <= x <= 99
                        year_constant = 1900 + x
                if 0  <= x <= 49
                        year_constant = 2000 + x

signed_numeric_constant :=
        [ '-' ] ws numeric_constant

signed_integer_constant :=
        [ '-' ] ws integer_constant

/*
** These definitions allow constants to have a prefix minus
** sign to indicate negative numbers.  A prefix plus sign is
** unnecessary.  The prefix minus sign applies only to
** constants.
*/

/*
** types of variables
```

```
*/
variable_types :=
        string_type | short_type | int_type | real_type |
        double_type | angle_type | time_type | date_type

string_type :=
        'string'

short_type :=
        'short'

int_type :=
        'int'

real_type :=
        'real'

double_type :=
        'double'

angle_type :=
        'angle'

time_type :=
        'time'

date_type :=
        'date'

/*
** The purpose of introducing the types angle, time, and date
** is to tell the interactive processor how to format the
** output of a 'show' command.  In addition, the types angle
** and time allow the proper conversion of input constants
** without trailing delimiters.
*/
```

```
/*
** DEFINITION OF BLOCK STRUCTURE OF THE COMMAND LANGUAGE
*/

RULE optional_white_space -->
        In the following presentation of syntax, any syntax
        unit may optionally be preceeded by ws.  If ws must
        be present, it is stated explicitly.

observing_program :=
        < block >...

block :=
        < pgm_block | func_block | proc_block |
          system_block >

RULE system_block_first -->
        If a system block is present, it must be the first
        block of the program.

RULE pgm_block_presence -->
        At least one block must be a pgm_block.

/*
** Execution begins at the first pgm_block.  If a program
** block is not present, the source text is compiled but
** marked as not executable.
*/

system_block :=
        'system' ws eos
        [ <system_statement_group>... ]
        'endsystem' eos

pgm_block :=
        <'program' | 'pgm'> ws pgm_name eos
        [ <statement_group>... ]
        <'endprogram' | 'endpgm'> eos

pgm_name :=
        name

func_block :=
        function_type <'func' | 'function'> ws
                  func_name '(' [parameter_list] ')' eos
        [ <statement_group>... ]
        <'endfunction' | 'endfunc'> eos

function_type :=
        integer_type | short_type | real_type | double_type |
        angle_type | time_type | date_type

func_name :=
        name
```

```
parameter_list :=
        list_of_names

/*
** The function_type indicates the type of variable returned
** by the function.  Functions may only return numeric
** variables.
*/

proc_block :=
        <'procedure' | 'proc'> ws proc_name
                '(' proc_parameter_list ')' eos
        [ <statement_group>... ]
        <'endprocedure' | 'endproc'> eos

proc_name :=
        name

proc_parameter_list :=
        [input_parameter_list] [';' output_parameter_list]

input_parameter_list :=
        list_of_names

output_parameter_list :=
        list_of_names

statement_group :=
        if_group |
        loop_group |
        null_statement |
        break_statement |
        return_statement |
        assignment_statement |
        variable_declaration_statement |
        procedure_reference_statement |
        function_reference_statement |
        catalog_statement |
        pause_statement |
        show_statement |
        resume_statement

system_statement_group :=
        variable_declaration_statement |
        func_block |
        proc_block |
        external_global_variable_declaration_statement |
        external_procedure_declaration_statement

/*
** All entities defined in the system block are, from the
** user's point of view, pre-defined entities belonging to
** the system.  The user cannot change these definitions.
** All variables defined in the system block are globally
** accessible from any other block.  These are the only
** globally defined variables it is possible to define.
```

```
*/

external_global_variable_declaration_statement :=
        'external' ws variable_declaration_statement

external_procedure_declaration_statement :=
        'external' ws procedure_declaration_statement

/*
** External variables and procedures are those which are
** external to the command language and which "belong to"
** the specific control system but which are accessible via
** the command language.
*/

RULE scope_of_names -->
        All variables defined within the system_block are
        global in scope, i.e., they are known and may be
        referenced in any block of the observing program.
        All variables declared outside the system_block are
        local in scope, e.i., they are known only in the
        block in which they are declared.

/*
** The following is a proposed list of pre-defined or
** built-in functions.
**      abs             absolute value
**      mod             modulo function
**      int             convert to int
**      real            convert to real
**      double          convert to double
**      exp             exponential function
**      pow             x to the y power
**      sqrt            square root
**      log             natural log
**      log10           base 10 log
**      sin             trig functions, etc.
**      cos
**      tan
**      asin
**      acos
**      atan
**      sinh
**      cosh
**      tanh
**      and             bitwise and function
**      or              bitwise or function
**      xor             bitwise exclusive-or function
**      not             bitwise not function
**      lshift          bitwise left shift
**      rshift          bitwise right shift
**      dim             maximum first dimension of an array
**      dim2            maximum second dimension of an array
**      dim3            maximum third dimension of an array
*/
```

```
/*
** The following is a proposed list of pre-defined or
** built-in constants.  These names function exactly like
** constants.
**      pi              3.141592653589793
**      twopi           2 * pi
**      halfpi          pi / 2
**      pisq            pi * pi
**      e               2.718281828459045
**      esq             e * e
**      sqrt2           1.414213562373095
**      sqrt3           1.732050807568877
**      c               299792458 (speed of light in m/sec)
**      csq             c * c
*/
```

```
/*
** DECLARATION OF VARIABLES
*/

variable_declaration_statement :=
        string_declaration |
        numeric_declaration |
        set_declaration |
        default_declaration

string_declaration :=
        string_type ws string_variable '(' string_length ')'
        < list_separator
                string_variable '(' string_length ')' >...
        eos

string_variable :=
        name

string_length :=
        integer_constant

numeric_declaration :=
        < short_type | int_type | real_type | double_type |
                angle_type | time_type | date_type >
        ws numeric_variable
            [ < list_separator numeric_variable >... ] eos >

numeric_variable :=
        array_variable | simple_numeric_variable

simple_numeric_variable :=
        name

array_variable :=
        array_name '(' dimension_list ')'

dimension_list :=
        integer_constant [ list_separator integer_constant
                [ list_separator integer_constant ] ]

RULE array_dimension_limits -->
        Arrays are limited to three dimensions.  Dimensions
        vary from 1 to integer_constant.

set_declaration :=
        'set' ws set_name [ ws 'catalog' ws
                '(' string_variable ')' ] eos
        set_member_specification
        'endset' eos

/*
** The catalog option on the set declaration specifies that
** the values to be assigned to this set come from a catalog
** which is indexed by the variable specified by
```

```
** string_variable.  This is how source data will be assigned
** from a source catalog.  More on this later.  Thus, sets
** divide into catalogued and non-catalogued sets.
*/

RULE set_membership -->
        The variable specified by string_variable is the
        first member of the set.

set_name :=
        name

set_member_specification :=
        set_member_item
        [ < list_separator set_member_item >... ]

set_member_item :=
        < string_variable | simple_numeric_variable
        | array_name | set_name >

default_declaration :=
        <'default' | 'def'> ws default_name
                ws 'of' ws set_name eos
        default_specification_list
                [ < ';' default_specification_list>... ] eos
        <'enddefault' | 'enddef'> eos

/*
** If a set is a non-catalogued set, the series of
** default_specification_lists (separated by semicolons)
** are accessed like arrays of one dimension and the 'dim'
** function works on them as well.  Thus, if a default is
** declared as:
**      def LO_setup of LO_settings
**              L1, L2, L3;
**              M1, M2, M3;
**              N1, N2, N3;
**      endef
** The vaules L1, L2, L3 are assigned by the statement:
**      LO_settings = LO_setup(1)
** The values M1, M2, M3 are assigned by the statement:
**      LO_settings = LO_setup(2)
**
** If a set is a catalogued set, the specific
** default_specification_list is referenced by the value of
** its index.  For example, if 'source' is a catalogued set
** indexed by 'source_name' and 3c277 is an entry in the
** catalog, the assignment statement
**      source = 3c277
** assigns all the data belonging to 3c277 to the proper
** items in the set.
*/

default_name :=
        name
```

```
default_specification_list :=
                default_specification_type_1 |
                default_specification_type_2

default_specification_type_1 :=
        default_item_type_1
        [ < list_separator default_item_type_1 >... ]

default_item_type_1 :=
        < < integer_constant '(' default_item_type_1 ')' > |
          constant | 'null' >

/*
** The integer_constant is a repetition factor and repeats
** the item in parentheses the number of times specified
** by integer_constant.
*/

RULE null_constant -->
        The keyword 'null' assigns a constant whose meaning
        is that the variable has no value.

default_specification_type_2 :=
        default_item_type_2
        [ < list_separator default_item_type_2 >... ]

default_item_type_2 :=
        <string_variable '=' string_constant > |
        < simple_numeric_variable '='
                signed_numeric_constant > |
        < set_name '=' < default_name | 'null' > |
        < array_name '='
                < integer_constant '(' array_init_item ')' |
                        array_init_item > >
        < restricted_array_reference '=' array_init_item >

array_init_item :=
        signed_numeric_constant | 'null'

restricted_array_reference :=
        array_name '(' integer_constant
        [ list_separator integer_constant
                [ list_separator integer_constant ] ] ')'

RULE order_of_default_items -->
        If the default specification is of type 1, the items
        must be in the same order in which they were declared
        in the set.  If the default specification is of type
        2, the items can appear in any order.
```

```
/*
** EXPRESSIONS
*/

arithmetic_expression :=
        < '(' arithmetic_expression ')' > |
        function_reference |
        array_reference |
        < arithmetic_expression a_op arithmetic_expression > |
        simple_numeric_variable |
        signed_numeric_constant

a_op :=
        '/' | '*' | '-' | '+'

logical_expression :=
        < '(' logical_expression ')' > |
        < l_op_1 '(' logical_expression ')' > |
        < logical_expression l_op_2 logical_expression > |
        < arithmetic_expression c_op arithmetic_expression > |
        < string_expression s_op string_expression >

l_op_1 :=
        '!'

l_op_2 :=
        '&' | '|'

c_op :=
        '=' | '!=' | '>' | '>=' | '<' | '<='

s_op :=
        '=' | '!='

string_expression :=
        string_variable | string_constant

function_reference :=
        function_name '(' function_argument_list ')'

function_argument_list :=
        function_argument_item
        [ < list_seperator function_argument_item >... ]

function_argument_item :=
        arithmetic_expression

/*
** Functions cannot return strings, arrays, sets, or
** defaults, nor can they take them as arguments.
*/

array_reference :=
        array_name '(' array_item_specification ')'
```

```
array_item_specification :=
        arithmetic_expression
        [ list_separator arithmetic_expression
                [ list_separator arithmetic_expression ] ]

RULE array_dimension_expression -->
        The arithmetic expression in array references is
        converted to type integer.
```

```
/*
** ASSIGNMENT STATEMENTS
*/

assignment_statement :=
        < string_variable '=' string_expression eos> |
        < < simple_numeric_variable | array_reference > '='
                arithmetic_expression eos > |
        < < simple_numeric_variable | array_reference > '='
                special_angle_time_constant eos > |
        < set_name '='
            <   < default_name
                        ['(' arithmetic_expression ')'] > |
                < string_expression |
                        <[ws] special_ascii_string [ws]> > |
                'null'
            eos >

special_angle_time_constant :=
        [ [integer_constant ws ] integer_constant ws ]
                basic_constant

RULE special_constant_validity -->
        The angle or time notation must conform to valid
        rules forming angle or time expressions.

special_ascii_string :=
        < '!'..'~' >...

/*
** This definition allows, as an option, the assignment to
** a set of any sequence of printable ascii characters
** without enclosing them in quotes.  The string is formed
** by stripping off the leading and trailing blanks.  No
** blanks are allowed in the string.
*/

RULE assignment_rule_1 -->
        The form 'name = special_angle_time_constant' applies
        only to variables of type angle or time.

RULE assignment_rule_2 -->
        The form 'set_name = default_name' can be applied to
        both catalogued and non-catalogued sets.  The right
        side of the assignment statement is first checked
        for a match on a proper default name.  If none is
        found, then an index search is performed.

RULE assignment_rule_3 -->
        The form 'set_name = ascii_string' applies
        only to cataloged sets.

RULE assignment_rule_4 -->
        The form 'set_name = null' applies to both catalogued
        and non-catalogued sets and assigns null values to
```

all members of the set.

```
/*
** CONTROL FLOW STATEMENTS
*/

if_group :=
        'if' '(' logical_expression ')' eos
            [ statement_group ]
        [ < 'elseif' '(' logical_expression ')' eos
                [ statement_group ] >... ]
        [ < 'else' eos
                [ statement_group ] > ]
        'endif' eos

loop_group :=
        while_group | for_group | repeat_group

while_group :=
        'while' '(' logical_expression ')' eos
            [ statement_group ]
        'endwhile' eos

repeat_group :=
        'repeat' eos
            [ statement_group ]
        'until' '(' logical_expression ')' eos

for_group :=
        'for' index_variable '=' initial_expression ','
            terminating_expression [',' increment_amount] eos
            [ statement_group ]
        'endfor' eos

index_variable :=
        integer_variable

RULE integer_variable_def -->
        An integer variable is a variable declared in a
        variable declaration beginning with int_type.

initial_expression :=
        arithmetic_expression

terminating_expression :=
        arithmetic_expression

increment_amount :=
        signed_integer_constant

RULE default_increment_amount -->
        If increment_amount is not specified it is taken
        to be 1.


/*
** The while_group has the following meaning:
```

```
**      L1: if (not logical_expression)
**              go to L2
**          statement_group
**          go to L1
**      L2:
**
** The repeat_group has the following meaning:
**      L1: statement_group
**          if (not logical_expression)
**              go to L1
**
** The for_group is derived from the while_group and has the
** following meaning:
** if increment_amount is positive
**      index_variable = initial_expression
**      while ( index_variable <= terminating_expression )
**              statement_group
**              index_variable = index_variable +
**                               increment_amount
**      endwhile
** if increment_amount is negative
**      index_variable = initial_expression
**      while ( index_variable >= terminating_expression )
**              statement_group
**              index_variable = index_variable -
**                               abs(increment_amount)
**      endwhile
*/

break_statement :=
        'break' eos

RULE break_interpretation -->
        The break statement only has a function inside a
        loop_group.  It terminates the loop and resumes
        execution at the end of the loop.
```

```
/*
** OTHER STATEMENTS
*/

procedure_reference_statement :=
        proc_name '(' procedure_argument_list ')' eos

procedure_argument_list :=
        [ input_argument_list ] [ ';' output_argument_list ]

input_argument_list := input_argument_list_item
        [ < list_separator input_argument_list_item >... ]

input_argument_list_item :=
        string_expression | arithmetic_expression | set_name
        default_name

output_argument_list := output_argument_list_item
        [ < list_separator output_argument_list_item >... ]

output_argument_list_item :=
        string_variable | numeric_variable | set_name

function_reference_statement :=
        function_reference eos

RULE function_return -->
        If a function is not used in an expression but merely
        in a stand-alone statement, any arithmetic value it
        may return is ignored.

null_statement :=
        eos

return_statement :=
        'return' [ arithmetic_expression ] eos

catalog_statement :=
        'catalog' '=' default_name
                [ <list_separator default_name>... ] eos

RULE catalog_statement_restriction -->
        The default names in the list must all be defaults
        of the same set.

/*
** The catalog statement specifies the order in which catalogs
** are to be searched to satisfy set assignment statements for
** catalogued sets.  Such a statement is necessary only if
** more than one catalog exists which applies to a given set
** and those catalogs might contain different entries for
** same index string.
*/

pause_statement :=
```

```
        'pause' [ ws 'until' '(' logical_expression ')' ] eos

/*
** The pause statements halts execution until receipt of a
** resume statement or until the optional logical condition
** is true.
*/

resume_statement :=
        'resume' eos

show_statement :=
        'show' [ show_item
                 [ <list_seperator show_item>... ] ] eos

show_item :=
        string_expression | numeric_variable | set_name

/*
** If the 'show' statement has no list of items, the
** currently executing position is displayed.
*/

/*
** Here, I will give a example of the use of catalogued sets.
**
** Suppose we have the following sets:
**      set source catalog (source_name)
**              ra, dec, epoch, gain
**      endset
**      set spectral_line catalog (frequency_name)
**              rest_frequency
**      endset
**
** Then suppose we have the following defaults:
**      def system_sourcelist of source
**              3c277, ra1, dec1, epoch1, gain1;
**              3c218, ra2, dec2, epoch2, gain2;
**              ...
**      enddef
**      def user_sourcelist of source
**              NGC7027, ura1, udec1, uepoch1, ugain1;
**              ...
**      enddef
**      def system_linelist of spectral_line
**              HI, rest_freq_h1;
**              OH, rest_freq_oh;
**              ...
**      enddef
**      def user_linelist of spectral_line
**              NH3, rest_freq_nh3;
**              ...
**      enddef
**
** Then a sequence of statements might be the following:
**      catalog = user_sourcelist, system_sourcelist
```

```
**      catalog = user_linelist, system_linelist
**      ...
**      spectral_line = HI
**      ...
**      source = 3c277
**      ...
**      spectral_line = NH3
**      ...
**      source = NGC7027
**
** Expanding the example a bit, we can add the following to
** create a list of sources to be used in conjuction with
** some observing procedure in a looping construct.  Suppose
** we add the following set definitions:
**      set sources
**              source
**      endset
** and the default definition
**      def actionlist of sources
**              3c277; 3c218; NGC7027;
**      enddef
** Then, we can write the following:
**      for i = 1, dim(actionlist)
**              sources = actionlist(i)
**              observing_procedure()
**      endfor
** Note that we can still use the following statements:
**      source = 3c218
**      observing_procedure()
*/
```