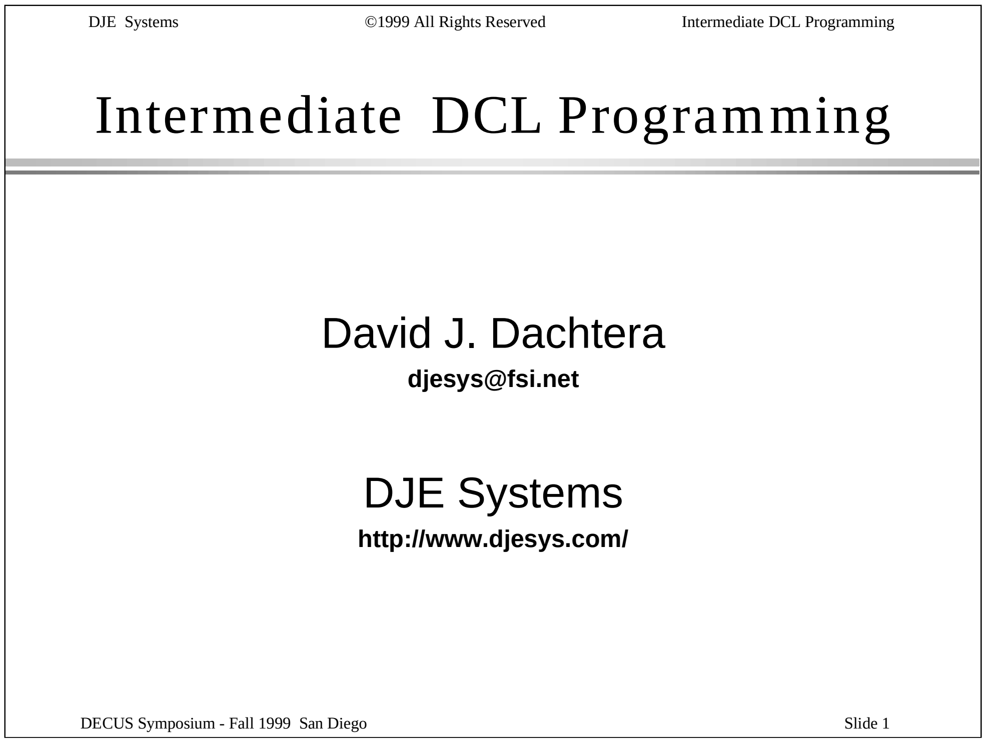


DJE Systems ©1999 All Rights Reserved Intermediate DCL Programming

Intermediate DCL Programming

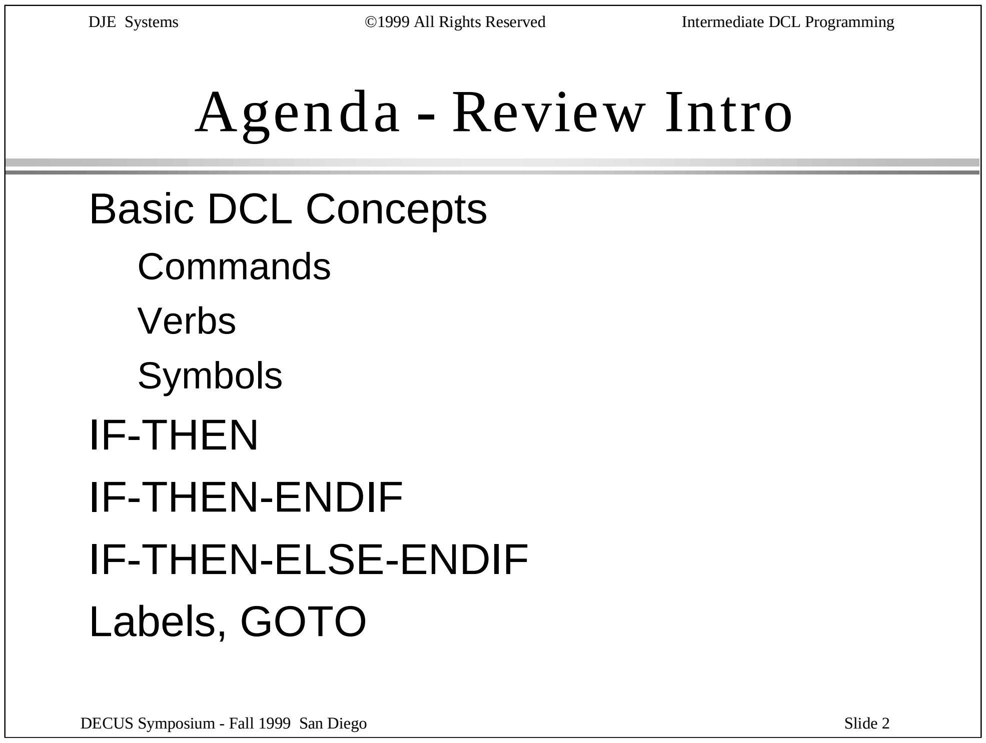
David J. Dachtera
djesys@fsi.net

DJE Systems
http://www.djesys.com/

DECUS Symposium - Fall 1999 San Diego Slide 1

In this presentation, we'll build on the concepts presented in the Introduction to DCL Programming.

We'll introduce some of the more advanced functions and operations, including Searching for files, File I/O, parsing strings read from files, and using symbol substitution.



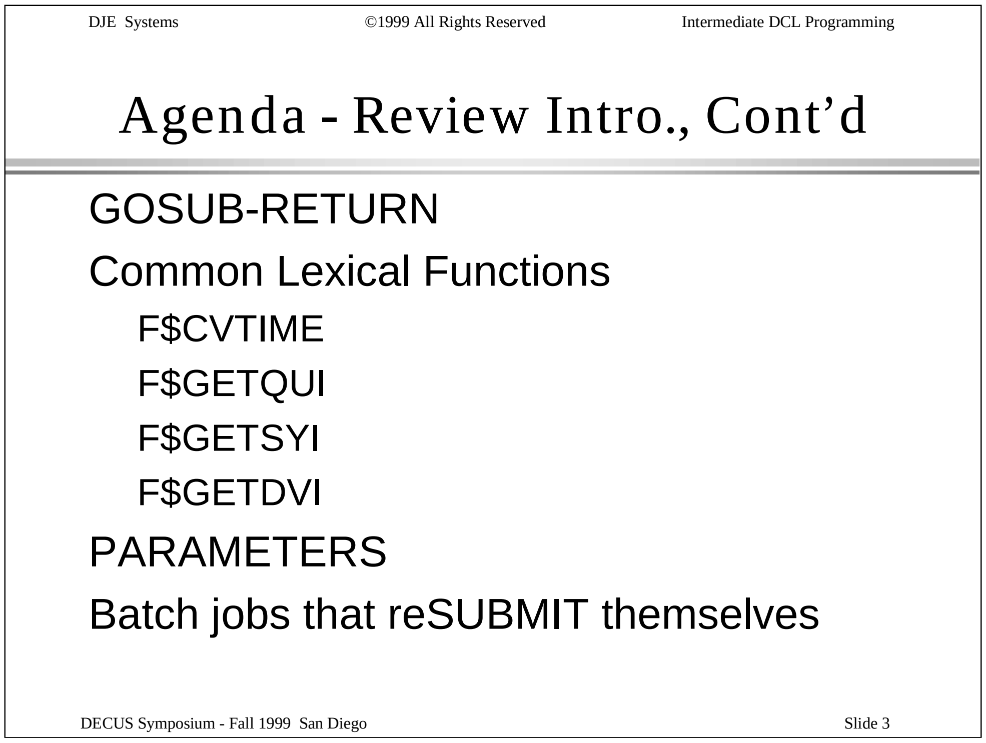
DJE Systems ©1999 All Rights Reserved Intermediate DCL Programming

Agenda - Review Intro

- Basic DCL Concepts
  - Commands
  - Verbs
  - Symbols
- IF-THEN
- IF-THEN-ENDIF
- IF-THEN-ELSE-ENDIF
- Labels, GOTO

DECUS Symposium - Fall 1999 San Diego Slide 2

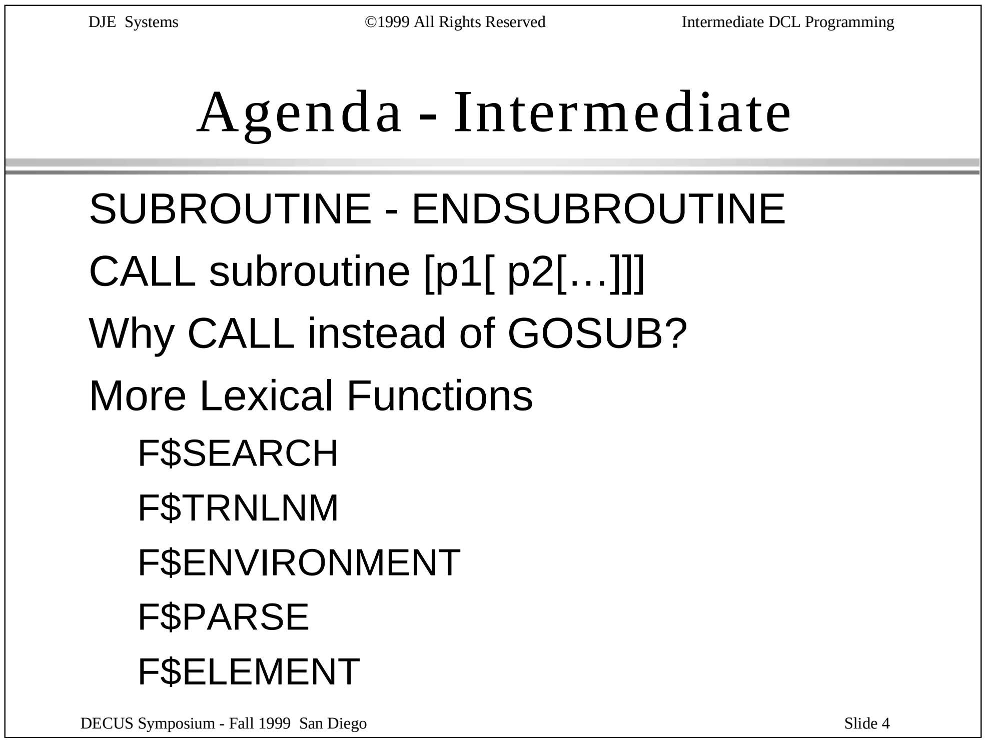
First, we’ll go over the basics we discussed in the introductory session. We’ll take another look at commands, verbs and symbols. We’ll review conditional expressions and basic logical controls.



Agenda - Review Intro., Cont'd

- GOSUB-RETURN
- Common Lexical Functions
  - F$CVTIME
  - F$GETQUI
  - F$GETSYI
  - F$GETDVI
- PARAMETERS
- Batch jobs that reSUBMIT themselves

We'll go over simple internal subroutines, common lexical functions, and processing parameters passed to a procedure. We'll review using simple F$GETQUI() items to gather the information needed to allow a procedure to resubmit itself.

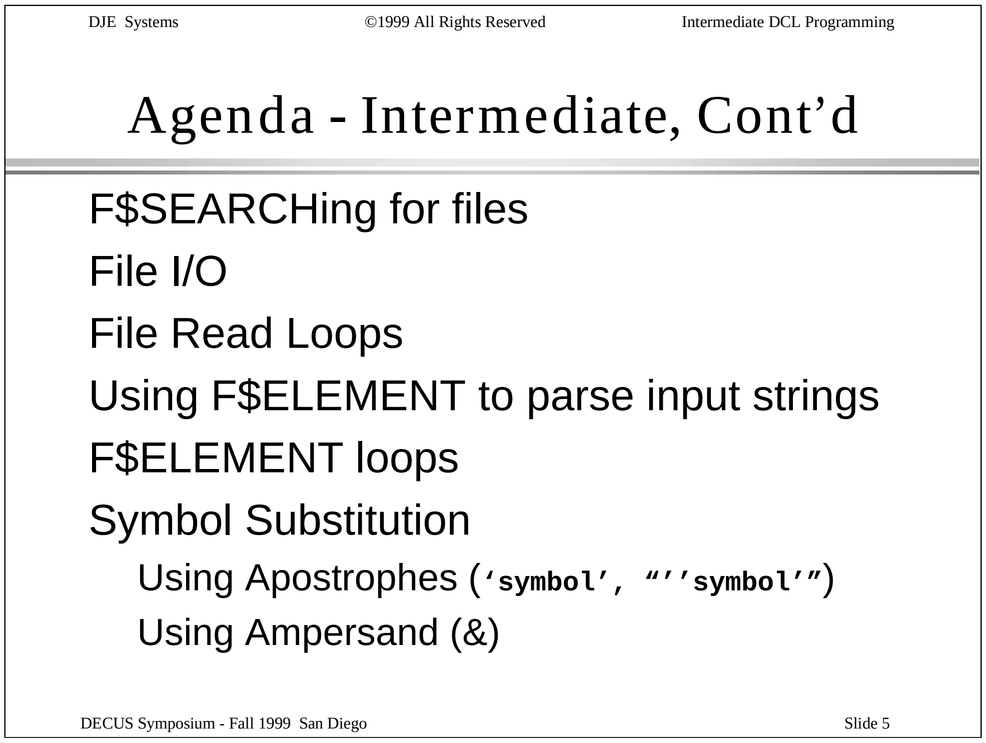

DJE Systems ©1999 All Rights Reserved Intermediate DCL Programming

Agenda - Intermediate

- SUBROUTINE - ENDSUBROUTINE
- CALL subroutine [p1[p2[…]]]
- Why CALL instead of GOSUB?
- More Lexical Functions
  - F$SEARCH
  - F$TRNLNM
  - F$ENVIRONMENT
  - F$PARSE
  - F$ELEMENT

DECUS Symposium - Fall 1999 San Diego Slide 4

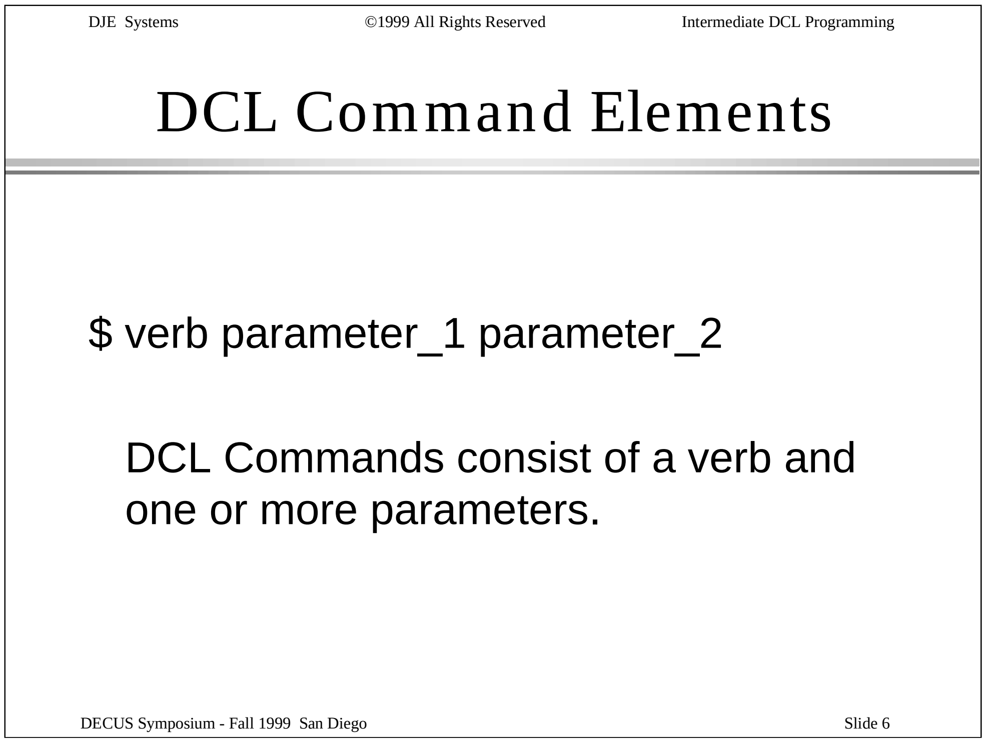
We'll then get deeper into some of the more advanced functions like complex internal subroutines, some of the more useful lexical functions, ...



Agenda - Intermediate, Cont’d

- F$SEARCHing for files
- File I/O
- File Read Loops
- Using F$ELEMENT to parse input strings
- F$ELEMENT loops
- Symbol Substitution
    - Using Apostrophes (`‘symbol’, “’’symbol’”`)
    - Using Ampersand (&)

...processing wildcarded file specifications, reading and writing disk files, parsing strings and parameters, and using symbol substitution.



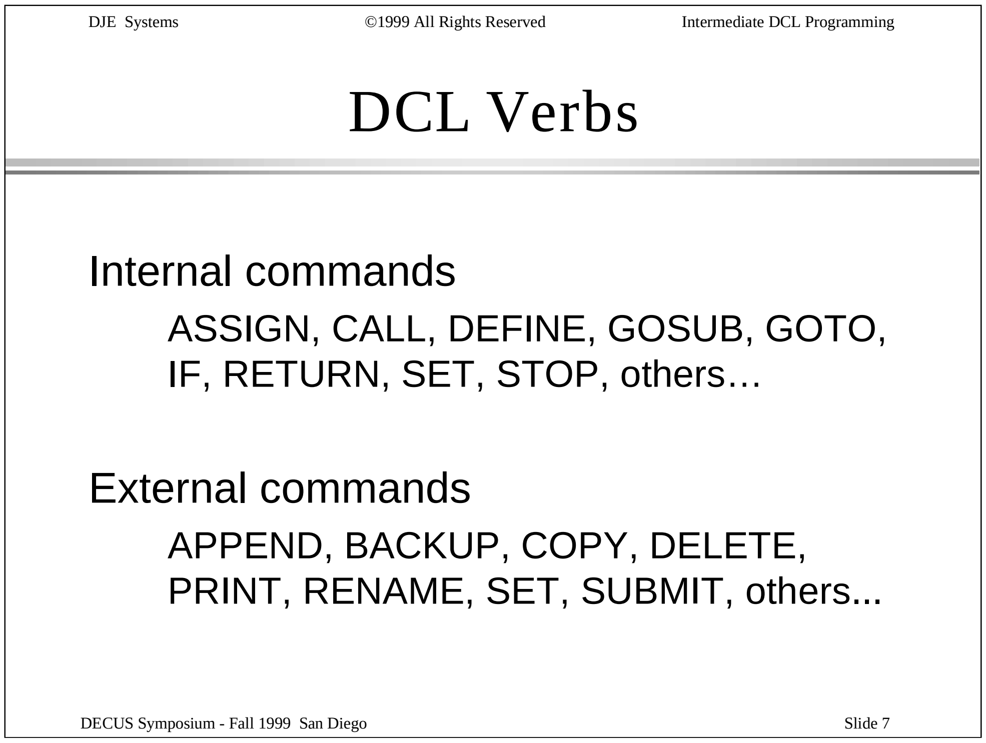
DCL Command Elements

```
$ verb parameter_1 parameter_2
```

DCL Commands consist of a verb and one or more parameters.

We begin our review with the basics of DCL commands.

A DCL command usually consists of a verb and one or more parameters.



DCL Verbs

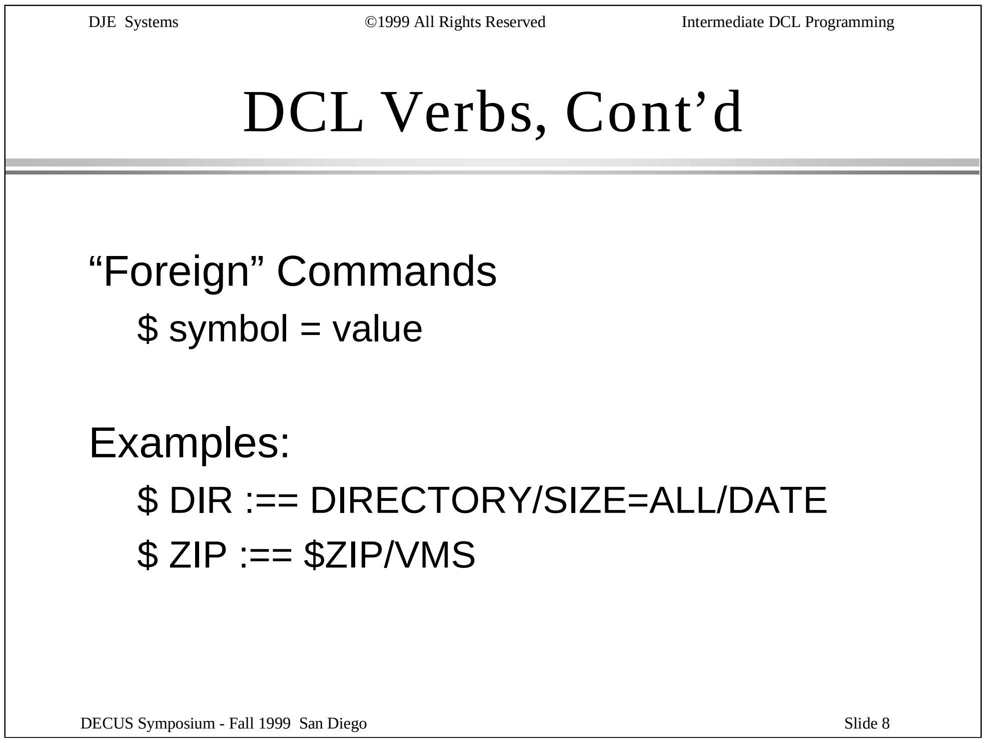
Internal commands

ASSIGN, CALL, DEFINE, GOSUB, GOTO, IF, RETURN, SET, STOP, others…

External commands

APPEND, BACKUP, COPY, DELETE, PRINT, RENAME, SET, SUBMIT, others...

Some commands are internal to DCL. Others are facilitated by programs external to DCL. Notice that some of the SET and STOP commands are internal and some are external.



DCL Verbs, Cont'd

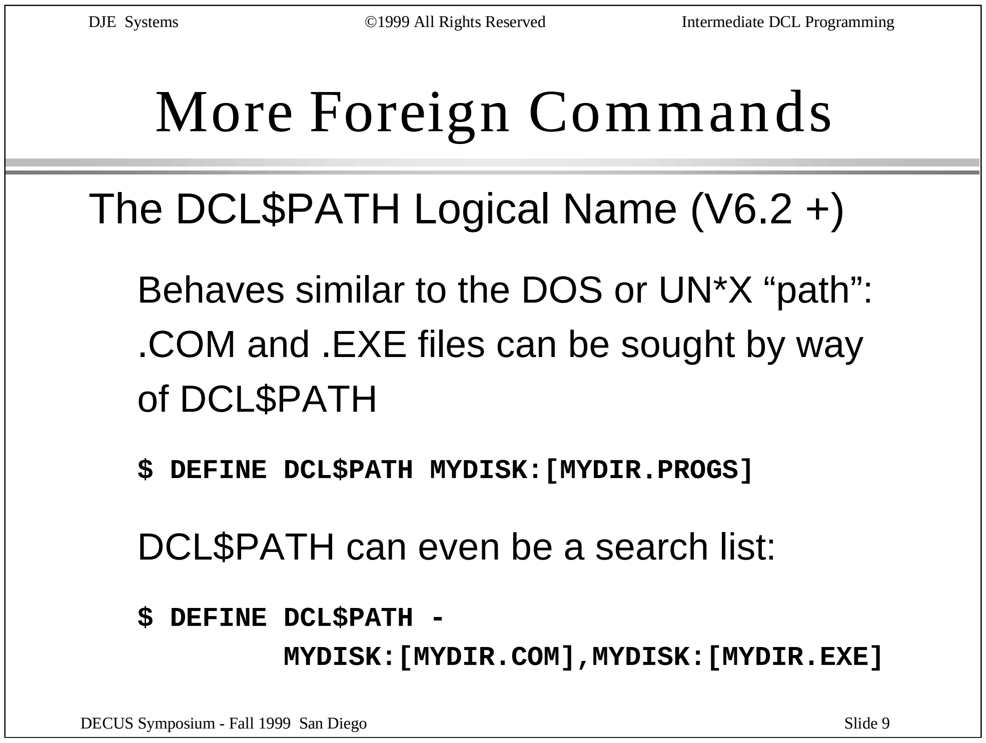
"Foreign" Commands

```
$ symbol = value
```

Examples:

```
$ DIR :== DIRECTORY/SIZE=ALL/DATE
$ ZIP :== $ZIP/VMS
```

Some commands are created by you, the user. These are called foreign commands, since they're not actually part of the command tables, or appear in the command tables with different default qualifiers and/or parameters.



More Foreign Commands

The DCL$PATH Logical Name (V6.2 +)

Behaves similar to the DOS or UN*X “path”: .COM and .EXE files can be sought by way of DCL$PATH

```
$ DEFINE DCL$PATH MYDISK:[MYDIR.PROGS]
```

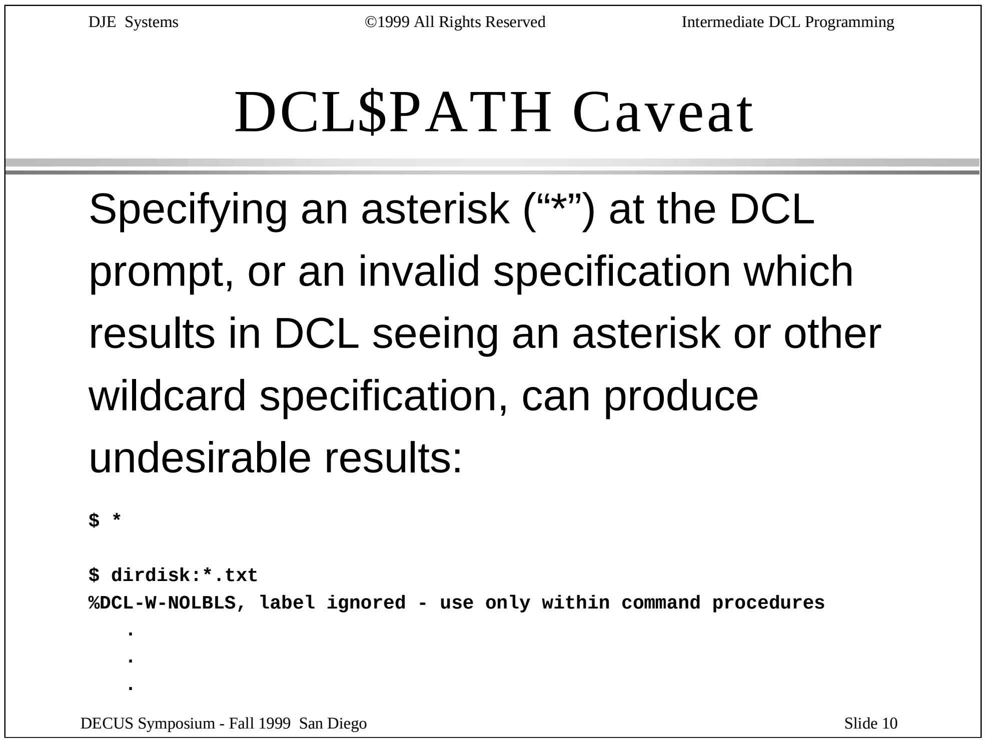
DCL$PATH can even be a search list:

```
$ DEFINE DCL$PATH -
         MYDISK:[MYDIR.COM],MYDISK:[MYDIR.EXE]
```

Another way to find “foreign” or external commands is to use the DCL$PATH logical name. DCL$PATH was introduced in OpenVMS V6.2.

DCL$PATH behaves very much like the DOS or UN*X “path” - .COM and .EXE files can be located via the DCL$PATH path.

DCL$PATH can have a single translation or it can be a search list.



DCL$PATH Caveat

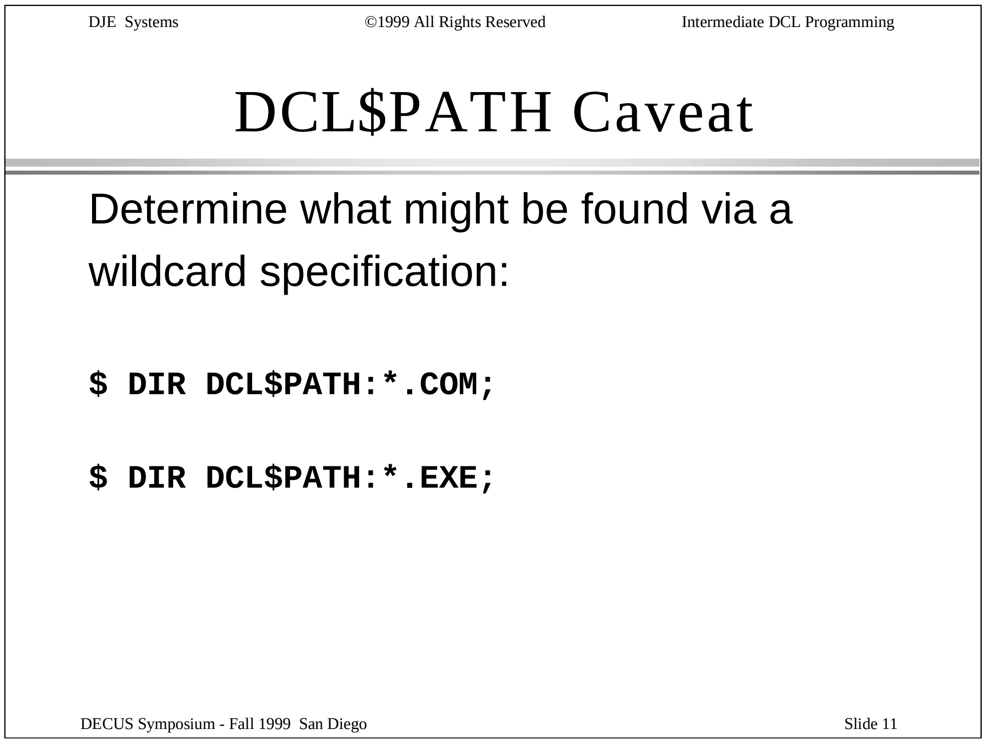
Specifying an asterisk ("*") at the DCL prompt, or an invalid specification which results in DCL seeing an asterisk or other wildcard specification, can produce undesirable results:

```
$ *

$ dirdisk:*.txt
%DCL-W-NOLBLS, label ignored - use only within command procedures
   .
   .
   .
```

There is an aspect of DCL$PATH of which you need to be aware:

DCL will observe wildcard specifications when seeking a file by way of DCL$PATH. This can produce undesired results.

Even an invalid specification might be interpreted as a wildcard specification. The second example shows what can happen if a space is left out of a DIRECTORY command.

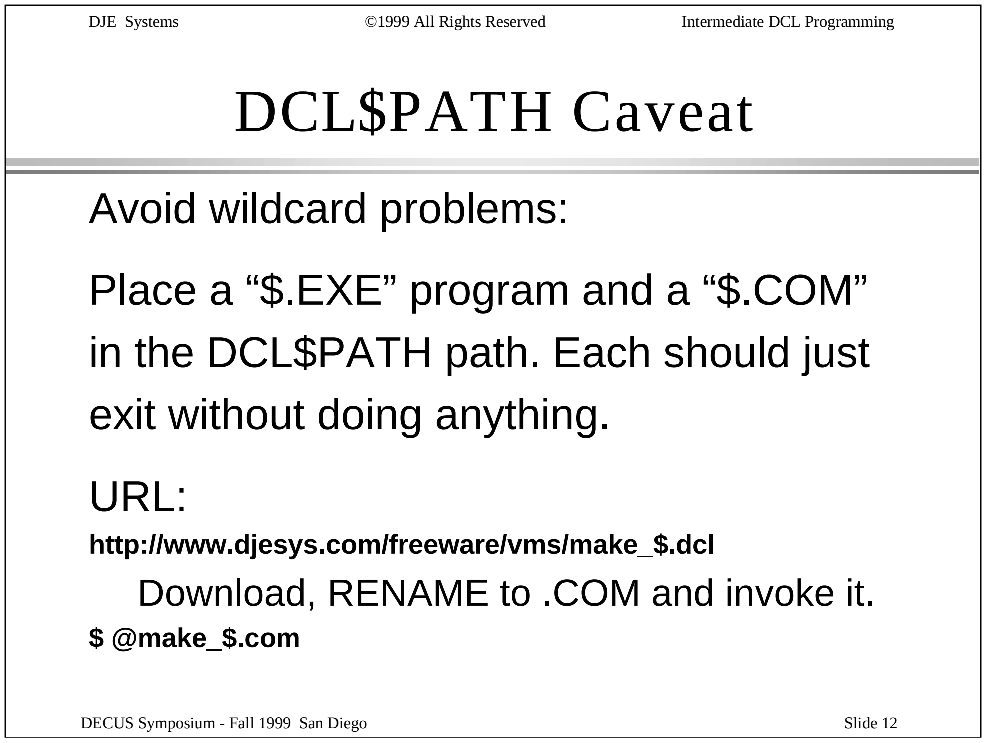


DCL$PATH Caveat

Determine what might be found via a wildcard specification:

```
$ DIR DCL$PATH:*.COM;

$ DIR DCL$PATH:*.EXE;
```

You can determine in advance what might be found via a wildcard specification. Just issue a DIRECTORY command for .COM files and another for .EXE files.



DCL$PATH Caveat

Avoid wildcard problems:

Place a “$.EXE” program and a “$.COM” in the DCL$PATH path. Each should just exit without doing anything.

URL:

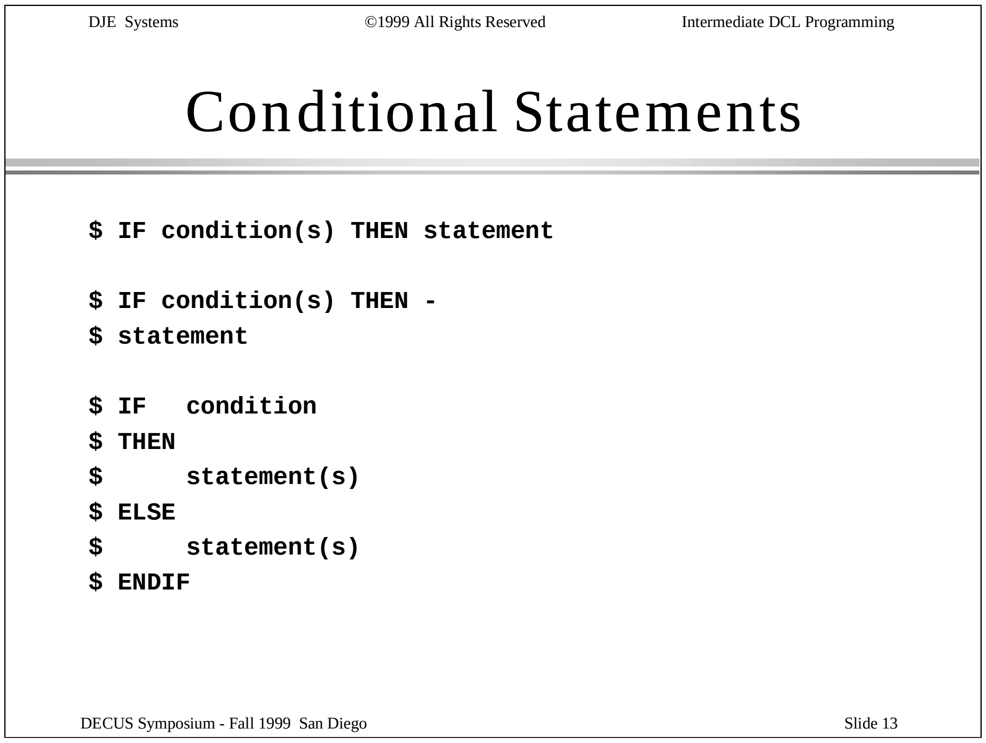
http://www.djesys.com/freeware/vms/make_$.dcl

Download, RENAME to .COM and invoke it.

$ @make_$.com

To avoid problems, you can place a “$.EXE” program or a “$.COM” procedure in the DCL$PATH path.

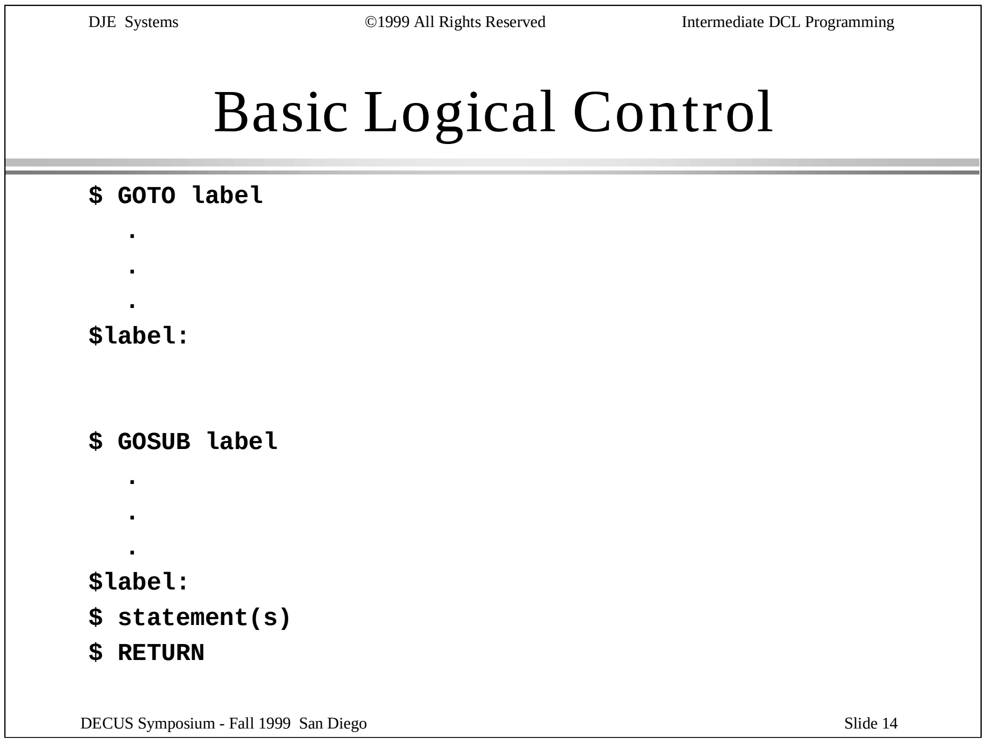
A DCL procedure to create these can be downloaded from the internet at the URL shown in the slide.



Conditional Statements

```
$ IF condition(s) THEN statement

$ IF condition(s) THEN -
$ statement

$ IF   condition
$ THEN
$      statement(s)
$ ELSE
$      statement(s)
$ ENDIF
```

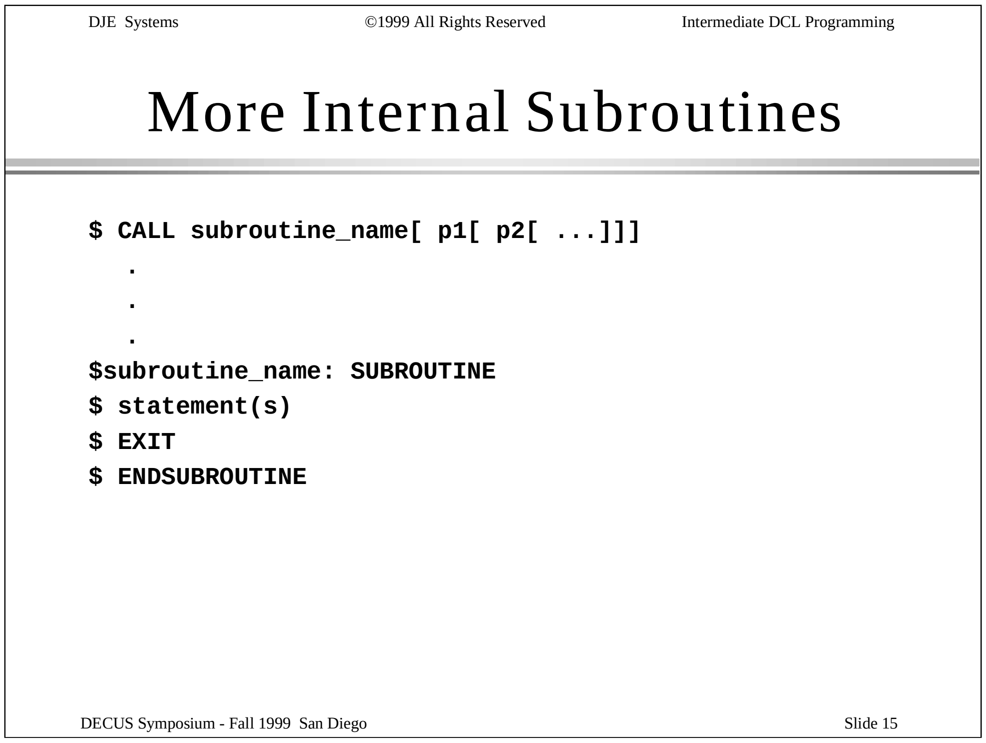
DCL provides conditional constructs including IF-THEN statements and IF-THEN-ELSE-ENDIF blocks.



Basic Logical Control

```
$ GOTO label
   .
   .
   .
$label:


$ GOSUB label
   .
   .
   .
$label:
$ statement(s)
$ RETURN
```

DCL provides for logical control of program flow by way of labels and the GOTO and GOSUB statements.

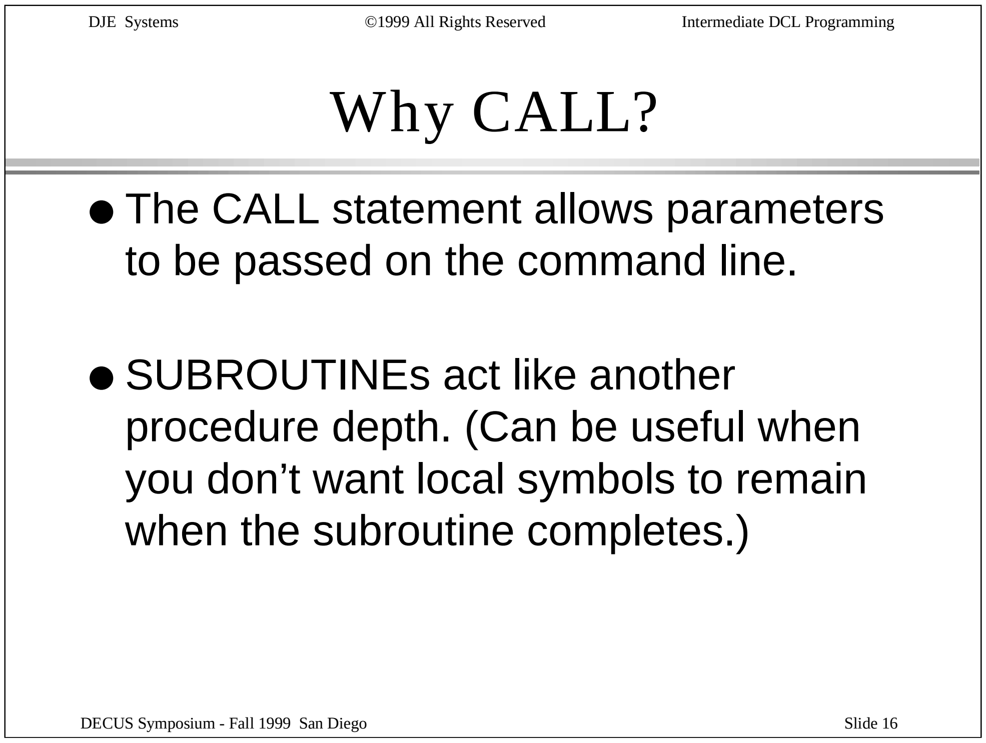

DJE Systems ©1999 All Rights Reserved Intermediate DCL Programming

More Internal Subroutines

```
$ CALL subroutine_name[ p1[ p2[ ...]]]
   .
   .
   .
$subroutine_name: SUBROUTINE
$ statement(s)
$ EXIT
$ ENDSUBROUTINE
```

DECUS Symposium - Fall 1999 San Diego Slide 15

DCL provides internal SUBROUTINEs that act like external procedures. This allows for easier parameter passing than GOSUB, also.


DJE Systems ©1999 All Rights Reserved Intermediate DCL Programming

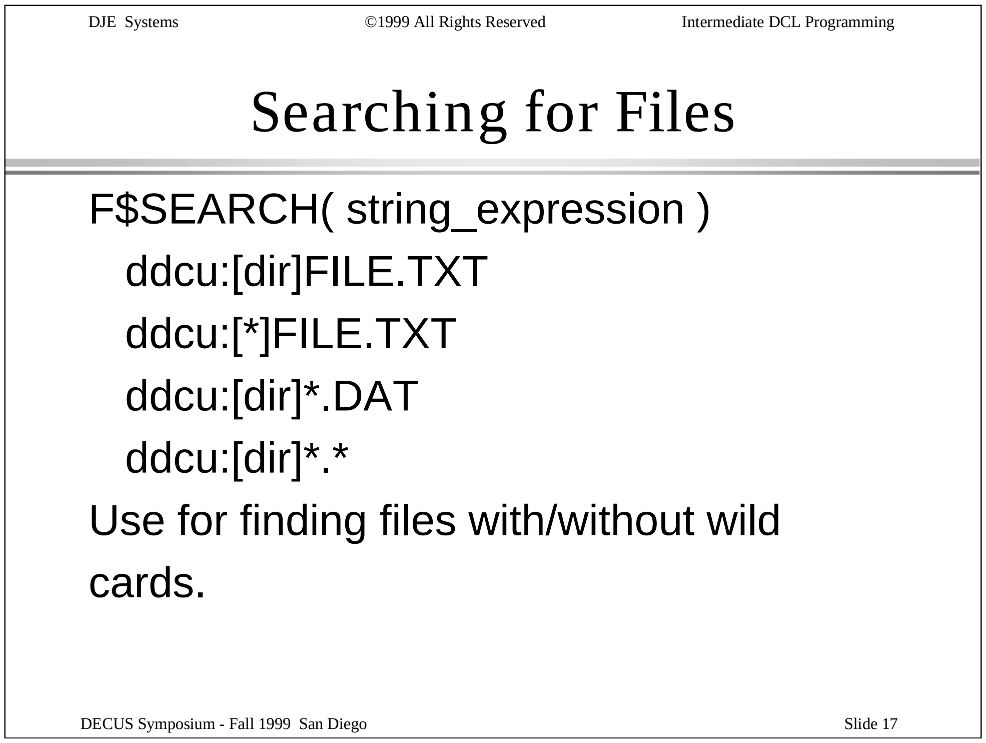
Why CALL?

- The CALL statement allows parameters to be passed on the command line.
- SUBROUTINEs act like another procedure depth. (Can be useful when you don’t want local symbols to remain when the subroutine completes.)

DECUS Symposium - Fall 1999 San Diego Slide 16

The CALL statement allows parameters to be passed to the SUBROUTINE.

SUBROUTINEs act like another procedure depth. Local symbols are local to the subroutine and global symbols are visible to the SUBROUTINE.



Searching for Files

```
F$SEARCH( string_expression )
  ddcu:[dir]FILE.TXT
  ddcu:[*]FILE.TXT
  ddcu:[dir]*.DAT
  ddcu:[dir]*.*
```

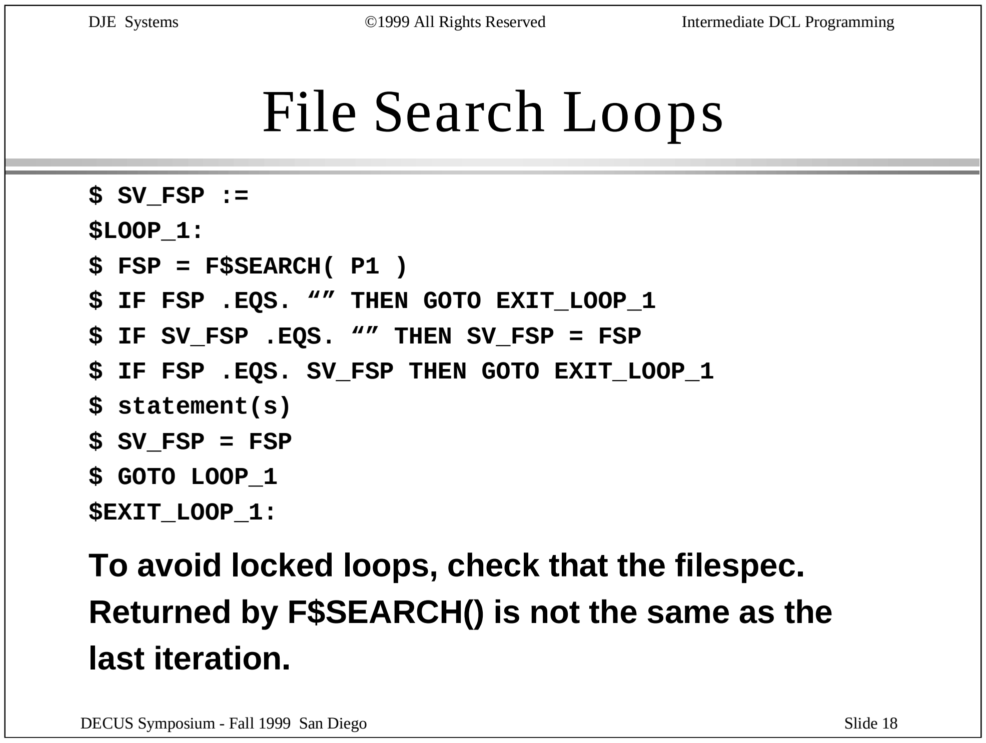
Use for finding files with/without wild cards.

Getting onto the intermediate topics, we begin with the F$SEARCH lexical function.

F$SEARCH() is useful for finding files using both absolute and wildcarded file specifications.

If the file you specify is not found, F$SEARCH returns a null string.

If you supply a wildcarded filespec, F$SEARCH returns the next matching filespec on each subsequent invocation. When there are no more matching files, a null string is returned.



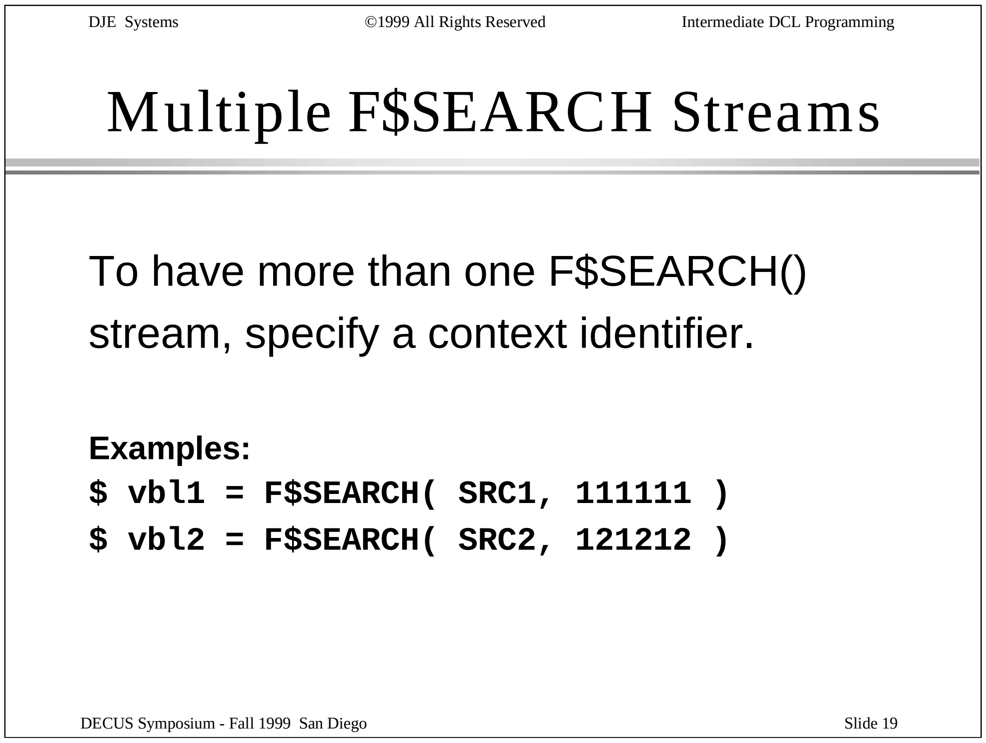
File Search Loops

```
$ SV_FSP :=
$LOOP_1:
$ FSP = F$SEARCH( P1 )
$ IF FSP .EQS. “” THEN GOTO EXIT_LOOP_1
$ IF SV_FSP .EQS. “” THEN SV_FSP = FSP
$ IF FSP .EQS. SV_FSP THEN GOTO EXIT_LOOP_1
$ statement(s)
$ SV_FSP = FSP
$ GOTO LOOP_1
$EXIT_LOOP_1:
```

To avoid locked loops, check that the filespec. Returned by F$SEARCH() is not the same as the last iteration.

Here’s an example of a loop which uses F$SEARCH.

Note that if the search specification is not wildcarded, F$SEARCH will return the same string over and over. The example shows how to avoid locked loops by saving the filespec after each invocation and comparing the previous string to the current string. If they match, exit the loop.



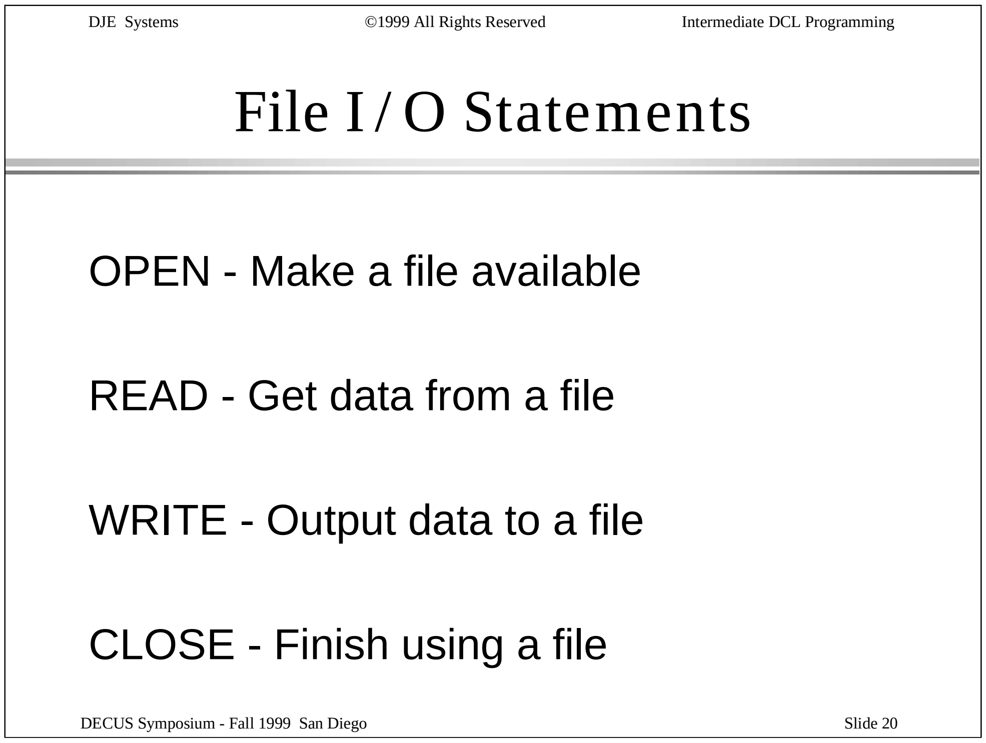
Multiple F$SEARCH Streams

To have more than one F$SEARCH() stream, specify a context identifier.

Examples:

```
$ vbl1 = F$SEARCH( SRC1, 111111 )
$ vbl2 = F$SEARCH( SRC2, 121212 )
```

You can have more than one F$SEARCH stream at a time. Just supply a unique context identifier for each stream.


DJE Systems ©1999 All Rights Reserved Intermediate DCL Programming

File I/O Statements

OPEN - Make a file available

READ - Get data from a file

WRITE - Output data to a file

CLOSE - Finish using a file

DECUS Symposium - Fall 1999 San Diego Slide 20

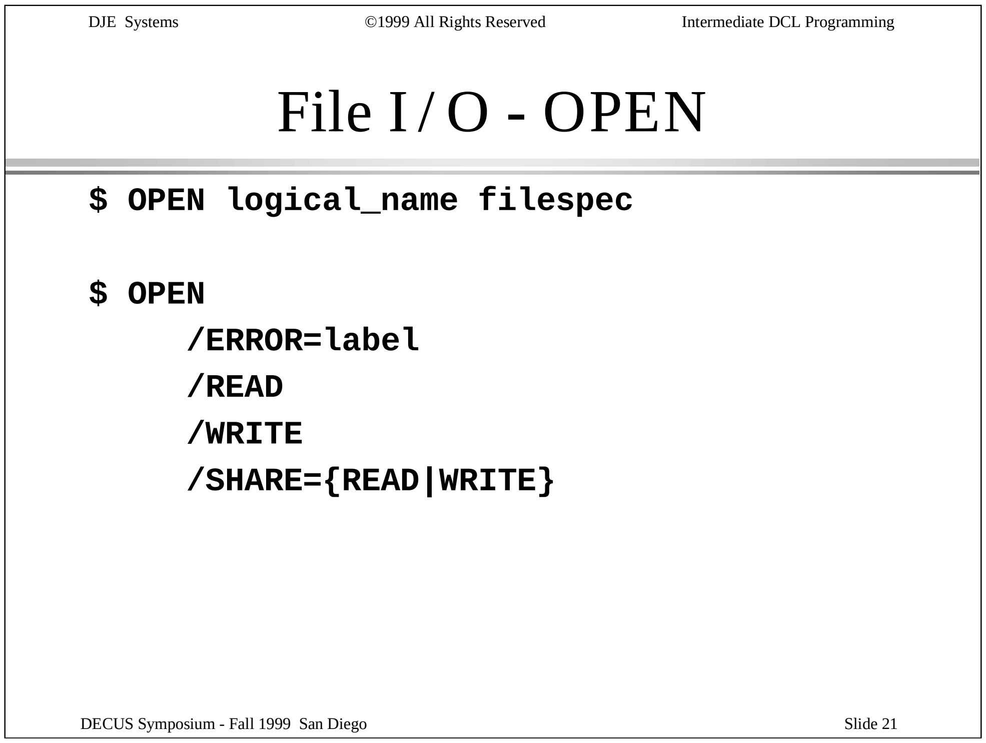
DCL provides four statements for performing file I/O: OPEN, READ, WRITE and CLOSE.

Use OPEN to begin using a file.

Use READ to get data from a file.

Use WRITE to write data to a file or to update existing records.

Use CLOSE to finish using a file and release the associated resources.



File I/O - OPEN

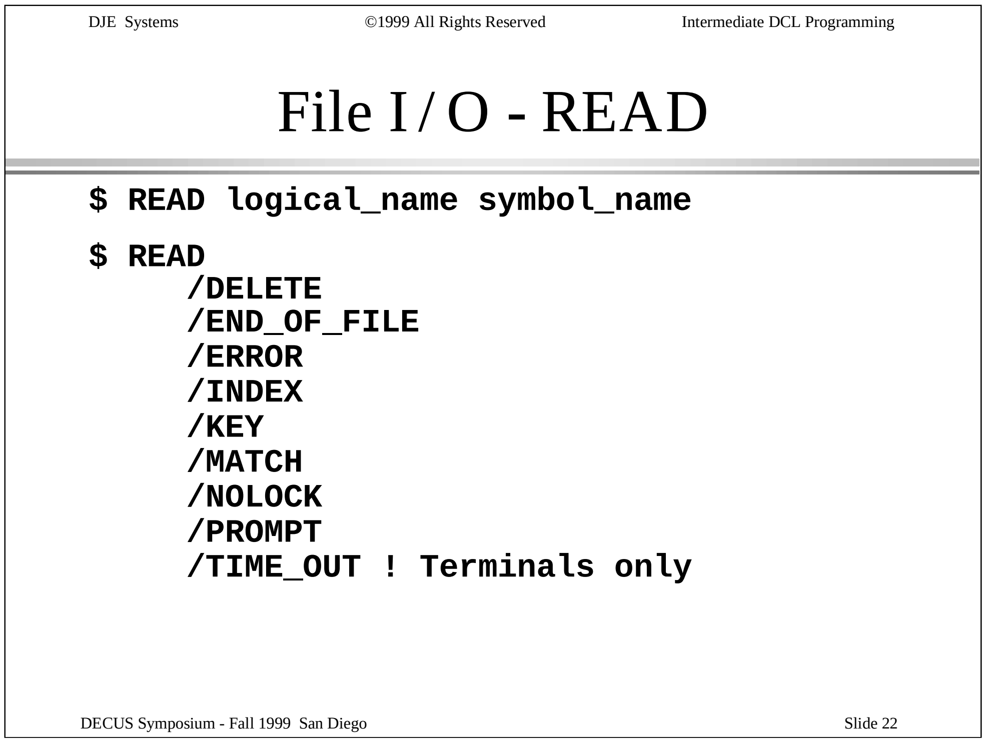
```
$ OPEN logical_name filespec

$ OPEN
     /ERROR=label
     /READ
     /WRITE
     /SHARE={READ|WRITE}
```

The OPEN statement makes a file available for processing. It establishes a “channel identifier” which you can use in READ and WRITE statements as well as in the CLOSE statement to finish using the file.

/READ opens the file for reading. The file must exist.

/WRITE opens the file for writing. If not accompanied by /READ, a new file is created.

/SHARE specifies how other I/O streams may use the file.

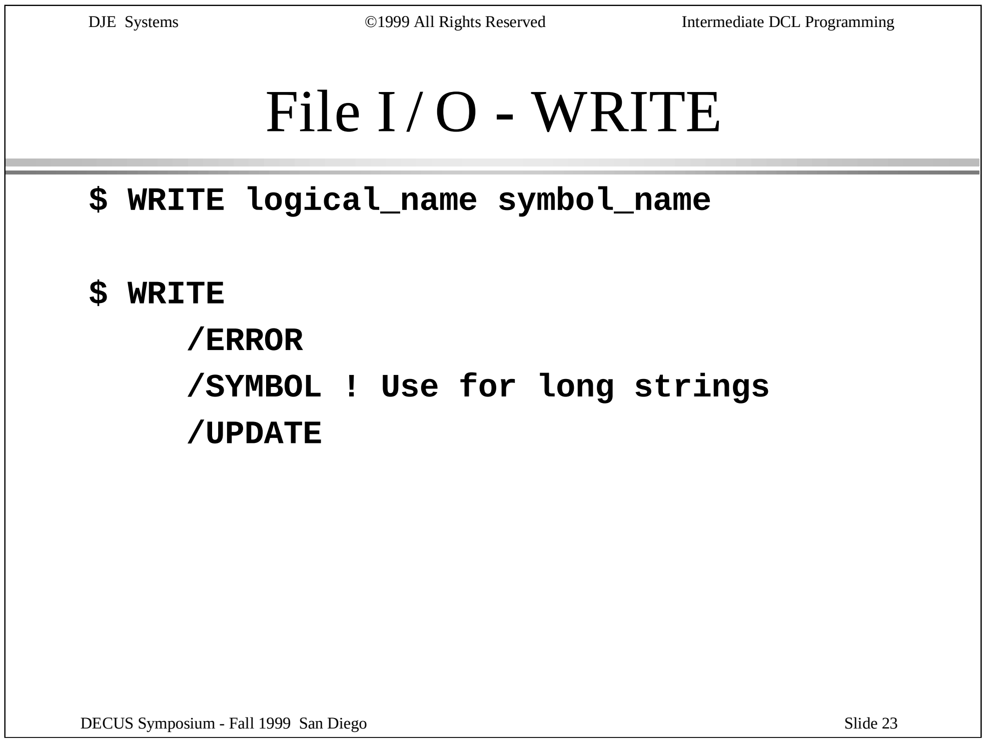


File I / O - READ

```
$ READ logical_name symbol_name

$ READ
     /DELETE
     /END_OF_FILE
     /ERROR
     /INDEX
     /KEY
     /MATCH
     /NOLOCK
     /PROMPT
     /TIME_OUT ! Terminals only
```

The READ statement retrieves data from a file.

The qualifiers shown provide for labels to receive control in case of error or at end of file, and provide ways to specify a key to match, which index to search, how to match the key value specified (RMS indexed files), a prompt string to use when READing from a terminal, and a TIME_OUT value for a time to wait for input from a terminal.



File I / O - WRITE

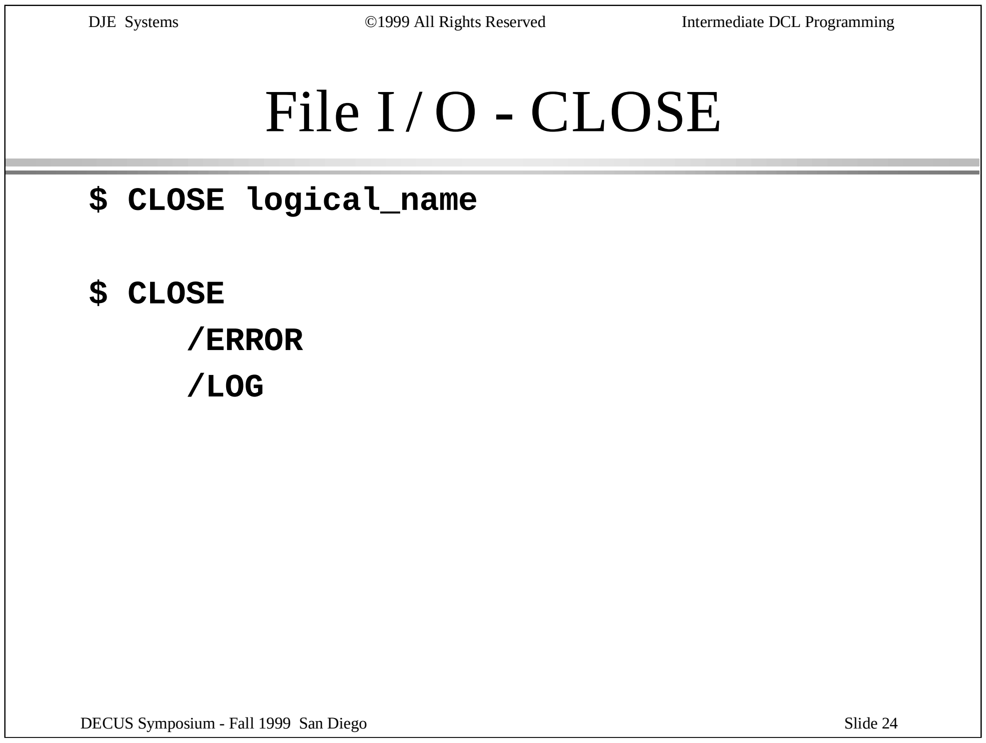
```
$ WRITE logical_name symbol_name

$ WRITE
     /ERROR
     /SYMBOL ! Use for long strings
     /UPDATE
```

The WRITE statement is used to write data to a file or to update an existing record in a file.

/ERROR is used to specify a label where control should be transferred when an error occurs.

/UPDATE is used to update an existing record.

/SYMBOL is used to write strings longer than 255 bytes.



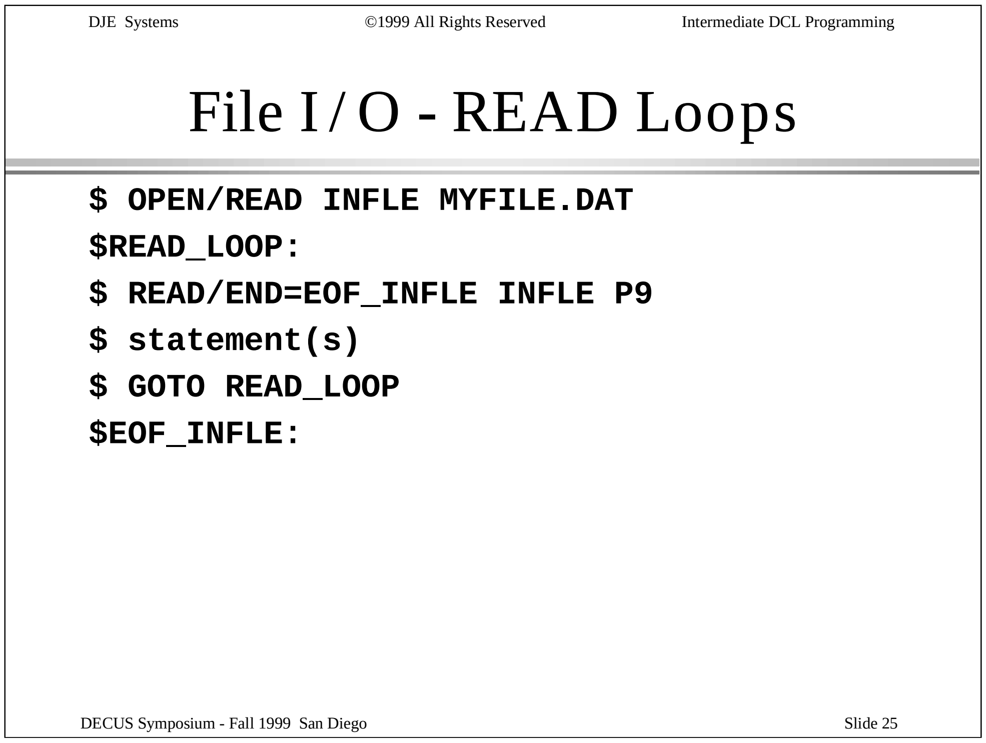
File I / O - CLOSE

```
$ CLOSE logical_name

$ CLOSE
     /ERROR
     /LOG
```

The CLOSE statement is used to finish using a file. The buffers are flushed and all associated resources are released.

/ERROR is used to specify a label where control should be transferred when an error occurs.

/LOG is used to avoid an error message when closing a file that isn’t open.

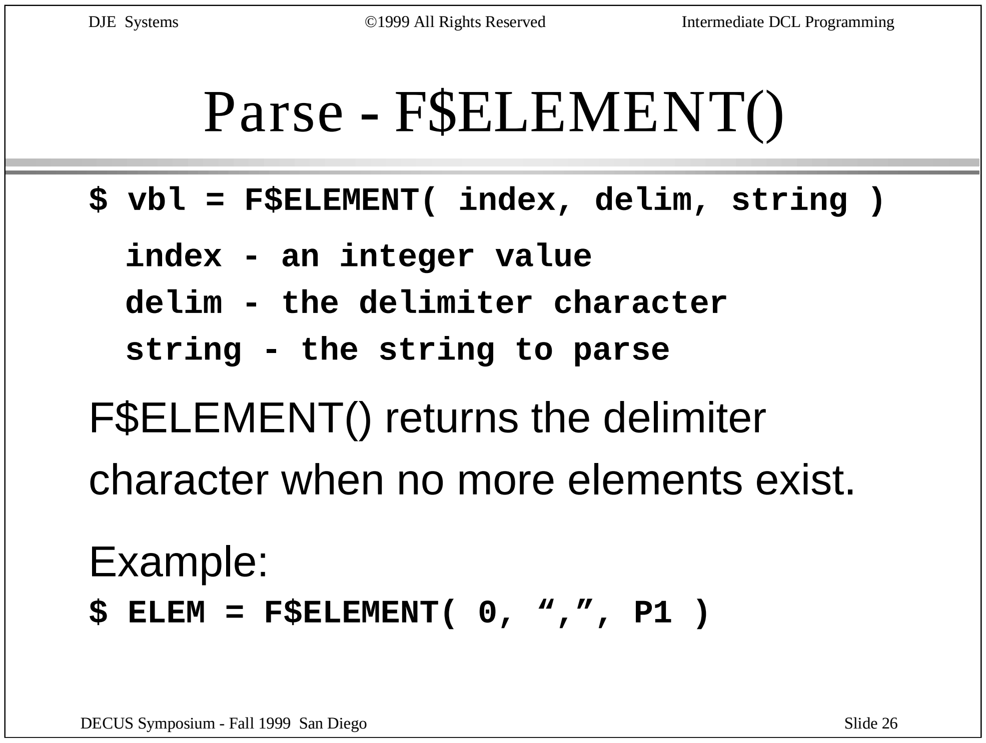


File I/O - READ Loops

```
$ OPEN/READ INFLE MYFILE.DAT
$READ_LOOP:
$ READ/END=EOF_INFLE INFLE P9
$ statement(s)
$ GOTO READ_LOOP
$EOF_INFLE:
```

Here’s an example of a loop to read a file and process its records.

The /END_OF_FILE qualifier is used to direct control to the “EOF_INFLE” label at end of file.



Parse - F$ELEMENT()

```
$ vbl = F$ELEMENT( index, delim, string )
  index - an integer value
  delim - the delimiter character
  string - the string to parse
```

F$ELEMENT() returns the delimiter character when no more elements exist.

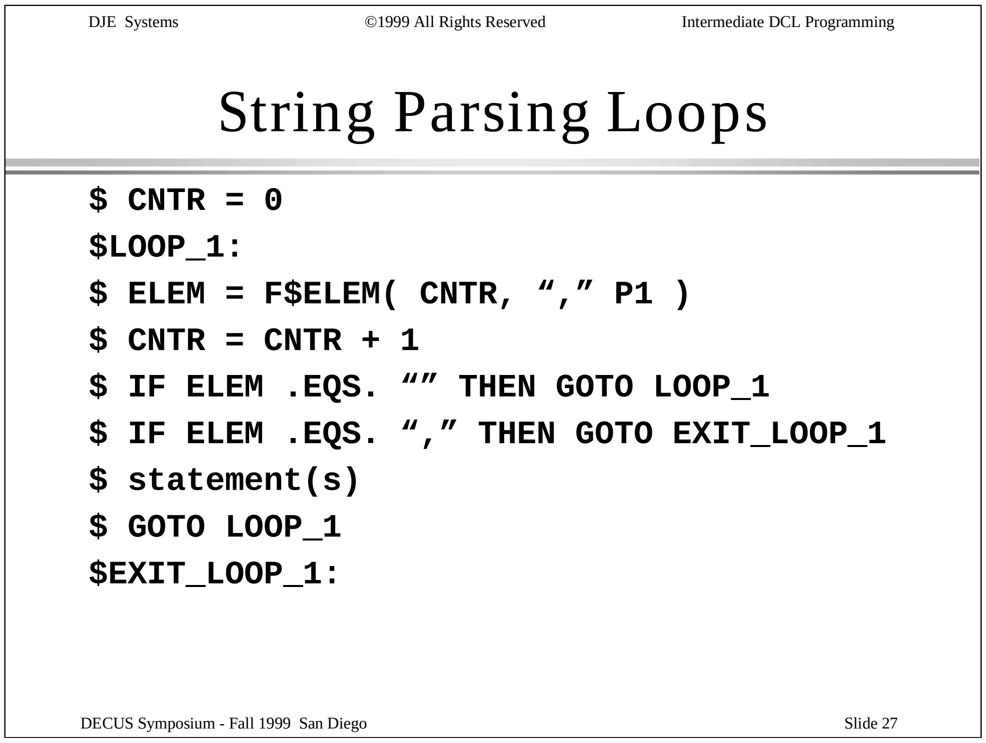
Example:

```
$ ELEM = F$ELEMENT( 0, “,”, P1 )
```

Moving on to some more advanced lexical functions, we start with F$ELEMENT. Use this to parse strings by searching for specific characters, such as comma, pipe symbol, space, etc.

The “index” starts at zero. The “delim” parameter can be any single character.

If the value of “index” points to an element beyond the end of the string, the function returns the delimiter character.



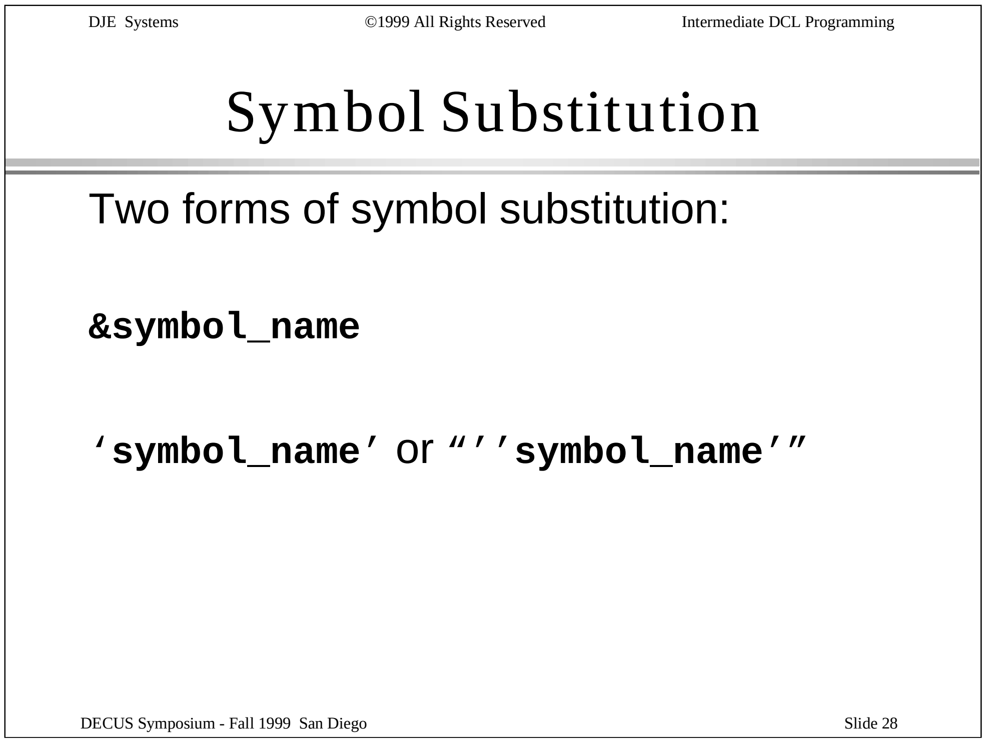
String Parsing Loops

```
$ CNTR = 0
$LOOP_1:
$ ELEM = F$ELEM( CNTR, “,” P1 )
$ CNTR = CNTR + 1
$ IF ELEM .EQS. “” THEN GOTO LOOP_1
$ IF ELEM .EQS. “,” THEN GOTO EXIT_LOOP_1
$ statement(s)
$ GOTO LOOP_1
$EXIT_LOOP_1:
```

Here’s an example of a loop for retrieving all the elements of a string. In the example, elements of the string are delimited by commas.

Note that the index is incremented before the element returned is examined. This is one way to help avoid locked loops. Ignoring null elements might not always be desirable.

When F$ELEMENT returns a comma, control is transferred to the EXIT_LOOP_1 label.



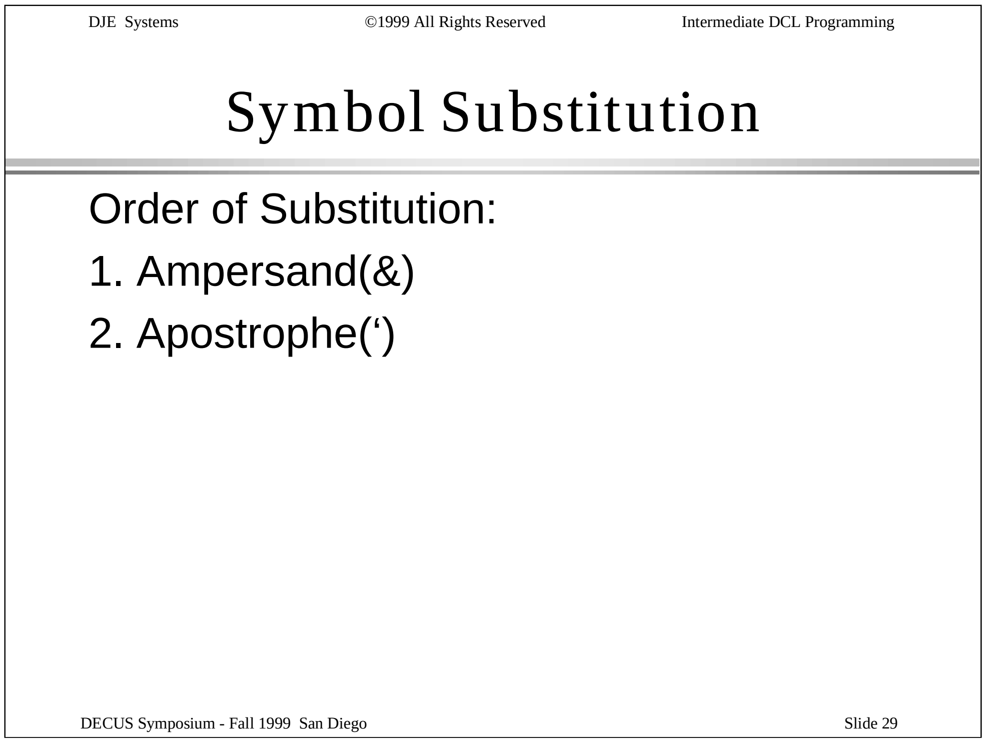
Symbol Substitution

Two forms of symbol substitution:

&symbol_name

‘symbol_name’ or **“’’symbol_name’’”**

Let’s look at symbol substitution. DCL provides two “passes”: one for symbols preceded by an ampersand (“&”) and another for symbols preceded by an apostrophe, or two apostrophes when used within a quoted string.


DJE Systems ©1999 All Rights Reserved Intermediate DCL Programming

Symbol Substitution

Order of Substitution:

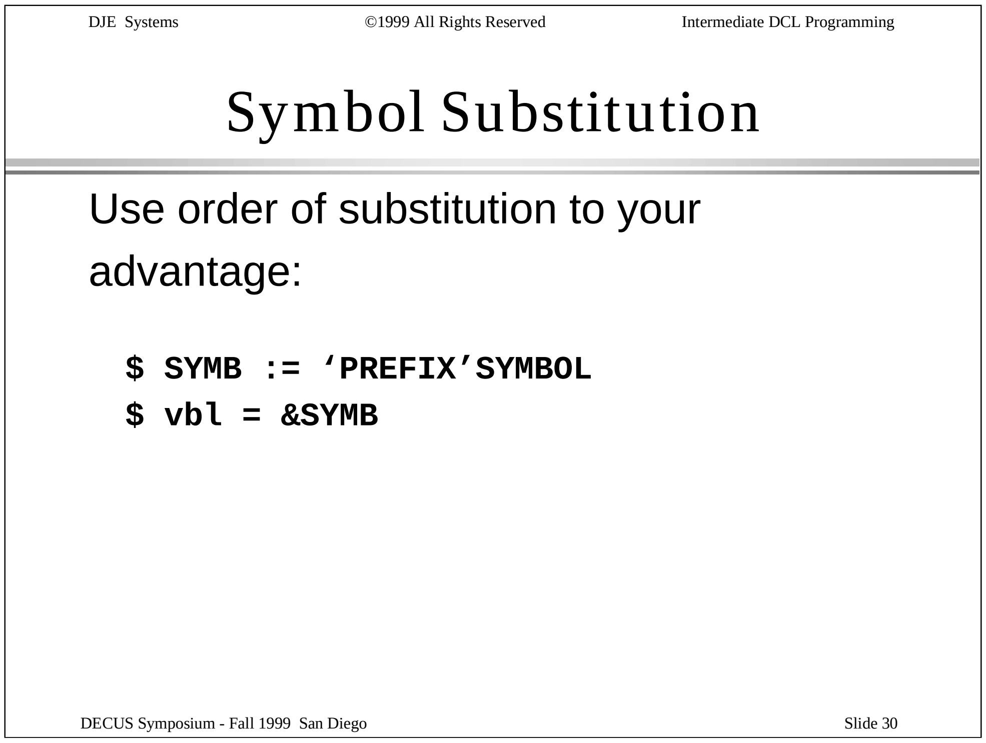
1. Ampersand(&)
2. Apostrophe(‘)

DECUS Symposium - Fall 1999 San Diego Slide 29

Symbol substitution occurs in the order shown.

First, symbols preceded by an ampersand are processed. The value of a symbol preceded by an ampersand is treated as a single “word” or token.

Second, symbols preceded by apostrophes are processed. When expanded, the value of a symbol preceded by apostrophe is treated as one or more “words” or tokens.



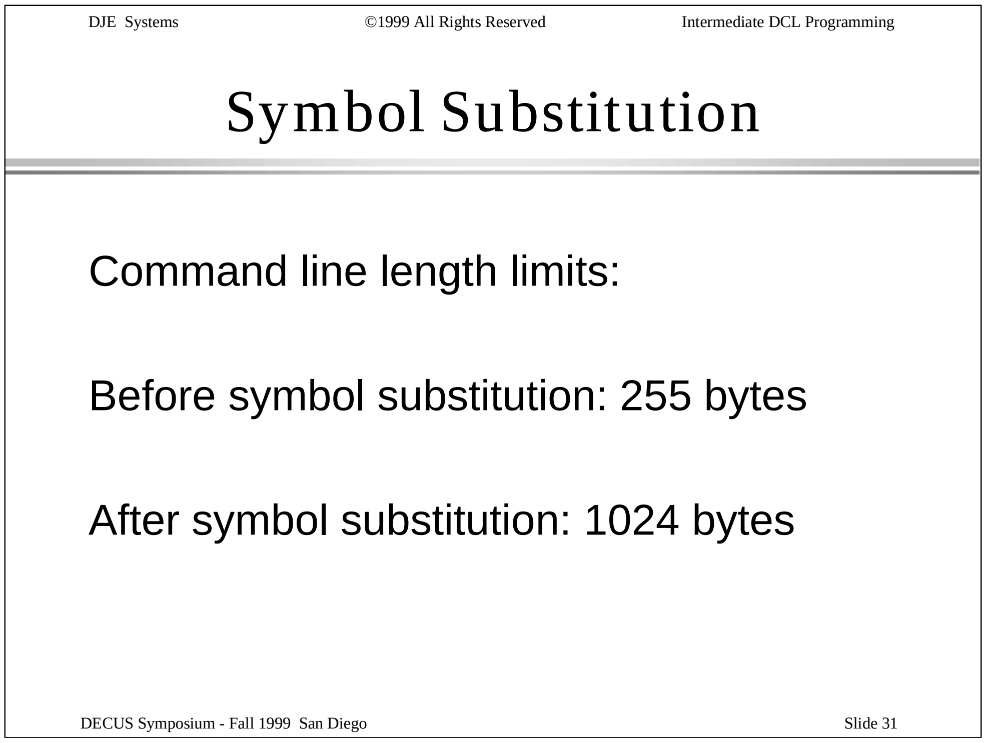
Symbol Substitution

Use order of substitution to your advantage:

```
$ SYMB := ‘PREFIX’SYMBOL
$ vbl = &SYMB
```

This slide shows how you can use symbol substitution to your own advantage.

This can be useful. This can also lead quickly to confusion.



Symbol Substitution

Command line length limits:

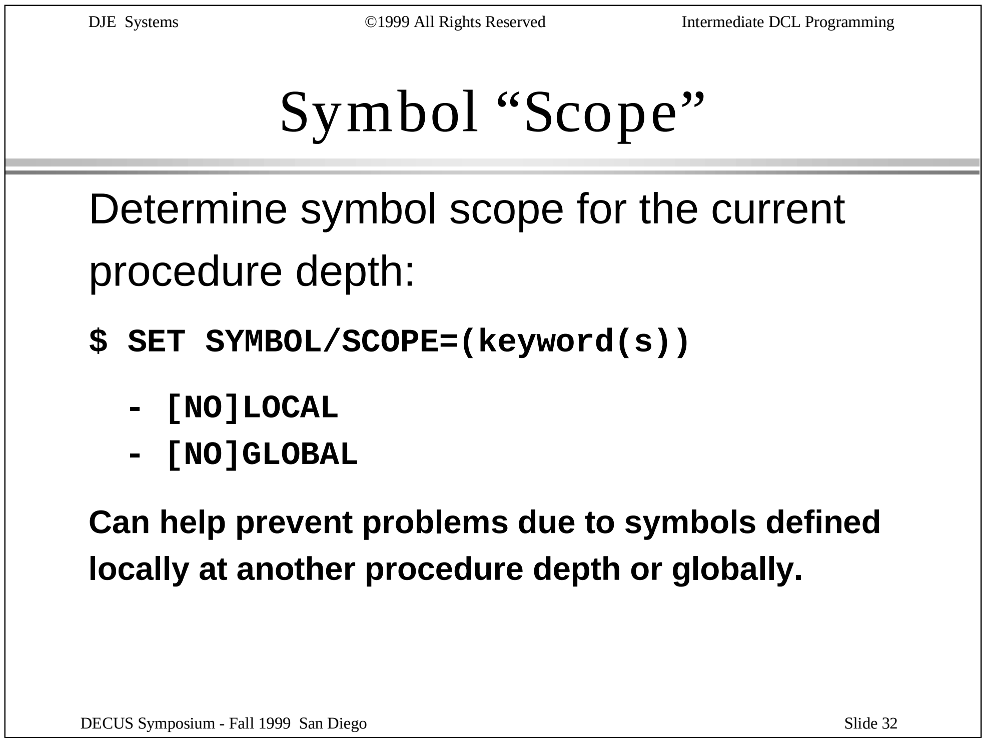
Before symbol substitution: 255 bytes

After symbol substitution: 1024 bytes

The maximum length of a command line before symbol substitution is 255 bytes.

DCL has a somewhat larger internal buffer which allows for the results of symbol substitution.

See the HELP topics “=“ and “:=“ for further information. (Steve Hoffman)



Symbol “Scope”

Determine symbol scope for the current procedure depth:

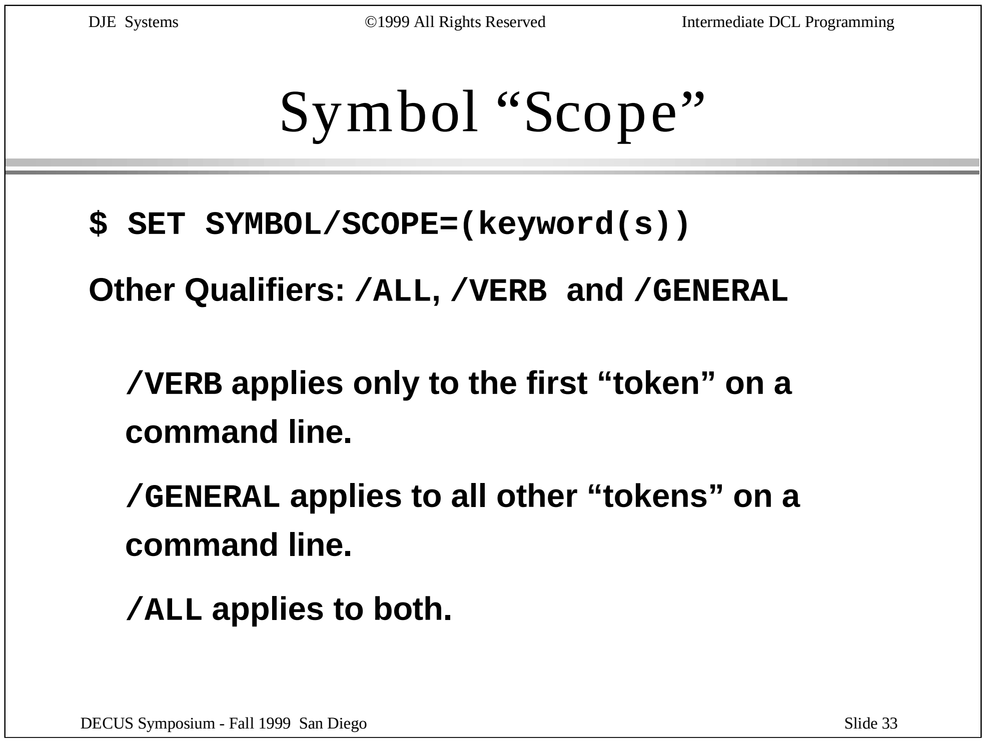
```
$ SET SYMBOL/SCOPE=(keyword(s))
  - [NO]LOCAL
  - [NO]GLOBAL
```

Can help prevent problems due to symbols defined locally at another procedure depth or globally.

Controlling symbol scope can help control confusion when a symbol name is used in more than one nested procedure.

When symbol scope is set to NOLOCAL, local symbols from lesser procedure depths are “invisible” to the current procedure depth all “greater” depths.

When symbol scope is set to NOGLOBAL, global symbols are “invisible” to the current procedure depth all “greater” depths.



Symbol “Scope”

```
$ SET SYMBOL/SCOPE=(keyword(s))
```

Other Qualifiers: `/ALL`, `/VERB` and `/GENERAL`

`/VERB` applies only to the first “token” on a command line.

`/GENERAL` applies to all other “tokens” on a command line.

`/ALL` applies to both.

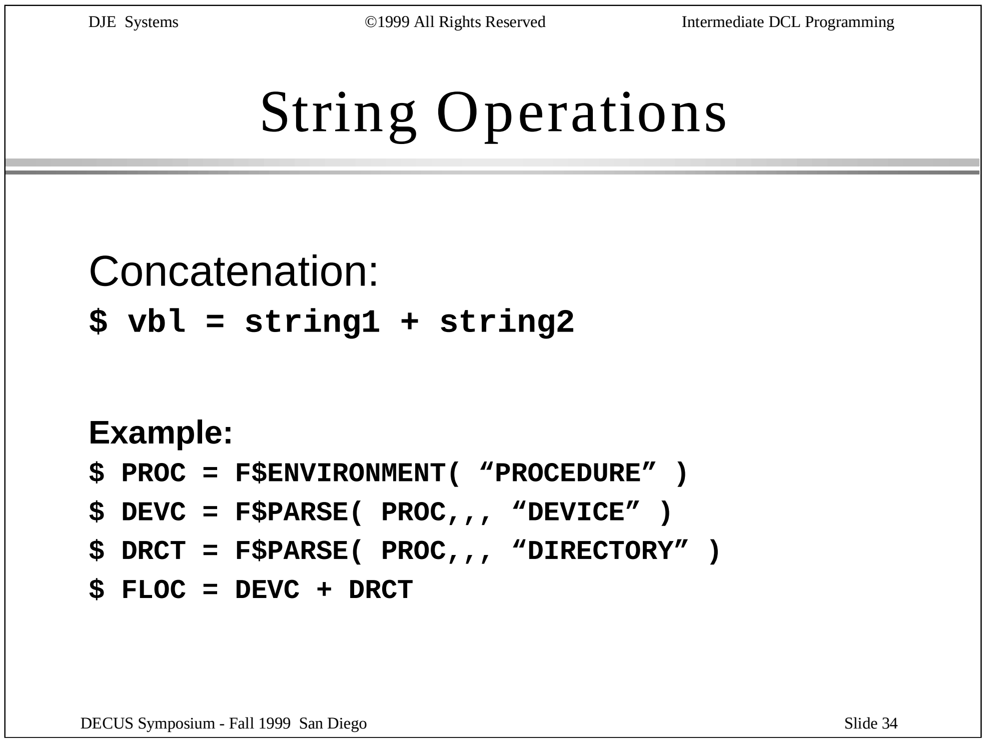
Other SET SYMBOL/SCOPE qualifiers control how the symbol scoping rules are applied.

/VERB applies to the first “token” (or “word”) on a command line.

/GENERAL applies to all other “tokens” (or “words”) on a command line.

/ALL applies to all of the “tokens” (or “words”) on a command line.

“Words” are delimted by the space (ASCII 32) character for DCL’s purposes.



String Operations

Concatenation:

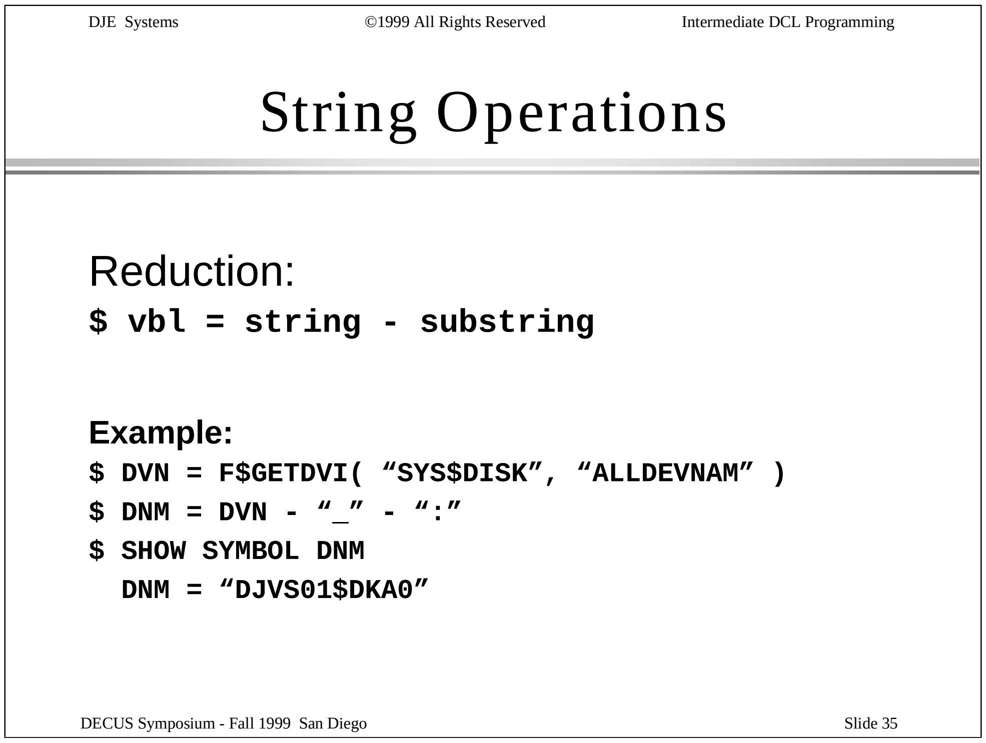
```
$ vbl = string1 + string2
```

Example:

```
$ PROC = F$ENVIRONMENT( “PROCEDURE” )
$ DEVC = F$PARSE( PROC,,, “DEVICE” )
$ DRCT = F$PARSE( PROC,,, “DIRECTORY” )
$ FLOC = DEVC + DRCT
```

Connecting two or more shorter strings together is known as “concatenating”. The original strings remain unchanged. The target string includes the contents of the original strings as one longer string. No spaces or other characters are inserted or appended.

The example illustrates the use of the F$ENVIRONMENT and F$PARSE lexical functions. Two elements of the procedure file specification are then concatenated together for use later on in the procedure.



String Operations

Reduction:

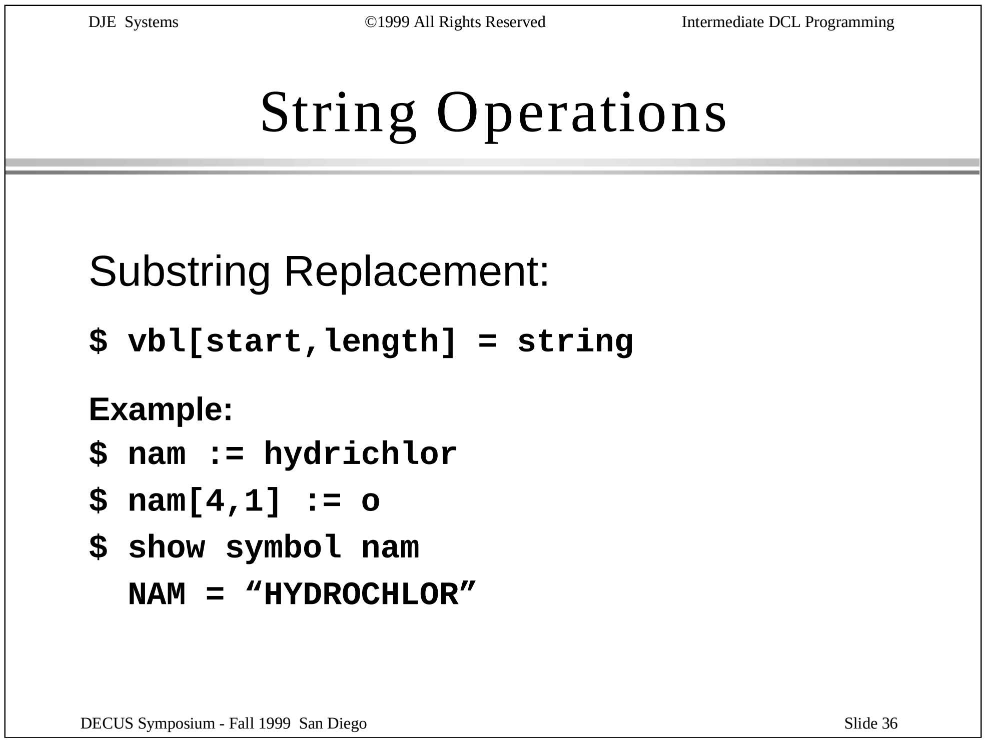
```
$ vbl = string - substring
```

Example:

```
$ DVN = F$GETDVI( “SYS$DISK”, “ALLDEVNAM” )
$ DNM = DVN - “_” - “:”
$ SHOW SYMBOL DNM
  DNM = “DJVS01$DKA0”
```

Removing a substring from a longer string is known as string reduction. The first instance of the substring is removed from the longer string, and the result is stored in the target symbol.

The example illustrates stripping the underscore and colon from a device name.



String Operations

Substring Replacement:

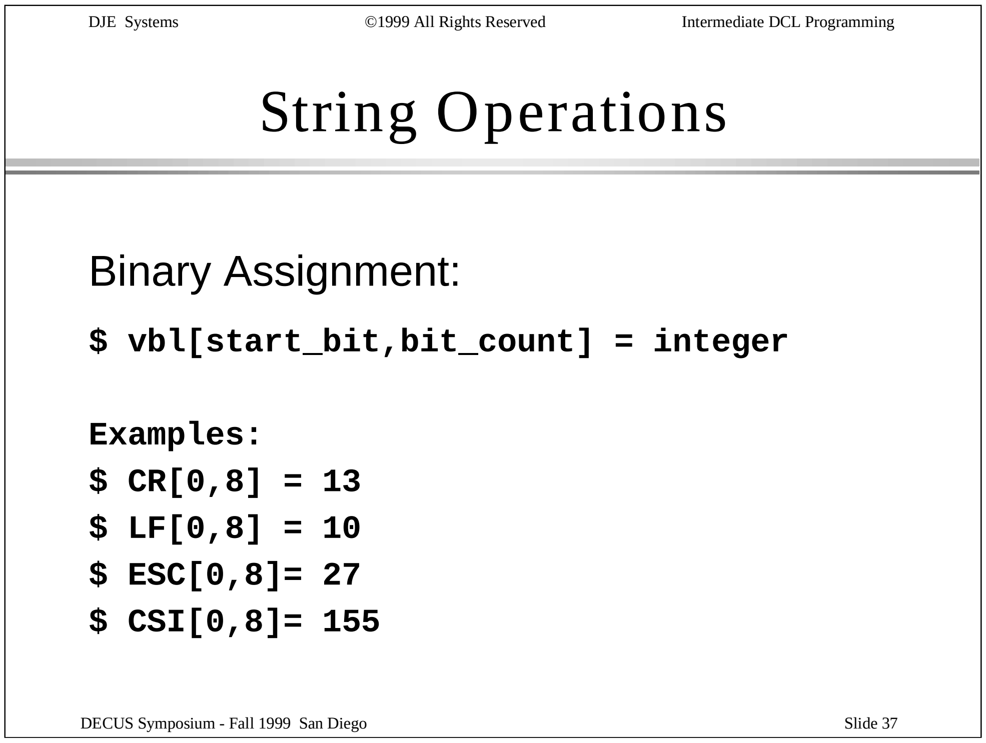
```
$ vbl[start,length] = string
```

Example:

```
$ nam := hydrichlor
$ nam[4,1] := o
$ show symbol nam
  NAM = “HYDROCHLOR”
```

Substrings within a longer string can be replaced. The starting position and the length of replacement (in bytes) are indicated within the square brackets following the name of the target string. The colon-equal operator (“:=“)or the colon-equal-equal operator (“:==“) **MUST** be used for this type of assignment.

The example illustrates replacing a single character near the middle of a string.



String Operations

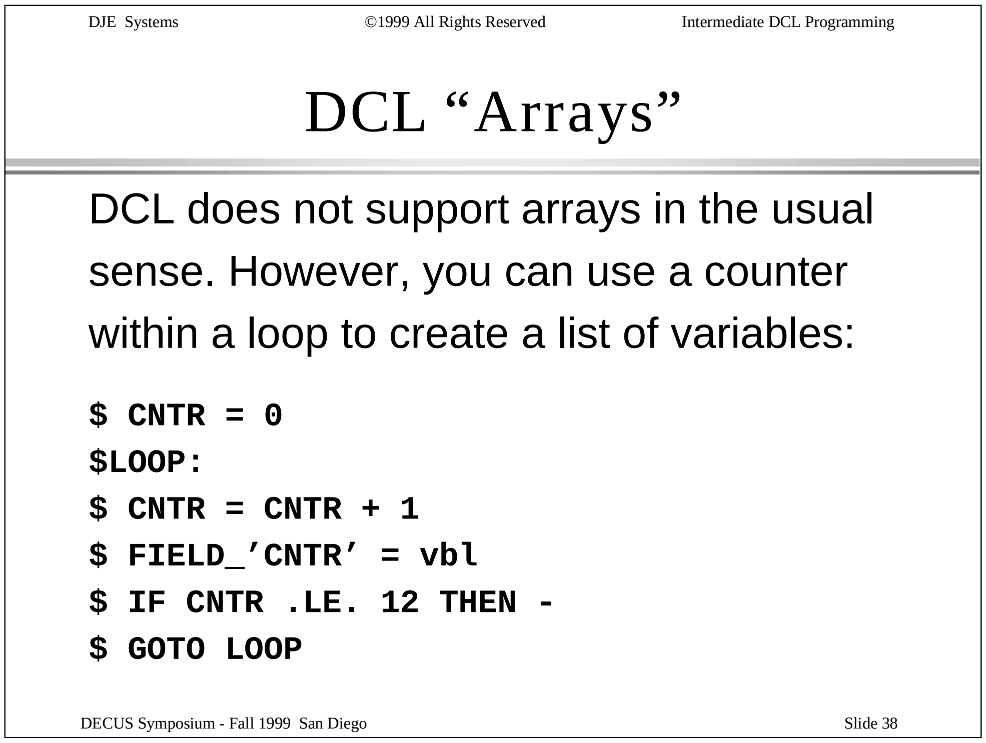
Binary Assignment:

```
$ vbl[start_bit,bit_count] = integer

Examples:
$ CR[0,8] = 13
$ LF[0,8] = 10
$ ESC[0,8]= 27
$ CSI[0,8]= 155
```

Binary values can also be assigned using the square brackets following the name of the target symbol.

The equal operator (“=“) or equal-equal operator (“==“) **MUST** be used for this type of assignment.

The first value within the brackets is the bit displacement into the target field. The second value is the number of bits to be affected by the assignment.

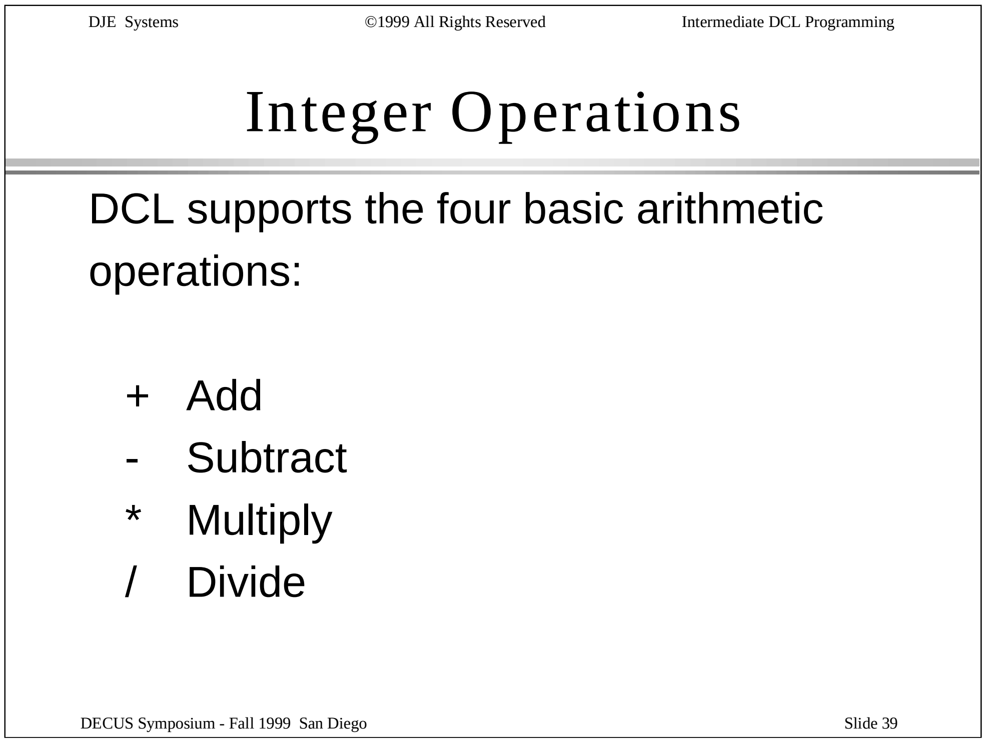


DCL “Arrays”

DCL does not support arrays in the usual sense. However, you can use a counter within a loop to create a list of variables:

```
$ CNTR = 0
$LOOP:
$ CNTR = CNTR + 1
$ FIELD_’CNTR’ = vbl
$ IF CNTR .LE. 12 THEN -
$ GOTO LOOP
```

While DCL does not support arrays in the sense of subscripted variables as one might find in a 3GL, a counter can be used to sequentially create variables with a number appended to the variable name. This can be done using symbol substitution to create a target variable name, as shown the example.

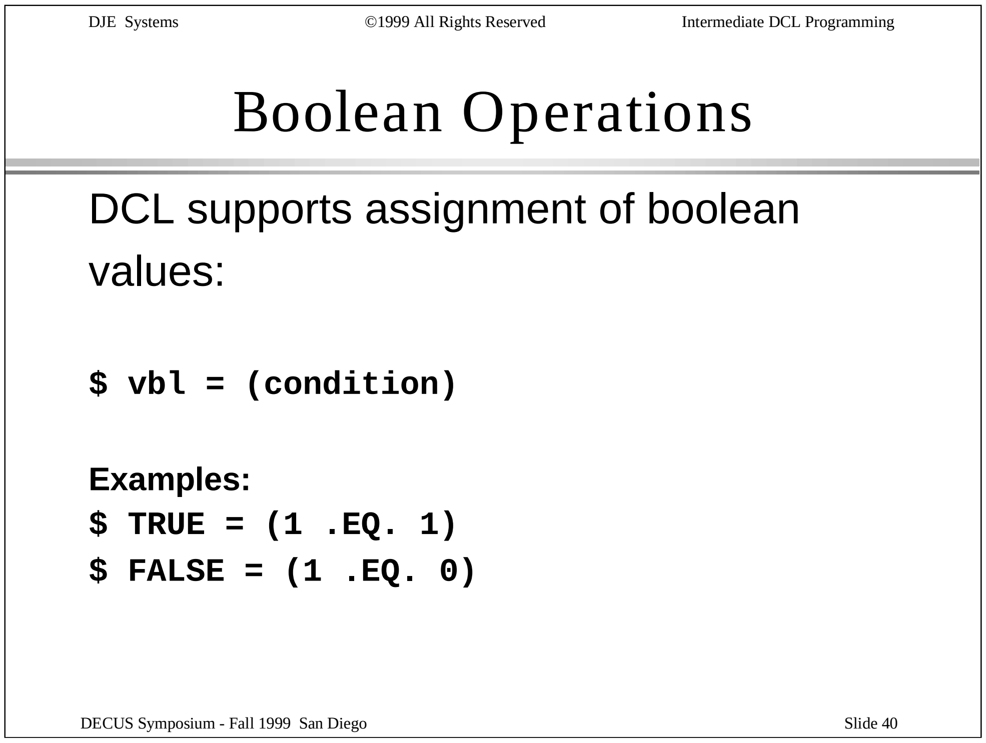


Integer Operations

DCL supports the four basic arithmetic operations:

+	Add
-	Subtract
*	Multiply
/	Divide

DCL can perform arithmetic operations on integer values. Integers are treated as signed longword (32 bit) values.



Boolean Operations

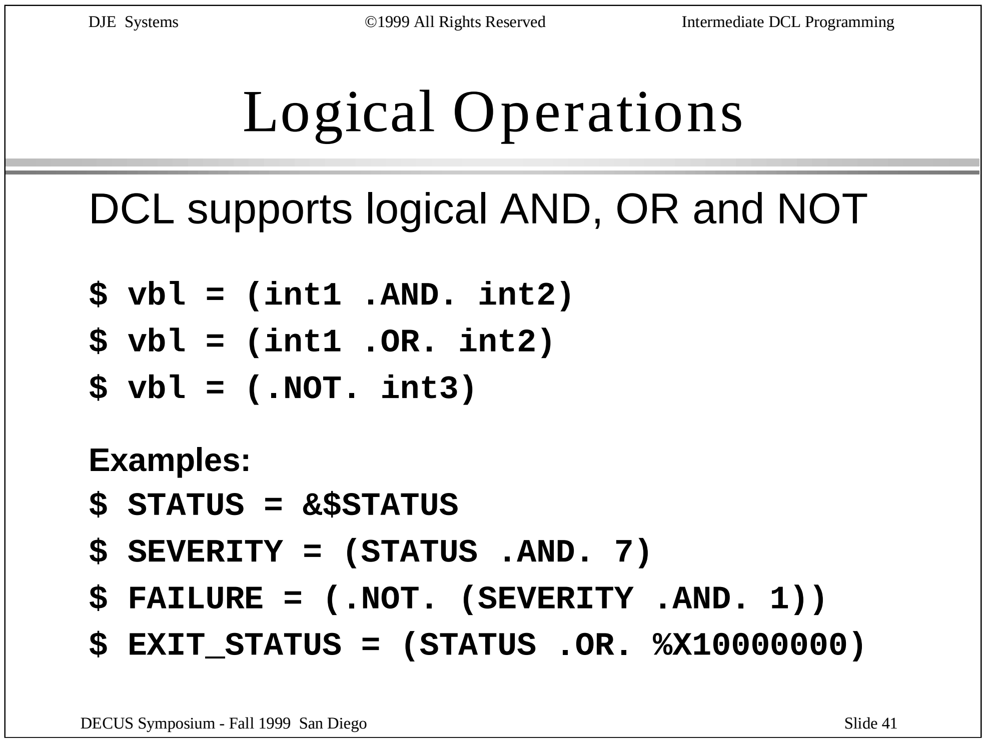
DCL supports assignment of boolean values:

```
$ vbl = (condition)
```

Examples:

```
$ TRUE = (1 .EQ. 1)
$ FALSE = (1 .EQ. 0)
```

The “truth” value of conditional expressions can be assigned to variables for use in multiple comparisons within a procedure.



Logical Operations

DCL supports logical AND, OR and NOT

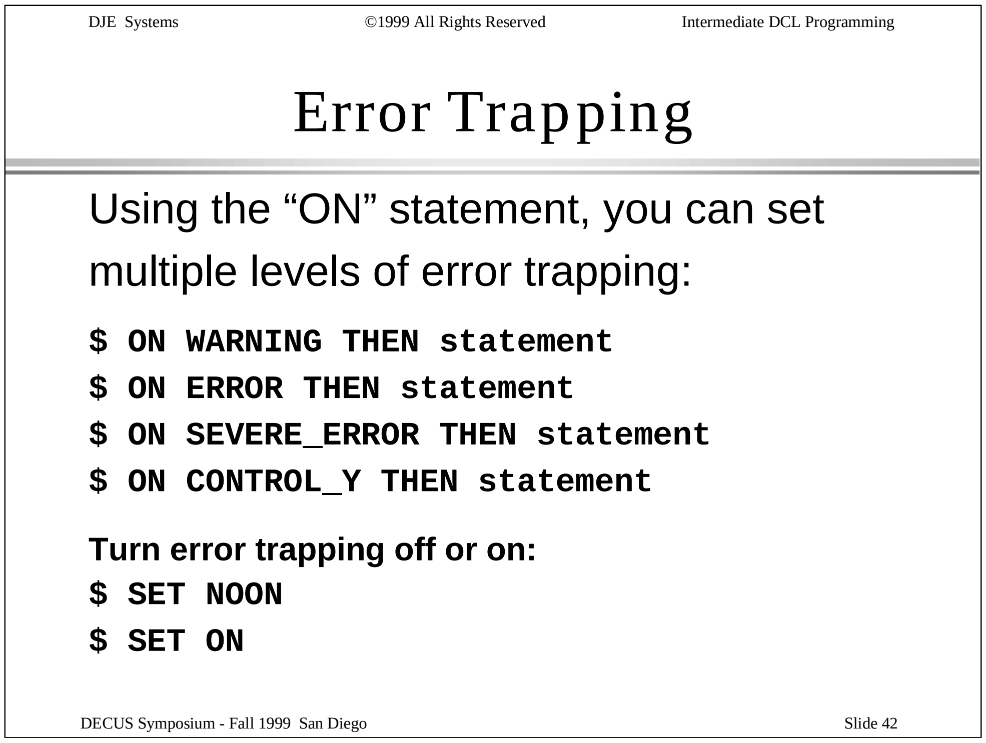
```
$ vbl = (int1 .AND. int2)
$ vbl = (int1 .OR. int2)
$ vbl = (.NOT. int3)
```

Examples:

```
$ STATUS = &$STATUS
$ SEVERITY = (STATUS .AND. 7)
$ FAILURE = (.NOT. (SEVERITY .AND. 1))
$ EXIT_STATUS = (STATUS .OR. %X10000000)
```

DCL provides the logical AND and logical OR boolean operators, and the NOT operator.

The examples show the use of the .AND., .NOT. and .OR. operators. Notice that FAILURE is taken as the logical NOT of a condition that would indicate success.



Error Trapping

Using the “ON” statement, you can set multiple levels of error trapping:

```
$ ON WARNING THEN statement
$ ON ERROR THEN statement
$ ON SEVERE_ERROR THEN statement
$ ON CONTROL_Y THEN statement
```

Turn error trapping off or on:

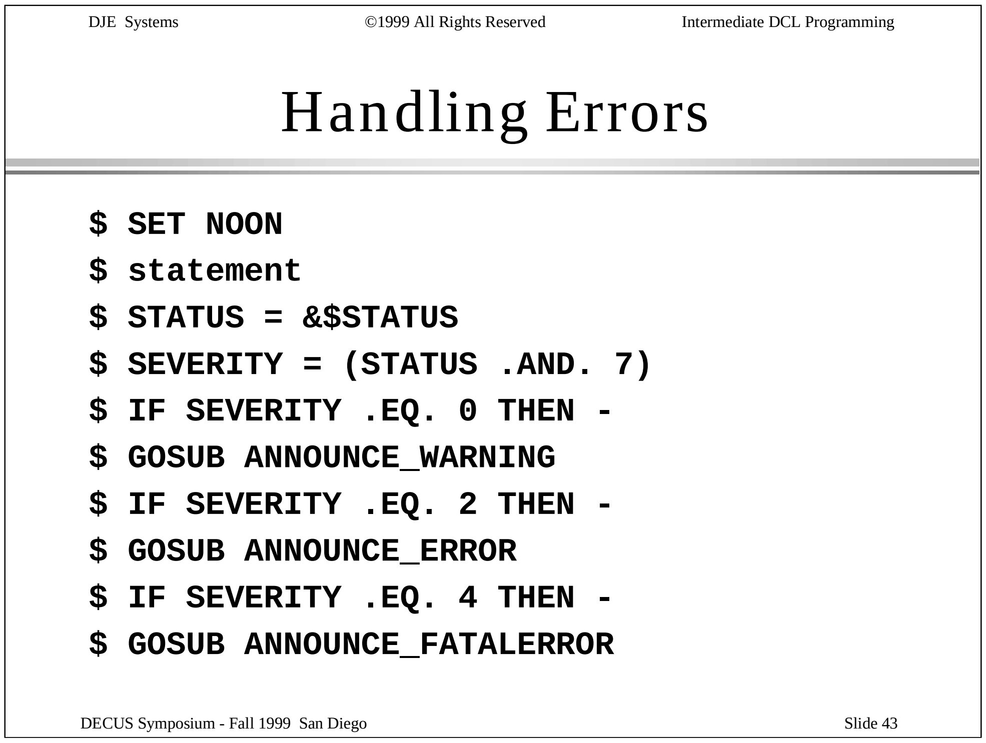
```
$ SET NOON
$ SET ON
```

Through the use of the “ON” statement and the “SET ON” and “SET NOON” statements, you can control how DCL behaves when various type of errors or events occur.

A WARNING event occurs when the severity is zero(0) (($STATUS .AND. 7) .EQ. 0).

An ERROR event occurs when the severity is two(2) (($STATUS .AND. 7) .EQ. 2).

A SEVERE ERROR event occurs when the severity is four(4) (($STATUS .AND. 7) .EQ. 4).



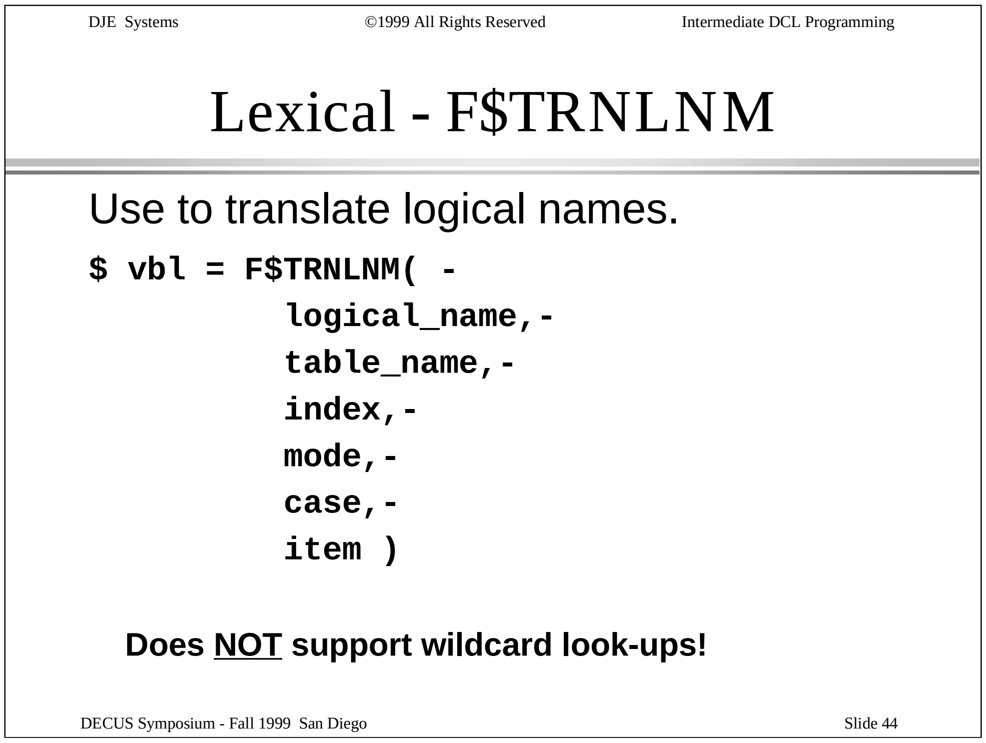
Handling Errors

```
$ SET NOON
$ statement
$ STATUS = &$STATUS
$ SEVERITY = (STATUS .AND. 7)
$ IF SEVERITY .EQ. 0 THEN -
$ GOSUB ANNOUNCE_WARNING
$ IF SEVERITY .EQ. 2 THEN -
$ GOSUB ANNOUNCE_ERROR
$ IF SEVERITY .EQ. 4 THEN -
$ GOSUB ANNOUNCE_FATALERROR
```

This code segment illustrates how to trap and handle errors without DCL's intervention.

After the "statement" is executed, the value of $STATUS is saved, and the SEVERITY is derived from the saved STATUS.

Different actions are taken depending upon the value of the SEVERITY symbol.



Lexical - F$TRNLNM

Use to translate logical names.

```
$ vbl = F$TRNLNM( -
          logical_name,-
          table_name,-
          index,-
          mode,-
          case,-
          item )
```

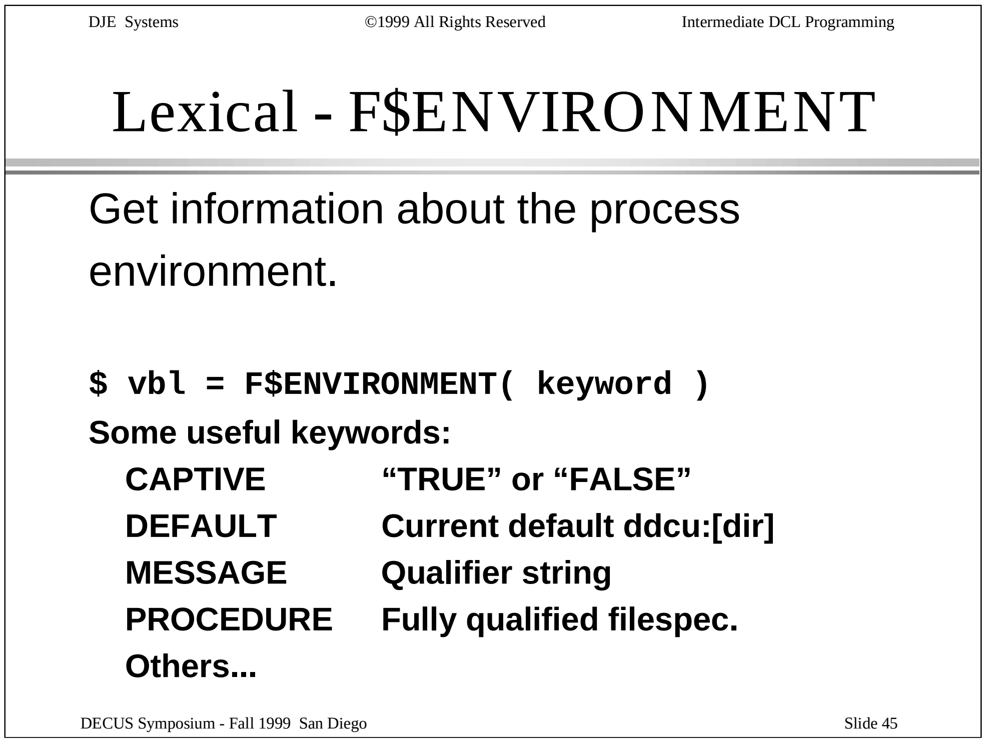
Does <u>NOT</u> support wildcard look-ups!

Now, we’ll look at some more lexical functions.

F$TRNLNM() is used to get the translation of a logical name, isolate the translation to a specific logical name table, index or mode, or to determine such characteristics about a logical name.

Notice that F$TRNLNM() does **<u>NOT</u>** support wildcarded logical name expressions.

F$TRNLNM() supercedes the older F$LOGICAL().



Lexical - F$ENVIRONMENT

Get information about the process environment.

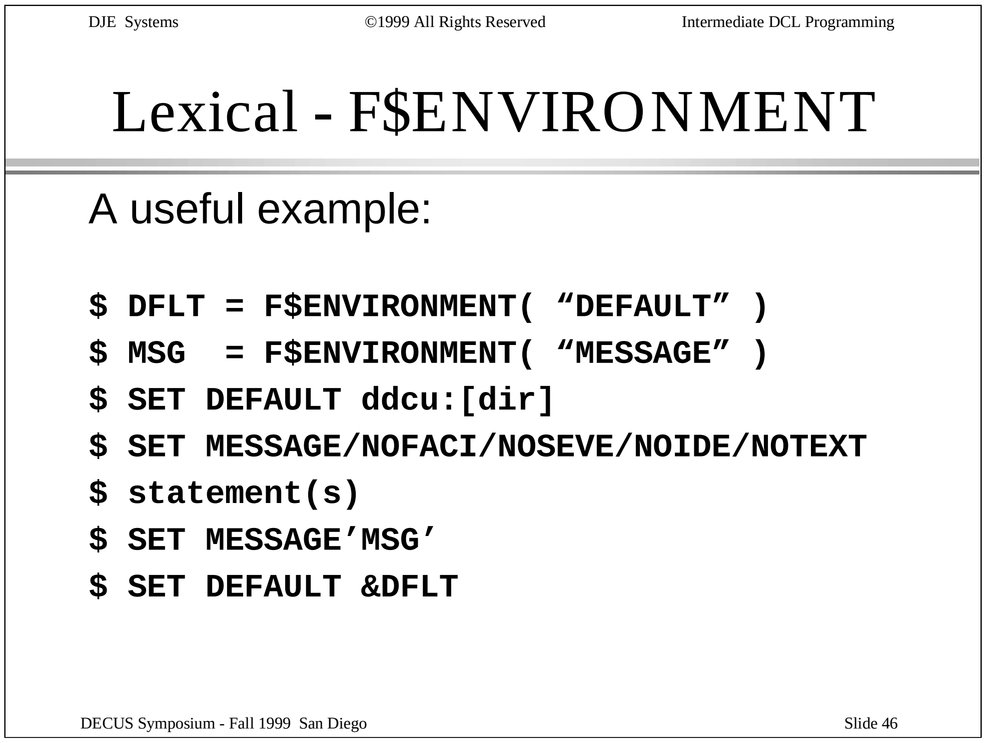
```
$ vbl = F$ENVIRONMENT( keyword )
```

Some useful keywords:

CAPTIVE	**“TRUE” or “FALSE”**
DEFAULT	**Current default ddcu:[dir]**
MESSAGE	**Qualifier string**
PROCEDURE	**Fully qualified filespec.**
Others...	

The F$ENVIRONMENT() lexical can be used to get information about the current process environment.

The example keywords shown are some of the more useful keywords. Other keywords are available to determine the CONTROL characters currently enabled, the current procedure depth, the current DCL prompt string, the current default file protection, the current symbol scope, etc.

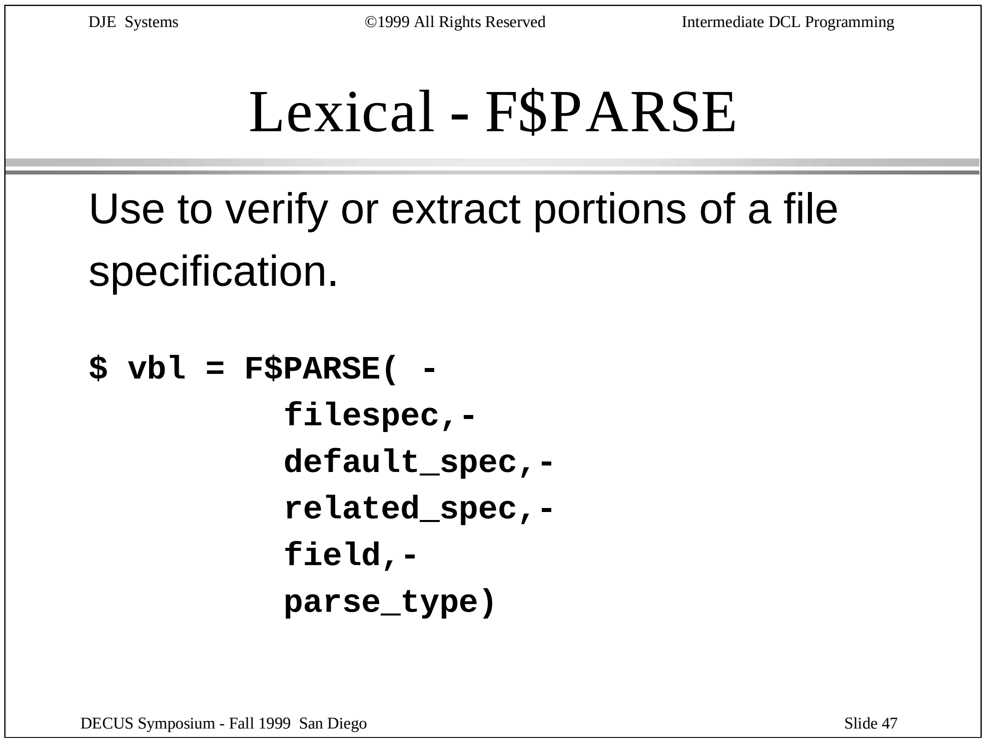


Lexical - F$ENVIRONMENT

A useful example:

```
$ DFLT = F$ENVIRONMENT( “DEFAULT” )
$ MSG  = F$ENVIRONMENT( “MESSAGE” )
$ SET DEFAULT ddcu:[dir]
$ SET MESSAGE/NOFACI/NOSEVE/NOIDE/NOTEXT
$ statement(s)
$ SET MESSAGE’MSG’
$ SET DEFAULT &DFLT
```

This example illustrates one method of suppressing messages selectively. The technique shown is to use the F$ENVIRONMENT lexical to save the MESSAGE display state, change it to what is wanted, then change it back to the original state.



Lexical - F$PARSE

Use to verify or extract portions of a file specification.

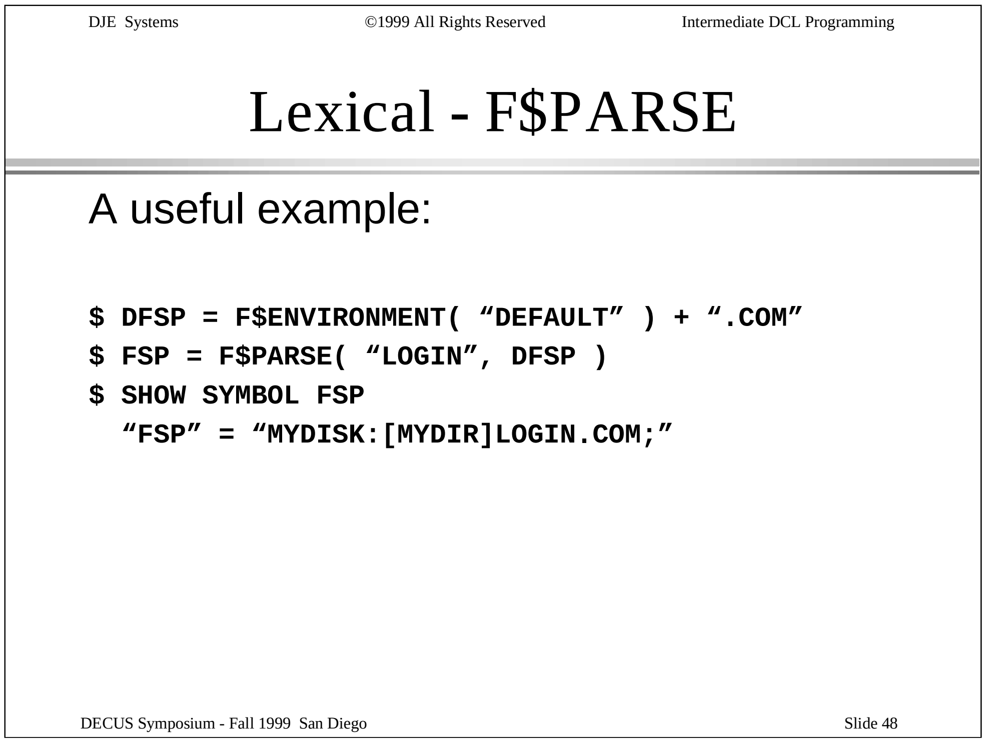
```
$ vbl = F$PARSE( -
          filespec,-
          default_spec,-
          related_spec,-
          field,-
          parse_type)
```

The F$PARSE() lexical can be used to verify a file specification or to extract portions of a valid file specification.

The “parse_type” keywords are SYNTAX_ONLY and NO_CONCEAL.

SYNTAX_ONLY parses a file spec and returns values whether the specified file exists or not.

NO_CONCEAL can be used to get the translation of a logical defined with the CONCEALED translation attribute.


DJE Systems ©1999 All Rights Reserved Intermediate DCL Programming

Lexical - F$PARSE

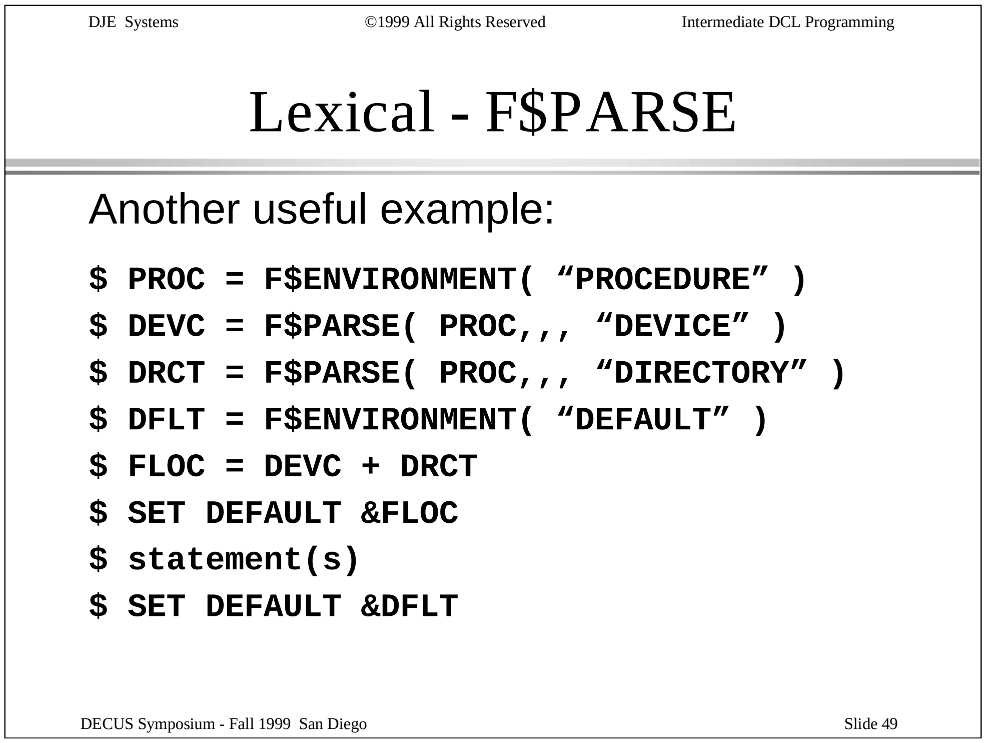
A useful example:

```
$ DFSP = F$ENVIRONMENT( “DEFAULT” ) + “.COM”
$ FSP = F$PARSE( “LOGIN”, DFSP )
$ SHOW SYMBOL FSP
  “FSP” = “MYDISK:[MYDIR]LOGIN.COM;”
```

DECUS Symposium - Fall 1999 San Diego Slide 48

Another use of F$PARSE is to complete a partial file specification, applying default values for those portions not specified.

The example shown illustrates how “LOGIN” can be expanded by applying appropriate default values for the other portions of the file specification.



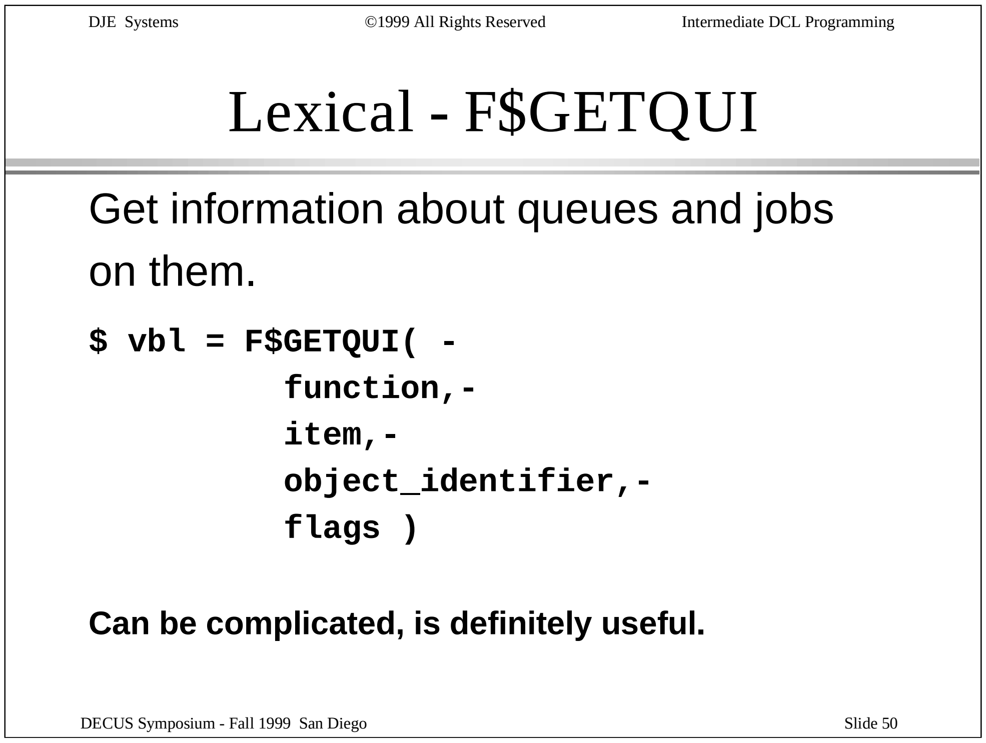
Lexical - F$PARSE

Another useful example:

```
$ PROC = F$ENVIRONMENT( “PROCEDURE” )
$ DEVC = F$PARSE( PROC,,, “DEVICE” )
$ DRCT = F$PARSE( PROC,,, “DIRECTORY” )
$ DFLT = F$ENVIRONMENT( “DEFAULT” )
$ FLOC = DEVC + DRCT
$ SET DEFAULT &FLOC
$ statement(s)
$ SET DEFAULT &DFLT
```

This example illustrates how F$PARSE can be used to extract portions of a file specification.

In the example, a new default disk and directory specification is derived from the disk and directory where the currently executing DCL procedure is found, the current default disk and directory are saved, then the new default is applied. A (series of) statement(s) is(are) executed, then the original default disk and directory specification is restored.



Lexical - F$GETQUI

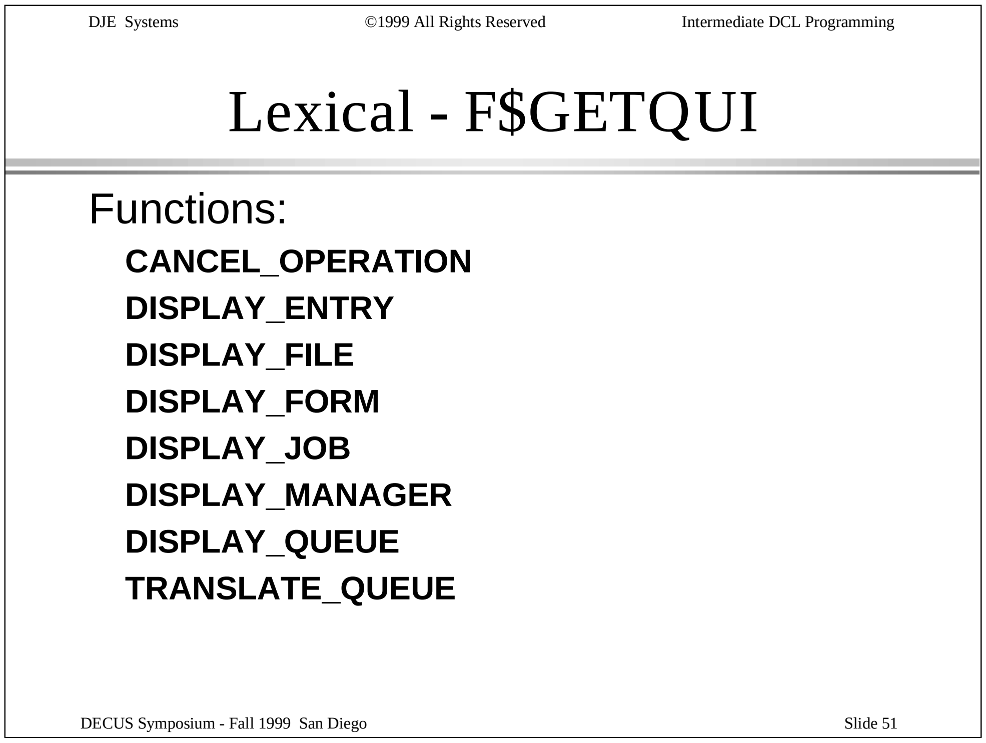
Get information about queues and jobs on them.

```
$ vbl = F$GETQUI( -
          function,-
          item,-
          object_identifier,-
          flags )
```

Can be complicated, is definitely useful.

The F$GETQUI() lexical function can be used to get information about queues and the jobs on those queues.

F$GETQUI() can be complicated to use; but its usefulness is well worth the effort.



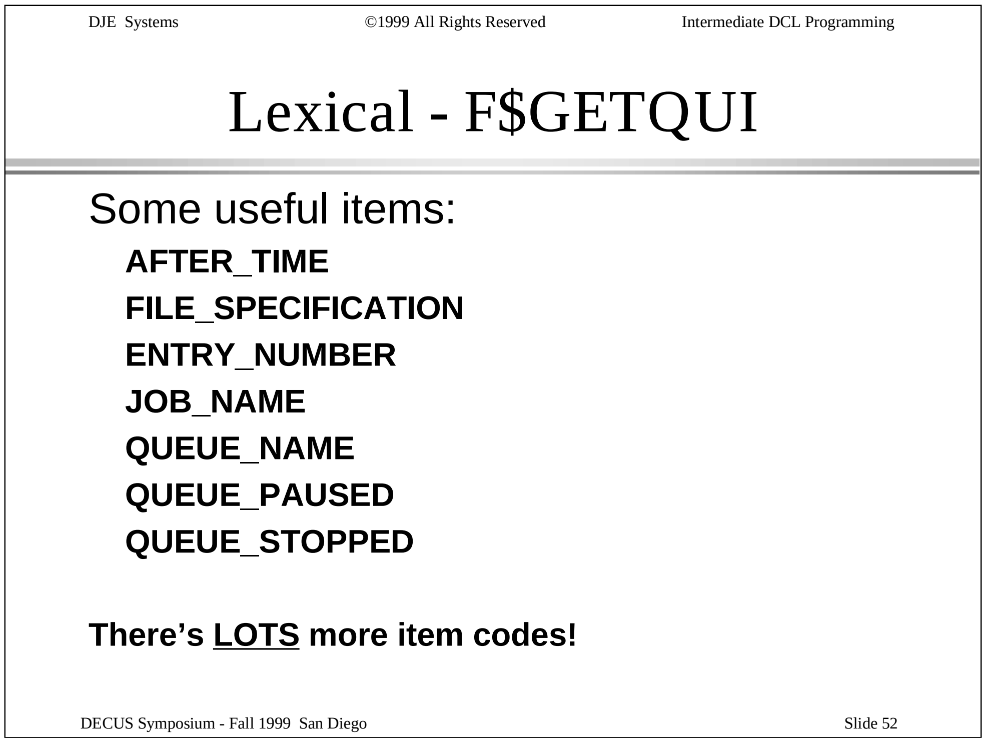
Lexical - F$GETQUI

Functions:

CANCEL_OPERATION
DISPLAY_ENTRY
DISPLAY_FILE
DISPLAY_FORM
DISPLAY_JOB
DISPLAY_MANAGER
DISPLAY_QUEUE
TRANSLATE_QUEUE

This slide lists some of the available function codes. Using these, your procedure can get information about a queue, an entry, a form, a job (where the entry number is not yet known), a queue manager or a logical queue.

The CANCEL_OPERATION function can be used to intentionally destroy the current context. This is useful before creating a new context, to ensure that any previous context has been cleared.



Lexical - F$GETQUI

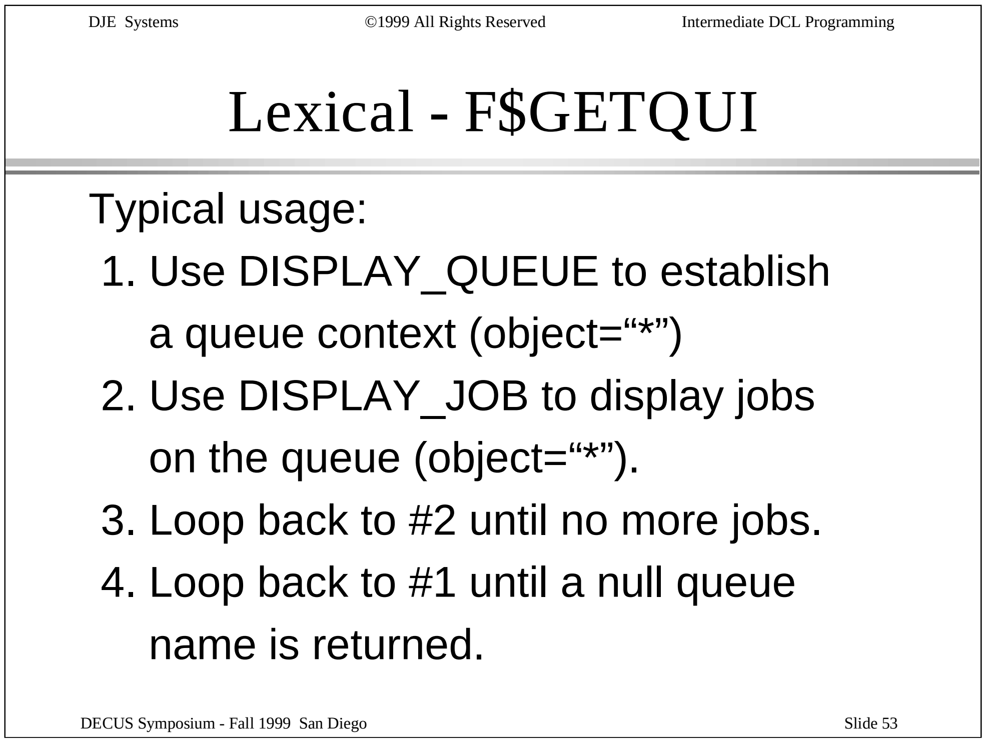
Some useful items:

AFTER_TIME
FILE_SPECIFICATION
ENTRY_NUMBER
JOB_NAME
QUEUE_NAME
QUEUE_PAUSED
QUEUE_STOPPED

There's LOTS more item codes!

This slide lists just a few of the many items that can be returned about a queue, a job, a form, etc.

The DCL Dictionary, Volume 1 is very useful to have at hand when using F$GETQUI(), as all of the available functions, item codes and other arguments are listed.

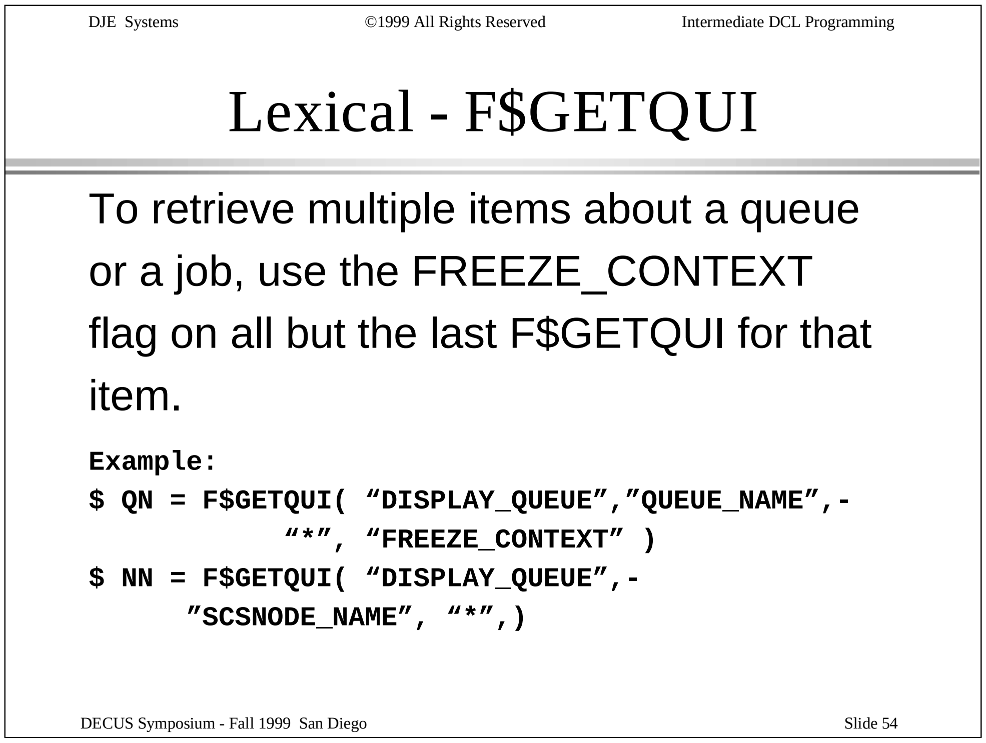


Lexical - F$GETQUI

Typical usage:

1. Use DISPLAY_QUEUE to establish a queue context (object="*")
2. Use DISPLAY_JOB to display jobs on the queue (object="*").
3. Loop back to #2 until no more jobs.
4. Loop back to #1 until a null queue name is returned.

When getting information about all the jobs on a queue, first create the queue context using DISPLAY_QUEUE. Then use DISPLAY_JOB repeatedly to loop through all the jobs in the queue.

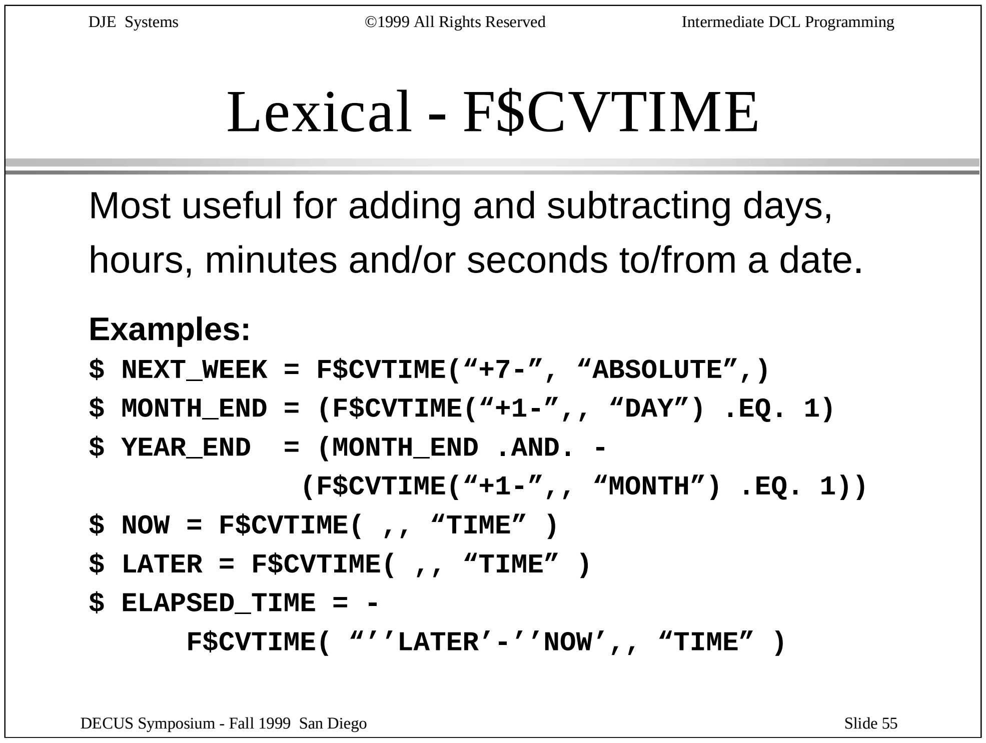


Lexical - F$GETQUI

To retrieve multiple items about a queue or a job, use the FREEZE_CONTEXT flag on all but the last F$GETQUI for that item.

```
Example:
$ QN = F$GETQUI( “DISPLAY_QUEUE”,”QUEUE_NAME”,-
            “*”, “FREEZE_CONTEXT” )
$ NN = F$GETQUI( “DISPLAY_QUEUE”,-
      ”SCSNODE_NAME”, “*”,)
```

When displaying multiple items about a queue, a job, etc., use the FREEZE_CONTEXT flag on all the items but the last. This prevents the current context from being advanced to the next queue, job, etc. until all the needed items about each queue, job, etc. have been retrieved.


DJE Systems ©1999 All Rights Reserved Intermediate DCL Programming

Lexical - F$CVTIME

Most useful for adding and subtracting days, hours, minutes and/or seconds to/from a date.

Examples:

```
$ NEXT_WEEK = F$CVTIME("+7-", "ABSOLUTE",)
$ MONTH_END = (F$CVTIME("+1-",, "DAY") .EQ. 1)
$ YEAR_END  = (MONTH_END .AND. -
             (F$CVTIME("+1-",, "MONTH") .EQ. 1))
$ NOW = F$CVTIME( ,, "TIME" )
$ LATER = F$CVTIME( ,, "TIME" )
$ ELAPSED_TIME = -
      F$CVTIME( "''LATER'-''NOW',, "TIME" )
```

DECUS Symposium - Fall 1999 San Diego Slide 55

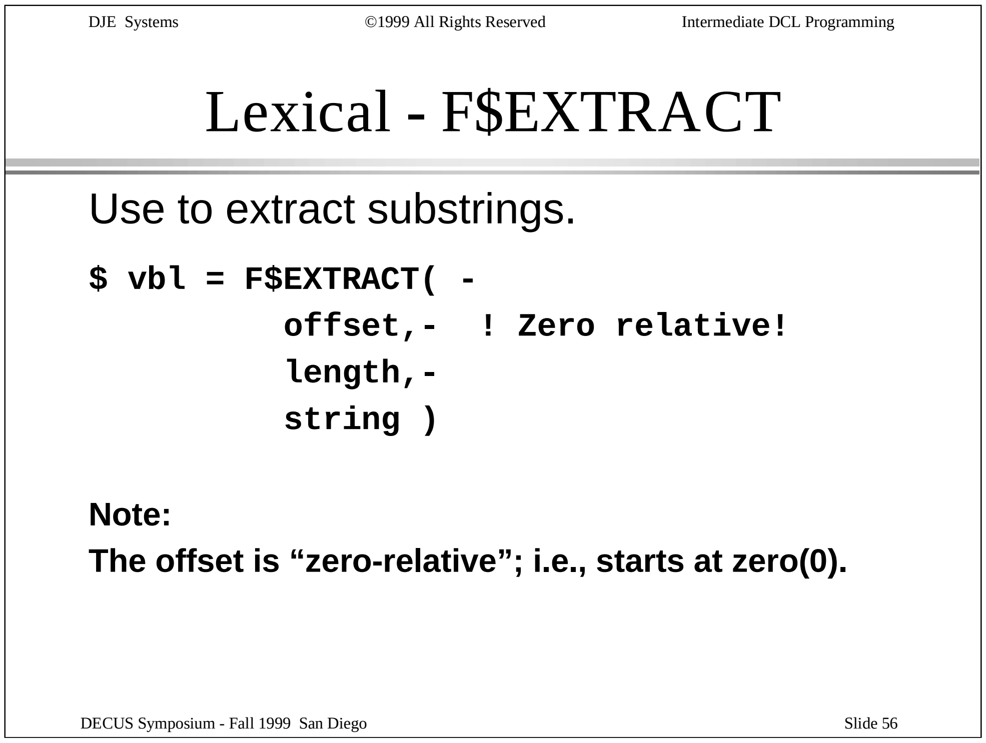
The F$CVTIME() function returns multiple elements of a date/time expression. It also performs conversion from one format to another, and allows for the addition or subtraction of days, hours, minutes, seconds, etc. from a known date/time or the current date time.

The examples show how to:

1. get the date/time for a week from now.

2. see if tomorrow is the first of the month/year.

3. get an elapsed time based on a starting time and an ending time (may not cross midnight more than once!).



Lexical - F$EXTRACT

Use to extract substrings.

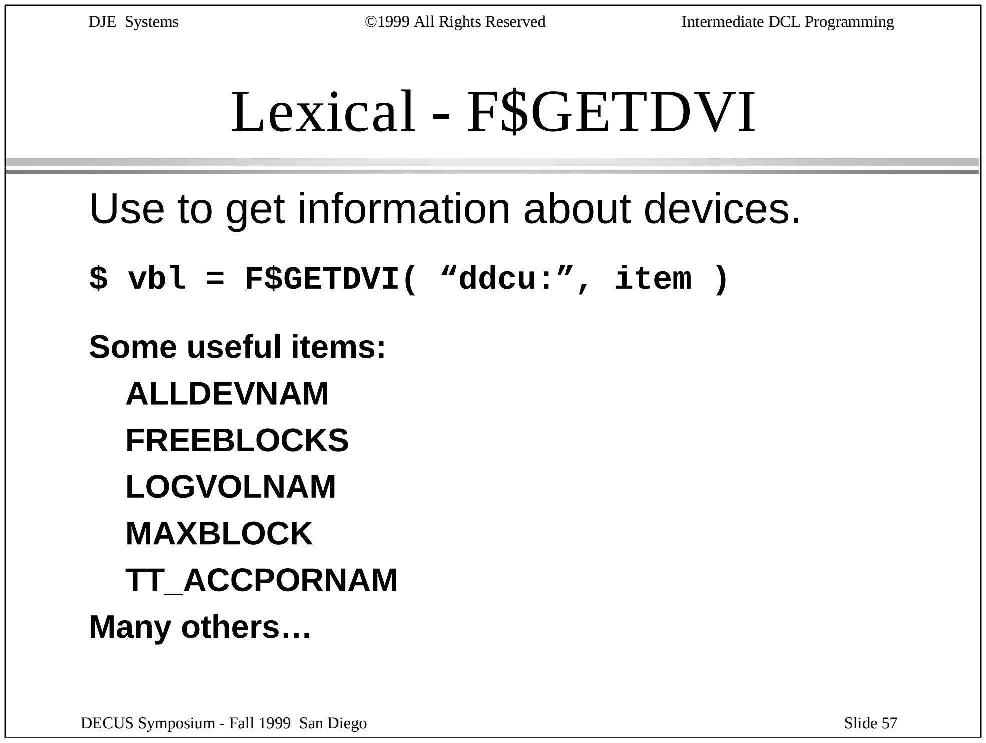
```
$ vbl = F$EXTRACT( -
          offset,-  ! Zero relative!
          length,-
          string )
```

Note:
The offset is “zero-relative”; i.e., starts at zero(0).

Earlier in this session, we looked at substring replacements. The F$EXTRACT() lexical function allows substring extraction.

The original string is left unchanged; only the contents of the requested substring are returned.

Note that the offset is “zero-relative”. The first character in a string has an offset of zero(0). The offset of the last character in a string is equal to the length of the string minus one(1) .


DJE Systems ©1999 All Rights Reserved Intermediate DCL Programming

Lexical - F$GETDVI

Use to get information about devices.

```
$ vbl = F$GETDVI( “ddcu:”, item )
```

Some useful items:

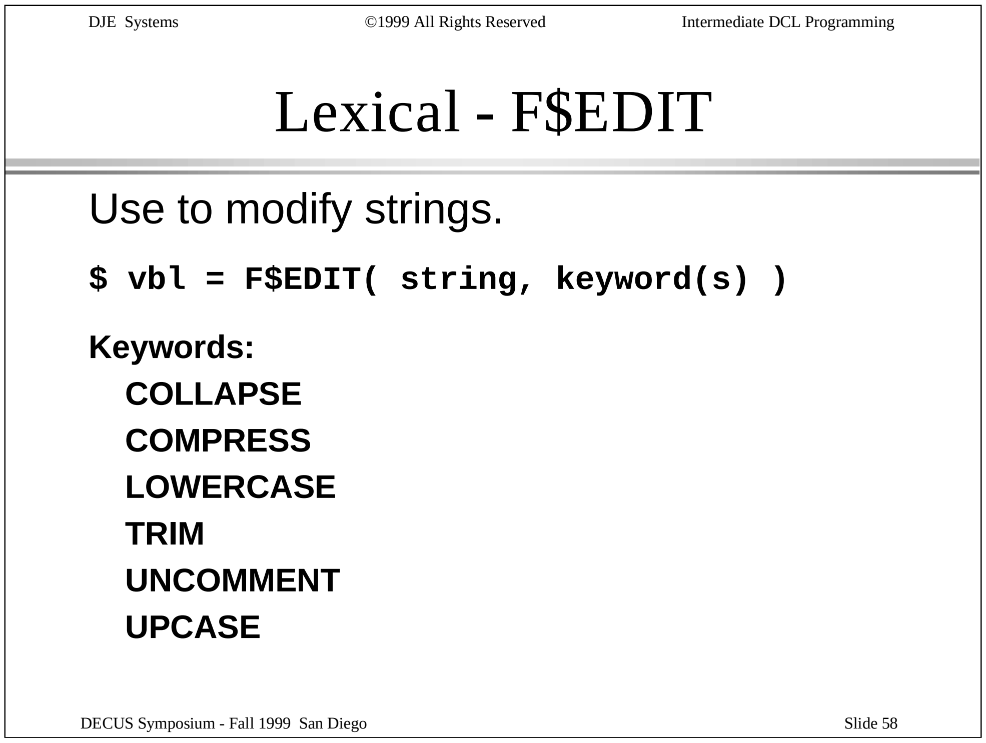
- **ALLDEVNAM**
- **FREEBLOCKS**
- **LOGVOLNAM**
- **MAXBLOCK**
- **TT_ACCPORNAM**

Many others…

DECUS Symposium - Fall 1999 San Diego Slide 57

The F$GETDVI() lexical is used to get information about devices in the system, or to see if a device exists.

Some of the valid items are listed. Many more items are available.



Lexical - F$EDIT

Use to modify strings.

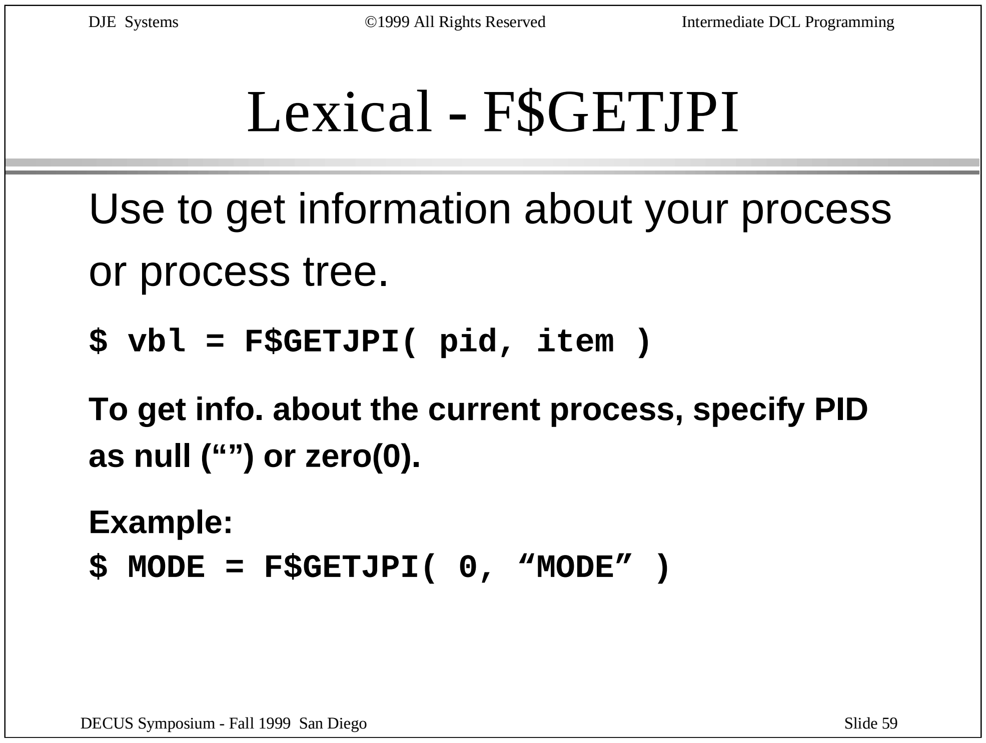
```
$ vbl = F$EDIT( string, keyword(s) )
```

Keywords:

- **COLLAPSE**
- **COMPRESS**
- **LOWERCASE**
- **TRIM**
- **UNCOMMENT**
- **UPCASE**

The F$EDIT() lexical function is used to modify strings.

Strings can be COLLAPSEd (all spaces and TABs are removed), COMPRESSed (spaces and TABs between words are reduced to a single space), converted to upper/lower case, TRIMmed of leading and trailing spaces and TABs, and comments can be stripped off. The comment delimiter is the exclamation point(“!”).



Lexical - F$GETJPI

Use to get information about your process or process tree.

```
$ vbl = F$GETJPI( pid, item )
```

To get info. about the current process, specify PID as null (“”) or zero(0).

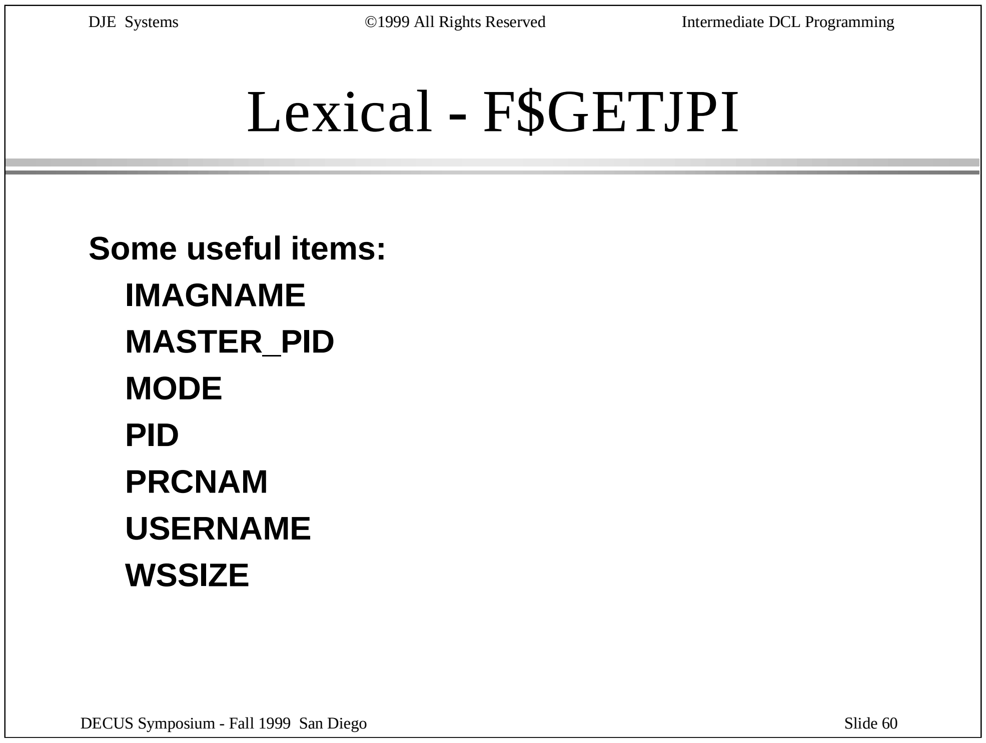
Example:

```
$ MODE = F$GETJPI( 0, “MODE” )
```

The F$GETJPI() lexical function is used to get information about the current job or process.

To get information about the current process, specify the PID as a null string (“”) or a zero(0).

The example shows how to retrieve the mode of the current process. This could also be retrieved using the F$MODE() lexical function.

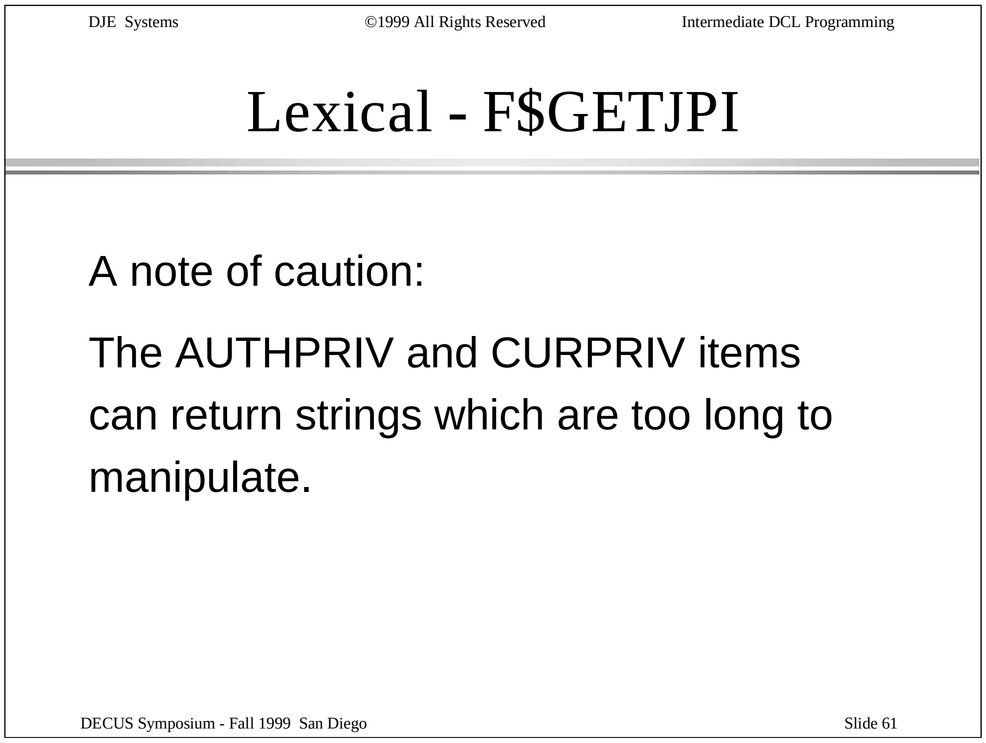


Lexical - F$GETJPI

Some useful items:
- **IMAGNAME**
- **MASTER_PID**
- **MODE**
- **PID**
- **PRCNAM**
- **USERNAME**
- **WSSIZE**

This slide shows just a few of the more useful items than can be retrieved for a process. Many item codes are available.

Refer to the on-line HELP or DCL Dictionary, Volume 1 for a complete listing.



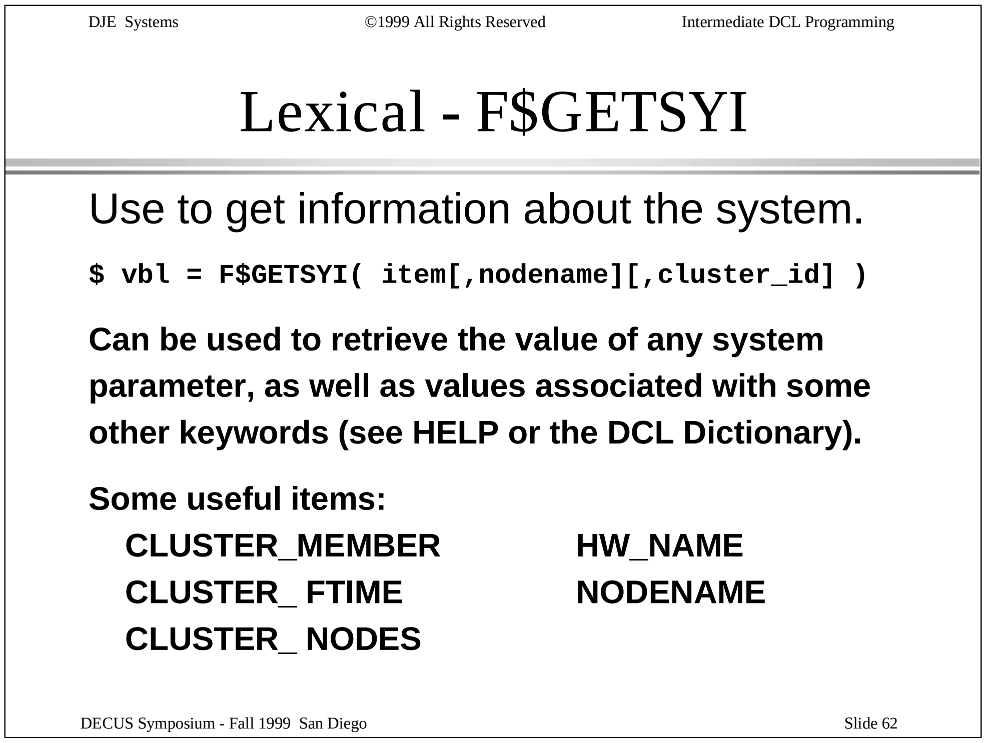
Lexical - F$GETJPI

A note of caution:

The AUTHPRIV and CURPRIV items can return strings which are too long to manipulate.

A note of caution about a couple F$GETJPI() items codes:

The AUTHPRIV and CURPRIV items can return strings which are too long to manipulate. Use them with caution.



Lexical - F$GETSYI

Use to get information about the system.

```
$ vbl = F$GETSYI( item[,nodename][,cluster_id] )
```

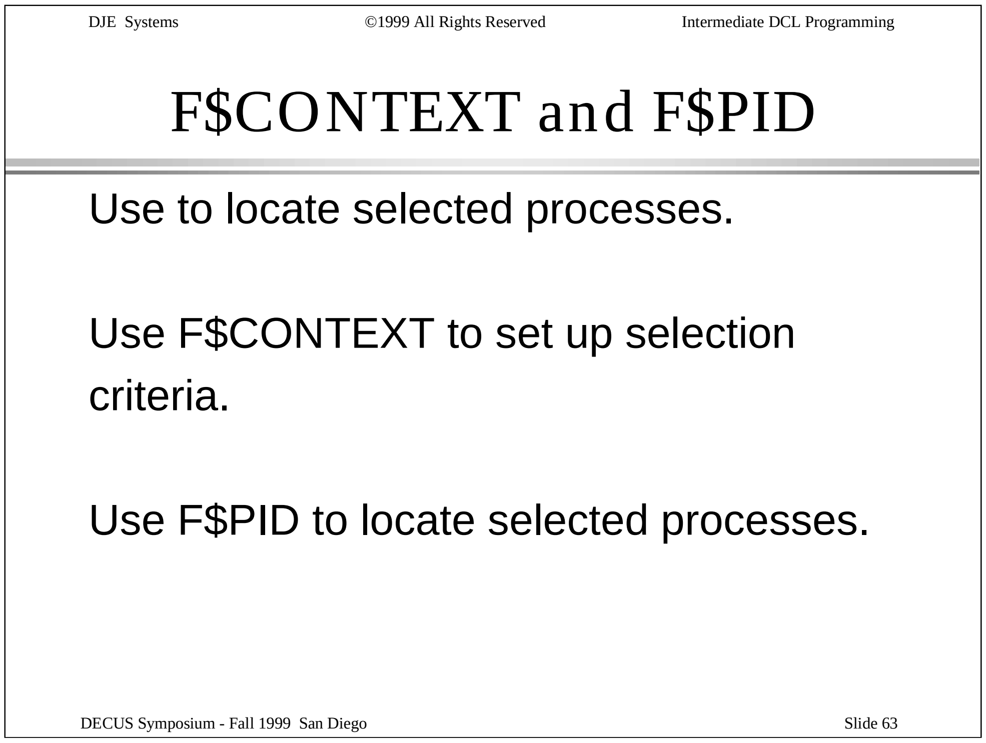
Can be used to retrieve the value of any system parameter, as well as values associated with some other keywords (see HELP or the DCL Dictionary).

Some useful items:

CLUSTER_MEMBER	**HW_NAME**
CLUSTER_ FTIME	**NODENAME**
CLUSTER_ NODES	

The F$GETSYI() lexical function can be used to retrieve many useful items of information about the running OpenVMS system.

Any system parameter value can be retrieved, as well as some information about the cluster and the hardware on which the system is running.



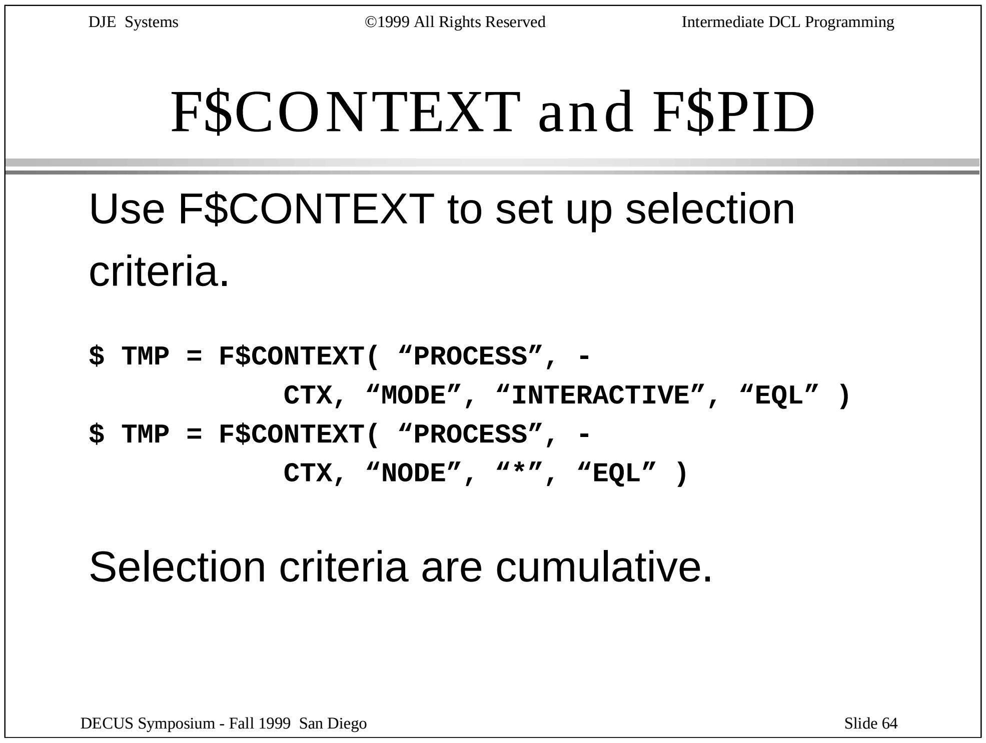
F$CONTEXT and F$PID

Use to locate selected processes.

Use F$CONTEXT to set up selection criteria.

Use F$PID to locate selected processes.

When used together, the F$CONTEXT and F$PID functions provide a means to look up processes on the system by a number of selection criteria.



F$CONTEXT and F$PID

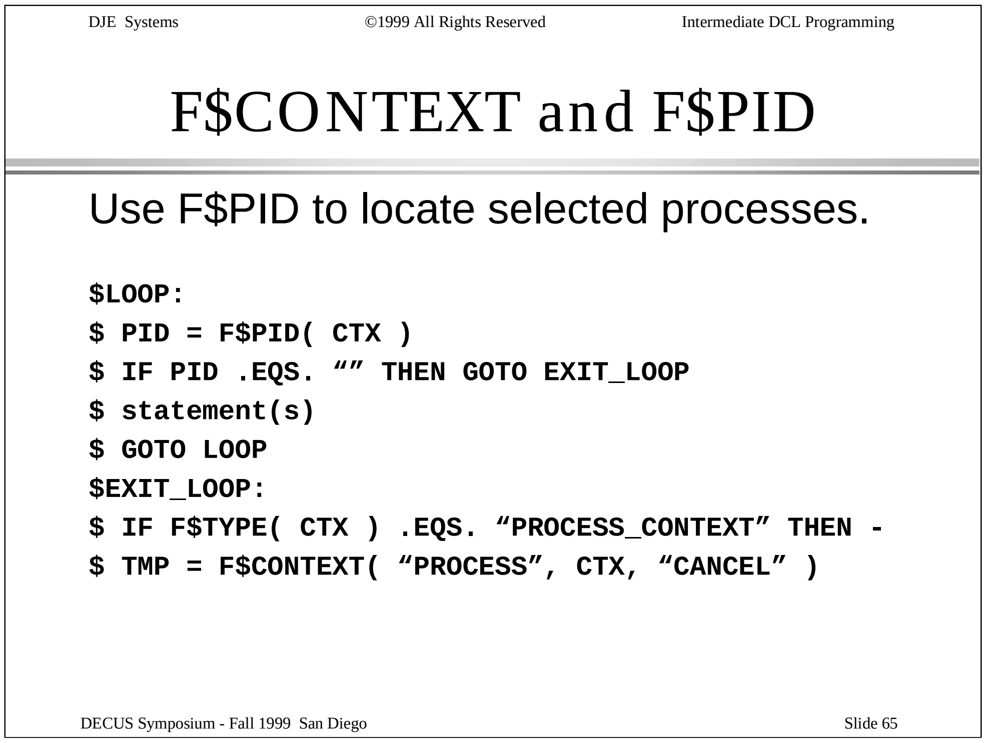
Use F$CONTEXT to set up selection criteria.

```
$ TMP = F$CONTEXT( “PROCESS”, -
            CTX, “MODE”, “INTERACTIVE”, “EQL” )
$ TMP = F$CONTEXT( “PROCESS”, -
            CTX, “NODE”, “*”, “EQL” )
```

Selection criteria are cumulative.

Use the F$CONTEXT function to set up your selection criteria. This allows a programmatic way of locating processes by name, by mode, by UIC, etc. without the need to use intermediate files or parse the output of a DCL command such as SHOW SYSTEM.

Multiple selection criteria can be specified by issuing multiple invocations of F$CONTEXT. The context constructed this way can be used with the F$PID function to return the PIDs of all processes matching the selection criteria specified.



F$CONTEXT and F$PID

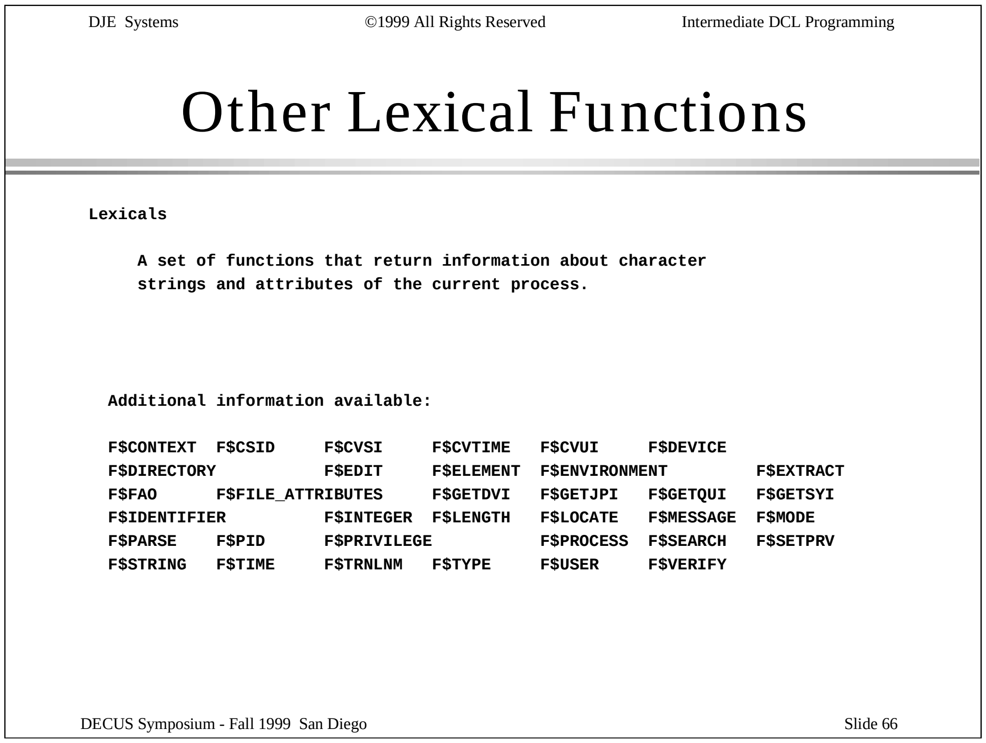
Use F$PID to locate selected processes.

```
$LOOP:
$ PID = F$PID( CTX )
$ IF PID .EQS. “” THEN GOTO EXIT_LOOP
$ statement(s)
$ GOTO LOOP
$EXIT_LOOP:
$ IF F$TYPE( CTX ) .EQS. “PROCESS_CONTEXT” THEN -
$ TMP = F$CONTEXT( “PROCESS”, CTX, “CANCEL” )
```

Once you have set up your selection context, pass that context symbol to F$PID() and invoke it in a loop until a null PID string if returned.

By default, F$PID() will return the PIDs of all processes in the system (local to a cluster node).

To locate all processes in the cluster matching the other selection criteria, include an F$CONTEXT invocation specifying an item code of NODENAME, a matching string of “*” and a match criterion of “EQL”.

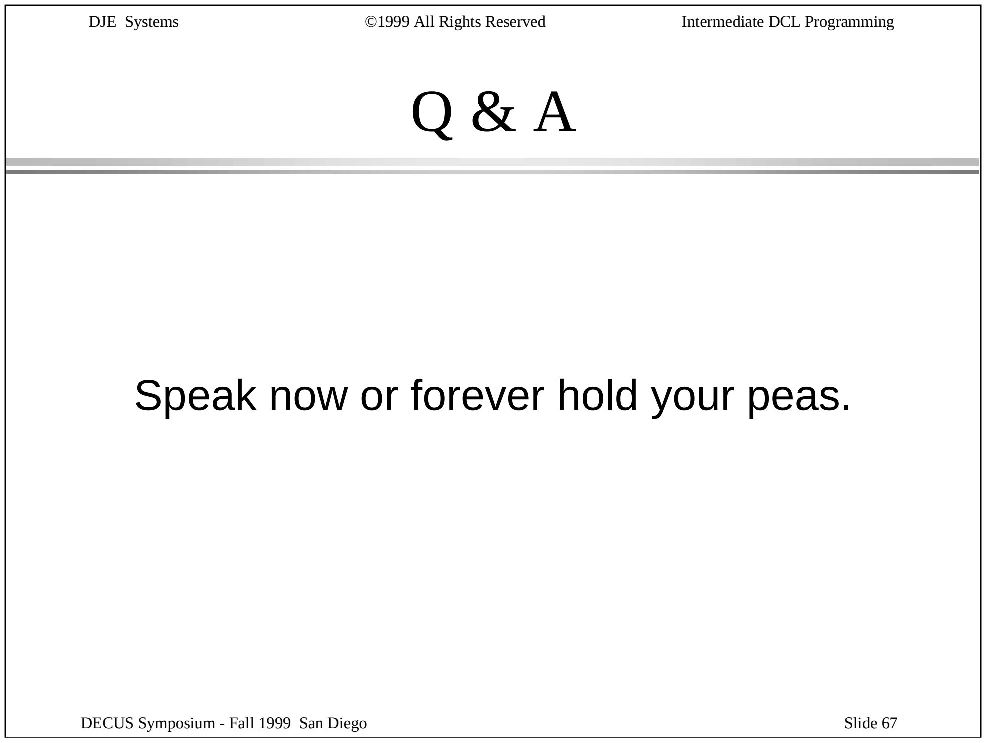


Other Lexical Functions

```
Lexicals

     A set of functions that return information about character
     strings and attributes of the current process.



   Additional information available:

   F$CONTEXT  F$CSID     F$CVSI     F$CVTIME   F$CVUI     F$DEVICE
   F$DIRECTORY           F$EDIT     F$ELEMENT  F$ENVIRONMENT         F$EXTRACT
   F$FAO      F$FILE_ATTRIBUTES     F$GETDVI   F$GETJPI   F$GETQUI   F$GETSYI
   F$IDENTIFIER          F$INTEGER  F$LENGTH   F$LOCATE   F$MESSAGE  F$MODE
   F$PARSE    F$PID      F$PRIVILEGE           F$PROCESS  F$SEARCH   F$SETPRV
   F$STRING   F$TIME     F$TRNLNM   F$TYPE     F$USER     F$VERIFY
```

Other lexical functions exist as well as those discussed in this presentation. The slide shows the output of “HELP Lexical” and shows all of the available lexical function names.

Refer to the DCL Dictionary, Volume 1 for complete information on the available lexical functions.


DJE Systems ©1999 All Rights Reserved Intermediate DCL Programming

Q & A

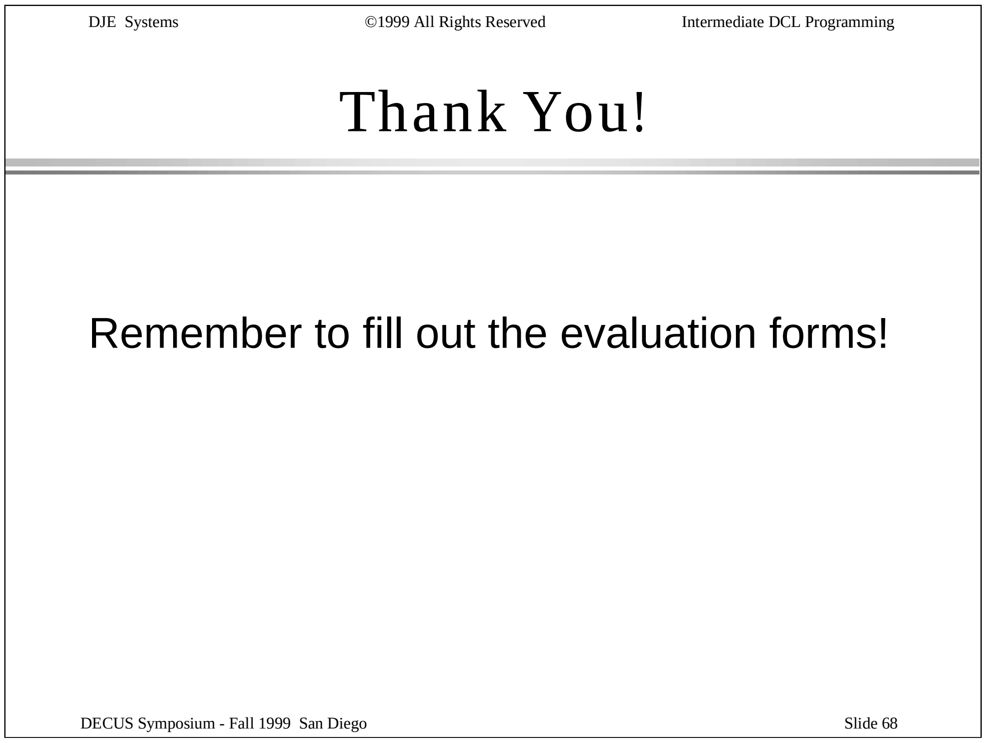
Speak now or forever hold your peas.

DECUS Symposium - Fall 1999 San Diego Slide 67

Please step to the microphone to ask your questions.

Please limit yourself to one question. If you have another, step to the end of the line and await another turn.

Practical examples and problem fixes are always welcome, of course!



DJE Systems ©1999 All Rights Reserved Intermediate DCL Programming

Thank You!

Remember to fill out the evaluation forms!

DECUS Symposium - Fall 1999 San Diego Slide 68

If evaluation forms are available, please remember to fill them out and return them to the presenter.