∆FIX command

Description

∆FIX__ is a preprocessor for *Dyalog APL* files following the formal specifications of the `2∘⎕FIX` command. Normally identified as dot-Dyalog (__.dyalog) files, these files contain one or more

- namespace-like objects (namespaces, classes, and interfaces),
- traditional functions (marked with `∇...∇`, unless the sole object in the file), and
- direct fns (*dfns*).

result ← [outSpec [comSpec [DEBUG]]] ∆FIX fileName

Description:
Takes an input file in 2 ⎕FIX format, preprocesses the file, then 2 ⎕FIX's it, and
returns the objects found or ⎕FIX error messages.
Like, ⎕FIX, accepts either a mix of namespace-like objects (namespaces, classes, interfaces)
and functions (marked with ∇) or a single function (whose first line must be its header,
with a ∇-prefix optional).

fileName: the full file identifier; if no type is indicated, .dyalog is appended.

outSpec: ∊0 (default), 1, 2. Indicates the format of the return value*.

On success, rc (return code) is 0.

0 - returns*: rc names -- names: the list of objects created by a ⎕FIX.\ 1 - returns*: rc names code -- code: output (vec of strings) from the preprocessor.\
2 - returns*: rc code -- rc: 0 on success\

- If an error occurs, returns:<br /> signalNum signalMsg -- signal…: APL ⎕SIGNAL number and message string

comSpec: ∊0 (default), 1, 2. Indicates how to handle preprocessor statements in output.

0: Keep all preprocessor statements, identified as comments with ⍝🅿 (path taken), ⍝ (not taken)\
1: Omit (⍝) paths not taken\
2: Omit also (⍝🅿) paths taken (leave other user comments)\

DEBUG: 0: not debug mode (default).\

1: debug mode. ⎕SIGNALs will not be trapped.

Preprocessor Directives

Directives are of the form `::DIRECTIVE name ← value` or `::DIRECTIVE (cond) action`.
Directives are always the first item on any line of input (leading spaces are ignored).

Special commands are of the form:
#COMMAND{argument}\
Or
name..COMMAND

Command Descriptions

::DEF[INE] name ← string

::UNDEF name

::LET name ← value\
::EVAL name ← value

::IFDEF name\
…\
::ELSE \
…\
::ENDIF[DEF]\

::IF cond\
::ELSEIF cond\
::ELSE\
::ENDIF

::INCLUDE fileID\
Include file right here, replacing the include statement, and preprocess it. fileID: (currently) a file identifier; if no filetype is indicated, .dyalog is assumed.\
::CINCLUDE fileID\
Include the file right here, as for ::INCLUDE, but only if not already included via ::INCLUDE or ::CINCLUDE. fileID: see ::INCLUDE.

::COND cond statement

::MESSAGE any text

::MSG any text

::ERROR [errcode] any text

#ENV{name}\
returns the value of the environment variable “name”.

#SH[ELL]{shell_cmd}
Returns the value of the shell command ⎕SH ‘shell_cmd’.

#EXEC{apl_cmd}
Returns the value of the APL command ⍎‘apl_cmd’.

name..CMD

Items of this form first undergo macro substitution (if applicable), before being quoted.
Thus, these commands are a handy way to check whether an object is defined or not, even if expected to be altered via macro substitution.\
Note: Ordinary quoted strings are ignored during macro substitution.

name..DEF becomes (0≠⎕NC ‘name’)\
name1.name2.name3..DEF becomes (0≠⎕NC ‘name1.name2.name3’)\

name..UNDEF becomes (0=⎕NC ‘name’)\
name1.name2.name3..UNDEF becomes (0=⎕NC ‘name1.name2.name3’)

APL STRINGS

APL strings in single quotes are handled as in APL. Strings may appear in double quotes (“…”), may contain unduplicated single quotes, and may extend over multiple lines. Double quoted strings are converted to single-quoted strings, after:

- Doubling internal single quotes
- Processing doubled internal double quotes.
- Converting newlines to ⎕UCS 10, in this way (↵ used to show newline):

“String1 ↵string2” → (‘String’,(⎕UCS 10),’string2’)
Blanks at the beginning of each continuation line are removed (the symbol · shows where the leading blanks are).

BEFORE:\
“This is line 1. ↵ \

·········This is line 2.”

AFTER:\
(‘This is line 1.’,(⎕UCS 10),’This is line 2.’)

Simple Macros

All names defined by ::DEF or ::LET (or synonym, ::EVAL) are replaced anywhere in APL text outside of quoted strings. If those objects contain non-text, they are converted to text; if they appear on multiple lines, it must make sense in the APL context.

Continuation lines in APL code

You may continue any APL line by placing two or more dots .. before any comments on that line.
In some cases, where the preprocessor handles arguments in parentheses or braces, those arguments may span multiple lines as left-hand parentheses or braces are matched by their right-hand counterparts. These will be documented in a later edition of this document.

Bugs

In this version, trailing (right-hand) comments are omitted from the preprocessor output. Lines containing nothing but comments (possibly with leading blanks) are maintained as is. This may cause problems for those using comments as “here text” or otherwise manipulating the comments in the (preprocessed) source file. Since most such uses depend on full comment lines, this should in most cases not be a problem.