```python
#!/usr/bin/env python
# tconfpy.py
# Copyright (c) 2003-2005 TundraWare Inc.  All Rights Reserved.
# For Updates See:  http://www.tundraware.com/Software/tconfpy

# Program Information

PROGNAME = "tconfpy"
RCSID = "$Id: tconfpy.py,v 2.112 2005/01/20 08:44:40 tundra Exp $"
VERSION = RCSID.split()[2]

# Copyright Information

CPRT         = "(c)"
DATE         = "2003-2005"
OWNER        = "TundraWare Inc."
RIGHTS       = "All Rights Reserved"
COPYRIGHT    = "Copyright %s %s %s,  %s." % (CPRT, DATE, OWNER, RIGHTS)
PROGINFO     = PROGNAME + " " + VERSION + " "
BANNER       = "%s\n%s\n%s\n" % (PROGINFO, COPYRIGHT, RCSID)


#----------------------------------------------------------#
#            Variables User Might Change                   #
#----------------------------------------------------------#


#------------------- Nothing Below Here Should Need Changing -----------------#


#----------------------------------------------------------#
#                         Imports                          #
#----------------------------------------------------------#

import copy
import os
import platform
import re
from sys import platform as sysplat


#----------------------------------------------------------#
#                 Aliases & Redefinitions                  #
#----------------------------------------------------------#



#----------------------------------------------------------#
#                Constants & Literals                      #
#----------------------------------------------------------#



###########
# Constants
###########

# Formatting Constants

ATEOF       = "EOF"                    # Use as line number when at EOF
FILENUM     = "[File: %s Line: %s]   " # Display filename and linenum
INMEMORY    = "In-Memory Configuration" # Configuration is in-memory
MSGPROMPT   = "%s>"
PTR         = "   --->   "             # Textual pointer for debug output
STARTUP     = "STARTUP"                # Indicates message before any lines processed

# Special And Reserved Symbols

HASH        = r'#'
COMMENT     = HASH         # Comment introducer character
DELIML      = r'['         # Left delimiter for vbl reference
DELIMR      = r']'         # Right delimiter for vbl reference
DOLLAR      = r'$'
ENVIRO      = DOLLAR       # Used to note environment variable
EQUAL       = r'='         # Used in vbl definition
EQUIV       = r"=="        # Used in conditional tests
NAMESPACE   = "NAMESPACE"  # Key used to get current namespace
```

```
PERIOD      = r'.'
NSSEP       = PERIOD       # Namespace separator character
NOTEQUIV    = r"!="        # Used in conditional tests

# Control and conditional symbols

CONDINTRO   = PERIOD       # Conditional introducer token
INCLUDE     = CONDINTRO + "include"
IF          = CONDINTRO + "if"
IFALL       = IF + "all"
IFANY       = IF + "any"
IFNONE      = IF + "none"
ELSE        = CONDINTRO + "else"
ENDIF       = CONDINTRO + "endif"
LITERAL     = CONDINTRO + "literal"
ENDLITERAL  = CONDINTRO + "endliteral"

# Pre-Defined System Symbols

PREDEFINTRO   = PERIOD

Predefined    = {"DELIML"                       : DELIML,
                 "DELIMR"                       : DELIMR,
                 "DOLLAR"                       : DOLLAR,
                 "ELSE"                         : ELSE,
                 "ENDIF"                        : ENDIF,
                 "ENDLITERAL"                   : ENDLITERAL,
                 "EQUAL"                        : EQUAL,
                 "EQUIV"                        : EQUIV,
                 "HASH"                         : HASH,
                 "IF"                           : IF,
                 "IFALL"                        : IFALL,
                 "IFANY"                        : IFANY,
                 "IFNONE"                       : IFNONE,
                 "INCLUDE"                      : INCLUDE,
                 "LITERAL"                      : LITERAL,
                 "NOTEQUIV"                     : NOTEQUIV,
                 "PERIOD"                       : PERIOD,
                 PREDEFINTRO + "MACHINENAME"   : platform.node(),
                 PREDEFINTRO + "OSDETAILS"     : platform.platform(),
                 PREDEFINTRO + "OSNAME"        : platform.system(),
                 PREDEFINTRO + "OSRELEASE"     : platform.release(),
                 PREDEFINTRO + "OSTYPE"        : sysplat,
                 PREDEFINTRO + "PLATFORM"      : os.name,
                 PREDEFINTRO + "PYTHONVERSION" : platform.python_version()
                }


# Symbols Illegal Anywhere In A Variable/Namespace Name

IllegalChars = (DELIML, DELIMR, COMMENT)


# Regular Expressions

reVARWHITE    = r".*\s+.*"                          # Look for embedded whitespace
reVARREF      = r"\%s.+?\%s" % (DELIML, DELIMR)    # Variable reference

VarWhite      = re.compile(reVARWHITE)
VarRef        = re.compile(reVARREF)



###########
# Literals
###########

# String Representations Of Booleans

Booleans = {"1"    : True, "0"     : False,
            "True" : True, "False" : False,
            "Yes"  : True, "No"    : False,
            "On"   : True, "Off"   : False
            }

sFALSE   = "False"
sTRUE    = "True"
```

```python



#----------------------------------------------------------#
#          Global Variables & Data Structures              #
#----------------------------------------------------------#


##########
# Symbol Table Related Classes
##########

class SymbolTable(object):

    def __init__(self):

        # Returned by the parser

        self.Symbols       = {}
        self.DebugMsgs     = []
        self.ErrMsgs       = []
        self.WarnMsgs      = []
        self.LiteralLines  = []
        self.TotalLines    = 0
        self.Visited       = []

        # Anything below here is used interally and not returned

        self.Templates     = Template()
        self.ALLOWNEWVAR   = True
        self.TEMPONLY      = False
        self.LITERALVARS   = False
        self.INLITERAL     = False
        self.DEBUG         = False
        self.CondStack     = [["", True],]  # Always has one entry as a sentinel

# End of class 'SymbolTable'

class Template(object):

    def __init__(self):
        self.Symbols = {}

# End of class 'Template'


# The descriptor is an object with the following attributes
#
# Value, Writeable, Type, Default, LegalVals = [list of legal vals], Min, Max]

# Object to hold full description and options for a given variable

class VarDescriptor(object):

    # Default variable type is a writeable string with no constraints
    def __init__(self):
        self.Value     = ""
        self.Writeable = True
        self.Type      = TYPE_STRING
        self.Default   = ""
        self.LegalVals = []
        self.Min       = None
        self.Max       = None

# End of class 'VarDescriptor'

# Legal Variable Types

TYPE_BOOL    = type(True)
TYPE_COMPLEX = type(1-1j)
TYPE_FLOAT   = type(3.14)
TYPE_INT     = type(1)
TYPE_STRING  = type('s')


##########
# Error, Warning, And Debug Message Strings Stored In A Global Dictionary
```

```
##########

Messages = {}


#----------------------------------------------------------#
#             Prompts, & Application Strings               #
#----------------------------------------------------------#




##########
# Debug Literals And Messages
##########

# Debug Literals

dDEBUG      = "DEBUG"
dLINEIGNORE = "Line Ignored"

dBLANKLINE  = "Parsed To Blank Line.  %s" % dLINEIGNORE
dNOTINCLUDE = "Current Conditional Block False.  %s" % dLINEIGNORE

# Debug Messages

Messages["dLINEIGNORE"] = FILENUM + "'%s'" + PTR + "%s"
Messages["dNAMESPACE"]  = FILENUM + "Setting Current Namespace To: '%s'"
Messages["dNUMLINES"]   = "Processing File '%s' Resulted In %d Total Lines Parsed"
Messages["dPARSEDLINE"] = FILENUM + "'%s'" + PTR + "'%s'"
Messages["dREGEXMATCH"] = FILENUM + "Value '%s' Matched Regex '%s' For Variable '%s'"
Messages["dVAREXISTS"]  = FILENUM + "Checking To See If Variable '%s' Exists"
Messages["dVARREF"]     = FILENUM + "Variable Dereference: '%s'" + PTR + "'%s'"
Messages["dVISITED"]    = "Configuration Files Processed: '%s'"


###########
# Error Literals And Messages
###########

# Error Literals

eERROR        = "ERROR"

eBADDEFAULT   = "Type Of Default Value Does Not Agree With Type Declared"
eBADLEGALVAL  = "Type Of One Or More LegalVals Does Not Agree With Type Declared"
eBADMINMAX    = "Type Of Min Or Max Value Not Appropriate For"
eBADTEMPLNAME = "Template Name Not Canonical"
eBADVALTYP    = "Initial Value Has Wrong Type"
eBADVARREF    = "Attempt To Get Value Of Non-Existent Variable"
eIFBAD        = "'%s' Or '%s' Missing" % (EQUIV, NOTEQUIV)
eINITSYMTBL   = "InitialSymTable"
eLEGALVALLIST = "The LegalVal Attribute Is Wrong Type (Must Be A List)"
eNOTDESCRIPT  = "Invalid Descriptor Type"
eNOVARS       = "This Conditional Requires At Least One Variable Name To Test"
eSTARTUP      = "<Program Starting>"
eTEMPLATES    = "Templates"

# Error Messages

Messages["eAPIPARAMBAD"]      = "API Error: %s For Variable '%s' In '%s' Passed To The API"
Messages["eBADCOND"]          = FILENUM + "Bad '%s' Directive. %s"
Messages["eBADREGEX"]         = FILENUM + "Bad Regular Expression, '%s', In Legal Values List For Variable '%s'"
Messages["eBADSYNTAX"]        = FILENUM + "Syntax Error.  Statement Not In Known Form"
Messages["eCONFIGTYPE"]       = "Don't Know How To Process Configurations Of Type '%s'"
Messages["eCONFOPEN"]         = FILENUM + "Cannot Open The File '%s'"
Messages["eELSENOIF"]         = FILENUM + "'%s' Without Preceding '%s' Form" % (ELSE, IF)
Messages["eENDIFEXTRA"]       = FILENUM + "'" + ENDIF + "' Without Matching Condition"
Messages["eENDIFMISS"]        = FILENUM + "Missing %d" + " '%s' " % ENDIF + " Statement(s)"
Messages["eEQUIVEXTRA"]       = FILENUM + "Only a single '%s' Or '%s' Operator Permitted" % (EQUIV, NOTEQUIV)
Messages["eIFEXTRATXT"]       = FILENUM + "Extra Text On Line.  '%s' Only Accepts Variable References As Arguments"
Messages["eNOTLEGALVAL"]      = FILENUM + "'%s' Not Found In List Of Legal Values For Variable '%s'"
Messages["eNOTSTRING"]        = FILENUM + "%s Is Not A String Type - Ignoring"
Messages["eCIRCULAR"]         = FILENUM + "Circular .includes Forbidden"
```

```
Messages["eSTRINGLONG"]       = FILENUM + "'%s' Too Long For Variable '%s'.  Maximum Length Is %s"
Messages["eSTRINGSHORT"]      = FILENUM + "'%s' Too Short For Variable '%s'.  Minimum Length is %s"
Messages["eSYMBADCHAR"]       = FILENUM + "Symbol '%s' Contains Illegal Character '%s'"
Messages["eSYMBADSTART"]      = FILENUM + "Symbol '%s' Begins With An Illegal Character"
Messages["eSYMNAMESPC"]       = FILENUM + "Symbol Names May Not Contain Whitespace"
Messages["eSYMNONAME"]        = FILENUM + "Symbol Name Evaluates To Null String.  Not Permitted"
Messages["eTEMPONLY"]         = FILENUM + "Cannot Create New Variable.  No Template For '%s'"
Messages["eTYPEBAD"]          = FILENUM + "Type Mismatch. '%s' Must Be Assigned Values Of Type %s Only"
Messages["eVALLARGE"]         = FILENUM + "%s Too Large For Variable '%s'.  Maximum Allowed is %s"
Messages["eVALSMALL"]         = FILENUM + "%s Too Small For Variable '%s'.  Minimum Allowed is %s"
Messages["eVARNEW"]           = FILENUM + "New Variable Creation Not Permitted"
Messages["eVARREADONLY"]      = FILENUM + "Variable '%s' Is Read-Only.  Cannot Change Its Value"
Messages["eVARREFNEST"]       = FILENUM + "Nested Variable References Are Not Permitted"
Messages["eVARUNDEF"]         = FILENUM + "Attempt To Reference Undefined Variable '%s'"


###########
# Warning Literals And Messages
###########

# Warning Literals

wWARNING     = "WARNING"

# Warning Messages

Messages["wENDLITEXTRA"] = FILENUM + "'%s' Statement Without Preceding '%s'. Statement Ignored" % (ENDLITERAL, LITERAL)
Messages["wENDLITMISS"]  = FILENUM + "Missing '%s' Statement.  All lines treated literally to end-of-file" % ENDLITERAL
Messages["wLITEXTRA"]    = FILENUM + "Already In A Literal Block. '%s' Statement Ignored" % LITERAL


# Determine Length Of Longest Message Name

MAXMSG = 0
for msg in Messages:
    l = len(msg)
    if l > MAXMSG:
        MAXMSG = l


#--------------------------- Code Begins Here ---------------------------------#


#----------------------------------------------------------#
#               Utility Function Definitions               #
#----------------------------------------------------------#


##########
# Create A Debug Message
##########

def DebugMsg(dmsg, args):

    global SymTable

    SymTable.DebugMsgs.append(mkmsg(Messages[dmsg] % args, dmsg))

# End of 'DebugMsg()'


##########
# Create An Error Message
##########

def ErrorMsg(error, args):

    global SymTable

    SymTable.ErrMsgs.append(mkmsg(Messages[error] % args + "!", error))

# End of 'ErrorMsg()'


```

```
##########
# Create A Warning Message
##########

def WarningMsg(warning, args):

    global SymTable

    SymTable.WarnMsgs.append(mkmsg(Messages[warning] % args + "!", warning))

# End of 'WarningMsg()'


##########
# Construct A Standard Application Message String
##########

def mkmsg(msg, msgtype):

    pad = " " * (MAXMSG - len(msgtype) + 2)

    return "%s%s%s%s" % (mkmsg.proginfo, MSGPROMPT % msgtype, pad, msg)


# End of 'mkmsg()'


#----------------------------------------------------------#
#                  Public API To Module                    #
#----------------------------------------------------------#

def ParseConfig(configuration,
                CallingProgram=PROGINFO,
                InitialSymTable=SymbolTable(),
                AllowNewVars=True,
                Templates=Template(),
                TemplatesOnly=False,
                LiteralVars=False,
                ReturnPredefs=True,
                Debug=False):

    global SymTable

    # Create a new symbol table
    SymTable = SymbolTable()

    # Set the name of the calling program for output messages
    mkmsg.proginfo = CallingProgram

    # Validate the passed symbol table
    symtblok = ValidateSymbolTable(InitialSymTable.Symbols, eINITSYMTBL)

    # Validate the passed template table
    templok = ValidateSymbolTable(Templates.Symbols, eTEMPLATES, istemplate=True)

    # Both of these must be OK or we're done

    if not (symtblok and templok):
        return SymTable

    # Everything was fine

    else:
        # Load the symbol table
        SymTable.Symbols = copy.copy(InitialSymTable.Symbols)

        # Load the template table
        SymTable.Templates.Symbols = copy.copy(Templates.Symbols)


    # Initialize the control structures

    SymTable.ALLOWNEWVAR   = AllowNewVars
    SymTable.TEMPONLY      = TemplatesOnly
    SymTable.LITERALVARS   = LiteralVars
    SymTable.DEBUG         = Debug

```

```
    # Load the predefined symbols

    for var in Predefined.keys():

        d = VarDescriptor()
        d.Value = Predefined[var]
        d.Writeable = False
        SymTable.Symbols[var] = d


    # Symbol Table passed to API was OK, so keep going

    # Make sure the symbol table has a valid namespace

    if NAMESPACE not in SymTable.Symbols:
        SymTable.Symbols[NAMESPACE] = VarDescriptor()
        SymTable.Symbols[NAMESPACE].Value = ""

    # Otherwise, ensure that the initial namespace is properly formed.
    # If not, revert to root namespace.

    elif not ValidateSymbolName(SymTable.Symbols[NAMESPACE].Value, STARTUP, STARTUP, AllowEmpty=True):
        SymTable.Symbols[NAMESPACE].Value = ""

    # Report namespace to debug output

    if SymTable.DEBUG:
        DebugMsg("dNAMESPACE", (STARTUP, STARTUP, SymTable.Symbols[NAMESPACE].Value))

    # We now can parse the configuration, either in a file or in-memory.

    cfgtype = type(configuration)

    # It's a file if the passed parameter is a string - assumed to be the filename

    if cfgtype == type('x'):
        ParseFile(configuration, eSTARTUP, 0)
        configname = configuration                # Name of the configuration

    # It's an in-memory configuration if the passed parameter is a list

    elif cfgtype == type([]):
        configname = INMEMORY
        ParseInMemory(configuration, configname)
        pass

    # Anything else is illegal

    else:
        ErrorMsg("eCONFIGTYPE", str(cfgtype).split()[-1][1:-2])

    # Make sure we had all condition blocks balanced with matching '.endif'

    finalcond = len(SymTable.CondStack)
    if finalcond != 1:
        ErrorMsg("eENDIFMISS", (configname, ATEOF, finalcond-1))

    # Make sure we ended any literal processing properly
    if SymTable.INLITERAL:
        WarningMsg("wENDLITMISS", (configname, ATEOF))

    # Return the parsing results

    if SymTable.DEBUG:
        DebugMsg("dNUMLINES", (configname, SymTable.TotalLines))
        DebugMsg("dVISITED", (SymTable.Visited))

    # Strip out Prefefined Variables if user does not want them

    if not ReturnPredefs:

        del SymTable.Symbols[NAMESPACE]
        for sym in Predefined:
            del SymTable.Symbols[sym]
```

```
    # Now populate a return symbol table containing only the stuff we
    # actually want

    retval = SymbolTable()

    retval.Symbols      = copy.copy(SymTable.Symbols)
    retval.DebugMsgs    = copy.copy(SymTable.DebugMsgs)
    retval.ErrMsgs      = copy.copy(SymTable.ErrMsgs)
    retval.WarnMsgs     = copy.copy(SymTable.WarnMsgs)
    retval.LiteralLines = copy.copy(SymTable.LiteralLines)
    retval.TotalLines   = copy.copy(SymTable.TotalLines)

    return retval


# End of 'ParseConfig()'


#----------------------------------------------------------#
#              Parser Support Functions                    #
#----------------------------------------------------------#


##########
# Condition A Line - Remove Comments & Leading/Trailing Whitespace
##########

def ConditionLine(line):

    return line.split(COMMENT)[0].strip()

# End of 'ConditionLine()'


##########
# Dereference Variables
##########

def DerefVar(line, cfgfile, linenum, RetBoolState=False):

    # Find all symbols refrences and replace w/sym table entry if present

    ref_ok = True

    for var in VarRef.findall(line):

        sym = var[1:-1]   # Strip delimiters

        # Make sure symbol name is properly formed

        ref_ok = ValidateSymbolName(sym, cfgfile, linenum, AllowEnviro=True)

        # If Preliminary tests found errors - quit now

        if not ref_ok:
            return line, ref_ok

        # By default, all variable names assumed to be relative to
        # current namespace unless escaped with NSSEP.  However,
        # environment variables and predefined variables are always
        # presumed to be relative to the top-level namespace and are
        # left untouched here.  This is also the case for the NAMESPACE
        # variable.

        if sym not in Predefined and sym != NAMESPACE and sym[0] != ENVIRO:

            # Look for an escape.  This makes the variable
            # reference "absolute" (relative to the top-level namespace).

            if sym and sym[0] == NSSEP:
                sym = sym[1:]

            # Prepend the current namespace unless we are in the top-level
            # namespace.

            elif SymTable.Symbols[NAMESPACE].Value:
                sym = "%s%s%s" % (SymTable.Symbols[NAMESPACE].Value, NSSEP, sym)
```

```python

        # Handle environment variables
        if sym and sym[0] == ENVIRO and sym[1:] in os.environ:
            envvar = os.getenv(sym[1:])
            line = line.replace(var, envvar)

            if SymTable.DEBUG:
                DebugMsg("dVARREF", (cfgfile, linenum, var, envvar))

        # Handle variables in symbol table

        # The RetBoolState API flag here means that we want to
        # replace references like [BooleanVar] based on the
        # *state* of the variable.  This is used by the conditional
        # logic for .ifall/any/none to allow logical conditions like
        #
        #       .ifall [Bool1] [Bool2]
        #
        # In this case, if the boolean is True, we return the name of
        # the boolean to make the existential test True.  Otherwise we
        # replace the reference with the name of a variable than can
        # never exist - it has a malformed name.  This will cause the
        # existential test count of found variables to change and
        # affect the final logical outcome.

        elif sym in SymTable.Symbols:
            symval = str(SymTable.Symbols[sym].Value)

            if RetBoolState and (SymTable.Symbols[sym].Type == TYPE_BOOL):

                if symval == sTRUE:
                    symval = sym
                else:
                    symval = IllegalChars[0]

            line = line.replace(var, symval)

            if SymTable.DEBUG:
                DebugMsg("dVARREF", (cfgfile, linenum, var, symval))


        # Reference to undefined variable

        else:
            ErrorMsg("eVARUNDEF", (cfgfile, linenum, sym))
            ref_ok = False

    return line, ref_ok

# End of 'DerefVar()'


##########
# Parse A File
##########

def ParseFile(cfgfile, current_cfg, current_linenum):

    global SymTable

    linenum = 0

    # Inhbit recursive includes

    if cfgfile in SymTable.Visited:

        ErrorMsg("eCIRCULAR", (current_cfg, current_linenum))
        return

    else:
        SymTable.Visited.append(cfgfile)

    try:
        cf = open(cfgfile)

        # Process and massage the configuration file
        for line in cf.read().splitlines():
```

```

            SymTable.TotalLines += 1
            linenum +=1

            # Parse this line
            ParseLine(line, cfgfile, linenum)

        # Close the config file
        cf.close()

    # File open failed for some reason
    except IOError:
        ErrorMsg("eCONFOPEN", (current_cfg, current_linenum, cfgfile))  # Record the error

# End of 'ParseFile()'


##########
# Parse An In-Memory Configuration
##########

def ParseInMemory(cfglist, configname):

    global SymTable

    linenum = 0

    SymTable.Visited.append(configname)

    # Process and massage the configuration file

    for line in cfglist:

        SymTable.TotalLines += 1
        linenum += 1

        # The entry must be a string
        if type(line) == TYPE_STRING:

            # Parse this line
            ParseLine(line, configname, linenum)

        # Anything else is a problem
        else:
            ErrorMsg("eNOTSTRING", (configname, linenum, str(line)))

# End of 'ParseInMemory()'


##########
# Parse A Line
##########

def ParseLine(line, cfgfile, linenum):

    global SymTable

    orig = line                 # May need copy of original for debug output


    ##########
    # Beginning Of Line Parser
    ##########

    #####
    # LITERAL and ENDLITERAL Processing
    #
    # These get highest precedence because they block everything else.
    # We keep track of them even in False conditional blocks.  This is
    # so that the text within a literal block does not cause the the
    # parser to throw a syntax error.
    #####

    if line.strip() in (LITERAL, ENDLITERAL):

        if line.strip() == LITERAL:
            if SymTable.INLITERAL:
```

```
                WarningMsg("wLITEXTRA", (cfgfile, linenum))
            else:
                SymTable.INLITERAL = True

        # Process ENDLITERAL statements
        else:
            if not SymTable.INLITERAL:
                WarningMsg("wENDLITEXTRA", (cfgfile, linenum))
            else:
                SymTable.INLITERAL = False


        if SymTable.DEBUG:
            DebugMsg("dPARSEDLINE", (cfgfile, linenum, orig, line))

        return


    #####
    # Process Content Of Literal Blocks
    #
    # We pass lines as-is, with optional variable replacement in
    # literal blocks.  But, only if we're inside a True conditional
    # block.
    #####

    if SymTable.INLITERAL:

        if SymTable.CondStack[-1][1]:

            if SymTable.LITERALVARS:
                line, ref_ok = DerefVar(line, cfgfile, linenum)

            SymTable.LiteralLines.append(line)

            if SymTable.DEBUG:
                DebugMsg("dPARSEDLINE", (cfgfile, linenum, orig, line))


        # Conditional block was false, so just throw away the line,
        # noting in debug output if requested.

        else:
            if SymTable.DEBUG:
                DebugMsg("dLINEIGNORE", (cfgfile, linenum, orig, dNOTINCLUDE))

        return


    #####
    # Process All Other Cases
    #####

    line = ConditionLine(line)  # Strip out comments and leading/trailing whitespace
    condstate = True            # Results of conditional tests kept here



    # Only attempt on non-blank lines for everything else
    # Line was blank
    if not line:

        # Note blank lines for debug purposes
        if SymTable.DEBUG:
            DebugMsg("dLINEIGNORE", (cfgfile, linenum, orig, dBLANKLINE))

        return


    # Get first token on the line
    FIRSTTOK = line.split()[0]


    #####
    # ELSE Processing
    #####

```

```
    if line == ELSE:

        # Get the enclosing block type and state

        btyp, bst = SymTable.CondStack.pop()

        # ELSE is only permitted after an immediately preceding IF form


        if btyp != IF:
            ErrorMsg("eELSENOIF", (cfgfile, linenum))
            SymTable.CondStack.append(["", False])   # Error makes all that follows False

        # We *are* in an IF block an ELSE is appropriate.
        # To determine whether the ELSE should be taken or not we have
        # to look at the state of that IF block AND the state of
        # the block that contains the IF.  This is because the IF
        # block state is the AND of the state of its parent block
        # and its own logical state - i.e. A block can be made
        # logically False by its containing block even if the condition
        # tested is True.

        else:

            # If the containing block is True, the contained IF state is legitimate.
            # The ELSE inverts the IF state in that case.

            if SymTable.CondStack[-1][1]:
                SymTable.CondStack.append([ELSE, not bst])

            # The containing block is false, so everything within it is also false

            else:
                SymTable.CondStack.append([ELSE, False])

        if SymTable.DEBUG:
            DebugMsg("dPARSEDLINE", (cfgfile, linenum, orig, line))

        return


    #####
    # ENDIF Processing
    #####

    if line == ENDIF:

        # Remove one level of conditional nesting
        SymTable.CondStack.pop()

        # Error, if there are more .endifs than conditionals
        if not SymTable.CondStack:
            ErrorMsg("eENDIFEXTRA", (cfgfile, linenum))
            SymTable.CondStack.append(["", False])  # Restore sentinel & inhibit further parsing

        if SymTable.DEBUG:
            DebugMsg("dPARSEDLINE", (cfgfile, linenum, orig, line))

        return


    #####
    # Namespace Processing
    #####

    # All we do here is detect the "[...]" form of the statement and
    # convert it into a "NAMESPACE = ..." form.  That way, the normal
    # variable setting routines below can handle it as usual.

    if line == DELIML+DELIMR or (len(VarRef.findall(line)) == 1 and \
                                 line[0] == DELIML and line[-1] == DELIMR and\
                                 line.count(DELIML) == line.count(DELIMR) == 1):

        # Get the new namespace
        line  = "%s%s%s" % (NAMESPACE, EQUAL, line[1:-1])
```

```
    #####
    # INCLUDE Processing
    #####

    if FIRSTTOK == INCLUDE:

        if not SymTable.CondStack[-1][1]:
            if SymTable.DEBUG:
                DebugMsg("dLINEIGNORE", (cfgfile, linenum, orig, dNOTINCLUDE))
            return

        line, ref_ok = DerefVar(line.split(INCLUDE)[1].strip(), cfgfile, linenum)

        # Only attempt the include if all the variable dereferencing was successful
        if ref_ok:
            ParseFile(line, cfgfile, linenum)

        if SymTable.DEBUG:
            DebugMsg("dPARSEDLINE", (cfgfile, linenum, orig, line))

        return


    #####
    # Conditional Processing
    #
    # Must be one of the following forms -
    #
    #    IFALL  [var] ...
    #    IFANY  [var] ...
    #    IFNONE [var] ...
    #    IF string == string
    #    IF string != string
    #
    #    where string can be any concatentation of text and variable references
    #
    #####

    # Is it any of the IF forms?

    elif FIRSTTOK in (IF, IFALL, IFANY, IFNONE):

        #####
        # Existential Conditionals
        #####

        if FIRSTTOK in (IFALL, IFANY, IFNONE):

            if FIRSTTOK == IFALL:
                line = line.split(IFALL)[1].strip()

            elif FIRSTTOK == IFANY:
                line = line.split(IFANY)[1].strip()

            else:
                line = line.split(IFNONE)[1].strip()


            # Dereference any variables

            line, ref_ok = DerefVar(line, cfgfile, linenum, RetBoolState=True)

            if ref_ok:

                vars = line.split()

                # There has to be at least one variable named
                if vars:
                    numexist = 0

                    # Iterate through all named variables to see if they exist
                    for v in vars:

                        # Condition the variable name with the current namespace
                        # unless:
                        #
                        #      1) We are in the root namespace
```

```
                    #      2) We're dealing with an environment variable
                    #      3) We're dealing with a predefined variable
                    #      4) We're dealing with the NAMESPACE variable
                    #      5) The variable name is not escaped

                    ns = SymTable.Symbols[NAMESPACE].Value

                    if ns and \
                       v[0] != ENVIRO and \
                       v not in Predefined and \
                       v != NAMESPACE:

                        # Convert escaped variable names to canonical form
                        if v[0] == NSSEP:
                            v = v[1:]

                        # Prepend current namespace to everything else
                        else:
                            v = "%s%s%s" % (ns, NSSEP, v)

                    # Produce debug ouput of actual variable name being checked
                    if SymTable.DEBUG:
                        DebugMsg("dVAREXISTS", (cfgfile, linenum, v))

                    # Handle environment variables
                    if v[0] == ENVIRO:
                        if v[1:] in os.environ:
                            numexist += 1

                    # Handle local local variables
                    elif v in SymTable.Symbols:
                        numexist += 1

                # And set the conditional state accordingly

                if FIRSTTOK == IFALL and numexist != len(vars):
                    condstate = False

                if FIRSTTOK == IFANY and numexist == 0:
                    condstate = False

                if FIRSTTOK == IFNONE and numexist != 0:
                    condstate = False

            # Bogus conditional syntax - no variables named
            else:
                ErrorMsg("eBADCOND", (cfgfile, linenum, FIRSTTOK, eNOVARS))
                condstate = False


        # Bogus conditional syntax - tried to reference non-existent variable
        else:
            ErrorMsg("eBADCOND", (cfgfile, linenum, FIRSTTOK, eBADVARREF))
            condstate = False

    #####
    # (In)Equality Conditionals - IF string EQUIV/NOTEQUIV string forms
    #####

    else:
        line = line.split(IF)[1].strip()

        if EQUIV in line or NOTEQUIV in line:

            # Only one operator permitted
            if (line.count(EQUIV) + line.count(NOTEQUIV)) > 1:
                ErrorMsg("eEQUIVEXTRA", (cfgfile, linenum))
                condstate = False

            else:

                # Dereference all variables
                line, ref_ok = DerefVar(line, cfgfile, linenum)

                # Reference to undefined variables forces False
                if not ref_ok:
                    condstate = False
```

```

                    # So does a failure of the equality test itself
                    else:
                        invert    = False
                        operator  = EQUIV
                        condstate = True

                        if operator not in line:  # Must be NOTEQUIV
                            invert = True
                            operator = NOTEQUIV

                        line = line.split(operator)

                        if line[0].strip() != line[1].strip():
                            condstate = False

                        if invert:
                            condstate = not condstate


            # Conditional Syntax Error
            else:
                ErrorMsg("eBADCOND", (cfgfile, linenum, FIRSTTOK, eIFBAD))

                # Force parse state to False on an error
                condstate = False

        # Set parser state based on a successful conditional test
        # But it has to be ANDed with the state of the enclosing block

        enclosing = SymTable.CondStack[-1][1]
        SymTable.CondStack.append([IF, condstate and enclosing])

        # Now reflect this in the parsed line
        line = sTRUE
        if not condstate:
            line = sFALSE

        if SymTable.DEBUG:
            DebugMsg("dPARSEDLINE", (cfgfile, linenum, orig, line))

        return


    #####
    # Handle Variable Declaration/Assignment
    #####

    # If we got here it means that none of the conditional forms
    # were found, so the only thing left might be a variable
    # definition/assignment.

    elif EQUAL in line:

        if not SymTable.CondStack[-1][1]:
            if SymTable.DEBUG:
                DebugMsg("dLINEIGNORE", (cfgfile, linenum, orig, dNOTINCLUDE))
            return

        # Catch attempts to dereference without name anywhere in the line

        if line.count(DELIML + DELIMR):
            ErrorMsg("eSYMNONAME", (cfgfile, linenum))

            if SymTable.DEBUG:
                DebugMsg("dPARSEDLINE", (cfgfile, linenum, orig, line))

            return

        # Do any necessary variable dereferencing
        line, ref_ok = DerefVar(line, cfgfile, linenum)

        # Only do this if all var references were valid
        if ref_ok:

            # Get left and right sides of the assignment
            e = line.index(EQUAL)
```

```
            l = line[:e].strip()
            r = line[e+1:].strip()
            templatename = l.split(NSSEP)[-1]


            # Make sure symbol name is properly formed

            if not  ValidateSymbolName(l, cfgfile, linenum):
                pass  # Validation function issues relevant error messages


            # Suppress any attempt to change a RO variable

            #                 ***NOTE***
            #
            # This test needs to happen here before any of the
            # namespace stuff below.  Doing it here guarantees we
            # catch an attempt to modify one of the Predefined
            # variables, which are always marked RO and are always
            # relative to the top-level namespace.  This way, we don't
            # have to put in an explicit check to ignore Predefined
            # variables in the namespace munging below.  IOW, Don't
            # move this test down into the body of the variable name
            # processing where namespace fiddling takes place.

            elif l in SymTable.Symbols and not SymTable.Symbols[l].Writeable:
                ErrorMsg("eVARREADONLY", (cfgfile, linenum, l))

            # Load variable into the symbol table
            else:

                # Munge the variable name to incorporate
                # the current namespace

                # The NAMESPACE variable is special, it is always
                # relative to the root namespace.

                if l in (NAMESPACE, NSSEP+NAMESPACE):

                    # Make sure it is in absolute format
                    varname = l = NAMESPACE

                # Handle absolute variable references

                elif l.startswith(NSSEP):
                    varname = l = l[1:]

                # In all other cases prepend current namespace
                else:
                    varname = l
                    ns =  SymTable.Symbols[NAMESPACE].Value

                    # Top level namespace variables don't need
                    # separator.

                    if ns:
                        l = "%s%s%s" % (ns, NSSEP, l)

                # If this is a newly defined variable, set its
                # default to be this first value assigned and
                # create the new entry

                if l not in SymTable.Symbols:

                    # Only do this if new variable creation allowed
                    if SymTable.ALLOWNEWVAR:

                        # Rules for new variable creation:
                        #
                        #    1) If var has a template, use it
                        #    2) If var has no template, but TemplatesOnly=True  -> Error
                        #    3) If var has no template, and TemplatesOnly=False -> Create new var

                        # Rule 1

                        if templatename in SymTable.Templates.Symbols:
                            # Create the new variable
```

```
                        SymTable.Symbols[l] = copy.copy(SymTable.Templates.Symbols[templatename])

                        # Load the proposed value only if valid

                        if ValidateValue(l, r, cfgfile, linenum):
                            SymTable.Symbols[l].Value = r

                        # If value is invalid, remove this
                        # variable.  We cannot create a new
                        # variable on an error.  Errors are
                        # actually reported by the validation
                        # function.

                        else:
                            del(SymTable.Symbols[l])

                    # Rule 2

                    elif SymTable.TEMPONLY:
                        ErrorMsg("eTEMPONLY", (cfgfile, linenum, varname))

                    # Rule 3

                    else:

                        # Build a new variable descriptor

                        d = VarDescriptor()
                        d.Default = r
                        d.Value = r
                        SymTable.Symbols[l] = d

                # New vars not allowed
                else:
                    ErrorMsg("eVARNEW", (cfgfile, linenum))

            # Otherwise, update an existing entry.  For existing
            # entries, the update only takes place if all the
            # validation tests specified in the descriptor are
            # passed successfully.

            else:

                # If all the validation testing is OK, update the
                # entry.  The validation function produces any
                # relevant errors.

                if ValidateValue(l, r, cfgfile, linenum):

                    # Write the new value for this variable

                    # Translate various strings to their
                    # equivalent logical value for boolean
                    # variables first

                    if SymTable.Symbols[l].Type == TYPE_BOOL:
                        r = Booleans[r.capitalize()]

                    SymTable.Symbols[l].Value = r

                    # Produce debug output when we change namespaces
                    if SymTable.DEBUG and l == NAMESPACE:
                        DebugMsg("dNAMESPACE",  (cfgfile, linenum, r))

    if SymTable.DEBUG:
        DebugMsg("dPARSEDLINE", (cfgfile, linenum, orig, line))

    return


#####
# Line Format Is Not In One Of The Recognized Forms - Syntax Error
#####

else:
```

```python
        ErrorMsg("eBADSYNTAX", (cfgfile, linenum))


    ##########
    # End Of Line Parser
    ##########


# End of 'ParseLine'


#####
# Ensure Symbol Name Is Properly Formed
#####

def ValidateSymbolName(sym, cfgfile, linenum, AllowEmpty=False, AllowEnviro=False):

    sym_ok = True

    # Check for whitespace

    if VarWhite.match(sym):
        ErrorMsg("eSYMNAMESPC", (cfgfile, linenum))
        sym_ok = False


    # Check for empty symbols if they are disallowed

    if not AllowEmpty and not sym:
        ErrorMsg("eSYMNONAME", (cfgfile, linenum))
        sym_ok = False


    # Check for illegal ENVIRO introducer

    if not AllowEnviro and sym and sym[0] == ENVIRO:
        ErrorMsg("eSYMBADSTART", (cfgfile, linenum, sym))
        sym_ok = False

    # Check for illegal characters in symbol name

    for c in sym:
        if c in IllegalChars:
            ErrorMsg("eSYMBADCHAR", (cfgfile, linenum, sym, c))
            sym_ok = False

    # Return symbol validity

    return sym_ok

# End Of 'ValidateSymbolName()'

#####
# Validate Symbol Table Content
#####

def ValidateSymbolTable(symtbl, name, istemplate=False):

    errors = 0

    for sym in symtbl:

        # Template names cannot contain a namespace separator character

        if istemplate and sym.count(NSSEP):
            ErrorMsg("eAPIPARAMBAD", (eBADTEMPLNAME, sym, name))
            errors += 1

        des = symtbl[sym]

        # Make sure a valid descriptor was passed for each variable
        # Make sure we got a Var Descriptor Object

        if not isinstance(des, VarDescriptor):
            errors += 1
            ErrorMsg("eAPIPARAMBAD", (eNOTDESCRIPT, sym, name))
            des = VarDescriptor()    # Make a fake one so the following tests don't blow up
```

```

        # Check various entries for type agreement

        dt = des.Type

        # Make sure default value agrees with variable type

        if des.Value and type(des.Value) != dt:
            errors += 1
            ErrorMsg("eAPIPARAMBAD", (eBADVALTYP, sym, name))

        # Make sure default value agrees with variable type

        if des.Default and type(des.Default) != dt:
            errors += 1
            ErrorMsg("eAPIPARAMBAD", (eBADDEFAULT, sym, name))

        # Make sure that LegalVals is a list type

        if type(des.LegalVals) != type([]):
            errors += 1
            ErrorMsg("eAPIPARAMBAD", (eLEGALVALLIST, sym, name))

        # Then check each value in the list for type agreement
        else:

            for lv in des.LegalVals:
                if type(lv) != dt:
                    errors += 1
                    ErrorMsg("eAPIPARAMBAD", (eBADLEGALVAL, sym, name))

        # Make sure min and max limits are of the correct type

        for mm in (des.Min, des.Max):

            # Floats can accept Float or Int boundaries
            # Ints, & Strings can accept Int boundaries
            # Boundaries not relevant for  Bool and Complex
            # Anything else is an error

            if mm and dt == TYPE_FLOAT and type(mm) != TYPE_FLOAT and type(mm) != TYPE_INT:
                errors += 1
                ErrorMsg("eAPIPARAMBAD", (eBADMINMAX, sym, name))

            if mm and dt in (TYPE_INT, TYPE_STRING) and type(mm) != TYPE_INT:
                errors += 1
                ErrorMsg("eAPIPARAMBAD", (eBADMINMAX, sym, name))

    # Return success of validation

    symtblok = True

    if errors:
        symtblok = False

    return symtblok

# End Of 'ValidateSymbolTable()'


#####
# Validate Value
#####

def ValidateValue(l, r, cfgfile, linenum):

    update = True

    # Do some special preprocessing for the NAMESPACE variable

    if l == NAMESPACE:

        # The root namespace is always permitted regardless of other
        # restrictions the calling program may have put in place.

        if not r:
            return True
```

```

        # Other NAMESPACE values must conform to the same rules
        # as variable names

        if not  ValidateSymbolName(r, cfgfile, linenum, AllowEmpty=True):
            update = False   # Validation function issues appropriate errors


    des = SymTable.Symbols[l]
    typ = des.Type
    lv  = des.LegalVals
    low = des.Min
    up  = des.Max

    #####
    # Type Enforcement
    #####

    # We try to coerce the new value into
    # the specified type.  If this works, we
    # go on to the rest of the validation tests,
    # otherwise mark the attempt as invalid.

    try:
        # Booleans are a special case - we accept only
        # a limited number of strings on the RHS

        if typ == TYPE_BOOL:
            r = Booleans[r.capitalize()]

        # For everything else, try an explicit coercion
        else:
            r = typ(r)

    except:
        ErrorMsg("eTYPEBAD", (cfgfile, linenum, l, str(typ).split()[1][:-1]))
        update = False


    #####
    # Legal Values Enforcement
    #####

    # These tests valid for everything except booleans.
    # An empty LegalVals list means this test is skipped.
    # For all numeric types, the test is to see if the new
    # value is one of the ones enumerated in LegalVals.
    #
    # For strings, LegalVals is presumed to contain a list
    # of regular expressions.  The test is to compile each
    # regex and see if any match the new value.


    if update and lv:

        if typ in (TYPE_COMPLEX, TYPE_FLOAT, TYPE_INT):
            if r not in lv:
                ErrorMsg("eNOTLEGALVAL", (cfgfile, linenum, r, l))
                update = False

        elif typ == TYPE_STRING:

            foundmatch = False
            for rex in lv:

                try:
                    rexc=re.compile(rex)
                    if rexc.match(r):
                        foundmatch = True

                        # This can be kind of complex, so produce
                        # debug output here to help the poor user

                        if SymTable.DEBUG:
                            DebugMsg("dREGEXMATCH", (cfgfile, linenum, r, rex, l))

                except:
```

```
                    ErrorMsg("eBADREGEX", (cfgfile, linenum, rex, l))
                    update = False

            if not foundmatch:
                ErrorMsg("eNOTLEGALVAL", (cfgfile, linenum, r, l))
                update = False


    #####
    # Bounds Checks
    #####

    # Check bounds for interger and floats

    if update and typ in (TYPE_FLOAT, TYPE_INT):

        if low != None and r < low:
            ErrorMsg("eVALSMALL", (cfgfile, linenum, r, l, low))
            update = False

        if up != None and r > up:
            ErrorMsg("eVALLARGE", (cfgfile, linenum, r, l, up))
            update = False

    # Check bounds for strings - these are min/max string lengths, if present

    if update and typ == TYPE_STRING:

        if low != None and len(r) < low:
            ErrorMsg("eSTRINGSHORT", (cfgfile, linenum, r, l, low))
            update = False

        if up != None and len(r) > up:
            ErrorMsg("eSTRINGLONG", (cfgfile, linenum, r, l, up))
            update = False

    return update


# End of 'ValidateValue()'


#----------------------------------------------------------#
#  List Of Public Names Available To Program Importing Us  #
#----------------------------------------------------------#


__all__ = ["ParseConfig",
           "SymbolTable",
           "Template",
           "TYPE_BOOL",
           "TYPE_COMPLEX",
           "TYPE_FLOAT",
           "TYPE_INT",
           "TYPE_STRING",
           "VarDescriptor"
           ]


#----------------------------------------------------------#
#               Entry Point On Direct Invocation            #
#----------------------------------------------------------#

if __name__ == '__main__':

    # Print program information
    print BANNER

    # Print things we know about the environment
    pk = Predefined.keys()
    pk.sort()
    for var in pk:

        if var[0] == PREDEFINTRO:
            val = Predefined[var]
            print "%s%s%s" % (var, (20-len(var)) * " ", val)

```