```
' ---------------------
' VM class
' ---------------------
VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
END
Attribute VB_Name = "VM"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = False
Attribute VB_Exposed = False
' Class Module: VM (AST executor)
Option Explicit
Private Const VTP_MAX_DEPTH As Long = 8    ' safety depth limit
Private Const VTP_MAX_ITEMS_INLINE As Long = 8 ' prefer inline for small containers
Private VERBOSE_ As Boolean
Private GLOBALS_ As Globals
Private OUTPUT__ As Variant

' Executes AST nodes produced by Compiler (AST).
' Uses Map node types and ScopeStack. Logs to gRuntimeLog.

Public Sub SetGlobals(aGlobals As Globals)
    Set GLOBALS_ = aGlobals
End Sub
Public Property Get OUTPUT_() As Variant
    vAssignment OUTPUT_, OUTPUT__
End Property
Public Property Get Verbose() As Boolean
    Verbose = VERBOSE_
End Property
Public Property Let Verbose(aValue As Boolean)
    VERBOSE_ = aValue
End Property

' Node helpers (Map-based)
Private Function MakeNode(nodeType As String) As Map
    Dim m As New Map
    m.Add "type", nodeType
    Set MakeNode = m
End Function

Public Sub RunProgramByIndex(idx As Long)
    GLOBALS_.ASF_InitGlobals
    If idx < 1 Or idx > GLOBALS_.gPrograms.Count Then Exit Sub
    Dim p As Variant: p = GLOBALS_.gPrograms(idx)
    Dim progName As String: progName = p(0)
    Dim stmts As Collection: Set stmts = p(1)
    Dim rawScope As Collection: Set rawScope = p(2)
    Dim progScope As New ScopeStack
    progScope.LoadRaw rawScope
    progScope.Push
    If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "RUN Program: " & progName
    Dim i As Long
    For i = 1 To stmts.Count
        Dim ctrl As String
        ctrl = ExecuteStmtNode(stmts(i), progScope)
        If ctrl = "RETURN" Then
            vAssignment OUTPUT__, progScope.GetValue("__return")
            Exit For
        End If
        If ctrl = "ERR" Then Exit For
    Next i
    progScope.Pop
End Sub

' Execute a statement node (Map). Return control signals: "", "BREAK", "CONTINUE",
"RETURN", "ERR"
```

```
Private Function ExecuteStmtNode(node As Map, progScope As ScopeStack) As String
    On Error GoTo ErrHandler
    Dim tp As String: tp = node.GetValue("type")
    Dim rval As Variant
    Dim i As Long
    Select Case tp
        Case "Print"
            Dim args As Collection: Set args = node.GetValue("args")
            Dim outParts As New Collection
            For i = 1 To args.Count
                Dim v As Variant: vAssignment v, EvalExprNode(args(i), progScope)
                outParts.Add ValueToStringForPrint(v)
            Next i
            Dim sb As String: sb = ""
            For i = 1 To outParts.Count
                If i > 1 Then sb = sb & ", "
                sb = sb & outParts(i)
            Next i
            If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "PRINT:" & sb
            Debug.Print sb
            ExecuteStmtNode = ""
            Exit Function

        Case "Assign"
            Dim left As Map: Set left = node.GetValue("left")
            Dim right As Map: Set right = node.GetValue("right")
            vAssignment rval, EvalExprNode(right, progScope)
            HandleAssignment left, rval, progScope
            If left.GetValue("type") = "Variable" Then
                Dim lName As String: lName = left.GetValue("name")
                If InStr(1, lName, ".") Then
                    Dim tLeft As Map
                    Dim tmpPropMap() As String
                    tmpPropMap() = Split(lName, ".")
                    Set tLeft = progScope.GetValue(tmpPropMap(LBound(tmpPropMap)))
                    For i = LBound(tmpPropMap) + 1 To UBound(tmpPropMap) - 1
                        Set tLeft = tLeft.GetValue(tmpPropMap(i))
                    Next i
                    tLeft.SetValue tmpPropMap(UBound(tmpPropMap)), rval
                End If
            End If
            ExecuteStmtNode = ""
            Exit Function

        Case "ExprStmt"
            Dim res As Variant: res = EvalExprNode(node.GetValue("expr"), progScope)
            ExecuteStmtNode = ""
            Exit Function

        Case "If"
            ExecuteStmtNode = ExecIfNode(node, progScope)
            Exit Function

        Case "For"
            ExecuteStmtNode = ExecForNode(node, progScope)
            Exit Function

        Case "While"
            ExecuteStmtNode = ExecWhileNode(node, progScope)
            Exit Function

        Case "Break"
            ExecuteStmtNode = "BREAK": Exit Function
        Case "Continue"
            ExecuteStmtNode = "CONTINUE": Exit Function

        Case "Return"
            Dim rex As Map: Set rex = node.GetValue("expr")
            If Not rex Is Nothing Then vAssignment rval, EvalExprNode(rex, progScope)
```

```
                Else rval = Empty
138             progScope.SetValue "__return", rval
139             ExecuteStmtNode = "RETURN": Exit Function
140
141         Case "TryCatch"
142             ExecuteStmtNode = ExecTryCatchNode(node, progScope)
143             Exit Function
144
145         Case "Switch"
146             ExecuteStmtNode = ExecSwitchNode(node, progScope)
147             Exit Function
148
149         Case Else
150             ' unknown node type
151             If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "Unknown statement node: " & tp
152             ExecuteStmtNode = ""
153             Exit Function
154     End Select
155
156 ErrHandler:
157     If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "VM statement error: " & err.Description
158     err.Clear
159     ExecuteStmtNode = "ERR"
160 End Function
161
162 ' Reuse your AssignToArray logic
163 Private Sub AssignToArray(arrName As String, idxV As Variant, val As Variant, progScope
    As ScopeStack)
164     Dim arr As Variant
165     Dim pos As Long
166     Dim ub As Long
167     Dim lb As Long
168
169     arr = progScope.GetValue(arrName)
170     pos = CLng(idxV)
171     If pos < 1 Then
172         err.Raise vbObjectError + 5001, "VM.AssignToArray", "Invalid array index (must
            be >=1)"
173     End If
174     If Not IsArray(arr) Then
175         If IsEmpty(arr) Then
176             arr = Array()
177         Else
178             err.Raise vbObjectError + 5002, "VM.AssignToArray", "Not an array"
179         End If
180     End If
181     ub = -1
182     If IsArray(arr) Then
183         ub = UBound(arr)
184     End If
185     If ub < 0 Then
186         ReDim arr(1 To pos)
187     Else
188         lb = LBound(arr)
189         If pos > (ub - lb + 1) Then
190             ReDim Preserve arr(lb To lb + pos - 1)
191         End If
192     End If
193     arr(LBound(arr) + pos - 1) = val
194     progScope.SetValue arrName, arr
195 End Sub
196
197 ' ---------------------
198 ' ExecIfNode: Evaluate condition, then execute proper block
199 ' ---------------------
200 Private Function ExecIfNode(node As Map, progScope As ScopeStack) As String
201     Dim condNode As Map: Set condNode = node.GetValue("cond")
202     If IsTruthy(EvalExprNode(condNode, progScope)) Then
203         Dim thenStmts As Collection: Set thenStmts = node.GetValue("then")
```

```
        Dim si As Long
        For si = 1 To thenStmts.Count
            Dim ctrl As String: ctrl = ExecuteStmtNode(thenStmts(si), progScope)
            If ctrl <> "" Then ExecIfNode = ctrl: Exit Function
        Next si
        ExecIfNode = ""
        Exit Function
    End If
    Dim elseifConds As Collection: Set elseifConds = node.GetValue("elseif_conds")
    Dim elseifBlocks As Collection: Set elseifBlocks = node.GetValue("elseif_blocks")
    Dim idx As Long
    For idx = 1 To elseifConds.Count
        If IsTruthy(EvalExprNode(elseifConds(idx), progScope)) Then
            Dim bl As Collection: Set bl = elseifBlocks(idx)
            Dim si2 As Long
            For si2 = 1 To bl.Count
                Dim ctrl2 As String: ctrl2 = ExecuteStmtNode(bl(si2), progScope)
                If ctrl2 <> "" Then ExecIfNode = ctrl2: Exit Function
            Next si2
            ExecIfNode = ""
            Exit Function
        End If
    Next idx
    If node.GetValue("hasElse") Then
        Dim els As Collection: Set els = node.GetValue("else")
        Dim ee As Long
        For ee = 1 To els.Count
            Dim ctrl3 As String: ctrl3 = ExecuteStmtNode(els(ee), progScope)
            If ctrl3 <> "" Then ExecIfNode = ctrl3: Exit Function
        Next ee
    End If
    ExecIfNode = ""
End Function

' ---------------------
' ExecForNode
' ---------------------
Private Function ExecForNode(node As Map, progScope As ScopeStack) As String
    Dim initNode As Map: Set initNode = node.GetValue("init")
    Dim condNode As Map: Set condNode = node.GetValue("cond")
    Dim stepNode As Map: Set stepNode = node.GetValue("step")
    Dim body As Collection: Set body = node.GetValue("body")
    Dim condOk As Boolean

    If Not initNode Is Nothing Then
        Dim initType As String
        initType = initNode.GetValue("type")
        Select Case initType
            Case "Assign", "Print", "If", "For", "While", "TryCatch", "Switch",
            "Return", "Break", "Continue", "ExprStmt"
                ' Already a statement node — execute directly so side-effects happen.
                ExecuteStmtNode initNode, progScope
            Case Else
                ' Expression node — wrap into ExprStmt for side-effect evaluation.
                ExecuteStmtNode MakeNodeExprStmt(initNode), progScope
        End Select
    End If

ForStart:
    condOk = True
    If Not condNode Is Nothing Then
        condOk = IsTruthy(EvalExprNode(condNode, progScope))
    End If
    If Not condOk Then GoTo ForEnd

    Dim s As Long
    For s = 1 To body.Count
        Dim ctrl As String: ctrl = ExecuteStmtNode(body(s), progScope)
        If ctrl = "BREAK" Then GoTo ForEnd
```

```
272         If ctrl = "CONTINUE" Then Exit For
273         If ctrl = "RETURN" Or ctrl = "ERR" Then ExecForNode = ctrl: Exit Function
274     Next s
275
276     If Not stepNode Is Nothing Then
277         Dim stepType As String
278         stepType = stepNode.GetValue("type")
279         Select Case stepType
280             Case "Assign", "Print", "If", "For", "While", "TryCatch", "Switch",
                "Return", "Break", "Continue", "ExprStmt"
281                 ExecuteStmtNode stepNode, progScope
282             Case Else
283                 ExecuteStmtNode MakeNodeExprStmt(stepNode), progScope
284         End Select
285     End If
286     GoTo ForStart
287
288 ForEnd:
289     ExecForNode = ""
290 End Function
291
292 ' Helper to wrap an expression node into an ExprStmt node
293 Private Function MakeNodeExprStmt(expr As Map) As Map
294     Dim m As Map: Set m = MakeNode("ExprStmt")
295     m.SetValue "expr", expr
296     Set MakeNodeExprStmt = m
297 End Function
298
299 ' ---------------------
300 ' ExecWhileNode
301 ' ---------------------
302 Private Function ExecWhileNode(node As Map, progScope As ScopeStack) As String
303     Dim condNode As Map: Set condNode = node.GetValue("cond")
304     Dim body As Collection: Set body = node.GetValue("body")
305
306 WhileStart:
307     If Not IsTruthy(EvalExprNode(condNode, progScope)) Then GoTo WhileEnd
308     Dim i As Long
309     For i = 1 To body.Count
310         Dim ctrl As String: ctrl = ExecuteStmtNode(body(i), progScope)
311         If ctrl = "BREAK" Then GoTo WhileEnd
312         If ctrl = "CONTINUE" Then Exit For
313         If ctrl = "RETURN" Or ctrl = "ERR" Then ExecWhileNode = ctrl: Exit Function
314     Next i
315     GoTo WhileStart
316
317 WhileEnd:
318     ExecWhileNode = ""
319 End Function
320
321 ' ---------------------
322 ' ExecTryCatchNode
323 ' ---------------------
324 Private Function ExecTryCatchNode(node As Map, progScope As ScopeStack) As String
325     On Error GoTo TryErr
326     Dim tryStmts As Collection: Set tryStmts = node.GetValue("try")
327     Dim i As Long
328     For i = 1 To tryStmts.Count
329         Dim ctrl As String: ctrl = ExecuteStmtNode(tryStmts(i), progScope)
330         If ctrl = "RETURN" Or ctrl = "ERR" Then
331             ExecTryCatchNode = ctrl
332             If ctrl = "ERR" Then GoTo TryErr
333             Exit Function
334         End If
335     Next i
336     ExecTryCatchNode = ""
337     Exit Function
338
339 TryErr:
```

```
340         err.Clear
341         Dim catchStmts As Collection: Set catchStmts = node.GetValue("catch")
342         If Not catchStmts Is Nothing Then
343             Dim j As Long
344             For j = 1 To catchStmts.Count
345                 Dim cctrl As String: cctrl = ExecuteStmtNode(catchStmts(j), progScope)
346                 If cctrl = "RETURN" Or cctrl = "ERR" Then ExecTryCatchNode = cctrl: Exit
                    Function
347             Next j
348         End If
349         ExecTryCatchNode = ""
350     End Function
351
352     ' ---------------------
353     ' ExecSwitchNode
354     ' ---------------------
355     Private Function ExecSwitchNode(node As Map, progScope As ScopeStack) As String
356         Dim switchVal As Variant: switchVal = EvalExprNode(node.GetValue("expr"), progScope)
357         Dim cases As Collection: Set cases = node.GetValue("cases")
358         Dim i As Long
359         Dim match As Boolean
360         For i = 1 To cases.Count
361             Dim pair As Collection: Set pair = cases(i)
362             Dim caseExpr As Map: Set caseExpr = pair(1)
363             Dim blockStmts As Collection: Set blockStmts = pair(2)
364             match = (EvalExprNode(caseExpr, progScope) = switchVal)
365             If match Then
366                 Dim s As Long
367                 For s = 1 To blockStmts.Count
368                     Dim ctrl As String: ctrl = ExecuteStmtNode(blockStmts(s), progScope)
369                     If ctrl = "BREAK" Then ExecSwitchNode = "": Exit Function
370                     If ctrl = "RETURN" Or ctrl = "ERR" Then ExecSwitchNode = ctrl: Exit
                        Function
371                 Next s
372             End If
373         Next i
374         If Not match Then
375             Dim defBlk As Collection: Set defBlk = node.GetValue("default")
376             If Not defBlk Is Nothing Then
377                 Dim d As Long
378                 For d = 1 To defBlk.Count
379                     Dim ctrl2 As String: ctrl2 = ExecuteStmtNode(defBlk(d), progScope)
380                     If ctrl2 = "BREAK" Then ExecSwitchNode = "": Exit Function
381                     If ctrl2 = "RETURN" Or ctrl2 = "ERR" Then ExecSwitchNode = ctrl2: Exit
                        Function
382                 Next d
383             End If
384         End If
385         ExecSwitchNode = ""
386     End Function
387
388     Private Sub vAssignment(ByRef var As Variant, ByRef vValue As Variant)
389         If IsObject(vValue) Then
390             Set var = vValue
391         Else
392             var = vValue
393         End If
394     End Sub
395     ' ---------------------
396     ' Expression evaluator: Evaluate Expr AST nodes to runtime values
397     ' ---------------------
398     Private Function EvalExprNode(node As Map, progScope As ScopeStack) As Variant
399         If node Is Nothing Then EvalExprNode = Empty: Exit Function
400         Dim tp As String: tp = node.GetValue("type")
401         Dim items As Collection
402         Dim tmpResult As Variant
403         Dim pi As Long
404         Dim funcIdx As Long
405         Dim baseExpr As Map
```

```
406
407         Select Case tp
408             Case "FuncLiteral", "AnonFunc"
409                 ' Create closure (capture env by reference for shared-write closures)
410                 Dim cparams As Collection: vAssignment cparams, node.GetValue("params")
411                 Dim cbody As Collection: vAssignment cbody, node.GetValue("body")
412                 Dim cmap As New Map
413                 Dim envCopy As ScopeStack
414                 cmap.Add "type", "Closure"
415                 cmap.SetValue "params", cparams
416                 cmap.SetValue "body", cbody
417                 ' capture current scope reference (shared-write)
418                 Set envCopy = New ScopeStack
419                 envCopy.LoadRawByRef progScope.RawByRef
420                 cmap.SetValue "env", envCopy
421                 vAssignment tmpResult, cmap
422                 GoTo exitfun
423             Case "VBEXPR"
424                 ' Forced VBAexpression node (from @(...)). The node stores the raw VB
                    expression string in "expr".
425                 Dim rawVB As String: rawVB = node.GetValue("expr")
426                 vAssignment tmpResult, EvalVBExpressionWithScope(rawVB, progScope)
427                 GoTo exitfun
428             Case "Object"
429                 ' Build and return a Map containing the evaluated properties
430                 Set items = node.GetValue("items")
431                 Dim om As New Map
432                 Dim valNode As Map
433                 Dim pair As Collection
434                 Dim key As String
435                 For pi = 1 To items.Count
436                     Set pair = items(pi)
437                     key = CStr(pair(1))
438                     Set valNode = pair(2)
439                     om.SetValue key, EvalExprNode(valNode, progScope)
440                 Next pi
441                 ' Return Map object (as Variant holding the object)
442                 vAssignment tmpResult, om
443                 GoTo exitfun
444             Case "Literal"
445                 vAssignment tmpResult, node.GetValue("value")
446                 GoTo exitfun
447             Case "Variable"
448                 ' Robust variable resolution:
449                 '  - return actual value from scope (covers closures stored as Map AST nodes)
450                 '  - if scope has no value, fallback to named functions table (gFuncTable)
451                 '  - optional fallback to gFuncObjects registry
452                 Dim vname As String: vname = CStr(node.GetValue("name"))
453                 Dim vVal As Variant
454                 vAssignment vVal, progScope.GetValue(vname)
455
456                 ' If scope contains a non-empty value, return it — this covers closures
457                 ' stored as Map/AST nodes (FuncLiteral/Closure/etc.).
458                 If Not IsEmpty(vVal) Then
459                     ' If it's a Map and looks like a function node, return as callable
460                     On Error Resume Next
461                     If TypeName(vVal) = "Map" Then
462                         Dim maybeType As Variant
463                         maybeType = Empty
464                         On Error Resume Next
465                         maybeType = vVal.GetValue("type")
466                         On Error GoTo 0
467                         If Not IsEmpty(maybeType) Then
468                             Dim tt As String: tt = CStr(maybeType)
469                             ' Common names for function-like AST nodes / closures:
470                             If tt = "FuncLiteral" Or tt = "Function" Or tt = "Closure" Or tt
                                = "AnonFunction" Then
471                                 vAssignment tmpResult, vVal
472                                 GoTo exitfun
```

```
                    End If
                End If
            End If
            ' Not necessarily a function-like Map — still return stored value.
            vAssignment tmpResult, vVal
            GoTo exitfun
        End If

        ' Not in scope (or Empty). Fallback: named compiled function table.
        On Error Resume Next
        If GLOBALS_.gFuncTable.Exists(vname) Then
            Dim fIdxVal As Variant: fIdxVal = GLOBALS_.gFuncTable.GetValue(vname)
            On Error GoTo 0
            If Not IsEmpty(fIdxVal) Then
                funcIdx = CLng(fIdxVal)

                ' Build a Closure map with the shape CallClosure expects:
                ' "type" = "Closure", "params" = Collection, "body" = Collection,
                "env" = ScopeStack
                Dim closureMap As New Map
                closureMap.Add "type", "Closure"

                ' params: convert stored gFuncParams (likely an array) into a
                Collection
                Dim paramsCol As Collection
                Set paramsCol = New Collection
                If GLOBALS_.gFuncParams.Exists(vname) Then
                    Dim paVar As Variant: paVar =
                    GLOBALS_.gFuncParams.GetValue(vname)
                    If IsArray(paVar) Then
                        If ArrayIsInit(paVar) Then
                            For pi = LBound(paVar) To UBound(paVar)
                                paramsCol.Add CStr(paVar(pi))
                            Next pi
                        Else
                            paramsCol.Add vbNullString
                        End If
                    ElseIf TypeName(paVar) = "Collection" Then
                        ' already a collection — copy it
                        Dim it As Variant
                        For Each it In paVar
                            paramsCol.Add it
                        Next it
                    End If
                End If
                closureMap.SetValue "params", paramsCol

                ' body and env come from gPrograms(funcIdx)
                Dim pinfo As Variant
                On Error Resume Next
                pinfo = GLOBALS_.gPrograms(funcIdx)
                If err.Number <> 0 Then
                    err.Clear
                    ' fallback: no program info — return Empty
                    EvalExprNode = Empty
                    Exit Function
                End If
                On Error GoTo 0

                ' pinfo layout: Array(name, stmtsCollection, rawScopeCollection)
                Dim bodyStmts As Collection
                Set bodyStmts = pinfo(1)
                closureMap.SetValue "body", bodyStmts

                ' env: create ScopeStack and LoadRaw with stored raw scope (so
                closure has env object)
                Dim envScope As ScopeStack
                Set envScope = New ScopeStack
                envScope.LoadRawByRef progScope.RawByRef
```

```
538                     Set pinfo(2) = envScope.RawByRef
539                     closureMap.SetValue "env", envScope
540
541                     ' optionally store a name/funcIdx for debugging
542                     closureMap.SetValue "name", vname
543                     closureMap.SetValue "funcIdx", funcIdx
544
545                     vAssignment tmpResult, closureMap
546                     GoTo exitfun
547                 End If
548             End If
549             On Error GoTo 0
550
551             ' Defensive fallback: optional function-objects registry (NOT IMPLEMENTED
                YET).
552 '           On Error Resume Next
553 '           If Not (gFuncObjects Is Nothing) Then
554 '               If gFuncObjects.Exists(vName) Then
555 '                   EvalExprNode = gFuncObjects.GetValue(vName)
556 '                   On Error GoTo 0
557 '                   GoTo exitfun
558 '               End If
559 '           End If
560 '           On Error GoTo 0
561
562             ' Not found — log to runtime log and return Empty
563             If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "VM: Variable '" & vname & "' is
                undefined or not callable (returned Empty)."
564             vAssignment tmpResult, Empty
565             GoTo exitfun
566         Case "Member"
567             ' Member(base, prop) — evaluate base, then property access
568             Set baseExpr = node.GetValue("base")
569             Dim propStr As String: propStr = CStr(node.GetValue("prop"))
570             Dim baseEvaluated As Variant
571             vAssignment baseEvaluated, EvalExprNode(baseExpr, progScope)
572             ' special-case array length: if prop == "length" and base is array, return
                length
573             If propStr = "length" Then
574                 If IsArray(baseEvaluated) Then
575                     Dim ub2 As Long
576                     ub2 = UBound(baseEvaluated)
577                     If ub2 < LBound(baseEvaluated) Then
578                         vAssignment tmpResult, 0
579                     Else
580                         vAssignment tmpResult, ub2 - LBound(baseEvaluated) + 1
581                     End If
582                     GoTo exitfun
583                 End If
584             End If
585             ' if base is Map, return stored property
586             If TypeName(baseEvaluated) = "Map" Then
587                 Dim bm As Map: Set bm = baseEvaluated
588                 vAssignment tmpResult, bm.GetValue(propStr)
589                 GoTo exitfun
590             End If
591             ' otherwise no property -> Empty
592             vAssignment tmpResult, Empty
593             GoTo exitfun
594         Case "Array"
595             Set items = node.GetValue("items")
596             If items.Count = 0 Then
597                 vAssignment tmpResult, Array()
598                 GoTo exitfun
599             End If
600             Dim a() As Variant
601             ReDim a(1 To items.Count)
602             Dim ii As Long
603             For ii = 1 To items.Count
```

```
604                 vAssignment a(ii), EvalExprNode(items(ii), progScope)
605             Next ii
606             vAssignment tmpResult, a
607             GoTo exitfun
608         Case "Index"
609             Dim base As Map: Set base = node.GetValue("base")
610             Dim idxNode As Map: Set idxNode = node.GetValue("index")
611             If base.GetValue("type") = "Variable" Then
612                 Dim arr As Variant: arr = progScope.GetValue(base.GetValue("name"))
613                 If Not IsArray(arr) Then vAssignment tmpResult, Empty: GoTo exitfun
614                 Dim pos As Long: pos = CLng(EvalExprNode(idxNode, progScope))
615                 vAssignment tmpResult, arr(LBound(arr) + pos - 1)
616                 GoTo exitfun
617             Else
618                 vAssignment tmpResult, Empty: GoTo exitfun
619             End If
620         Case "Call"
621             ' Call can be either:
622             '  - Call with "name" (string) and args (legacy)
623             '  - Call with "callee" (an expression node) and args
624             '       (new: allows variable holding closures)
625
626             Dim fname As String
627             Dim hasCalleeExpr As Boolean
628             If node.Exists("name") Then
629                 fname = CStr(node.GetValue("name"))
630             End If
631             If fname = "" Then
632                 ' try to derive name from callee expression when callee is a simple
                    Variable node
633                 If node.Exists("callee") Then
634                     hasCalleeExpr = True
635                     Dim calleeExpr As Map: Set calleeExpr = node.GetValue("callee")
636                     If Not calleeExpr Is Nothing Then
637                         If calleeExpr.GetValue("type") = "Variable" Then
638                             fname = CStr(calleeExpr.GetValue("name"))
639                         End If
640                     End If
641                 End If
642             End If
643             ' builtin length function
644             If fname = ".__len__" Then
645                 Dim argn As Collection: Set argn = node.GetValue("args")
646                 Dim av As Variant: av = EvalExprNode(argn(1), progScope)
647                 If Not IsArray(av) Then
648                     vAssignment tmpResult, 0
649                 Else
650                     vAssignment tmpResult, UBound(av) - LBound(av) + 1
651                 End If
652                 GoTo exitfun
653             End If
654
655             Dim argsColl As Collection: Set argsColl = node.GetValue("args")
656             Dim evaluated As Collection
657             Set evaluated = New Collection
658             Dim k As Long
659             For k = 1 To argsColl.Count
660                 evaluated.Add EvalExprNode(argsColl(k), progScope)
661             Next k
662
663             If Not hasCalleeExpr Then
664                 ' named call (existing behavior + fallback to VBAexpressions)
665                 If GLOBALS_.gFuncTable.Exists(fname) Then
666                     funcIdx = CLng(GLOBALS_.gFuncTable.GetValue(fname))
667                     vAssignment tmpResult, CallFuncByIndex_AST(funcIdx, evaluated)
668                 Else
669                     ' named but not internal -> try VBAexpressions function call
670                     vAssignment tmpResult, EvalVBFunctionCall(fname, evaluated,
                        progScope)
```

```
671                 End If
672             Else
673                 ' dynamic callee: evaluate callee expression -> it must yield a closure
                    map or call via VB
674                 Dim calleeVal As Variant
675                 Dim hasThisVal As Boolean: hasThisVal = False
676                 Dim thisVal As Variant
677                 ' If callee is a member or index expression, compute base as `this` for
                    method call binding.
678                 If Not calleeExpr Is Nothing Then
679                     Dim ct As String: ct = IIf(calleeExpr.Exists("type"),
                        CStr(calleeExpr.GetValue("type")), "")
680                     If ct = "Member" Or ct = "Index" Then
681                         hasThisVal = True
682                         Set baseExpr = calleeExpr.GetValue("base")
683                         thisVal = EvalExprNode(baseExpr, progScope)
684                     End If
685                 End If
686                 vAssignment calleeVal, EvalExprNode(calleeExpr, progScope)
687                 If TypeName(calleeVal) = "Map" Then
688                     Dim cM As Map: Set cM = calleeVal
689                     If cM.GetValue("type") = "Closure" Then
690                         If hasThisVal Then
691                             vAssignment tmpResult, CallClosure(cM, evaluated, thisVal)
692                         Else
693                             vAssignment tmpResult, CallClosure(cM, evaluated)
694                         End If
695                     Else
696                         ' not a closure -> fail gracefully
697                         vAssignment tmpResult, Empty
698                     End If
699                 Else
700                     ' if calleeVal is string -> call via VBAexpressions
701                     If VarType(calleeVal) = vbString Then
702                         vAssignment tmpResult, EvalVBFunctionCall(CStr(calleeVal),
                            evaluated, progScope)
703                     Else
704                         vAssignment tmpResult, Empty
705                     End If
706                 End If
707             End If
708             GoTo exitfun
709
710         Case "Unary"
711             Dim op As String: op = node.GetValue("op")
712             Dim ev As Variant: ev = EvalExprNode(node.GetValue("expr"), progScope)
713             If op = "!" Then
714                 vAssignment tmpResult, Not IsTruthy(ev)
715             ElseIf op = "-" Then
716                 vAssignment tmpResult, -CDbl(ev)
717             Else
718                 vAssignment tmpResult, ev
719             End If
720             GoTo exitfun
721         Case "Ternary"
722             Dim cnd As Variant: cnd = EvalExprNode(node.GetValue("cond"), progScope)
723             If IsTruthy(cnd) Then
724                 vAssignment tmpResult, EvalExprNode(node.GetValue("trueExpr"), progScope)
725             Else
726                 vAssignment tmpResult, EvalExprNode(node.GetValue("falseExpr"),
                    progScope)
727             End If
728             GoTo exitfun
729         Case "Binary"
730             Dim lop As Variant: lop = EvalExprNode(node.GetValue("left"), progScope)
731             Dim rop As Variant
732             Dim op2 As String: op2 = node.GetValue("op")
733             If op2 = "&&" Then
734                 If Not IsTruthy(lop) Then vAssignment tmpResult, False: GoTo exitfun
```

```
                rop = EvalExprNode(node.GetValue("right"), progScope)
                vAssignment tmpResult, (IsTruthy(lop) And IsTruthy(rop))
                GoTo exitfun
            ElseIf op2 = "||" Then
                If IsTruthy(lop) Then vAssignment tmpResult, True: GoTo exitfun
                rop = EvalExprNode(node.GetValue("right"), progScope)
                vAssignment tmpResult, (IsTruthy(lop) Or IsTruthy(rop))
                GoTo exitfun
            Else
                rop = EvalExprNode(node.GetValue("right"), progScope)
                Select Case op2
                    Case "+"
                        vAssignment tmpResult, SafeAdd(lop, rop)
                    Case "-": vAssignment tmpResult, lop - rop
                    Case "*": vAssignment tmpResult, lop * rop
                    Case "/"
                        If rop = 0 Then err.Raise vbObjectError + 2001,
                        "VM.EvalExprNode", "Division by zero" Else vAssignment
                        tmpResult, lop / rop
                    Case "%": vAssignment tmpResult, lop Mod rop
                    Case "^": vAssignment tmpResult, lop ^ rop
                    Case "==", "=": vAssignment tmpResult, (lop = rop)
                    Case "!=": vAssignment tmpResult, (lop <> rop)
                    Case "<": vAssignment tmpResult, (lop < rop)
                    Case ">": vAssignment tmpResult, (lop > rop)
                    Case "<=": vAssignment tmpResult, (lop <= rop)
                    Case ">=": vAssignment tmpResult, (lop >= rop)
                    Case Else
                        vAssignment tmpResult, Empty
                End Select
                GoTo exitfun
            End If
    End Select

    vAssignment tmpResult, Empty
exitfun:
    vAssignment EvalExprNode, tmpResult
End Function

Private Function ReturnCollection(ByRef aColl As Variant) As Collection
    Set ReturnCollection = aColl
End Function
Private Function ArrayIsInit(aArray As Variant) As Boolean
    Dim ub As Long
    ArrayIsInit = True
    On Error GoTo err_handler
    ub = UBound(aArray)
    Exit Function
err_handler:
    err.Clear
    ArrayIsInit = False
End Function
' Call function program by index (AST). "args" is Collection of evaluated arg values.
Private Function CallFuncByIndex_AST(funcIdx As Long, args As Collection) As Variant
    Dim p As Variant: p = GLOBALS_.gPrograms(funcIdx)
    Dim rawScope As Collection: Set rawScope = p(2)
    Dim callScope As New ScopeStack
    Dim i As Long
    callScope.LoadRaw rawScope
    callScope.Push
    Dim fname As String: fname = p(0)
    ' set params from gFuncParams
    If GLOBALS_.gFuncParams.Exists(fname) Then
        Dim pa As Variant: pa = GLOBALS_.gFuncParams.GetValue(fname)
        For i = LBound(pa) To UBound(pa)
            If i - LBound(pa) + 1 <= args.Count Then
                callScope.SetValue CStr(pa(i)), args(i - LBound(pa) + 1)
            Else
                callScope.SetValue CStr(pa(i)), Empty
```

```
802             End If
803         Next i
804     End If
805
806     Dim stmts As Collection: Set stmts = p(1)
807     For i = 1 To stmts.Count
808         Dim ctrl As String: ctrl = ExecuteStmtNode(stmts(i), callScope)
809         If ctrl = "RETURN" Then
810             CallFuncByIndex_AST = callScope.GetValue("__return")
811             callScope.Pop
812             Exit Function
813         ElseIf ctrl = "ERR" Then
814             err.Raise vbObjectError + 3000, "VM.CallFunc", "Error during function
                execution"
815         End If
816     Next i
817     CallFuncByIndex_AST = Empty
818     callScope.Pop
819 End Function
820
821 ' ---------------------
822 ' Utilities adapted from your previous VM
823 ' ---------------------
824 Private Function IsTruthy(v As Variant) As Boolean
825     If IsObject(v) Then IsTruthy = Not v Is Nothing: Exit Function
826     If IsNull(v) Then IsTruthy = False: Exit Function
827     If IsEmpty(v) Then IsTruthy = False: Exit Function
828     If VarType(v) = vbBoolean Then IsTruthy = CBool(v): Exit Function
829     If IsNumeric(v) Then IsTruthy = (CDbl(v) <> 0): Exit Function
830     If VarType(v) = vbString Then
831         If IsBoolean(CStr(v)) Then IsTruthy = CBool(v) Else IsTruthy = (CStr(v) <> "")
832         Exit Function
833     End If
834     IsTruthy = True
835 End Function
836
837 Private Function IsBoolean(ByRef expression As String) As Boolean
838     IsBoolean = (LCase(expression) = "true")
839     If Not IsBoolean Then IsBoolean = (LCase(expression) = "false")
840 End Function
841
842 Private Function SafeAdd(a As Variant, b As Variant) As Variant
843     If IsNumeric(a) And IsNumeric(b) Then SafeAdd = a + b Else SafeAdd = CStr(a) &
    CStr(b)
844 End Function
845
846 'Private Function ValueToStringForPrint(v As Variant) As String
847 '    Dim ub As Long, s As String, i As Long
848 '    If Not IsArray(v) Then
849 '        If IsNull(v) Then ValueToStringForPrint = "NULL": Exit Function
850 '        If IsEmpty(v) Then ValueToStringForPrint = "": Exit Function
851 '        ' pretty-print Map objects as {k: v, ...}
852 '        If TypeName(v) = "Map" Then
853 '            Dim outMap As String: outMap = "{"
854 '            Dim keysCol As Collection
855 '            Set keysCol = v.keys
856 '            Dim kk As Long
857 '            For kk = 1 To keysCol.Count
858 '                If kk > 1 Then outMap = outMap & ", "
859 '                Dim kname As String: kname = CStr(keysCol(kk))
860 '                Dim kval As Variant: kval = v.GetValue(kname)
861 '                outMap = outMap & kname & ": " & ValueToStringForPrint(kval)
862 '            Next kk
863 '            outMap = outMap & "}"
864 '            ValueToStringForPrint = outMap
865 '            Exit Function
866 '        End If
867 '        ValueToStringForPrint = CStr(v): Exit Function
868 '    End If
```

```
'    ub = UBound(v)
'    If ub < 0 Then ValueToStringForPrint = "[]": Exit Function
'    s = "["
'    For i = LBound(v) To ub
'        If i > LBound(v) Then s = s & ", "
'        s = s & CStr(v(i))
'    Next i
'    ValueToStringForPrint = s & "]"
'End Function
Private Function ValueToStringForPrint(v As Variant) As String
    Dim visited As New Collection
    ValueToStringForPrint = ValueToStringWithCtx(v, visited, 0)
End Function
' core recursive printer with context
Private Function ValueToStringWithCtx(v As Variant, visited As Collection, depth As
Long) As String
    On Error GoTo ErrHandler

    ' Depth guard
    If depth > VTP_MAX_DEPTH Then
        ValueToStringWithCtx = "..."
        Exit Function
    End If

    ' Null / Empty
    If IsNull(v) Then
        ValueToStringWithCtx = "NULL": Exit Function
    End If
    If IsEmpty(v) Then
        ValueToStringWithCtx = "": Exit Function
    End If

    ' Scalars
    If VarType(v) = vbString Then
        ValueToStringWithCtx = CStr(v): Exit Function
    End If
    If VarType(v) = vbBoolean Then
        If CBool(v) Then ValueToStringWithCtx = "True" Else ValueToStringWithCtx =
        "False"
        Exit Function
    End If
    If IsNumeric(v) Then
        ValueToStringWithCtx = CStr(v): Exit Function
    End If

    ' Arrays (native VBA arrays)
    If IsArray(v) Then
        ValueToStringWithCtx = ArrayToString(v, visited, depth)
        Exit Function
    End If

    ' Objects
    If IsObject(v) Then
        Dim tn As String: tn = TypeName(v)
        Select Case tn
            Case "Map"
                ' cycle detection by object identity
                Dim i As Long
                For i = 1 To visited.Count
                    If visited(i) Is v Then
                        ValueToStringWithCtx = "[Circular]": Exit Function
                    End If
                Next i
                visited.Add v
                ValueToStringWithCtx = MapToString(v, visited, depth + 1)
                visited.Remove visited.Count
                Exit Function

            Case "Collection"
```

```
                ' cycle detection
                Dim j As Long
                For j = 1 To visited.Count
                    If visited(j) Is v Then
                        ValueToStringWithCtx = "[Circular]": Exit Function
                    End If
                Next j
                visited.Add v
                ValueToStringWithCtx = CollectionToString(v, visited, depth + 1)
                visited.Remove visited.Count
                Exit Function

            Case Else
                ' generic objects: try to call a ToString-like property if present, else
                fallback
                ValueToStringWithCtx = ObjectToString(v, visited, depth + 1)
                Exit Function
        End Select
    End If

    ' Fallback
    ValueToStringWithCtx = CStr(v)
    Exit Function

ErrHandler:
    ' On any unexpected error, return a safe placeholder and continue
    On Error Resume Next
    ValueToStringWithCtx = "[error: " & err.Number & "]"
    err.Clear
End Function

' Convert Map -> string
Private Function MapToString(m As Variant, visited As Collection, depth As Long) As
String
    On Error GoTo ErrHandler
    Dim keys As Collection: Set keys = m.keys
    Dim kcnt As Long: kcnt = keys.Count

    If kcnt = 0 Then
        MapToString = "{}": Exit Function
    End If

    ' For small maps and shallow depth prefer inline representation
    If kcnt <= VTP_MAX_ITEMS_INLINE And depth <= 2 Then
        Dim parts() As String
        ReDim parts(1 To kcnt)
        Dim i As Long
        For i = 1 To kcnt
            Dim key As String: key = CStr(keys(i))
            Dim val As Variant: val = m.GetValue(key)
            parts(i) = CStr(key) & ": " & ValueToStringWithCtx(val, visited, depth)
        Next i
        MapToString = "{ " & Join(parts, ", ") & " }"
        Exit Function
    End If

    ' Multi-line pretty print
    Dim sb As String
    Dim indent As String: indent = String(depth * 2, " ")
    Dim innerIndent As String: innerIndent = String((depth + 1) * 2, " ")
    sb = "{"
    Dim first As Boolean: first = True
    Dim kk As Variant
    For Each kk In keys
        If Not first Then sb = sb & vbCrLf
        sb = sb & innerIndent & CStr(kk) & ": " &
        ValueToStringWithCtx(m.GetValue(CStr(kk)), visited, depth + 1)
        first = False
    Next kk
```

```
    sb = sb & vbCrLf & indent & "}"
    MapToString = sb
    Exit Function

ErrHandler:
    MapToString = "{<error>}"
    err.Clear
End Function

' Convert Collection -> string (treat as list)
Private Function CollectionToString(col As Variant, visited As Collection, depth As Long) As String
    On Error GoTo ErrHandler
    Dim n As Long: n = col.Count
    If n = 0 Then CollectionToString = "[]": Exit Function
    If n <= VTP_MAX_ITEMS_INLINE And depth <= 2 Then
        Dim tmp() As String: ReDim tmp(1 To n)
        Dim ii As Long
        For ii = 1 To n
            tmp(ii) = ValueToStringWithCtx(col(ii), visited, depth)
        Next ii
        CollectionToString = "[ " & Join(tmp, ", ") & " ]"
        Exit Function
    End If

    Dim sb As String: sb = "["
    Dim i As Long
    Dim indent As String: indent = String((depth + 1) * 2, " ")
    For i = 1 To n
        If i > 1 Then sb = sb & vbCrLf
        sb = sb & indent & ValueToStringWithCtx(col(i), visited, depth + 1)
    Next i
    sb = sb & vbCrLf & String(depth * 2, " ") & "]"
    CollectionToString = sb
    Exit Function

ErrHandler:
    CollectionToString = "[<error>]"
    err.Clear
End Function

' Convert native VBA array -> string
Private Function ArrayToString(arr As Variant, visited As Collection, depth As Long) As String
    On Error GoTo ErrHandler
    Dim lb As Long, ub As Long
    lb = LBound(arr): ub = UBound(arr)
    Dim n As Long: n = ub - lb + 1
    If n <= 0 Then ArrayToString = "[]": Exit Function
    If n <= VTP_MAX_ITEMS_INLINE And depth <= 2 Then
        Dim tmp() As String: ReDim tmp(1 To n)
        Dim i As Long
        For i = lb To ub
            tmp(i - lb + 1) = ValueToStringWithCtx(arr(i), visited, depth)
        Next i
        ArrayToString = "[ " & Join(tmp, ", ") & " ]"
        Exit Function
    End If

    Dim sb As String: sb = "["
    Dim indent As String: indent = String((depth + 1) * 2, " ")
    Dim ii As Long
    For ii = lb To ub
        If ii > lb Then sb = sb & vbCrLf
        sb = sb & indent & ValueToStringWithCtx(arr(ii), visited, depth + 1)
    Next ii
    sb = sb & vbCrLf & String(depth * 2, " ") & "]"
    ArrayToString = sb
    Exit Function
```

```
1069
1070    ErrHandler:
1071        ArrayToString = "[<error>]"
1072        err.Clear
1073    End Function
1074
1075    ' Generic object to string fallback:
1076    ' - If the object is a Map-like (has Keys and GetValue), will attempt to treat it as Map.
1077    ' - Else TypeName + simple to-string
1078    Private Function ObjectToString(obj As Variant, visited As Collection, depth As Long) As
        String
1079        On Error GoTo Fallback
1080        Dim tn As String: tn = TypeName(obj)
1081
1082        ' Attempt Map-like duck typing: presence of Keys and GetValue
1083        ' (use On Error to bail out if methods missing)
1084        Dim dummy As Collection
1085        Dim tryKeys As Collection
1086        On Error GoTo Fallback2
1087        Set tryKeys = obj.keys
1088        ' if successful, treat as Map
1089        Dim i As Long
1090        For i = 1 To visited.Count
1091            If visited(i) Is obj Then
1092                ObjectToString = "[Circular]": Exit Function
1093            End If
1094        Next i
1095        visited.Add obj
1096        ObjectToString = MapToString(obj, visited, depth)
1097        visited.Remove visited.Count
1098        Exit Function
1099
1100    Fallback2:
1101        ' Not a Map-like object: try default string
1102        On Error GoTo Fallback
1103        ObjectToString = "<" & tn & ">"
1104        Exit Function
1105
1106    Fallback:
1107        ObjectToString = "<object>"
1108        err.Clear
1109    End Function
1110
1111    ' Utility to escape short strings for printing if you want (optional)
1112    Private Function EscapeStringForPrint(s As String) As String
1113        ' currently returns s raw; adapt if you want quoted output
1114        EscapeStringForPrint = s
1115    End Function
1116     ' VBAexpressions integration helpers
1117     ' ----------------------
1118     ' Evaluate a raw VBAexpressions expression string using a VBAexpressions instance,
1119     ' seeding it with the current ASF scope variables so VB expressions can reference ASF
        variables.
1120    Private Function EvalVBExpressionWithScope(expr As String, progScope As ScopeStack) As
        Variant
1121         On Error GoTo ErrHandler
1122         Dim exprEval As VBAexpressions
1123         Set exprEval = New VBAexpressions
1124
1125         ' Create expression in evaluator
1126         exprEval.Create expr
1127
1128         ' Inject variables from progScope (shadowing: global frames first, then locals)
1129         Dim frame As Variant
1130         Dim m As Map
1131         Dim keyCol As Collection
1132         Dim key As Variant
1133         For Each frame In progScope.Raw
1134             Set m = frame
```

```
1135            Set keyCol = m.keys
1136            For Each key In keyCol
1137                exprEval.VarValue(CStr(key)) = m.GetValue(CStr(key))
1138            Next key
1139        Next frame
1140
1141        ' Evaluate
1142        exprEval.Eval
1143        If exprEval.ErrorType = 0 Then
1144            EvalVBExpressionWithScope = exprEval.result
1145        Else
1146            ' On error, raise to caller; the try/catch at Exec layer can handle it
1147            err.Raise vbObjectError + 7001, "VM.EvalVBExpressionWithScope", "VBAexpressions
                eval error"
1148        End If
1149        Exit Function
1150    ErrHandler:
1151        ' convert to runtime log and return Empty
1152        If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "VBAexpr error: " & err.Description
1153        err.Clear
1154        EvalVBExpressionWithScope = Empty
1155    End Function
1156
1157     ' Evaluate a function call in VBAexpressions. Args is a Collection of evaluated values.
1158     ' We create a temporary variable for each argument inside the VBAexpressions environment
1159     ' to avoid needing to serialize complex values into textual literals.
1160    Private Function EvalVBFunctionCall(fname As String, args As Collection, progScope As
        ScopeStack) As Variant
1161        On Error GoTo ErrHandler
1162        Dim exprEval As VBAexpressions
1163        Set exprEval = New VBAexpressions
1164
1165        ' Seed evaluator with ASF scope variables
1166        Dim frame As Variant
1167        Dim m As Map
1168        Dim keyCol As Collection
1169        Dim key As Variant
1170        For Each frame In progScope.Raw
1171            Set m = frame
1172            Set keyCol = m.keys
1173            For Each key In keyCol
1174                exprEval.VarValue(CStr(key)) = m.GetValue(CStr(key))
1175            Next key
1176        Next frame
1177
1178        ' Inject arguments as temporary variables: __asf_vbarg_1, __asf_vbarg_2, ...
1179        Dim i As Long
1180        Dim tmpNames As New Collection
1181        For i = 1 To args.Count
1182            Dim tname As String: tname = "__asf_vbarg_" & CStr(i)
1183            tmpNames.Add tname
1184            exprEval.VarValue(tname) = args(i)
1185        Next i
1186
1187        ' build call string referencing temp names
1188        Dim callStr As String: callStr = fname & "("
1189        For i = 1 To tmpNames.Count
1190            If i > 1 Then callStr = callStr & ","
1191            callStr = callStr & tmpNames(i)
1192        Next i
1193        callStr = callStr & ")"
1194
1195        ' Evaluate
1196        exprEval.Create callStr
1197        exprEval.Eval
1198        If exprEval.ErrorType = 0 Then
1199            EvalVBFunctionCall = exprEval.result
1200        Else
1201            err.Raise vbObjectError + 7002, "VM.EvalVBFunctionCall", "VBAexpressions
```

```
            function call error"
1202        End If
1203        Exit Function
1204    ErrHandler:
1205        If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "VBAexpr function-call error: " &
            err.Description
1206        err.Clear
1207        EvalVBFunctionCall = Empty
1208    End Function
1209
1210     ' ---------------------
1211     ' Closure / function-value support (runtime helpers)
1212     ' ---------------------
1213
1214     ' Create and call closures (closureMap is a Map with keys:
1215     '  "type" = "Closure", "params" = Collection, "body" = Collection (stmts), "env" =
         ScopeStack)
1216     Private Function CallClosure(closureMap As Map, evaluatedArgs As Collection, Optional
         thisVal As Variant) As Variant
1217        On Error GoTo ErrHandler
1218        Dim env As ScopeStack
1219        Set env = closureMap.GetValue("env") ' shared reference (shared-write semantics)
1220
1221        ' push a new frame for this call
1222        env.Push
1223        Dim i As Long
1224        Dim pa As Variant
1225        Dim elm As Variant
1226        Dim fc As Boolean
1227
1228        vAssignment pa, closureMap.GetValue("params")
1229        If IsObject(pa) Then
1230           fc = (Not pa Is Nothing)
1231        Else
1232           fc = Not IsEmpty(pa)
1233        End If
1234        If fc Then
1235            For Each elm In pa
1236                i = i + 1
1237                If i <= evaluatedArgs.Count Then
1238                    env.SetValue CStr(elm), evaluatedArgs(i)
1239                Else
1240                    env.SetValue CStr(elm), Empty
1241                End If
1242            Next elm
1243        End If
1244
1245        ' set 'this' if provided
1246        If Not IsMissing(thisVal) Then
1247            env.SetValue "this", thisVal
1248        End If
1249
1250        ' execute body
1251        Dim stmts As Collection: Set stmts = closureMap.GetValue("body")
1252        Dim ctrl As String
1253        Dim stmtIdx As Long
1254        For stmtIdx = 1 To stmts.Count
1255            ctrl = ExecuteStmtNode(stmts(stmtIdx), env)
1256            If ctrl = "RETURN" Then
1257                vAssignment CallClosure, env.GetValue("__return")
1258                env.Pop
1259                Exit Function
1260            ElseIf ctrl = "ERR" Then
1261                err.Raise vbObjectError + 8001, "VM.CallClosure", "Error during closure
                    execution"
1262            End If
1263        Next stmtIdx
1264
1265        ' normal return -> Empty
```

```
1266        env.Pop
1267        CallClosure = Empty
1268        Exit Function
1269    ErrHandler:
1270        If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "CallClosure error: " & err.Description
1271        err.Clear
1272        CallClosure = Empty
1273     End Function
1274     ' ----------------- LValue resolution and helpers -----------------
1275     Function ResolveLValue(lvalueNode As Map, progScope As ScopeStack) As Variant
1276        Dim result() As Variant
1277        If lvalueNode Is Nothing Then
1278            err.Raise vbObjectError + 9000, "VM.ResolveLValue", "Nil lvalue node"
1279        End If
1280
1281        Dim t As String: t = ""
1282        If lvalueNode.Exists("type") Then t = CStr(lvalueNode.GetValue("type"))
1283        Dim idxVal2 As Variant
1284
1285        Select Case t
1286        Case "Variable"
1287            Dim vname As String: vname = CStr(lvalueNode.GetValue("name"))
1288            ReDim result(0 To 3)
1289            result(0) = "scopeVar"
1290            result(1) = vname
1291            result(2) = vname
1292            ResolveLValue = result
1293            Exit Function
1294
1295        Case "Member"
1296           ' member: compute/evaluate base container, ensure it is a Map if needed,
1297           ' and return ("mapObj", mapRef, propertyName)
1298            Dim propName As String
1299            If lvalueNode.Exists("name") Then
1300                propName = CStr(lvalueNode.GetValue("name"))
1301            ElseIf lvalueNode.Exists("property") Then
1302                propName = CStr(lvalueNode.GetValue("property"))
1303            Else
1304                err.Raise vbObjectError + 9001, "VM.ResolveLValue", "Member node missing
                    property name"
1305            End If
1306
1307            Dim baseNode As Map: Set baseNode = lvalueNode.GetValue("base")
1308           ' If base is a bare variable and empty, create Map and store it into the scope
               so further member sets work
1309           ' Special-case: if the base is an Index (array-element), we must obtain the
               actual
1310           ' element *stored in the array* (not a temporary value) so member writes mutate
               the stored element.
1311            If Not baseNode Is Nothing Then
1312                Dim baseTypeAhead As String
1313                If baseNode.Exists("type") Then baseTypeAhead =
                    CStr(baseNode.GetValue("type")) Else baseTypeAhead = ""
1314                If baseTypeAhead = "Index" Then
1315                    ' Resolve the index target to the actual element Map (creating it if
                        necessary)
1316                    Dim elemMap As Map: Set elemMap = GetElementMapFromIndexNode(baseNode,
                        progScope)
1317                    If elemMap Is Nothing Then
1318                        err.Raise vbObjectError + 9020, "VM.ResolveLValue", "Index element
                            not available for member assignment"
1319                    End If
1320                    ' Build a 4-element descriptor so HandleAssignment will find mapObj at
                        resolved(1) and prop at resolved(3)
1321                    Dim outArrSpecial As Variant: outArrSpecial = Array("mapObj", elemMap,
                        "", propName)
1322                    ResolveLValue = outArrSpecial
1323                    Exit Function
1324                End If
```

```
1325                Dim baseType As String
1326                If baseNode.Exists("type") Then baseType = CStr(baseNode.GetValue("type"))
                    Else baseType = ""
1327                If baseType = "Variable" Then
1328                    Dim baseVarName As String: baseVarName = CStr(baseNode.GetValue("name"))
1329                    Dim baseVal As Variant: baseVal = progScope.GetValue(baseVarName)
1330                    If IsEmpty(baseVal) Then
1331                        Dim newMap As Map: Set newMap = MakeNode("Map")
1332                        progScope.SetValue baseVarName, newMap
1333                        baseVal = newMap
1334                    End If
1335                    If TypeName(baseVal) = "Map" Then
1336                        ReDim result(0 To 3)
1337                        result(0) = "mapObj"
1338                        vAssignment result(1), baseVal
1339                        vAssignment result(2), propName
1340                        ResolveLValue = result
1341                        Exit Function
1342                    Else
1343                        err.Raise vbObjectError + 9002, "VM.ResolveLValue", "Cannot assign
                            member on non-object base"
1344                    End If
1345                Else
1346                    Dim baseVal2 As Variant: baseVal2 = EvalExprNode(baseNode, progScope)
1347                    If TypeName(baseVal2) = "Map" Then
1348                        ReDim result(0 To 3)
1349                        result(0) = "mapObj"
1350                        vAssignment result(1), baseVal2
1351                        result(2) = propName
1352                        ResolveLValue = result
1353                        Exit Function
1354                    Else
1355                        err.Raise vbObjectError + 9003, "VM.ResolveLValue", "Cannot assign
                            member on non-object base"
1356                    End If
1357                End If
1358            Else
1359                err.Raise vbObjectError + 9004, "VM.ResolveLValue", "Member node missing
                    base"
1360            End If
1361
1362        Case "Index"
1363            Dim idxNode As Map: Set idxNode = lvalueNode.GetValue("index")
1364            Dim baseIdxNode As Map: Set baseIdxNode = lvalueNode.GetValue("base")
1365            If baseIdxNode Is Nothing Then
1366                err.Raise vbObjectError + 9005, "VM.ResolveLValue", "Index node missing
                    base"
1367            End If
1368            Dim baseIdxType As String: baseIdxType = ""
1369            If baseIdxNode.Exists("type") Then baseIdxType =
                CStr(baseIdxNode.GetValue("type"))
1370
1371            If baseIdxType = "Variable" Then
1372                Dim arrVarName As String: arrVarName = CStr(baseIdxNode.GetValue("name"))
1373                Dim arrVal As Variant: arrVal = progScope.GetValue(arrVarName)
1374                If IsEmpty(arrVal) Then
1375                    Dim emptyArr() As Variant
1376                    ReDim emptyArr(0 To 0)
1377                    progScope.SetValue arrVarName, emptyArr
1378                    arrVal = progScope.GetValue(arrVarName)
1379                End If
1380                Dim idxVal As Variant: idxVal = EvalExprNode(idxNode, progScope)
1381                If Not IsNumeric(idxVal) Then err.Raise vbObjectError + 9006,
                    "VM.ResolveLValue", "Array index must be numeric"
1382                ReDim result(0 To 3)
1383                result(0) = "arrayInScope"
1384                result(1) = arrVarName
1385                result(2) = CLng(idxVal)
1386                ResolveLValue = result
```

```
1387                 Exit Function
1388             Else
1389                 ' base is an expression (e.g., a.prop[...]) — if base is Member(baseObj,
                     propName)
1390                 ' we want the parent map object and property name so we can assign into the
                     array stored in the property.
1391                 If baseIdxNode.Exists("type") And CStr(baseIdxNode.GetValue("type")) =
                     "Member" Then
1392                     Dim parentNode As Map: Set parentNode = baseIdxNode.GetValue("base")
1393                     If parentNode Is Nothing Then err.Raise vbObjectError + 9015,
                     "VM.ResolveLValue", "Member base missing"
1394                     ' Evaluate parent object (should yield the Map that holds the property)
1395                     Dim parentVal As Variant: parentVal = EvalExprNode(parentNode, progScope)
1396                     If IsEmpty(parentVal) Then
1397                         ' If the parent is a variable and is empty, create a Map to hold
                         properties (match other behaviors)
1398                         If parentNode.Exists("type") And CStr(parentNode.GetValue("type")) =
                         "Variable" Then
1399                             Dim newParentMap As Map: Set newParentMap = MakeNode("Map")
1400                             progScope.SetValue CStr(parentNode.GetValue("name")),
                             newParentMap
1401                             parentVal = newParentMap
1402                         End If
1403                     End If
1404                     If TypeName(parentVal) <> "Map" Then
1405                         err.Raise vbObjectError + 9016, "VM.ResolveLValue", "Parent of
                         member index must be an object (Map)"
1406                     End If
1407                     Dim innerPropName As String: innerPropName =
                     CStr(baseIdxNode.GetValue("name"))
1408                     idxVal2 = EvalExprNode(idxNode, progScope)
1409                     If Not IsNumeric(idxVal2) Then err.Raise vbObjectError + 9007,
                     "VM.ResolveLValue", "Array index must be numeric"
1410                     result = Array("arrayInMap", parentVal, CLng(idxVal2), innerPropName)
1411                     ResolveLValue = result
1412                     Exit Function
1413                 Else
1414                     ' fallback for generic expression bases: evaluate base to a value
1415                     Dim baseVal3 As Variant: baseVal3 = EvalExprNode(baseIdxNode, progScope)
1416                     idxVal2 = EvalExprNode(idxNode, progScope)
1417                     If Not IsNumeric(idxVal2) Then err.Raise vbObjectError + 9007,
                     "VM.ResolveLValue", "Array index must be numeric"
1418                     If TypeName(baseVal3) = "Map" Then
1419                         ' indexing a Map directly is not supported for numeric indices
1420                         err.Raise vbObjectError + 9008, "VM.ResolveLValue", "Cannot assign
                         indexed member on non-variable/map base"
1421                     ElseIf IsArray(baseVal3) Then
1422                         err.Raise vbObjectError + 9009, "VM.ResolveLValue", "Cannot assign
                         into temporary array expression"
1423                     Else
1424                         err.Raise vbObjectError + 9010, "VM.ResolveLValue", "Unsupported
                         index base for assignment"
1425                     End If
1426                 End If
1427             End If
1428
1429         Case Else
1430             err.Raise vbObjectError + 9011, "VM.ResolveLValue", "Unsupported LValue node
                 type: " & t
1431         End Select
1432
1433     End Function
1434
1435     Sub HandleAssignment(lhs As Map, rhsValue As Variant, progScope As ScopeStack)
1436     '     Dim resolved As Variant
1437     '     On Error GoTo HandlerErr
1438     '     resolved = ResolveLValue(lhs, progScope)
1439     '     Dim kind As String: kind = CStr(resolved(0))
1440     '     Select Case kind
```

```
'     Case "scopeVar"
'         Dim vname As String: vname = CStr(resolved(1))
'         progScope.SetValue vname, rhsValue
'     Case "mapObj"
'         Dim mapObj As Map: Set mapObj = resolved(1)
'         Dim propName2 As String: propName2 = CStr(resolved(2))
'         mapObj.SetValue propName2, rhsValue
'     Case "arrayInScope"
'         Dim arrName As String: arrName = CStr(resolved(1))
'         Dim idx As Long: idx = CLng(resolved(2))
'         AssignToArrayByName arrName, idx, rhsValue, progScope
'     Case "arrayInMap"
'         If UBound(resolved) >= 4 Then
'             Dim mapObj2 As Map: Set mapObj2 = resolved(1)
'             Dim idx2 As Long: idx2 = CLng(resolved(2))
'             Dim propName3 As String: propName3 = CStr(resolved(3))
'             Dim arrVal As Variant: arrVal = mapObj2.GetValue(propName3)
'             If IsEmpty(arrVal) Then
'                 Dim newA() As Variant
'                 ReDim newA(1 To 0)
'                 arrVal = newA
'             End If
'             Call AssignToArrayValueInPlace(arrVal, idx2, rhsValue)
'             mapObj2.SetValue propName3, arrVal
'         Else
'             err.Raise vbObjectError + 9110, "VM.HandleAssignment", "Malformed
ResolveLValue arrayInMap result"
'         End If
'     Case Else
'         err.Raise vbObjectError + 9111, "VM.HandleAssignment", "Unhandled resolved
LValue kind: " & kind
'     End Select
'     Exit Sub
    Dim resolved As Variant
    Dim propName2 As String
    On Error GoTo HandlerErr
    resolved = ResolveLValue(lhs, progScope)
    ' Support both returned shapes: 3-element (standard) or 4-element (arrayInMap with
    prop name)
    Dim kind As String: kind = CStr(resolved(0))
    Select Case kind
    Case "scopeVar"
        Dim vname As String: vname = CStr(resolved(1))
        ' If the scope variable name contains dots (e.g. "o.a") — try to resolve to a
        parent Map in scope.
        If InStr(vname, ".") > 0 Then
            Dim curName As String: curName = vname
            Dim done As Boolean: done = False
            Do While InStr(curName, ".") > 0 And Not done
                Dim lastDot As Long: lastDot = InStrRev(curName, ".")
                Dim parentName As String: parentName = left$(curName, lastDot - 1)
                Dim propName As String: propName = Mid$(curName, lastDot + 1)
                Dim parentVal As Variant: vAssignment parentVal,
                progScope.GetValue(parentName)
                If Not IsEmpty(parentVal) Then
                    If TypeName(parentVal) = "Map" Then
                        ' write into parentVal[propName]
                        parentVal.SetValue propName, rhsValue
                        done = True
                        Exit Do
                    ElseIf IsArray(parentVal) Then
                        ' parentVal is array stored in a top-level variable; assign into
                        it if propName is a numeric index
                        If IsNumeric(propName) Then
                            Call AssignToArrayValueInPlace(parentVal, CLng(propName),
                            rhsValue)
                            progScope.SetValue parentName, parentVal
                            done = True
                            Exit Do
```

```
                        End If
                    End If
                End If
                curName = parentName
            Loop
            If done Then Exit Sub
            ' If not resolvable, fallthrough and set top-level variable (will create
            variable named "o.a" which is undesirable)
        End If
        progScope.SetValue vname, rhsValue
    Case "mapObj"
        Dim mapObj As Map
        ' resolved(1) can be either a Map object or (unexpectedly) a string like "o.a"
        in some flows.
        If TypeName(resolved(1)) = "String" Then
            ' try to resolve dotted name to a Map + prop
            Dim dotted As String: dotted = CStr(resolved(1))
            If InStr(dotted, ".") > 0 Then
                Dim leftPart As String: leftPart = left$(dotted, InStrRev(dotted, ".") -
                1)
                Dim rightPart As String: rightPart = Mid$(dotted, InStrRev(dotted, ".")
                + 1)
                Dim leftVal As Variant: vAssignment leftVal, progScope.GetValue(leftPart)
                If Not IsEmpty(leftVal) And TypeName(leftVal) = "Map" Then
                    Set mapObj = leftVal
                    propName2 = rightPart
                    mapObj.SetValue propName2, rhsValue
                    Exit Sub
                End If
            End If
            ' fallback: treat resolved(1) as string variable name
            Dim xname As String: xname = CStr(resolved(1))
            If Not IsEmpty(progScope.GetValue(xname)) Then
                If TypeName(progScope.GetValue(xname)) = "Map" Then
                    Set mapObj = progScope.GetValue(xname)
                    Dim propName2b As String: propName2b = CStr(resolved(2))
                    mapObj.SetValue propName2b, rhsValue
                    Exit Sub
                Else
                    err.Raise vbObjectError + 9130, "VM.HandleAssignment", "Cannot
                    resolve mapObj from string: " & dotted
                End If
            End If
        Else
            Set mapObj = resolved(1)
            propName2 = CStr(resolved(2))
            mapObj.SetValue propName2, rhsValue
        End If
     Case "arrayInScope"
        Dim arrName As String: arrName = CStr(resolved(1))
        Dim idx As Long: idx = CLng(resolved(2))
        ' If arrName is dotted (like "o.a"), resolve to parent Map and property name,
        then operate on that property's array
        If InStr(arrName, ".") > 0 Then
            Dim curD As String: curD = arrName
            Dim resolvedOK As Boolean: resolvedOK = False
            Do While InStr(curD, ".") > 0 And Not resolvedOK
                Dim lastDot2 As Long: lastDot2 = InStrRev(curD, ".")
                Dim pName As String: pName = left$(curD, lastDot2 - 1)
                Dim propN As String: propN = Mid$(curD, lastDot2 + 1)
                Dim pVal As Variant: vAssignment pVal, progScope.GetValue(pName)
                If Not IsEmpty(pVal) And TypeName(pVal) = "Map" Then
                    Dim arrVal2 As Variant: vAssignment arrVal2, pVal.GetValue(propN)
                    If IsEmpty(arrVal2) Then
                        Dim tmpC() As Variant
                        ReDim tmpC(1 To 0)
                        arrVal2 = tmpC
                    End If
                    Call AssignToArrayValueInPlace(arrVal2, idx, rhsValue)
```

```
                    pVal.SetValue propN, arrVal2
                    resolvedOK = True
                    Exit Do
                End If
                curD = pName
            Loop
            If resolvedOK Then Exit Sub
            ' fallthrough to try as top-level arr var
        End If
        AssignToArrayByName arrName, idx, rhsValue, progScope
    Case "arrayInMap"
        ' in the special shape the resolver returned Array(kind,map,index,propName)
        If UBound(resolved) >= 4 Then
            Dim mapObj2 As Map: Set mapObj2 = resolved(1)
            Dim idx2 As Long: idx2 = CLng(resolved(2))
            Dim propName3 As String: propName3 = CStr(resolved(3))
            Dim arrVal As Variant: vAssignment arrVal, mapObj2.GetValue(propName3)
            ' if not array, create
            If IsEmpty(arrVal) Then
                Dim newA() As Variant
                ReDim newA(0 To 0)
                arrVal = newA
            End If
            ' Assign into the array stored in the map property (use 1-based semantics)
            Call AssignToArrayValueInPlace(arrVal, idx2, rhsValue)
            ' store back the possibly resized array
            mapObj2.SetValue propName3, arrVal
        Else
            err.Raise vbObjectError + 9110, "VM.HandleAssignment", "Malformed
            ResolveLValue arrayInMap result"
        End If
    End Select
    Exit Sub
HandlerErr:
    err.Raise err.Number, err.Source, err.Description
End Sub

Sub AssignToArrayByName(arrVarName As String, idx As Long, newValue As Variant,
progScope As ScopeStack)
    Dim arrVal As Variant: arrVal = progScope.GetValue(arrVarName)
    If IsEmpty(arrVal) Then
        Dim tmpA() As Variant
        ReDim tmpA(1 To 0)
        arrVal = tmpA
    End If
    Call AssignToArrayValueInPlace(arrVal, idx, newValue)
    progScope.SetValue arrVarName, arrVal
End Sub

Sub AssignToArrayValueInPlace(ByRef arrVal As Variant, idx As Long, newValue As Variant)
    If Not IsArray(arrVal) Then
        Dim na() As Variant
        ReDim na(1 To idx)
        na(idx) = newValue
        arrVal = na
        Exit Sub
    End If
    Dim lb As Long: lb = LBound(arrVal)
    Dim ub As Long: ub = UBound(arrVal)
    Dim pos As Long: pos = lb + idx - 1
    If pos < lb Then
        err.Raise vbObjectError + 9120, "VM.AssignToArrayValueInPlace", "Index out of
        bounds (below LBound)"
    End If
    If pos > ub Then
        ReDim Preserve arrVal(lb To pos)
    End If
    arrVal(pos) = newValue
End Sub
```

```
'------------------------------------------------------------------------
' GetElementMapFromIndexNode
' Given an Index AST node, resolve the array container and return the actual
' element Map stored at the index (creating a Map element if empty).
'------------------------------------------------------------------------
Function GetElementMapFromIndexNode(indexNode As Map, progScope As ScopeStack) As Map
    Dim resolved As Variant
    resolved = ResolveLValue(indexNode, progScope)
    If IsEmpty(resolved) Then
        Set GetElementMapFromIndexNode = Nothing
        Exit Function
    End If
    Dim kind As String: kind = CStr(resolved(0))
    Select Case kind
    Case "arrayInScope"
        Dim arrName As String: arrName = CStr(resolved(1))
        Dim idx As Long: idx = CLng(resolved(2))
        Dim arrVal As Variant: arrVal = progScope.GetValue(arrName)
        If IsEmpty(arrVal) Then
            Dim tmpA() As Variant
            ReDim tmpA(1 To 0)
            arrVal = tmpA
        End If
        Dim lb As Long: lb = LBound(arrVal)
        Dim pos As Long: pos = lb + idx - 1
        If pos > UBound(arrVal) Then
            ' extend array so the slot exists
            ReDim Preserve arrVal(lb To pos)
        End If
        Dim elem As Variant: elem = arrVal(pos)
        If IsEmpty(elem) Then
            Dim nm As Map: Set nm = MakeNode("Map")
            arrVal(pos) = nm
            progScope.SetValue arrName, arrVal
            Set GetElementMapFromIndexNode = nm
            Exit Function
        ElseIf TypeName(elem) = "Map" Then
            Set GetElementMapFromIndexNode = elem
            Exit Function
        Else
            ' not a Map stored in the array slot
            err.Raise vbObjectError + 9021, "VM.GetElementMapFromIndexNode", "Array
            element is not an object"
        End If
    Case "arrayInMap"
        ' resolved shape: Array("arrayInMap", mapObj, index, propName)
        Dim mapObj As Map: Set mapObj = resolved(1)
        Dim theIdx As Long: theIdx = CLng(resolved(2))
        Dim propName As String: propName = CStr(resolved(3))
        Dim arrVal2 As Variant: arrVal2 = mapObj.GetValue(propName)
        If IsEmpty(arrVal2) Then
            Dim tmpB() As Variant
            ReDim tmpB(1 To 0)
            arrVal2 = tmpB
        End If
        Dim lb2 As Long: lb2 = LBound(arrVal2)
        Dim pos2 As Long: pos2 = lb2 + theIdx - 1
        If pos2 > UBound(arrVal2) Then
            ReDim Preserve arrVal2(lb2 To pos2)
        End If
        Dim elem2 As Variant: elem2 = arrVal2(pos2)
        If IsEmpty(elem2) Then
            Dim nm2 As Map: Set nm2 = MakeNode("Map")
            arrVal2(pos2) = nm2
            mapObj.SetValue propName, arrVal2
            Set GetElementMapFromIndexNode = nm2
            Exit Function
        ElseIf TypeName(elem2) = "Map" Then
            Set GetElementMapFromIndexNode = elem2
```

```
            Exit Function
        Else
            err.Raise vbObjectError + 9022, "VM.GetElementMapFromIndexNode", "Array
            element is not an object (map)"
        End If
    Case Else
        err.Raise vbObjectError + 9023, "VM.GetElementMapFromIndexNode", "Unsupported
        resolved kind for index -> " & kind
    End Select
End Function
 ' ----------------- end LValue helpers -----------------
' ---------------------
' End VM class
' ---------------------


' ---------------------
' Compiler class
' ---------------------
VERSION 1.0 CLASS
BEGIN
  MultiUse = -1  'True
END
Attribute VB_Name = "Compiler"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = False
Attribute VB_Exposed = False
' Class Module: Compiler
Option Explicit
Private GLOBALS_ As Globals

' Produces AST (Maps) for program and functions.
' Relies on Parser.Tokenize, Map, ScopeStack and globals:
'   ProgramCache, gPrograms (Collection), gFuncTable (Map), gFuncParams (Map)

Public Sub SetGlobals(ByRef aGlobals As Globals)
    Set GLOBALS_ = aGlobals
End Sub

' Node helpers (Map-based)
Private Function MakeNode(nodeType As String) As Map
    Dim m As New Map
    m.Add "type", nodeType
    Set MakeNode = m
End Function
' --- Helper: parse collapsed IDENT strings like "o.x", "a[3].b[2].c", "arr[1].length"
Private Function ParseCollapsedIdentToNode(ByVal name As String) As Map
    ' Returns a node representing the chain: Variable/Index/Member nodes
    Dim parts() As String
    Dim curNode As Map
    Dim pos As Long, nlen As Long
    Dim ch As String
    Dim i As Long
    Dim token As String
    nlen = Len(name)
    pos = 1

    ' convenience: iterate and consume segments
    Do While pos <= nlen
        ch = Mid$(name, pos, 1)
        If ch = "." Then
            ' dot should not appear at start; skip
            pos = pos + 1
            GoTo ContinueLoop
        End If
        ' extract next segment (up to '.' or '[')
        Dim nextDot As Long: nextDot = InStr(pos, name, ".")
        Dim nextBr As Long: nextBr = InStr(pos, name, "[")
```

```
1767          Dim seg As String
1768          If nextBr = 0 And nextDot = 0 Then
1769              seg = Mid$(name, pos)
1770              pos = nlen + 1
1771          ElseIf nextBr > 0 And (nextDot = 0 Or nextBr < nextDot) Then
1772              seg = Mid$(name, pos, nextBr - pos)
1773              ' we'll handle bracket after seg
1774              pos = nextBr
1775          Else
1776              seg = Mid$(name, pos, nextDot - pos)
1777              pos = nextDot
1778          End If
1779
1780          If seg <> "" Then
1781              If curNode Is Nothing Then
1782                  ' first segment => base variable
1783                  Dim v As Map: Set v = MakeNode("Variable")
1784                  v.SetValue "name", seg
1785                  Set curNode = v
1786              Else
1787                  ' dot-access to a property name
1788                  Dim mem As Map: Set mem = MakeNode("Member")
1789                  mem.SetValue "base", curNode
1790                  mem.SetValue "prop", seg
1791                  Set curNode = mem
1792              End If
1793          End If
1794
1795          ' handle bracketed index(s) immediately following
1796          Do While pos <= nlen And Mid$(name, pos, 1) = "["
1797              Dim closePos As Long: closePos = InStr(pos, name, "]")
1798              If closePos = 0 Then
1799                  ' malformed, bail with current node
1800                  Exit Do
1801              End If
1802              Dim idxStr As String: idxStr = Mid$(name, pos + 1, closePos - pos - 1)
1803              ' build index expression node from literal or identifier (numbers expected
                  from collapse)
1804              Dim idxNode As Map
1805              If IsNumeric(idxStr) Then
1806                  Set idxNode = MakeNode("Literal")
1807                  idxNode.SetValue "value", CDbl(idxStr)
1808              Else
1809                  ' fallback: create Variable or Literal string if quoted
1810                  If left$(idxStr, 1) = "'" And right$(idxStr, 1) = "'" Then
1811                      Set idxNode = MakeNode("Literal")
1812                      idxNode.SetValue "value", Mid$(idxStr, 2, Len(idxStr) - 2)
1813                  Else
1814                      Set idxNode = MakeNode("Variable")
1815                      idxNode.SetValue "name", idxStr
1816                  End If
1817              End If
1818              ' create Index node: base = curNode
1819              Dim inNode As Map: Set inNode = MakeNode("Index")
1820              inNode.SetValue "base", curNode
1821              inNode.SetValue "index", idxNode
1822              Set curNode = inNode
1823              pos = closePos + 1
1824          Loop
1825
1826          ' handle trailing dot (will be processed in next loop iteration)
1827      ContinueLoop:
1828          Loop
1829
1830          ' Special-case '.length' when last part is 'length':
1831          ' If last node is Member with prop "length", convert into Call to builtin ".__len__"
1832          If Not curNode Is Nothing Then
1833              If curNode.GetValue("type") = "Member" Then
1834                  If LCase$(CStr(curNode.GetValue("prop"))) = "length" Then
```

```
                Dim baseNode As Map: Set baseNode = curNode.GetValue("base")
                Dim lenCall As Map: Set lenCall = MakeNode("Call")
                ' builtin callee as Variable ".__len__" (VM will treat this as builtin)
                Dim builtinLen As Map: Set builtinLen = MakeNode("Variable")
                builtinLen.SetValue "name", ".__len__"
                lenCall.SetValue "callee", builtinLen
                Dim args As New Collection
                args.Add baseNode
                lenCall.SetValue "args", args
                Set ParseCollapsedIdentToNode = lenCall
                Exit Function
            End If
        End If
    End If

    Set ParseCollapsedIdentToNode = curNode
End Function

' ---------------------
' Utility: deep-clone a Collection of Maps or primitive values
' ---------------------
Private Function CloneCollectionOfVariants(src As Variant) As Collection
    Dim out As New Collection
    Dim it As Variant
    For Each it In src
        If IsObject(it) Then
            If TypeName(it) = "Map" Then
                out.Add it.Clone
            ElseIf TypeName(it) = "Collection" Then
                out.Add CloneCollectionOfVariants(it)
            Else
                out.Add it
            End If
        Else
            out.Add it
        End If
    Next it
    Set CloneCollectionOfVariants = out
End Function

' ---------------------

Public Function CompileProgram(src As String, Optional progName As String = "@anon") As
Long
    Dim p As Parser
    Dim toks As Collection
    Dim progScope As ScopeStack
    Dim stmtsAST As Collection
    Dim stmtTokens As Collection
    Dim i As Long
    Dim node As Map

    GLOBALS_.ASF_InitGlobals

    If GLOBALS_.ProgramCache.Exists(src) Then
        CompileProgram = GLOBALS_.ProgramCache.GetValue(src)
        Exit Function
    End If

    Set p = New Parser
    With p
        .SetGlobals GLOBALS_
        Set toks = .Tokenize(src)
    End With

    Set progScope = New ScopeStack
    progScope.Push

    Set stmtsAST = New Collection
```

```
1903        i = 1
1904
1905    Compiler_MainLoop:
1906        Do While i <= toks.Count
1907            ' skip comments at top level
1908            If toks(i)(0) = "COMMENT" Then
1909                i = i + 1
1910                GoTo Compiler_MainLoop
1911            End If
1912
1913            ' Function definition at top-level
1914            If toks(i)(0) = "IDENT" And LCase$(toks(i)(1)) = "fun" Then
1915                i = i + 1
1916                ' skip comments
1917                Do While i <= toks.Count And toks(i)(0) = "COMMENT"
1918                    i = i + 1
1919                Loop
1920                If i > toks.Count Then err.Raise vbObjectError + 1, , "Unexpected end after
                    fun"
1921                Dim fname As String: fname = toks(i)(1)
1922                i = i + 1
1923                Do While i <= toks.Count And toks(i)(0) = "COMMENT"
1924                    i = i + 1
1925                Loop
1926
1927                Dim argList As New Collection
1928                If i <= toks.Count And toks(i)(0) = "PAREN" And toks(i)(1) = "(" Then
1929                    i = i + 1
1930                    Do While i <= toks.Count And Not (toks(i)(0) = "PAREN" And toks(i)(1) =
                        ")")
1931                        If toks(i)(0) = "COMMENT" Then
1932                            i = i + 1
1933                        Else
1934                            If toks(i)(0) = "IDENT" Then argList.Add toks(i)(1)
1935                            i = i + 1
1936                        End If
1937                    Loop
1938                    If i <= toks.Count And toks(i)(0) = "PAREN" And toks(i)(1) = ")" Then i
                        = i + 1
1939                End If
1940
1941                Do While i <= toks.Count And toks(i)(0) = "COMMENT"
1942                    i = i + 1
1943                Loop
1944                If i > toks.Count Or Not (toks(i)(0) = "PAREN" And toks(i)(1) = "{") Then
1945                    err.Raise vbObjectError + 2, , "Expected function body"
1946                End If
1947                i = i + 1
1948                Dim depth As Long: depth = 0
1949                Dim bodyTokens As New Collection
1950                Do While i <= toks.Count
1951                    If toks(i)(0) = "PAREN" Then
1952                        If toks(i)(1) = "{" Then
1953                            depth = depth + 1
1954                        ElseIf toks(i)(1) = "}" Then
1955                            If depth = 0 Then
1956                                Exit Do
1957                            Else
1958                                depth = depth - 1
1959                            End If
1960                        End If
1961                    End If
1962                    bodyTokens.Add toks(i)
1963                    i = i + 1
1964                Loop
1965
1966                Dim funcIdx As Long
1967                funcIdx = CompileFunctionAST(fname, argList, bodyTokens)
1968                GLOBALS_.gFuncTable.Add fname, funcIdx
```

```
1969
1970                Dim pa() As String
1971                If argList.Count > 0 Then ReDim pa(0 To argList.Count - 1)
1972                Dim j As Long
1973                For j = 1 To argList.Count: pa(j - 1) = argList(j): Next j
1974                GLOBALS_.gFuncParams.Add fname, pa
1975
1976                If i <= toks.Count Then i = i + 1 ' skip the trailing '}' if present
1977                GoTo Compiler_MainLoop
1978            Else
1979                ' collect a top-level statement tokens (nesting-aware) then parse to AST node
1980                Dim braceDepth As Long: braceDepth = 0
1981                Dim parenDepth As Long: parenDepth = 0
1982                Dim bracketDepth As Long: bracketDepth = 0
1983
1984                Set stmtTokens = New Collection
1985
1986                Do While i <= toks.Count
1987                    If toks(i)(0) = "COMMENT" Then
1988                        i = i + 1
1989                        GoTo CollectNextTop
1990                    End If
1991
1992                    If toks(i)(0) = "PAREN" Then
1993                        Select Case toks(i)(1)
1994                            Case "{"
1995                                braceDepth = braceDepth + 1
1996                            Case "}"
1997                                If braceDepth > 0 Then
1998                                    braceDepth = braceDepth - 1
1999                                End If
2000                            Case "("
2001                                parenDepth = parenDepth + 1
2002                            Case ")"
2003                                If parenDepth > 0 Then
2004                                    parenDepth = parenDepth - 1
2005                                End If
2006                            Case "["
2007                                bracketDepth = bracketDepth + 1
2008                            Case "]"
2009                                If bracketDepth > 0 Then
2010                                    bracketDepth = bracketDepth - 1
2011                                End If
2012                        End Select
2013                    End If
2014
2015                    ' Top-level statement separator: semicolon (commas are argument
                        separators) if we are not nested
2016                    If toks(i)(0) = "SEP" And toks(i)(1) = ";" And braceDepth = 0 And
                        parenDepth = 0 And bracketDepth = 0 Then
2017                        i = i + 1
2018                        Exit Do
2019                    End If
2020
2021                    stmtTokens.Add toks(i)
2022                    i = i + 1
2023    CollectNextTop:
2024                Loop
2025
2026                If stmtTokens.Count > 0 Then
2027                    Set node = ParseStatementTokensToAST(stmtTokens)
2028                    If Not node Is Nothing Then
2029                        ' add a clone to avoid accidental reference reuse
2030                        stmtsAST.Add node.Clone
2031                    End If
2032                End If
2033                GoTo Compiler_MainLoop
2034            End If
2035        Loop
```

```
2036        Dim pIndex As Long
2037        Dim normalizedStmts As Collection
2038        ' Normalize assignment-like expression-statements into proper Assign AST nodes
2039        Set normalizedStmts = NormalizeAssignsInStmts(stmtsAST)
2040        ' Normalize compound assignments (+=, -=, etc.)
2041        Set normalizedStmts = NormalizeCompoundAssigns(normalizedStmts)
2042        pIndex = GLOBALS_.gPrograms.Count + 1
2043        GLOBALS_.gPrograms.Add Array(progName, normalizedStmts, progScope.Raw)
2044        GLOBALS_.ProgramCache.Add src, pIndex
2045        CompileProgram = pIndex
2046    End Function
2047
2048    ' Compile a function body (AST) and register it as separate program
2049    Private Function CompileFunctionAST(fname As String, argList As Collection, bodyTokens
      As Collection) As Long
2050        Dim funcScope As ScopeStack
2051        Dim i As Long
2052        Dim bodyStmtsAST As Collection
2053        Set funcScope = New ScopeStack
2054        funcScope.Push
2055        For i = 1 To argList.Count
2056            funcScope.SetValue argList(i), Empty
2057        Next i
2058        Dim idx As Long
2059        Set bodyStmtsAST = ParseTokensToAST(bodyTokens)
2060        ' Normalize assignments inside function body as well
2061        Set bodyStmtsAST = NormalizeAssignsInStmts(bodyStmtsAST)
2062        idx = GLOBALS_.gPrograms.Count + 1
2063        GLOBALS_.gPrograms.Add Array(fname, bodyStmtsAST, funcScope.Raw)
2064        CompileFunctionAST = idx
2065    End Function
2066
2067    ' Parse a list of tokens representing a block into AST statements (returns Collection of
      Map nodes)
2068    Private Function ParseTokensToAST(toks As Collection) As Collection
2069        Dim out As Collection
2070        Dim stmtTokens As Collection
2071        Set out = New Collection
2072        Dim i As Long: i = 1
2073        Dim closedTopLevelBlock As Boolean
2074
2075    ParseMain:
2076        Do While i <= toks.Count
2077            If toks(i)(0) = "COMMENT" Then
2078                i = i + 1
2079                GoTo ParseMain
2080            End If
2081            Set stmtTokens = New Collection
2082            Dim braceDepth As Long: braceDepth = 0
2083            Dim parenDepth As Long: parenDepth = 0
2084            Dim bracketDepth As Long: bracketDepth = 0
2085
2086    ParseInner:
2087            Do While i <= toks.Count
2088                If toks(i)(0) = "COMMENT" Then
2089                    i = i + 1
2090                    GoTo ParseInner
2091                End If
2092                If toks(i)(0) = "PAREN" Then
2093                    Select Case toks(i)(1)
2094                        Case "{"
2095                            braceDepth = braceDepth + 1
2096                        Case "}"
2097                            ' decrement only if we have an inner brace to close
2098                            If braceDepth > 0 Then
2099                                braceDepth = braceDepth - 1
2100                            End If
2101                            ' If we've just closed a top-level block (braceDepth now 0),
2102                            ' that usually signals the end of a statement (e.g. end of an
```

```
                        if/for/while block).
2103                    ' Finalize the current statement tokens so adjacent statements
                        are parsed separately.
2104                    If braceDepth = 0 Then
2105                        ' include this '}' token into the current stmtTokens and
                            finish the statement
2106                        stmtTokens.Add toks(i)
2107                        i = i + 1
2108                        ' mark that we closed a top-level block for follow-up checks
2109                        closedTopLevelBlock = True
2110                        Exit Do
2111                    End If
2112                Case "("
2113                    parenDepth = parenDepth + 1
2114                Case ")"
2115                    If parenDepth > 0 Then
2116                        parenDepth = parenDepth - 1
2117                    End If
2118                Case "["
2119                    bracketDepth = bracketDepth + 1
2120                Case "]"
2121                    If bracketDepth > 0 Then
2122                        bracketDepth = bracketDepth - 1
2123                    End If
2124            End Select
2125        End If
2126
2127        ' Only a semicolon (;) is a top-level statement separator.
2128        If toks(i)(0) = "SEP" And toks(i)(1) = ";" And braceDepth = 0 And parenDepth
            = 0 And bracketDepth = 0 Then
2129            i = i + 1
2130            Exit Do
2131        End If
2132
2133        ' default: append current token and continue
2134        stmtTokens.Add toks(i)
2135        i = i + 1
2136    Loop
2137
2138    If stmtTokens.Count > 0 Then
2139        Dim tmpNode As Map
2140        Set tmpNode = ParseStatementTokensToAST(stmtTokens)
2141        If Not tmpNode Is Nothing Then
2142            ' If we closed a top-level block and the next token is not a semicolon,
2143            ' warn the user (do not silently swallow).
2144            If closedTopLevelBlock Then
2145                If i <= toks.Count Then
2146                    If Not (toks(i)(0) = "SEP" And toks(i)(1) = ";") Then
2147                        ' If the next token is not a comment and not end-of-input,
                            log warning.
2148                        If Not (toks(i)(0) = "COMMENT") Then
2149                            On Error Resume Next
2150                            ' best-effort: add to runtime log if present, and print
                                to Immediate
2151                            If Not GLOBALS_.gRuntimeLog Is Nothing Then
                                GLOBALS_.gRuntimeLog.Add "COMPILER: missing ';' after
                                block near token index " & CStr(i)
2152                            On Error GoTo 0
2153                        End If
2154                    Else
2155                        ' consume explicit semicolon
2156                        i = i + 1
2157                    End If
2158                End If
2159                closedTopLevelBlock = False
2160            End If
2161            out.Add tmpNode.Clone
2162        End If
2163    End If
```

```
2164            GoTo ParseMain
2165        Loop
2166
2167        Set ParseTokensToAST = out
2168    End Function
2169
2170    ' Parse a single statement token collection into an AST node (Map)
2171    Private Function ParseStatementTokensToAST(stmtTokens As Collection) As Map
2172        If stmtTokens.Count = 0 Then
2173            Set ParseStatementTokensToAST = Nothing
2174            Exit Function
2175        End If
2176
2177        ' quick check for print
2178        If stmtTokens.Count >= 2 Then
2179            If stmtTokens(1)(0) = "IDENT" And LCase$(stmtTokens(1)(1)) = "print" Then
2180                Dim k As Long: k = 2
2181                Do While k <= stmtTokens.Count And Not (stmtTokens(k)(0) = "PAREN" And
                    stmtTokens(k)(1) = "(")
2182                    k = k + 1
2183                Loop
2184                If k <= stmtTokens.Count Then
2185                    Dim opened As Long: opened = 1
2186                    k = k + 1
2187                    Dim innerToks As Collection
2188                    Set innerToks = New Collection
2189                    Do While k <= stmtTokens.Count And opened > 0
2190                        If stmtTokens(k)(0) = "PAREN" Then
2191                            If stmtTokens(k)(1) = "(" Then
2192                                opened = opened + 1
2193                            ElseIf stmtTokens(k)(1) = ")" Then
2194                                opened = opened - 1
2195                            End If
2196                        End If
2197                        If opened > 0 Then innerToks.Add stmtTokens(k)
2198                        k = k + 1
2199                    Loop
2200                    Dim args As Collection: Set args = ParseArgsTokensToExprNodes(innerToks)
2201                    Dim node As Map: Set node = MakeNode("Print")
2202                    node.SetValue "args", args
2203                    Set ParseStatementTokensToAST = node
2204                    Exit Function
2205                End If
2206            End If
2207        End If
2208
2209        ' keywords
2210        If stmtTokens(1)(0) = "IDENT" Then
2211            Dim kw As String: kw = LCase$(stmtTokens(1)(1))
2212            Select Case kw
2213                Case "if"
2214                    Set ParseStatementTokensToAST = ParseIfAST(stmtTokens)
2215                    Exit Function
2216                Case "for"
2217                    Set ParseStatementTokensToAST = ParseForAST(stmtTokens)
2218                    Exit Function
2219                Case "while"
2220                    Set ParseStatementTokensToAST = ParseWhileAST(stmtTokens)
2221                    Exit Function
2222                Case "try"
2223                    Set ParseStatementTokensToAST = ParseTryCatchAST(stmtTokens)
2224                    Exit Function
2225                Case "switch"
2226                    Set ParseStatementTokensToAST = ParseSwitchAST(stmtTokens)
2227                    Exit Function
2228                Case "break"
2229                    Dim b As Map: Set b = MakeNode("Break")
2230                    Set ParseStatementTokensToAST = b
2231                    Exit Function
```

```
2232                Case "continue"
2233                    Dim c As Map: Set c = MakeNode("Continue")
2234                    Set ParseStatementTokensToAST = c
2235                    Exit Function
2236                Case "return"
2237                    Dim rv As Map: Set rv = MakeNode("Return")
2238                    If stmtTokens.Count >= 2 Then
2239                        Dim rhs As New Collection, m As Long
2240                        For m = 2 To stmtTokens.Count: rhs.Add stmtTokens(m): Next m
2241                        Dim rexpr As Map: Set rexpr = ParseExprTokensToNode(rhs)
2242                        rv.SetValue "expr", rexpr
2243                    Else
2244                        rv.SetValue "expr", Nothing
2245                    End If
2246                    Set ParseStatementTokensToAST = rv
2247                    Exit Function
2248            End Select
2249        End If
2250
2251        ' assignment: find top-level "=" operator position
2252        Dim assignPos As Long: assignPos = 0
2253        Dim ii As Long
2254        Dim depthB As Long, depthP As Long, depthBr As Long
2255        For ii = 1 To stmtTokens.Count
2256            If stmtTokens(ii)(0) = "PAREN" Then
2257                Select Case stmtTokens(ii)(1)
2258                    Case "{"
2259                        depthB = depthB + 1
2260                    Case "}"
2261                        If depthB > 0 Then depthB = depthB - 1
2262                    Case "("
2263                        depthP = depthP + 1
2264                    Case ")"
2265                        If depthP > 0 Then depthP = depthP - 1
2266                    Case "["
2267                        depthBr = depthBr + 1
2268                    Case "]"
2269                        If depthBr > 0 Then depthBr = depthBr - 1
2270                End Select
2271            End If
2272            If stmtTokens(ii)(0) = "OP" And stmtTokens(ii)(1) = "=" And depthB = 0 And
                depthP = 0 And depthBr = 0 Then
2273                assignPos = ii
2274                Exit For
2275            End If
2276        Next ii
2277
2278        If assignPos > 0 Then
2279            Dim lhsT As Collection, rhsT As Collection, t As Long
2280            Set lhsT = New Collection: Set rhsT = New Collection
2281            For t = 1 To assignPos - 1: lhsT.Add stmtTokens(t): Next t
2282            For t = assignPos + 1 To stmtTokens.Count: rhsT.Add stmtTokens(t): Next t
2283            Dim lhsNode As Map
2284            If lhsT.Count = 1 And lhsT(1)(0) = "IDENT" Then
2285                Dim nm As String: nm = lhsT(1)(1)
2286                If InStr(nm, "[") > 0 Then
2287                    Dim baseName As String: baseName = left$(nm, InStr(nm, "[") - 1)
2288                    Dim idxStr As String: idxStr = Mid$(nm, InStr(nm, "[") + 1, Len(nm) -
                        InStr(nm, "[") - 1)
2289                    Dim idxNode As Map: Set idxNode = ParseExprFromStringToNode(idxStr)
2290                    Set lhsNode = MakeNode("Index")
2291                    Dim tmpVar As Map: Set tmpVar = MakeNode("Variable")
2292                    tmpVar.SetValue "name", baseName
2293                    lhsNode.SetValue "base", tmpVar
2294                    lhsNode.SetValue "index", idxNode
2295                Else
2296                    Set lhsNode = MakeNode("Variable")
2297                    lhsNode.SetValue "name", nm
2298                End If
```

```
2299          Else
2300              Set lhsNode = ParseExprTokensToNode(lhsT)
2301          End If
2302          Dim rhsNode As Map: Set rhsNode = ParseExprTokensToNode(rhsT)
2303          Dim asn As Map: Set asn = MakeNode("Assign")
2304          asn.SetValue "left", lhsNode
2305          asn.SetValue "right", rhsNode
2306          Set ParseStatementTokensToAST = asn
2307          Exit Function
2308      End If
2309
2310      ' fallback: expression statement
2311      Dim exprNode As Map
2312      Set exprNode = ParseExprTokensToNode(stmtTokens)
2313      Dim es As Map: Set es = MakeNode("ExprStmt")
2314      es.SetValue "expr", exprNode
2315      Set ParseStatementTokensToAST = es
2316  End Function
2317
2318  ' ---------------------
2319  ' Expression parsing -> Expr AST nodes
2320  ' ---------------------
2321
2322  Private Function ParseArgsTokensToExprNodes(argTokens As Collection) As Collection
2323      Dim out As New Collection
2324      If argTokens.Count = 0 Then
2325          Set ParseArgsTokensToExprNodes = out
2326          Exit Function
2327      End If
2328      Dim i As Long: i = 1
2329      Dim braceDepth As Long: braceDepth = 0
2330      Dim parenDepth As Long: parenDepth = 0
2331      Dim bracketDepth As Long: bracketDepth = 0
2332      Dim cur As New Collection
2333  ParseArgsMain:
2334      Do While i <= argTokens.Count
2335          If argTokens(i)(0) = "PAREN" Then
2336              Select Case argTokens(i)(1)
2337                  Case "{"
2338                      braceDepth = braceDepth + 1
2339                  Case "}"
2340                      If braceDepth > 0 Then braceDepth = braceDepth - 1
2341                  Case "("
2342                      parenDepth = parenDepth + 1
2343                  Case ")"
2344                      If parenDepth > 0 Then parenDepth = parenDepth - 1
2345                  Case "["
2346                      bracketDepth = bracketDepth + 1
2347                  Case "]"
2348                      If bracketDepth > 0 Then bracketDepth = bracketDepth - 1
2349              End Select
2350          End If
2351          ' Argument/element separator: comma only (semicolon is NOT an argument separator)
2352          If argTokens(i)(0) = "SEP" And argTokens(i)(1) = "," And braceDepth = 0 And
              parenDepth = 0 And bracketDepth = 0 Then
2353              If cur.Count > 0 Then out.Add ParseExprTokensToNode(cur)
2354              Set cur = New Collection
2355              i = i + 1
2356              GoTo ParseArgsMain
2357          End If
2358
2359          cur.Add argTokens(i)
2360          i = i + 1
2361      Loop
2362      If cur.Count > 0 Then out.Add ParseExprTokensToNode(cur)
2363      Set ParseArgsTokensToExprNodes = out
2364  End Function
2365
2366  Private Function ParseExprFromStringToNode(exprStr As String) As Map
```

```
2367        Dim p As Parser
2368        Set p = New Parser
2369        Dim toks As Collection
2370        Set toks = p.Tokenize(exprStr)
2371        Set ParseExprFromStringToNode = ParseExprTokensToNode(toks)
2372    End Function
2373
2374    Private Function ParseExprTokensToNode(toks As Collection) As Map
2375        Dim n As Long: n = toks.Count
2376        If n = 0 Then
2377            Set ParseExprTokensToNode = MakeNode("Literal"): ParseExprTokensToNode.SetValue
            "value", Empty
2378            Exit Function
2379        End If
2380
2381        Dim types() As String, vals() As Variant, i As Long
2382        ReDim types(0 To n - 1)
2383        ReDim vals(0 To n - 1)
2384        For i = 1 To n
2385            types(i - 1) = toks(i)(0)
2386            vals(i - 1) = toks(i)(1)
2387        Next i
2388
2389        Dim idx As Long: idx = 0
2390        Set ParseExprTokensToNode = ParseTernaryNode(types, vals, n, idx)
2391    End Function
2392
2393    ' Recursive descent building nodes
2394
2395    Private Function ParseTernaryNode(types() As String, vals() As Variant, n As Long, ByRef
    idx As Long) As Map
2396        ' parse cond ? trueExpr : falseExpr (right-assoc)
2397        Dim cond As Map
2398        Set cond = ParseLogicalOrNode(types, vals, n, idx)
2399        If idx < n Then
2400            If types(idx) = "OP" And vals(idx) = "?" Then
2401                idx = idx + 1
2402                Dim trueExpr As Map
2403                Set trueExpr = ParseTernaryNode(types, vals, n, idx) ' right-assoc: allow
                nested ternary
2404                ' expect ':'
2405                If idx < n And types(idx) = "OP" And vals(idx) = ":" Then
2406                    idx = idx + 1
2407                Else
2408                    err.Raise vbObjectError + 8001, "Compiler.ParseTernaryNode", "Expected
                    ':' in ternary expression"
2409                End If
2410                Dim falseExpr As Map
2411                Set falseExpr = ParseTernaryNode(types, vals, n, idx)
2412                Dim tn As New Map
2413                tn.Add "type", "Ternary"
2414                tn.SetValue "cond", cond
2415                tn.SetValue "trueExpr", trueExpr
2416                tn.SetValue "falseExpr", falseExpr
2417                Set ParseTernaryNode = tn
2418                Exit Function
2419            End If
2420        End If
2421        Set ParseTernaryNode = cond
2422    End Function
2423
2424    Private Function ParseLogicalOrNode(types() As String, vals() As Variant, n As Long,
    ByRef idx As Long) As Map
2425        Dim left_ As Map
2426        Dim right_ As Map
2427        Dim node As Map
2428        Dim op As String
2429        Set left_ = ParseLogicalAndNode(types, vals, n, idx)
2430        Do While idx < n
```

```
2431            If Not (types(idx) = "OP" And vals(idx) = "||") Then Exit Do
2432            op = vals(idx)
2433            idx = idx + 1
2434            Set right_ = ParseLogicalAndNode(types, vals, n, idx)
2435            Set node = MakeNode("Binary")
2436            node.SetValue "op", op
2437            node.SetValue "left", left_.Clone
2438            node.SetValue "right", right_.Clone
2439            Set left_ = node
2440        Loop
2441        Set ParseLogicalOrNode = left_.Clone
2442        Set left_ = Nothing
2443        Set right_ = Nothing
2444        Set node = Nothing
2445    End Function
2446
2447    Private Function ParseLogicalAndNode(types() As String, vals() As Variant, n As Long,
      ByRef idx As Long) As Map
2448        Dim left_ As Map
2449        Dim right_ As Map
2450        Dim node As Map
2451        Set left_ = ParseEqualityNode(types, vals, n, idx)
2452        Do While idx < n
2453            If Not (types(idx) = "OP" And vals(idx) = "&&") Then Exit Do
2454            idx = idx + 1
2455            Set right_ = ParseEqualityNode(types, vals, n, idx)
2456            Set node = MakeNode("Binary")
2457            node.SetValue "op", "&&"
2458            node.SetValue "left", left_.Clone
2459            node.SetValue "right", right_.Clone
2460            Set left_ = node
2461        Loop
2462        Set ParseLogicalAndNode = left_.Clone
2463        Set left_ = Nothing
2464        Set right_ = Nothing
2465        Set node = Nothing
2466    End Function
2467
2468    Private Function ParseEqualityNode(types() As String, vals() As Variant, n As Long,
      ByRef idx As Long) As Map
2469        Dim left_ As Map
2470        Dim right_ As Map
2471        Dim node As Map
2472        Dim op As String
2473        Set left_ = ParseRelationalNode(types, vals, n, idx)
2474        Do While idx < n
2475            If Not (types(idx) = "OP" And (vals(idx) = "==" Or vals(idx) = "=" Or vals(idx)
            = "!=")) Then Exit Do
2476            op = vals(idx)
2477            idx = idx + 1
2478            Set right_ = ParseRelationalNode(types, vals, n, idx)
2479            Set node = MakeNode("Binary")
2480            node.SetValue "op", op
2481            node.SetValue "left", left_.Clone
2482            node.SetValue "right", right_.Clone
2483            Set left_ = node
2484        Loop
2485        Set ParseEqualityNode = left_.Clone
2486        Set left_ = Nothing
2487        Set right_ = Nothing
2488        Set node = Nothing
2489    End Function
2490
2491    Private Function ParseRelationalNode(types() As String, vals() As Variant, n As Long,
      ByRef idx As Long) As Map
2492        Dim left_ As Map
2493        Dim right_ As Map
2494        Dim node As Map
2495        Dim op As String
```

```
2496        Set left_ = ParseAddNode(types, vals, n, idx)
2497        Do While idx < n
2498            If Not (types(idx) = "OP" And (vals(idx) = "<" Or vals(idx) = ">" Or vals(idx) =
                "<=" Or vals(idx) = ">=")) Then Exit Do
2499            op = vals(idx)
2500            idx = idx + 1
2501            Set right_ = ParseAddNode(types, vals, n, idx)
2502            Set node = MakeNode("Binary")
2503            node.SetValue "op", op
2504            node.SetValue "left", left_.Clone
2505            node.SetValue "right", right_.Clone
2506            Set left_ = node
2507        Loop
2508        Set ParseRelationalNode = left_.Clone
2509        Set left_ = Nothing
2510        Set right_ = Nothing
2511        Set node = Nothing
2512    End Function
2513
2514    Private Function ParseAddNode(types() As String, vals() As Variant, n As Long, ByRef idx
        As Long) As Map
2515        Dim left_ As Map
2516        Dim op As String
2517        Dim right_ As Map
2518        Dim node As Map
2519        Set left_ = ParseMulNode(types, vals, n, idx)
2520        Do While idx < n
2521            If Not (types(idx) = "OP" And (vals(idx) = "+" Or vals(idx) = "-")) Then Exit Do
2522            op = vals(idx)
2523            idx = idx + 1
2524            Set right_ = ParseMulNode(types, vals, n, idx)
2525            Set node = MakeNode("Binary")
2526            node.SetValue "op", op
2527            node.SetValue "left", left_.Clone
2528            node.SetValue "right", right_.Clone
2529            Set left_ = node
2530        Loop
2531        Set ParseAddNode = left_.Clone
2532        Set left_ = Nothing
2533        Set right_ = Nothing
2534        Set node = Nothing
2535    End Function
2536
2537    Private Function ParseMulNode(types() As String, vals() As Variant, n As Long, ByRef idx
        As Long) As Map
2538        Dim left_ As Map
2539        Dim right_ As Map
2540        Dim node As Map
2541        Dim op As String
2542        Set left_ = ParsePowNode(types, vals, n, idx)
2543        Do While idx < n
2544            If Not (types(idx) = "OP" And (vals(idx) = "*" Or vals(idx) = "/" Or vals(idx) =
                "%")) Then Exit Do
2545            op = vals(idx)
2546            idx = idx + 1
2547            Set right_ = ParsePowNode(types, vals, n, idx)
2548            Set node = MakeNode("Binary")
2549            node.SetValue "op", op
2550            node.SetValue "left", left_.Clone
2551            node.SetValue "right", right_.Clone
2552            Set left_ = node
2553        Loop
2554        Set ParseMulNode = left_.Clone
2555        Set left_ = Nothing
2556        Set right_ = Nothing
2557        Set node = Nothing
2558    End Function
2559
2560    Private Function ParsePowNode(types() As String, vals() As Variant, n As Long, ByRef idx
```

```
As Long) As Map
    Dim left_ As Map
    Dim right_ As Map
    Dim node As Map
    Set left_ = ParseUnaryNode(types, vals, n, idx)
    Do While idx < n
        ' Right-associative exponentiation:
        ' if we see '^' after the left operand, parse the RHS with ParsePow
        ' so a ^ b ^ c => a ^ (b ^ c)
        If Not (types(idx) = "OP" And vals(idx) = "^") Then Exit Do
        idx = idx + 1
        ' parse the right-hand side with ParsePow to ensure right-assoc
        Set right_ = ParsePowNode(types, vals, n, idx)
        Set node = MakeNode("Binary")
        node.SetValue "op", "^"
        node.SetValue "left", left_.Clone
        node.SetValue "right", right_.Clone
        Set left_ = node
    Loop
    Set ParsePowNode = left_.Clone
    Set left_ = Nothing
    Set right_ = Nothing
    Set node = Nothing
End Function

Private Function ParseUnaryNode(types() As String, vals() As Variant, n As Long, ByRef
idx As Long) As Map
    If idx < n And types(idx) = "OP" Then
        Dim op As String: op = vals(idx)
        If op = "!" Or op = "-" Then
            idx = idx + 1
            Dim v As Map: Set v = ParseUnaryNode(types, vals, n, idx)
            Dim node As Map: Set node = MakeNode("Unary")
            node.SetValue "op", op
            node.SetValue "expr", v
            Set ParseUnaryNode = node
            Exit Function
        End If
    End If
    Set ParseUnaryNode = ParsePrimaryNode(types, vals, n, idx)
End Function

Private Function ParsePrimaryNode(types() As String, vals() As Variant, n As Long, ByRef
idx As Long) As Map
    If idx >= n Then
        Set ParsePrimaryNode = MakeNode("Literal")
        ParsePrimaryNode.SetValue "value", Empty
        Exit Function
    End If
    Dim t As String: t = types(idx)
    Dim v As Variant: v = vals(idx)

    Select Case t
        Case "NUMBER"
            Dim nNode As Map: Set nNode = MakeNode("Literal")
            nNode.SetValue "value", CDbl(v)
            idx = idx + 1
            Set ParsePrimaryNode = nNode
            Exit Function
        Case "STRING"
            Dim sNode As Map: Set sNode = MakeNode("Literal")
            sNode.SetValue "value", v
            idx = idx + 1
            Set ParsePrimaryNode = sNode
            Exit Function
        Case "VBEXPR"
            ' Special token produced by the parser for @(...) or direct VBAexpressions
            block.
            ' Create a VBExpr node with the raw expression string (to be evaluated by
```

```
            VBAexpressions at runtime).
2626        Dim vbNode As Map: Set vbNode = MakeNode("VBExpr")
2627        vbNode.SetValue "expr", CStr(v)
2628        idx = idx + 1
2629        Set ParsePrimaryNode = vbNode
2630        Exit Function
2631    Case "IDENT"
2632        Dim name As String: name = v
2633        Dim idxNode As Map
2634        Dim idxExpr As Map
2635        If LCase$(name) = "true" Then
2636            Dim bt As Map: Set bt = MakeNode("Literal")
2637            bt.SetValue "value", True
2638            idx = idx + 1
2639            Set ParsePrimaryNode = bt
2640            Exit Function
2641        ElseIf LCase$(name) = "false" Then
2642            Dim bf As Map: Set bf = MakeNode("Literal")
2643            bf.SetValue "value", False
2644            idx = idx + 1
2645            Set ParsePrimaryNode = bf
2646            Exit Function
2647        End If
2648
2649        ' Expression-level anonymous function literal: fun (p1, p2) { ... }
2650        ' This reuses the same 'fun' token used for top-level function declarations
            but emits
2651        ' a FuncLiteral node usable inside expressions.
2652        If LCase$(name) = "fun" Then
2653            ' advance past 'fun'
2654            idx = idx + 1
2655            ' expect parameter list
2656            Dim params As Collection
2657            Set params = New Collection
2658            If idx < n And types(idx) = "PAREN" And vals(idx) = "(" Then
2659                idx = idx + 1
2660                Do While idx < n And Not (types(idx) = "PAREN" And vals(idx) = ")")
2661                    If types(idx) = "IDENT" Then
2662                        params.Add CStr(vals(idx))
2663                        idx = idx + 1
2664                        If idx < n And types(idx) = "SEP" And vals(idx) = "," Then
                            idx = idx + 1
2665                    ElseIf types(idx) = "SEP" And vals(idx) = "," Then
2666                        idx = idx + 1
2667                    Else
2668                        ' skip unexpected tokens until ')'
2669                        idx = idx + 1
2670                    End If
2671                Loop
2672                If idx < n And types(idx) = "PAREN" And vals(idx) = ")" Then idx =
                    idx + 1
2673            End If
2674
2675            ' parse body { ... }
2676            If idx < n And types(idx) = "PAREN" And vals(idx) = "{" Then
2677                idx = idx + 1
2678                Dim depthBody As Long: depthBody = 0
2679                Dim bodyTok As Collection
2680                Set bodyTok = New Collection
2681                Do While idx < n
2682                    If types(idx) = "PAREN" Then
2683                        If vals(idx) = "{" Then
2684                            depthBody = depthBody + 1
2685                        ElseIf vals(idx) = "}" Then
2686                            If depthBody = 0 Then
2687                                Exit Do
2688                            Else
2689                                depthBody = depthBody - 1
2690                            End If
```

```
                            End If
                        End If
                        bodyTok.Add Array(types(idx), vals(idx))
                        idx = idx + 1
                    Loop
                    ' consume closing '}'
                    If idx < n And types(idx) = "PAREN" And vals(idx) = "}" Then idx =
                    idx + 1
                    Dim bodyStmts As Collection: Set bodyStmts =
                    ParseTokensToAST(bodyTok)
                    Dim fnode As Map: Set fnode = MakeNode("FuncLiteral")
                    fnode.SetValue "params", params
                    fnode.SetValue "body", bodyStmts
                    Set ParsePrimaryNode = fnode.Clone
                    Exit Function
                Else
                    ' no body found -> emit empty function literal
                    Dim fnode2 As Map: Set fnode2 = MakeNode("FuncLiteral")
                    fnode2.SetValue "params", params
                    Dim emptyCol As New Collection
                    fnode2.SetValue "body", emptyCol
                    Set ParsePrimaryNode = fnode2
                    Exit Function
                End If
            End If

            Dim lenCall As Map
            Dim builtinLen As Map
            Dim al As Collection
            ' Handle collapsed identifiers with array/index and optional trailing
            ".length"
            ' Examples:
            '   a[3]
            '   a[3].length
            If InStr(name, "[") > 0 Then
                Dim base As String: base = left$(name, InStr(name, "[") - 1)
                ' Find matching closing bracket for the first '[' (supports nested
                brackets)
                Dim posOpen As Long: posOpen = InStr(name, "[")
                Dim posClose As Long: posClose = 0
                Dim depthBr As Long: depthBr = 0
                Dim iCh As Long
                For iCh = posOpen To Len(name)
                    Dim ch As String: ch = Mid$(name, iCh, 1)
                    If ch = "[" Then
                        depthBr = depthBr + 1
                    ElseIf ch = "]" Then
                        depthBr = depthBr - 1
                        If depthBr = 0 Then
                            posClose = iCh
                            Exit For
                        End If
                    End If
                Next iCh
                If posClose = 0 Then
                    err.Raise vbObjectError + 8002, "Compiler.ParsePrimary", "Invalid
                    collapsed identifier: missing ']'"
                End If
                Dim idxStr As String: idxStr = Mid$(name, posOpen + 1, posClose -
                posOpen - 1)
                Dim idxNodeFromStr As Map: Set idxNodeFromStr =
                ParseExprFromStringToNode(idxStr)
                Dim inNode As Map: Set inNode = MakeNode("Index")
                Dim vn As Map: Set vn = MakeNode("Variable")
                vn.SetValue "name", base
                inNode.SetValue "base", vn
                inNode.SetValue "index", idxNodeFromStr
                idx = idx + 1
                ' If next tokens are ". length" treat as builtin length call on the
```

```
                    indexed result.
                    If idx < n Then
                        If types(idx) = "SYM" And vals(idx) = "." Then
                            If (idx + 1) < n And types(idx + 1) = "IDENT" And
                            LCase$(CStr(vals(idx + 1))) = "length" Then
                                ' consume '.' and 'length'
                                idx = idx + 2
                                Set lenCall = MakeNode("Call")
                                ' set both 'callee' and 'name' for maximum compatibility
                                with VM variants
                                Set builtinLen = MakeNode("Variable")
                                builtinLen.SetValue "name", ".__len__"
                                lenCall.SetValue "callee", builtinLen
                                lenCall.SetValue "name", ".__len__"
                                Set al = New Collection
                                al.Add inNode
                                lenCall.SetValue "args", al
                                Set ParsePrimaryNode = lenCall
                                Exit Function
                            End If
                        End If
                    End If
                    Set ParsePrimaryNode = inNode
                    Exit Function
                End If
                ' if collapsed dotted/indexed form => convert into AST nodes
                If InStr(name, ".") Then
                    Dim complexNode As Map
                    Set complexNode = ParseCollapsedIdentToNode(name)
                    If Not complexNode Is Nothing Then
                        idx = idx + 1
                        ' return currentNode (no postfix loop needed because we've already
                        built the chain)
                        Set ParsePrimaryNode = complexNode
                        Exit Function
                    End If
                End If
                ' create initial variable node for an IDENT and then apply postfix operators:
                Dim currentNode As Map: Set currentNode = MakeNode("Variable")
                currentNode.SetValue "name", name
                idx = idx + 1

                ' Postfix loop: handle .prop, [...], func calls (...) as postfixes chaining
                onto currentNode
                Do While idx < n
                    ' member access a.b
                    If types(idx) = "SYM" And vals(idx) = "." Then
                        idx = idx + 1
                        If idx < n And types(idx) = "IDENT" Then
                            Dim mem As Map: Set mem = MakeNode("Member")
                            mem.SetValue "base", currentNode
                            mem.SetValue "prop", CStr(vals(idx))
                            Set currentNode = mem.Clone
                            idx = idx + 1
                            ' continue loop
                        Else
                            ' invalid member access; stop postfixing
                            Exit Do
                        End If
                    ' index access a[expr]
                    ElseIf types(idx) = "PAREN" And vals(idx) = "[" Then
                        idx = idx + 1
                        Dim idxTok As Collection
                        Set idxTok = New Collection
                        Dim depthIdx As Long: depthIdx = 0
                        Do While idx < n
                            If types(idx) = "PAREN" Then
                                If vals(idx) = "[" Then
                                    depthIdx = depthIdx + 1
```

```
                            ElseIf vals(idx) = "]" Then
                                If depthIdx = 0 Then
                                    Exit Do
                                Else
                                    depthIdx = depthIdx - 1
                                End If
                            End If
                        End If
                        idxTok.Add Array(types(idx), vals(idx))
                        idx = idx + 1
                    Loop
                    Dim idxExprNode As Map: Set idxExprNode =
                    ParseExprTokensToNode(idxTok)
                    If idx < n And types(idx) = "PAREN" And vals(idx) = "]" Then idx =
                    idx + 1
                    Dim indexNode As Map: Set indexNode = MakeNode("Index")
                    indexNode.SetValue "base", currentNode
                    indexNode.SetValue "index", idxExprNode
                    Set currentNode = indexNode.Clone
                ' function / method call: ( arglist )
                ElseIf types(idx) = "PAREN" And vals(idx) = "(" Then
                    ' parse args with nested depth counters (borrowed pattern from
                    previous call parsing logic)
                    idx = idx + 1
                    Dim argNodes As Collection
                    Dim argTok As Collection
                    Dim argTokDepthParen As Long
                    Dim argTokDepthBr As Long
                    Dim argTokDepthB As Long
                    Set argNodes = New Collection
                    Do
                        Set argTok = New Collection
                        argTokDepthParen = 0
                        argTokDepthBr = 0
                        argTokDepthB = 0
                        Do While idx < n
                            If types(idx) = "PAREN" Then
                                Select Case CStr(vals(idx))
                                    Case "("
                                        argTokDepthParen = argTokDepthParen + 1
                                    Case ")"
                                        If argTokDepthParen = 0 Then Exit Do Else
                                        argTokDepthParen = argTokDepthParen - 1
                                    Case "["
                                        argTokDepthBr = argTokDepthBr + 1
                                    Case "]"
                                        If argTokDepthBr = 0 Then Exit Do Else
                                        argTokDepthBr = argTokDepthBr - 1
                                    Case "{"
                                        argTokDepthB = argTokDepthB + 1
                                    Case "}"
                                        If argTokDepthB = 0 Then Exit Do Else
                                        argTokDepthB = argTokDepthB - 1
                                End Select
                            ElseIf types(idx) = "SEP" And vals(idx) = "," And
                            argTokDepthParen = 0 And argTokDepthBr = 0 And argTokDepthB
                            = 0 Then
                                Exit Do
                            End If
                            argTok.Add Array(types(idx), vals(idx))
                            idx = idx + 1
                        Loop
                        If argTok.Count > 0 Then
                            argNodes.Add ParseExprTokensToNode(argTok)
                        Else
                            Dim litEmpty As Map: Set litEmpty = MakeNode("Literal")
                            litEmpty.SetValue "value", Empty
                            argNodes.Add litEmpty.Clone
                        End If
```

```
                        If idx < n Then
                            If types(idx) = "SEP" And vals(idx) = "," Then
                                idx = idx + 1
                                ' continue parsing next arg
                            ElseIf types(idx) = "PAREN" And vals(idx) = ")" Then
                                idx = idx + 1
                                Exit Do
                            Else
                                ' unexpected token -> stop args parsing
                                Exit Do
                            End If
                        Else
                            Exit Do
                        End If
                    Loop
                    Dim callNode As Map: Set callNode = MakeNode("Call")
                    callNode.SetValue "callee", currentNode
                    callNode.SetValue "args", argNodes
                    Set currentNode = callNode.Clone
                Else
                    Exit Do
                End If
            Loop
            Set ParsePrimaryNode = currentNode.Clone
            Exit Function
        Case "PAREN"
            If v = "(" Then
                idx = idx + 1
                Dim innerNode As Map: Set innerNode = ParseLogicalOrNode(types, vals, n,
                idx)
                If idx < n And types(idx) = "PAREN" And vals(idx) = ")" Then idx = idx +
                1
                Set ParsePrimaryNode = innerNode.Clone
                Exit Function
            ElseIf v = "[" Then
                idx = idx + 1
                Dim arrList As Collection
                Dim elemTok As Collection
                Set arrList = New Collection
                Do While idx < n And Not (types(idx) = "PAREN" And vals(idx) = "]")
                    Set elemTok = New Collection
                    ' Collect element tokens until a comma (element separator) or
                    closing bracket
                    Do While idx < n And Not ((types(idx) = "SEP" And vals(idx) = ",")
                    Or (types(idx) = "PAREN" And vals(idx) = "]"))
                        elemTok.Add Array(types(idx), vals(idx))
                        idx = idx + 1
                    Loop
                    arrList.Add ParseExprTokensToNode(elemTok)
                    If idx < n And types(idx) = "SEP" And vals(idx) = "," Then idx = idx
                    + 1
                Loop
                If idx < n And types(idx) = "PAREN" And vals(idx) = "]" Then idx = idx +
                1
                Dim arrNode As Map: Set arrNode = MakeNode("Array")
                arrNode.SetValue "items", arrList
                Set ParsePrimaryNode = arrNode.Clone
                Exit Function
            ElseIf v = "{" Then
                ' object literal: { key: value, key2: value2 }
                idx = idx + 1
                Dim objItems As Collection
                Dim pair As Collection
                Set objItems = New Collection
                ' key can be IDENT or STRING
                Dim keyTok As String
                Do While idx < n And Not (types(idx) = "PAREN" And vals(idx) = "}")
                    If types(idx) = "IDENT" Or types(idx) = "STRING" Then
                        keyTok = CStr(vals(idx))
```

```
                        idx = idx + 1
                    Else
                        err.Raise vbObjectError + 8002, "Compiler.ParsePrimary",
                        "Invalid object key"
                    End If
                    ' expect ':'
                    If idx < n And types(idx) = "OP" And vals(idx) = ":" Then
                        idx = idx + 1
                    Else
                        err.Raise vbObjectError + 8003, "Compiler.ParsePrimary",
                        "Expected ':' after object key"
                    End If
                    ' parse value expression
                    Dim valNode As Map
                    Set valNode = ParseTernaryNode(types, vals, n, idx)
                    ' store pair (key, node)
                    Set pair = New Collection
                    pair.Add keyTok
                    pair.Add valNode.Clone
                    objItems.Add pair
                    ' optional comma
                    If idx < n And types(idx) = "SEP" And vals(idx) = "," Then idx = idx
                    + 1
                Loop
                If idx < n And types(idx) = "PAREN" And vals(idx) = "}" Then idx = idx +
                1
                Dim onode As Map: Set onode = MakeNode("Object")
                onode.SetValue "items", objItems
                Set ParsePrimaryNode = onode.Clone
                Exit Function
            End If
    End Select

    Dim defN As Map: Set defN = MakeNode("Literal")
    defN.SetValue "value", Empty
    Set ParsePrimaryNode = defN
End Function

' ---------------------
' AST builders for control structures
' ---------------------

Private Function ParseIfAST(stmtTokens As Collection) As Map
    Dim node As Map: Set node = MakeNode("If")
    Dim i As Long: i = 2
    Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
    stmtTokens(i)(1) = "(")
        i = i + 1
    Loop
    If i > stmtTokens.Count Then Set ParseIfAST = node: Exit Function
    i = i + 1
    Dim condTokens As Collection
    Set condTokens = New Collection
    Dim condDepth As Long: condDepth = 0
    Do While i <= stmtTokens.Count
        If stmtTokens(i)(0) = "PAREN" Then
            If stmtTokens(i)(1) = "(" Then
                condDepth = condDepth + 1
            ElseIf stmtTokens(i)(1) = ")" Then
                If condDepth = 0 Then
                    Exit Do
                Else
                    condDepth = condDepth - 1
                End If
            End If
        End If
        condTokens.Add stmtTokens(i)
        i = i + 1
    Loop
```

```
3005        node.SetValue "cond", ParseExprTokensToNode(condTokens)
3006
3007        Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
            stmtTokens(i)(1) = "{")
3008            i = i + 1
3009        Loop
3010        If i > stmtTokens.Count Then Set ParseIfAST = node.Clone: GoTo clean_
3011        Dim j As Long: j = i + 1
3012        Dim depth As Long: depth = 0
3013        Dim thenTokens As Collection
3014        Set thenTokens = New Collection
3015        Do While j <= stmtTokens.Count
3016            If stmtTokens(j)(0) = "PAREN" Then
3017                If stmtTokens(j)(1) = "{" Then
3018                    depth = depth + 1
3019                ElseIf stmtTokens(j)(1) = "}" Then
3020                    If depth = 0 Then
3021                        Exit Do
3022                    Else
3023                        depth = depth - 1
3024                    End If
3025                End If
3026            End If
3027            thenTokens.Add stmtTokens(j)
3028            j = j + 1
3029        Loop
3030        node.SetValue "then", ParseTokensToAST(thenTokens)
3031
3032        Dim pos As Long: pos = j + 1
3033        Dim elseifConds As Collection, elseifBlocks As Collection
3034        Dim hasElse As Boolean: hasElse = False
3035        Dim tname As String
3036        Dim eCondTok As Collection
3037        Dim eBlockTok As Collection
3038        Dim elseTok As Collection
3039        Dim eDepth As Long
3040        Set elseifConds = New Collection
3041        Set elseifBlocks = New Collection
3042        Do While pos <= stmtTokens.Count
3043            If stmtTokens(pos)(0) = "COMMENT" Then
3044                pos = pos + 1
3045                GoTo IfParseNext
3046            End If
3047            If stmtTokens(pos)(0) = "IDENT" Then
3048                tname = LCase$(stmtTokens(pos)(1))
3049                If tname = "elseif" Then
3050                    pos = pos + 1
3051                    Do While pos <= stmtTokens.Count And Not (stmtTokens(pos)(0) = "PAREN"
                    And stmtTokens(pos)(1) = "(")
3052                        pos = pos + 1
3053                    Loop
3054                    If pos > stmtTokens.Count Then Exit Do
3055                    pos = pos + 1
3056                    eDepth = 0
3057                    Set eCondTok = New Collection
3058                    Do While pos <= stmtTokens.Count
3059                        If stmtTokens(pos)(0) = "PAREN" Then
3060                            If stmtTokens(pos)(1) = "(" Then
3061                                eDepth = eDepth + 1
3062                            ElseIf stmtTokens(pos)(1) = ")" Then
3063                                If eDepth = 0 Then
3064                                    Exit Do
3065                                Else
3066                                    eDepth = eDepth - 1
3067                                End If
3068                            End If
3069                        End If
3070                        eCondTok.Add stmtTokens(pos)
3071                        pos = pos + 1
```

```
3072                    Loop
3073                    Do While pos <= stmtTokens.Count And Not (stmtTokens(pos)(0) = "PAREN"
                        And stmtTokens(pos)(1) = "{")
3074                        pos = pos + 1
3075                    Loop
3076                    If pos > stmtTokens.Count Then Exit Do
3077                    pos = pos + 1
3078                    Set eBlockTok = New Collection
3079                    depth = 0
3080                    Do While pos <= stmtTokens.Count
3081                        If stmtTokens(pos)(0) = "PAREN" Then
3082                            If stmtTokens(pos)(1) = "{" Then
3083                                depth = depth + 1
3084                            ElseIf stmtTokens(pos)(1) = "}" Then
3085                                If depth = 0 Then
3086                                    Exit Do
3087                                Else
3088                                    depth = depth - 1
3089                                End If
3090                            End If
3091                        End If
3092                        eBlockTok.Add stmtTokens(pos)
3093                        pos = pos + 1
3094                    Loop
3095                    elseifConds.Add ParseExprTokensToNode(eCondTok)
3096                    elseifBlocks.Add ParseTokensToAST(eBlockTok)
3097                    pos = pos + 1
3098                    GoTo IfParseNext
3099                ElseIf tname = "else" Then
3100                    pos = pos + 1
3101                    Do While pos <= stmtTokens.Count And Not (stmtTokens(pos)(0) = "PAREN"
                        And stmtTokens(pos)(1) = "{")
3102                        pos = pos + 1
3103                    Loop
3104                    If pos > stmtTokens.Count Then Exit Do
3105                    pos = pos + 1
3106                    Set elseTok = New Collection
3107                    depth = 0
3108                    Do While pos <= stmtTokens.Count
3109                        If stmtTokens(pos)(0) = "PAREN" Then
3110                            If stmtTokens(pos)(1) = "{" Then
3111                                depth = depth + 1
3112                            ElseIf stmtTokens(pos)(1) = "}" Then
3113                                If depth = 0 Then
3114                                    Exit Do
3115                                Else
3116                                    depth = depth - 1
3117                                End If
3118                            End If
3119                        End If
3120                        elseTok.Add stmtTokens(pos)
3121                        pos = pos + 1
3122                    Loop
3123                    hasElse = True
3124                    node.SetValue "else", ParseTokensToAST(elseTok)
3125                    Exit Do
3126                Else
3127                    Exit Do
3128                End If
3129            Else
3130                Exit Do
3131            End If
3132    IfParseNext:
3133        Loop
3134
3135        node.SetValue "elseif_conds", CloneCollectionOfVariants(elseifConds)
3136        node.SetValue "elseif_blocks", CloneCollectionOfVariants(elseifBlocks)
3137        node.SetValue "hasElse", hasElse
3138        Set ParseIfAST = node
```

```
clean_:
    Set node = Nothing
    Set eCondTok = Nothing
    Set eBlockTok = Nothing
    Set elseTok = Nothing
    Set elseifConds = Nothing
    Set elseifBlocks = Nothing
End Function

Private Function ParseForAST(stmtTokens As Collection) As Map
    Dim node As Map: Set node = MakeNode("For")
    Dim initTok As Collection
    Dim condTok As Collection
    Dim stepTok As Collection
    Dim bodyToks As Collection
    Dim i As Long: i = 2
    Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
    stmtTokens(i)(1) = "(")
        i = i + 1
    Loop
    If i > stmtTokens.Count Then Set ParseForAST = node: Exit Function
    i = i + 1
    Set initTok = New Collection
    Dim hdrParen As Long, hdrBracket As Long, hdrBrace As Long
    hdrParen = 0: hdrBracket = 0: hdrBrace = 0
    Do While i <= stmtTokens.Count
        If stmtTokens(i)(0) = "PAREN" Then
            Select Case stmtTokens(i)(1)
                Case "("
                    hdrParen = hdrParen + 1
                Case ")"
                    If hdrParen > 0 Then
                        hdrParen = hdrParen - 1
                    Else
                        Exit Do
                    End If
                Case "["
                    hdrBracket = hdrBracket + 1
                Case "]"
                    If hdrBracket > 0 Then hdrBracket = hdrBracket - 1
                Case "{"
                    hdrBrace = hdrBrace + 1
                Case "}"
                    If hdrBrace > 0 Then hdrBrace = hdrBrace - 1
            End Select
        End If
        If hdrParen = 0 And hdrBracket = 0 And hdrBrace = 0 And stmtTokens(i)(0) = "SEP"
        And _
                    (stmtTokens(i)(1) = "," Or stmtTokens(i)(1) = ";") Then
            i = i + 1
            Exit Do
        End If
        initTok.Add stmtTokens(i)
        i = i + 1
    Loop
    node.SetValue "init", ParseExprTokensToNode(initTok)

    Set condTok = New Collection
    hdrParen = 0: hdrBracket = 0: hdrBrace = 0
    Do While i <= stmtTokens.Count
        If stmtTokens(i)(0) = "PAREN" Then
            Select Case stmtTokens(i)(1)
                Case "("
                    hdrParen = hdrParen + 1
                Case ")"
                    If hdrParen > 0 Then
                        hdrParen = hdrParen - 1
                    Else
                        Exit Do
```

```
                    End If
                Case "["
                    hdrBracket = hdrBracket + 1
                Case "]"
                    If hdrBracket > 0 Then hdrBracket = hdrBracket - 1
                Case "{"
                    hdrBrace = hdrBrace + 1
                Case "}"
                    If hdrBrace > 0 Then hdrBrace = hdrBrace - 1
            End Select
        End If
        If hdrParen = 0 And hdrBracket = 0 And hdrBrace = 0 And stmtTokens(i)(0) = "SEP"
        And _
                    (stmtTokens(i)(1) = "," Or stmtTokens(i)(1) = ";") Then
            i = i + 1
            Exit Do
        End If
        condTok.Add stmtTokens(i)
        i = i + 1
    Loop
    node.SetValue "cond", ParseExprTokensToNode(condTok)

    Set stepTok = New Collection
    hdrParen = 0: hdrBracket = 0: hdrBrace = 0
    Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
    stmtTokens(i)(1) = ")")
        If stmtTokens(i)(0) = "PAREN" Then
            Select Case stmtTokens(i)(1)
                Case "("
                    hdrParen = hdrParen + 1
                Case ")"
                    If hdrParen > 0 Then hdrParen = hdrParen - 1
                Case "["
                    hdrBracket = hdrBracket + 1
                Case "]"
                    If hdrBracket > 0 Then hdrBracket = hdrBracket - 1
                Case "{"
                    hdrBrace = hdrBrace + 1
                Case "}"
                    If hdrBrace > 0 Then hdrBrace = hdrBrace - 1
            End Select
        End If
        stepTok.Add stmtTokens(i)
        i = i + 1
    Loop
    node.SetValue "step", ParseExprTokensToNode(stepTok)

    Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
    stmtTokens(i)(1) = "{")
        i = i + 1
    Loop
    If i > stmtTokens.Count Then Set ParseForAST = node.Clone: GoTo clean_
    Dim j As Long: j = i + 1
    Dim depth As Long: depth = 0
    Set bodyToks = New Collection
    Do While j <= stmtTokens.Count
        If stmtTokens(j)(0) = "PAREN" Then
            If stmtTokens(j)(1) = "{" Then
                depth = depth + 1
            ElseIf stmtTokens(j)(1) = "}" Then
                If depth = 0 Then
                    Exit Do
                Else
                    depth = depth - 1
                End If
            End If
        End If
        bodyToks.Add stmtTokens(j)
        j = j + 1
```

```
3272        Loop
3273        node.SetValue "body", ParseTokensToAST(bodyToks)
3274        Set ParseForAST = node.Clone
3275    clean_:
3276        Set node = Nothing
3277        Set bodyToks = Nothing
3278        Set stepTok = Nothing
3279        Set condTok = Nothing
3280        Set initTok = Nothing
3281    End Function
3282
3283    Private Function ParseWhileAST(stmtTokens As Collection) As Map
3284        Dim node As Map: Set node = MakeNode("While")
3285        Dim i As Long: i = 2
3286        Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
            stmtTokens(i)(1) = "(")
3287            i = i + 1
3288        Loop
3289        If i > stmtTokens.Count Then Set ParseWhileAST = node: Exit Function
3290        i = i + 1
3291        Dim condTok As Collection
3292        Set condTok = New Collection
3293        Dim depthP As Long: depthP = 0
3294        Do While i <= stmtTokens.Count
3295            If stmtTokens(i)(0) = "PAREN" Then
3296                If stmtTokens(i)(1) = "(" Then
3297                    depthP = depthP + 1
3298                ElseIf stmtTokens(i)(1) = ")" Then
3299                    If depthP = 0 Then
3300                        Exit Do
3301                    Else
3302                        depthP = depthP - 1
3303                    End If
3304                End If
3305            End If
3306            condTok.Add stmtTokens(i)
3307            i = i + 1
3308        Loop
3309        node.SetValue "cond", ParseExprTokensToNode(condTok)
3310
3311        Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
            stmtTokens(i)(1) = "{")
3312            i = i + 1
3313        Loop
3314        If i > stmtTokens.Count Then Set ParseWhileAST = node: Exit Function
3315        Dim j As Long: j = i + 1
3316        Dim depth As Long: depth = 0
3317        Dim bodyToks As Collection
3318        Set bodyToks = New Collection
3319        Do While j <= stmtTokens.Count
3320            If stmtTokens(j)(0) = "PAREN" Then
3321                If stmtTokens(j)(1) = "{" Then
3322                    depth = depth + 1
3323                ElseIf stmtTokens(j)(1) = "}" Then
3324                    If depth = 0 Then
3325                        Exit Do
3326                    Else
3327                        depth = depth - 1
3328                    End If
3329                End If
3330            End If
3331            bodyToks.Add stmtTokens(j)
3332            j = j + 1
3333        Loop
3334        node.SetValue "body", ParseTokensToAST(bodyToks)
3335        Set ParseWhileAST = node
3336    End Function
3337
3338    Private Function ParseTryCatchAST(stmtTokens As Collection) As Map
```

```
3339        Dim node As Map: Set node = MakeNode("TryCatch")
3340        Dim i As Long: i = 2
3341        Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
            stmtTokens(i)(1) = "{")
3342            i = i + 1
3343        Loop
3344        If i > stmtTokens.Count Then Set ParseTryCatchAST = node: Exit Function
3345        Dim j As Long: j = i + 1
3346        Dim depth As Long: depth = 0
3347        Dim tryTok As Collection
3348        Set tryTok = New Collection
3349        Do While j <= stmtTokens.Count
3350            If stmtTokens(j)(0) = "PAREN" Then
3351                If stmtTokens(j)(1) = "{" Then
3352                    depth = depth + 1
3353                ElseIf stmtTokens(j)(1) = "}" Then
3354                    If depth = 0 Then
3355                        Exit Do
3356                    Else
3357                        depth = depth - 1
3358                    End If
3359                End If
3360            End If
3361            tryTok.Add stmtTokens(j)
3362            j = j + 1
3363        Loop
3364        node.SetValue "try", ParseTokensToAST(tryTok)
3365
3366        Dim k As Long: k = j + 1
3367        Dim catchTok As Collection
3368        Set catchTok = New Collection
3369        Do While k <= stmtTokens.Count
3370            If stmtTokens(k)(0) = "IDENT" And LCase$(stmtTokens(k)(1)) = "catch" Then
3371                Dim kk As Long: kk = k + 1
3372                Do While kk <= stmtTokens.Count And Not (stmtTokens(kk)(0) = "PAREN" And
                    stmtTokens(kk)(1) = "{")
3373                    kk = kk + 1
3374                Loop
3375                If kk <= stmtTokens.Count Then
3376                    kk = kk + 1
3377                    Dim depth2 As Long: depth2 = 0
3378                    Do While kk <= stmtTokens.Count
3379                        If stmtTokens(kk)(0) = "PAREN" Then
3380                            If stmtTokens(kk)(1) = "{" Then
3381                                depth2 = depth2 + 1
3382                            ElseIf stmtTokens(kk)(1) = "}" Then
3383                                If depth2 = 0 Then
3384                                    Exit Do
3385                                Else
3386                                    depth2 = depth2 - 1
3387                                End If
3388                            End If
3389                        End If
3390                        catchTok.Add stmtTokens(kk)
3391                        kk = kk + 1
3392                    Loop
3393                End If
3394                Exit Do
3395            End If
3396            k = k + 1
3397        Loop
3398        node.SetValue "catch", ParseTokensToAST(catchTok)
3399        Set ParseTryCatchAST = node
3400    End Function
3401
3402    Private Function ParseSwitchAST(stmtTokens As Collection) As Map
3403        Dim node As Map: Set node = MakeNode("Switch")
3404        Dim i As Long: i = 2
3405        Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
```

```
        stmtTokens(i)(1) = "("
3406        i = i + 1
3407    Loop
3408    If i > stmtTokens.Count Then Set ParseSwitchAST = node: Exit Function
3409    i = i + 1
3410    Dim condTok As New Collection
3411    Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
        stmtTokens(i)(1) = ")")
3412        condTok.Add stmtTokens(i)
3413        i = i + 1
3414    Loop
3415    node.SetValue "expr", ParseExprTokensToNode(condTok)
3416
3417    Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
        stmtTokens(i)(1) = "{")
3418        i = i + 1
3419    Loop
3420    If i > stmtTokens.Count Then Set ParseSwitchAST = node: Exit Function
3421    i = i + 1
3422    Dim cases As Collection
3423    Set cases = New Collection
3424    Dim defaultBlock As Collection
3425    Dim valTok As Collection
3426    Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
        stmtTokens(i)(1) = "}")
3427        If stmtTokens(i)(0) = "IDENT" And LCase$(stmtTokens(i)(1)) = "case" Then
3428            i = i + 1
3429            Set valTok = New Collection
3430            Dim cvParen As Long: cvParen = 0
3431            Dim cvBracket As Long: cvBracket = 0
3432            Dim cvBrace As Long: cvBrace = 0
3433            Do While i <= stmtTokens.Count
3434                If stmtTokens(i)(0) = "PAREN" Then
3435                    Select Case stmtTokens(i)(1)
3436                        Case "("
3437                            cvParen = cvParen + 1
3438                        Case ")"
3439                            If cvParen > 0 Then cvParen = cvParen - 1
3440                        Case "["
3441                            cvBracket = cvBracket + 1
3442                        Case "]"
3443                            If cvBracket > 0 Then cvBracket = cvBracket - 1
3444                        Case "{"
3445                            If cvParen = 0 And cvBracket = 0 Then
3446                                Exit Do
3447                            Else
3448                                cvBrace = cvBrace + 1
3449                            End If
3450                        Case "}"
3451                            If cvBrace > 0 Then cvBrace = cvBrace - 1
3452                    End Select
3453                End If
3454                valTok.Add stmtTokens(i)
3455                i = i + 1
3456            Loop
3457            Dim caseValNode As Map: Set caseValNode = ParseExprTokensToNode(valTok)
3458            If i <= stmtTokens.Count And stmtTokens(i)(0) = "PAREN" And stmtTokens(i)(1)
                = "{" Then
3459                i = i + 1
3460                Dim depth As Long: depth = 0
3461                Dim blockTok As Collection
3462                Dim pair As Collection
3463                Dim caseBlock As Collection
3464                Set blockTok = New Collection
3465                Do While i <= stmtTokens.Count
3466                    If stmtTokens(i)(0) = "PAREN" Then
3467                        If stmtTokens(i)(1) = "{" Then
3468                            depth = depth + 1
3469                        ElseIf stmtTokens(i)(1) = "}" Then
```

```
                            If depth = 0 Then
                                Exit Do
                            Else
                                depth = depth - 1
                            End If
                        End If
                    End If
                    blockTok.Add stmtTokens(i)
                    i = i + 1
                Loop
                Set caseBlock = ParseTokensToAST(blockTok)
                Set pair = New Collection
                pair.Add caseValNode
                pair.Add caseBlock
                cases.Add pair
                i = i + 1
            End If
        ElseIf stmtTokens(i)(0) = "IDENT" And LCase$(stmtTokens(i)(1)) = "default" Then
            i = i + 1
            Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
            stmtTokens(i)(1) = "{")
                i = i + 1
            Loop
            If i <= stmtTokens.Count And stmtTokens(i)(0) = "PAREN" And stmtTokens(i)(1)
            = "{" Then
                i = i + 1
                Dim depth2 As Long: depth2 = 0
                Dim defTok As New Collection
                Do While i <= stmtTokens.Count
                    If stmtTokens(i)(0) = "PAREN" Then
                        If stmtTokens(i)(1) = "{" Then
                            depth2 = depth2 + 1
                        ElseIf stmtTokens(i)(1) = "}" Then
                            If depth2 = 0 Then
                                Exit Do
                            Else
                                depth2 = depth2 - 1
                            End If
                        End If
                    End If
                    defTok.Add stmtTokens(i)
                    i = i + 1
                Loop
                Set defaultBlock = ParseTokensToAST(defTok)
                i = i + 1
            End If
        Else
            i = i + 1
        End If
    Loop
    node.SetValue "cases", cases
    node.SetValue "default", defaultBlock
    Set ParseSwitchAST = node
End Function


' Compiler AST normalization pass
' Converts expression statements like (Binary op "=" ...) into Assign nodes
' when the left-hand side is a valid assignment target (Variable or Index).
' Also normalizes `for` init/step and recursively visits blocks.
'
Private Function IsAssignableNode(n As Map) As Boolean
    If n Is Nothing Then
        IsAssignableNode = False: Exit Function
    End If
    Dim t As String: t = n.GetValue("type")
    ' Accept variable, indexed access (arr[i]) and member access (obj.prop)
    If t = "Variable" Or t = "Index" Or t = "Member" Then
        IsAssignableNode = True
```

```
3537        Else
3538            IsAssignableNode = False
3539        End If
3540    End Function
3541
3542    Private Function NormalizeExprNodeRecursive(expr As Map) As Map
3543        ' Recursively walk expression nodes and normalize inner expressions as needed.
3544        If expr Is Nothing Then
3545            Set NormalizeExprNodeRecursive = Nothing: Exit Function
3546        End If
3547        Dim t As String: t = expr.GetValue("type")
3548        Select Case t
3549            Case "Binary"
3550                Dim left_ As Map: Set left_ =
                    NormalizeExprNodeRecursive(expr.GetValue("left"))
3551                Dim right_ As Map: Set right_ =
                    NormalizeExprNodeRecursive(expr.GetValue("right"))
3552                ' create new Binary node only if children changed, otherwise return original
3553                If (Not left_ Is expr.GetValue("left")) Or (Not right_ Is
                    expr.GetValue("right")) Then
3554                    Dim nb As New Map
3555                    nb.Add "type", "Binary"
3556                    nb.SetValue "op", expr.GetValue("op")
3557                    nb.SetValue "left", IIf(left_ Is Nothing, expr.GetValue("left"), left_)
3558                    nb.SetValue "right", IIf(right_ Is Nothing, expr.GetValue("right"),
                        right_)
3559                    Set NormalizeExprNodeRecursive = nb
3560                    Exit Function
3561                End If
3562                Set NormalizeExprNodeRecursive = expr
3563                Exit Function
3564            Case "Unary"
3565                Dim rec As Map: Set rec = NormalizeExprNodeRecursive(expr.GetValue("expr"))
3566                If Not rec Is expr.GetValue("expr") Then
3567                    Dim nu As New Map
3568                    nu.Add "type", "Unary"
3569                    nu.SetValue "op", expr.GetValue("op")
3570                    nu.SetValue "expr", rec
3571                    Set NormalizeExprNodeRecursive = nu
3572                    Exit Function
3573                End If
3574                Set NormalizeExprNodeRecursive = expr
3575                Exit Function
3576            Case "Call"
3577                ' normalize args
3578                Dim args As Collection: Set args = expr.GetValue("args")
3579                If Not args Is Nothing Then
3580                    Dim newArgs As New Collection
3581                    Dim changed As Boolean: changed = False
3582                    Dim i As Long
3583                    For i = 1 To args.Count
3584                        Dim ae As Map: Set ae = NormalizeExprNodeRecursive(args(i))
3585                        If Not ae Is args(i) Then changed = True: newArgs.Add ae Else
                            newArgs.Add args(i)
3586                    Next i
3587                    If changed Then
3588                        Dim nc As New Map
3589                        nc.Add "type", "Call"
3590                        nc.SetValue "name", expr.GetValue("name")
3591                        nc.SetValue "args", newArgs
3592                        Set NormalizeExprNodeRecursive = nc
3593                        Exit Function
3594                    End If
3595                End If
3596                Set NormalizeExprNodeRecursive = expr
3597                Exit Function
3598            Case "Index"
3599                Dim b As Map: Set b = NormalizeExprNodeRecursive(expr.GetValue("base"))
3600                Dim idx As Map: Set idx = NormalizeExprNodeRecursive(expr.GetValue("index"))
```

```
3601                If Not b Is expr.GetValue("base") Or Not idx Is expr.GetValue("index") Then
3602                    Dim ni As New Map
3603                    ni.Add "type", "Index"
3604                    ni.SetValue "base", IIf(b Is Nothing, expr.GetValue("base"), b)
3605                    ni.SetValue "index", IIf(idx Is Nothing, expr.GetValue("index"), idx)
3606                    Set NormalizeExprNodeRecursive = ni
3607                    Exit Function
3608                End If
3609                Set NormalizeExprNodeRecursive = expr
3610                Exit Function
3611            Case "Array"
3612                Dim items As Collection: Set items = expr.GetValue("items")
3613                If Not items Is Nothing Then
3614                    Dim newit As New Collection
3615                    Dim ch As Boolean: ch = False
3616                    Dim ii As Long
3617                    For ii = 1 To items.Count
3618                        Dim ei As Map: Set ei = NormalizeExprNodeRecursive(items(ii))
3619                        If Not ei Is items(ii) Then ch = True: newit.Add ei Else newit.Add
                            items(ii)
3620                    Next ii
3621                    If ch Then
3622                        Dim na As New Map
3623                        na.Add "type", "Array"
3624                        na.SetValue "items", newit
3625                        Set NormalizeExprNodeRecursive = na
3626                        Exit Function
3627                    End If
3628                End If
3629                Set NormalizeExprNodeRecursive = expr
3630                Exit Function
3631            Case Else
3632                Set NormalizeExprNodeRecursive = expr
3633                Exit Function
3634        End Select
3635    End Function
3636
3637    Private Function NormalizeNodeRecursive(node As Map) As Map
3638        ' Normalize a statement node and recursively process inner blocks.
3639        If node Is Nothing Then
3640            Set NormalizeNodeRecursive = Nothing: Exit Function
3641        End If
3642        Dim expr As Map
3643        Dim left_ As Map
3644        Dim a As Map
3645        Dim r As Map
3646        Dim newExpr As Map
3647        Dim ns As Map
3648        Dim cond As Map
3649        Dim thenBlk As Collection
3650        Dim outThen As Collection
3651        Dim elseifConds As Collection
3652        Dim elseifBlocks As Collection
3653        Dim outElseIfConds As Collection
3654        Dim outElseIfBlocks As Collection
3655        Dim eb As Collection
3656        Dim nb As Collection
3657        Dim elseBlk As Collection
3658        Dim outElse As Collection
3659        Dim nif As Map
3660        Dim initNode As Map
3661        Dim condNode As Map
3662        Dim stepNode As Map
3663        Dim body As Collection
3664        Dim nInit As Map, nCond As Map, nStep As Map
3665        Dim na As Map
3666        Dim L As Map
3667        Dim e As Map
3668        Dim L2 As Map
```

```
3669        Dim na2 As Map
3670        Dim SL As Map
3671        Dim na3 As Map
3672        Dim newBody As Collection
3673        Dim nf As Map
3674        Dim cnd As Map
3675        Dim bdy As Collection
3676        Dim nbdy As Collection
3677        Dim nw As Map
3678        Dim tryBlk As Collection
3679        Dim catchBlk As Collection
3680        Dim nt As Collection, nc As Collection
3681        Dim ntc As Map
3682        Dim cases As Collection
3683        Dim ncases As Collection
3684        Dim pair As Collection
3685        Dim caseExpr As Map
3686        Dim blockStmts As Collection
3687        Dim newPair As Collection
3688        Dim def As Collection
3689        Dim ndef As Collection
3690        Dim nsw As Map
3691        Dim leftA As Map
3692        Dim rightA As Map
3693        Dim naNode As Map
3694        Dim tp As String: tp = node.GetValue("type")
3695        Select Case tp
3696            Case "ExprStmt"
3697                Set expr = node.GetValue("expr")
3698                ' If expression is a Binary "=" with assignable LHS, convert to Assign node.
3699                If Not expr Is Nothing Then
3700                    If expr.GetValue("type") = "Binary" Then
3701                        If CStr(expr.GetValue("op")) = "=" Then
3702                            Set left_ = expr.GetValue("left")
3703                            If Not left_ Is Nothing Then
3704                                If IsAssignableNode(left_) Then
3705                                    a.Add "type", "Assign"
3706                                    a.SetValue "left", left_
3707                                    ' right: normalize recursively as expression
3708                                    Set r =
                                    NormalizeExprNodeRecursive(expr.GetValue("right"))
3709                                    If r Is Nothing Then Set r = expr.GetValue("right")
3710                                    a.SetValue "right", r
3711                                    Set NormalizeNodeRecursive = a
3712                                    Exit Function
3713                                End If
3714                            End If
3715                        End If
3716                    End If
3717                End If
3718                ' otherwise just normalize inner expression (if any)
3719                If Not expr Is Nothing Then
3720                    Set newExpr = NormalizeExprNodeRecursive(expr)
3721                    If Not newExpr Is expr Then
3722                        Set ns = New Map
3723                        ns.Add "type", "ExprStmt"
3724                        ns.SetValue "expr", newExpr
3725                        Set NormalizeNodeRecursive = ns
3726                        Exit Function
3727                    End If
3728                End If
3729                Set NormalizeNodeRecursive = node
3730                Exit Function
3731
3732            Case "If"
3733                Set cond = NormalizeExprNodeRecursive(node.GetValue("cond"))
3734                Set thenBlk = node.GetValue("then")
3735                Dim i As Long
3736                If Not thenBlk Is Nothing Then
```

```
3737                    Set outThen = New Collection
3738                    For i = 1 To thenBlk.Count
3739                        outThen.Add NormalizeNodeRecursive(thenBlk(i))
3740                    Next i
3741                End If
3742                Set elseifConds = node.GetValue("elseif_conds")
3743                Set elseifBlocks = node.GetValue("elseif_blocks")
3744                Set outElseIfConds = New Collection
3745                Set outElseIfBlocks = New Collection
3746                If Not elseifConds Is Nothing Then
3747                    For i = 1 To elseifConds.Count
3748                        outElseIfConds.Add NormalizeExprNodeRecursive(elseifConds(i))
3749                    Next i
3750                End If
3751                If Not elseifBlocks Is Nothing Then
3752                    For i = 1 To elseifBlocks.Count
3753                        Set eb = elseifBlocks(i)
3754                        Set nb = New Collection
3755                        Dim j As Long
3756                        For j = 1 To eb.Count
3757                            nb.Add NormalizeNodeRecursive(eb(j))
3758                        Next j
3759                        outElseIfBlocks.Add nb
3760                    Next i
3761                End If
3762                Set elseBlk = node.GetValue("else")
3763                If Not elseBlk Is Nothing Then
3764                    Set outElse = New Collection
3765                    For i = 1 To elseBlk.Count
3766                        outElse.Add NormalizeNodeRecursive(elseBlk(i))
3767                    Next i
3768                End If
3769                Set nif = New Map
3770                nif.Add "type", "If"
3771                nif.SetValue "cond", IIf(cond Is Nothing, node.GetValue("cond"), cond)
3772                nif.SetValue "then", outThen
3773                nif.SetValue "elseif_conds", outElseIfConds
3774                nif.SetValue "elseif_blocks", outElseIfBlocks
3775                nif.SetValue "hasElse", node.GetValue("hasElse")
3776                If Not outElse Is Nothing Then nif.SetValue "else", outElse
3777                Set NormalizeNodeRecursive = nif
3778                Exit Function
3779
3780            Case "For"
3781                ' normalize init/cond/step and body
3782                Set initNode = node.GetValue("init")
3783                Set condNode = node.GetValue("cond")
3784                Set stepNode = node.GetValue("step")
3785                Set body = node.GetValue("body")
3786                If Not initNode Is Nothing Then
3787                    If initNode.GetValue("type") = "Binary" And
                       CStr(initNode.GetValue("op")) = "=" Then
3788                        Set L = initNode.GetValue("left")
3789                        If Not L Is Nothing And IsAssignableNode(L) Then
3790                            Set na = New Map
3791                            na.Add "type", "Assign"
3792                            na.SetValue "left", L
3793                            na.SetValue "right",
                                NormalizeExprNodeRecursive(initNode.GetValue("right"))
3794                            Set nInit = na
3795                        Else
3796                            Set nInit = NormalizeExprNodeRecursive(initNode)
3797                        End If
3798                    ElseIf initNode.GetValue("type") = "ExprStmt" Then
3799                        ' exprstmt could wrap a binary expression
3800                        Set e = initNode.GetValue("expr")
3801                        If Not e Is Nothing And e.GetValue("type") = "Binary" And
                           CStr(e.GetValue("op")) = "=" Then
3802                            Set L2 = e.GetValue("left")
```

```
                        If Not L2 Is Nothing And IsAssignableNode(L2) Then
                            Set na2 = New Map
                            na2.Add "type", "Assign"
                            na2.SetValue "left", L2
                            na2.SetValue "right",
                            NormalizeExprNodeRecursive(e.GetValue("right"))
                            Set nInit = na2
                        Else
                            Set nInit = NormalizeExprNodeRecursive(e)
                        End If
                    Else
                        Set nInit = NormalizeExprNodeRecursive(e)
                    End If
                Else
                    Set nInit = NormalizeExprNodeRecursive(initNode)
                End If
            End If
            If Not condNode Is Nothing Then Set nCond =
            NormalizeExprNodeRecursive(condNode)
            If Not stepNode Is Nothing Then
                If stepNode.GetValue("type") = "Binary" And
                CStr(stepNode.GetValue("op")) = "=" Then
                    Set SL = stepNode.GetValue("left")
                    If Not SL Is Nothing And IsAssignableNode(SL) Then
                        Set na3 = New Map
                        na3.Add "type", "Assign"
                        na3.SetValue "left", SL
                        na3.SetValue "right",
                        NormalizeExprNodeRecursive(stepNode.GetValue("right"))
                        Set nStep = na3
                    Else
                        Set nStep = NormalizeExprNodeRecursive(stepNode)
                    End If
                Else
                    Set nStep = NormalizeExprNodeRecursive(stepNode)
                End If
            End If
            Set newBody = New Collection
            If Not body Is Nothing Then
                Dim bidx As Long
                For bidx = 1 To body.Count
                    newBody.Add NormalizeNodeRecursive(body(bidx))
                Next bidx
            End If
            Set nf = New Map
            nf.Add "type", "For"
            If Not nInit Is Nothing Then nf.SetValue "init", nInit
            If Not nCond Is Nothing Then nf.SetValue "cond", nCond
            If Not nStep Is Nothing Then nf.SetValue "step", nStep
            nf.SetValue "body", newBody
            Set NormalizeNodeRecursive = nf
            Exit Function

        Case "While"
            Set cnd = NormalizeExprNodeRecursive(node.GetValue("cond"))
            Set bdy = node.GetValue("body")
            Set nbdy = New Collection
            If Not bdy Is Nothing Then
                Dim bi As Long
                For bi = 1 To bdy.Count
                    nbdy.Add NormalizeNodeRecursive(bdy(bi))
                Next bi
            End If
            Set nw = New Map
            nw.Add "type", "While"
            nw.SetValue "cond", IIf(cnd Is Nothing, node.GetValue("cond"), cnd)
            nw.SetValue "body", nbdy
            Set NormalizeNodeRecursive = nw
            Exit Function
```

```
3868
3869            Case "TryCatch"
3870                Set tryBlk = node.GetValue("try")
3871                Set catchBlk = node.GetValue("catch")
3872                Set nt = New Collection: Set nc = New Collection
3873                Dim ti As Long
3874                If Not tryBlk Is Nothing Then
3875                    For ti = 1 To tryBlk.Count: nt.Add NormalizeNodeRecursive(tryBlk(ti)):
                    Next ti
3876                End If
3877                If Not catchBlk Is Nothing Then
3878                    For ti = 1 To catchBlk.Count: nc.Add
                    NormalizeNodeRecursive(catchBlk(ti)): Next ti
3879                End If
3880                Set ntc = New Map
3881                ntc.Add "type", "TryCatch"
3882                ntc.SetValue "try", nt
3883                ntc.SetValue "catch", nc
3884                Set NormalizeNodeRecursive = ntc
3885                Exit Function
3886
3887            Case "Switch"
3888                Set cases = node.GetValue("cases")
3889                Set ncases = New Collection
3890                If Not cases Is Nothing Then
3891                    Dim ci As Long
3892                    For ci = 1 To cases.Count
3893                        Set pair = cases(ci)
3894                        Set caseExpr = NormalizeExprNodeRecursive(pair(1))
3895                        Set blockStmts = pair(2)
3896                        Set nb = New Collection
3897                        Dim bi2 As Long
3898                        For bi2 = 1 To blockStmts.Count
3899                            nb.Add NormalizeNodeRecursive(blockStmts(bi2))
3900                        Next bi2
3901                        Set newPair = New Collection
3902                        newPair.Add caseExpr
3903                        newPair.Add nb
3904                        ncases.Add newPair
3905                    Next ci
3906                End If
3907                Set def = node.GetValue("default")
3908                If Not def Is Nothing Then
3909                    Set ndef = New Collection
3910                    Dim di As Long
3911                    For di = 1 To def.Count: ndef.Add NormalizeNodeRecursive(def(di)): Next
                    di
3912                End If
3913                Set nsw = New Map
3914                nsw.Add "type", "Switch"
3915                nsw.SetValue "expr", NormalizeExprNodeRecursive(node.GetValue("expr"))
3916                nsw.SetValue "cases", ncases
3917                If Not ndef Is Nothing Then nsw.SetValue "default", ndef
3918                Set NormalizeNodeRecursive = nsw
3919                Exit Function
3920
3921            Case Else
3922                ' Assign, Print, Return, Break, Continue, etc. often have expr fields to
                    normalize.
3923                If node.GetValue("type") = "Assign" Then
3924                    Set leftA = node.GetValue("left")
3925                    Set rightA = NormalizeExprNodeRecursive(node.GetValue("right"))
3926                    Set naNode = New Map
3927                    naNode.Add "type", "Assign"
3928                    naNode.SetValue "left", leftA
3929                    naNode.SetValue "right", IIf(rightA Is Nothing, node.GetValue("right"),
                    rightA)
3930                    Set NormalizeNodeRecursive = naNode
3931                    Exit Function
```

```
            End If
            Set NormalizeNodeRecursive = node
            Exit Function
    End Select
End Function

Private Function NormalizeAssignsInStmts(stmts As Collection) As Collection
    Dim out As New Collection
    If stmts Is Nothing Then
        Set NormalizeAssignsInStmts = out: Exit Function
    End If
    Dim i As Long
    Dim n As Map
    For i = 1 To stmts.Count
        Set n = stmts(i)
        out.Add NormalizeNodeRecursive(n)
    Next i
    Set NormalizeAssignsInStmts = out
End Function

 ' Normalize compound assignments like a += b  -> Assign(a, Binary(a, +, b))
 Private Function NormalizeCompoundAssigns(stmts As Collection) As Collection
     Dim out As New Collection
     If stmts Is Nothing Then
         Set NormalizeCompoundAssigns = out: Exit Function
     End If
     Dim i As Long
     Dim e As Map
     Dim n As Map
     Dim op As String
     Dim left_ As Map
     Dim right_ As Map
     Dim bin As Map
     Dim baseOp As String
     Dim a As Map
     For i = 1 To stmts.Count
         Set n = stmts(i)
         If Not n Is Nothing And n.GetValue("type") = "ExprStmt" Then
             Set e = n.GetValue("expr")
             If Not e Is Nothing And e.GetValue("type") = "Binary" Then
                 op = CStr(e.GetValue("op"))
                 Select Case op
                     Case "+=", "-=", "*=", "/=", "%=", "^="
                         ' build Assign node: left = left <op> right
                         Set left_ = e.GetValue("left")
                         Set right_ = e.GetValue("right")
                         Set bin = New Map
                         bin.Add "type", "Binary"
                         baseOp = left$(op, Len(op) - 1)
                         bin.SetValue "op", baseOp
                         ' left needs to be used as a copy for the RHS binary left
                         operand
                         bin.SetValue "left", left_
                         bin.SetValue "right", right_
                         Set a = New Map
                         a.Add "type", "Assign"
                         a.SetValue "left", left_
                         a.SetValue "right", bin
                         out.Add a
                         GoTo NormNext
                 End Select
             End If
         End If
         ' default: pass-through (but recursively normalize inner blocks too if desired)
         out.Add n
NormNext:
     Next i
     Set NormalizeCompoundAssigns = out
 End Function
```

```
4000    ' ---------------------
4001    ' End Compiler class
4002    ' ---------------------
```