

```

1  ' -----
2  ' Compiler class
3  ' -----
4  ' Class Module: Compiler
5  Option Explicit
6  Private GLOBALS_ As Globals
7
8  ' Produces AST (Maps) for program and functions.
9  ' Relies on Parser.Tokenize, Map, ScopeStack and globals:
10 '   ProgramCache, gPrograms (Collection), gFuncTable (Map), gFuncParams (Map)
11
12 Public Sub SetGlobals(ByRef aGlobals As Globals)
13     Set GLOBALS_ = aGlobals
14 End Sub
15
16 ' Node helpers (Map-based)
17 Private Function MakeNode(nodeType As String) As Map
18     Dim m As New Map
19     m.Add "type", nodeType
20     Set MakeNode = m
21 End Function
22 ' --- Helper: parse collapsed IDENT strings like "o.x", "a[3].b[2].c", "arr[1].length"
23 Private Function ParseCollapsedIdentToNode(ByVal name As String) As Map
24     ' Returns a node representing the chain: Variable/Index/Member nodes
25     Dim parts() As String
26     Dim curNode As Map
27     Dim pos As Long, nlen As Long
28     Dim ch As String
29     Dim i As Long
30     Dim token As String
31     nlen = Len(name)
32     pos = 1
33
34     ' convenience: iterate and consume segments
35     Do While pos <= nlen
36         ch = Mid$(name, pos, 1)
37         If ch = "." Then
38             ' dot should not appear at start; skip
39             pos = pos + 1
40             GoTo ContinueLoop
41         End If
42         ' extract next segment (up to '.' or '[')
43         Dim nextDot As Long: nextDot = InStr(pos, name, ".")
44         Dim nextBr As Long: nextBr = InStr(pos, name, "[")
45         Dim seg As String
46         If nextBr = 0 And nextDot = 0 Then
47             seg = Mid$(name, pos)
48             pos = nlen + 1
49         ElseIf nextBr > 0 And (nextDot = 0 Or nextBr < nextDot) Then
50             seg = Mid$(name, pos, nextBr - pos)
51             ' we'll handle bracket after seg
52             pos = nextBr
53         Else
54             seg = Mid$(name, pos, nextDot - pos)
55             pos = nextDot
56         End If
57
58         If seg <> "" Then
59             If curNode Is Nothing Then
60                 ' first segment => base variable
61                 Dim v As Map: Set v = MakeNode("Variable")
62                 v.SetValue "name", seg
63                 Set curNode = v
64             Else
65                 ' dot-access to a property name
66                 Dim mem As Map: Set mem = MakeNode("Member")
67                 mem.SetValue "base", curNode
68                 mem.SetValue "prop", seg
69                 Set curNode = mem

```

```

70         End If
71     End If
72
73     ' handle bracketed index(s) immediately following
74     Do While pos <= nlen And Mid$(name, pos, 1) = "["
75         Dim closePos As Long: closePos = InStr(pos, name, "]")
76         If closePos = 0 Then
77             ' malformed, bail with current node
78             Exit Do
79         End If
80         Dim idxStr As String: idxStr = Mid$(name, pos + 1, closePos - pos - 1)
81         ' build index expression node from literal or identifier (numbers expected
            from collapse)
82         Dim idxNode As Map: Set idxNode = ParseExprFromStringToNode(idxStr)
83         ' create Index node: base = curNode
84         Dim inNode As Map: Set inNode = MakeNode("Index")
85         inNode.SetValue "base", curNode
86         inNode.SetValue "index", idxNode
87         Set curNode = inNode
88         pos = closePos + 1
89     Loop
90
91     ' handle trailing dot (will be processed in next loop iteration)
92 ContinueLoop:
93     Loop
94
95     ' Special-case '.length' when last part is 'length':
96     ' If last node is Member with prop "length", convert into Call to builtin ".__len__"
97     If Not curNode Is Nothing Then
98         If curNode.GetValue("type") = "Member" Then
99             If LCase$(CStr(curNode.GetValue("prop"))) = "length" Then
100                 Dim baseNode As Map: Set baseNode = curNode.GetValue("base")
101                 Dim lenCall As Map: Set lenCall = MakeNode("Call")
102                 ' builtin callee as Variable ".__len__" (VM will treat this as builtin)
103                 Dim builtinLen As Map: Set builtinLen = MakeNode("Variable")
104                 builtinLen.SetValue "name", ".__len__"
105                 lenCall.SetValue "callee", builtinLen
106                 Dim args As New Collection
107                 args.Add baseNode
108                 lenCall.SetValue "args", args
109                 Set ParseCollapsedIdentToNode = lenCall
110                 Exit Function
111             End If
112         End If
113     End If
114
115     Set ParseCollapsedIdentToNode = curNode
116 End Function
117
118 ' -----
119 ' Utility: deep-clone a Collection of Maps or primitive values
120 ' -----
121 Private Function CloneCollectionOfVariants(src As Variant) As Collection
122     Dim out As New Collection
123     Dim it As Variant
124     For Each it In src
125         If IsObject(it) Then
126             If TypeName(it) = "Map" Then
127                 out.Add it.Clone
128             ElseIf TypeName(it) = "Collection" Then
129                 out.Add CloneCollectionOfVariants(it)
130             Else
131                 out.Add it
132             End If
133         Else
134             out.Add it
135         End If
136     Next it
137     Set CloneCollectionOfVariants = out

```

```

138 End Function
139
140 ' -----
141
142 Public Function CompileProgram(src As String, Optional progName As String = "@anon") As
Long
143     Dim p As Parser
144     Dim toks As Collection
145     Dim progScope As ScopeStack
146     Dim stmtsAST As Collection
147     Dim stmtTokens As Collection
148     Dim i As Long
149     Dim node As Map
150
151     GLOBALS_.ASF_InitGlobals
152
153     If GLOBALS_.ProgramCache.Exists(src) Then
154         CompileProgram = GLOBALS_.ProgramCache.GetValue(src)
155         Exit Function
156     End If
157
158     Set p = New Parser
159     With p
160         .SetGlobals GLOBALS_
161         Set toks = .Tokenize(src)
162     End With
163
164     Set progScope = New ScopeStack
165     progScope.Push
166
167     Set stmtsAST = New Collection
168     i = 1
169
170 Compiler_MainLoop:
171     Do While i <= toks.Count
172         ' skip comments at top level
173         If toks(i)(0) = "COMMENT" Then
174             i = i + 1
175             GoTo Compiler_MainLoop
176         End If
177
178         ' Function definition at top-level
179         If toks(i)(0) = "IDENT" And LCase$(toks(i)(1)) = "fun" Then
180             i = i + 1
181             ' skip comments
182             Do While i <= toks.Count And toks(i)(0) = "COMMENT"
183                 i = i + 1
184             Loop
185             If i > toks.Count Then err.Raise vbObjectError + 1, , "Unexpected end after
fun"
186             Dim fname As String: fname = toks(i)(1)
187             i = i + 1
188             Do While i <= toks.Count And toks(i)(0) = "COMMENT"
189                 i = i + 1
190             Loop
191
192             Dim argList As New Collection
193             If i <= toks.Count And toks(i)(0) = "PAREN" And toks(i)(1) = "(" Then
194                 i = i + 1
195                 Do While i <= toks.Count And Not (toks(i)(0) = "PAREN" And toks(i)(1) =
")")
196                     If toks(i)(0) = "COMMENT" Then
197                         i = i + 1
198                     Else
199                         If toks(i)(0) = "IDENT" Then argList.Add toks(i)(1)
200                         i = i + 1
201                     End If
202                 Loop
203                 If i <= toks.Count And toks(i)(0) = "PAREN" And toks(i)(1) = ")" Then i

```



```

272         bracketDepth = bracketDepth + 1
273     Case "]"
274         If bracketDepth > 0 Then
275             bracketDepth = bracketDepth - 1
276         End If
277     End Select
278 End If
279
280     ' Top-level statement separator: semicolon (commas are argument
281     ' separators) if we are not nested
282     If toks(i)(0) = "SEP" And toks(i)(1) = ";" And braceDepth = 0 And
283     parenDepth = 0 And bracketDepth = 0 Then
284         i = i + 1
285         Exit Do
286     End If
287     stmtTokens.Add toks(i)
288     i = i + 1
289 CollectNextTop:
290     Loop
291     If stmtTokens.Count > 0 Then
292         Set node = ParseStatementTokensToAST(stmtTokens)
293         If Not node Is Nothing Then
294             ' add a clone to avoid accidental reference reuse
295             stmtsAST.Add node.Clone
296         End If
297     End If
298     GoTo Compiler_MainLoop
299 End If
300 Loop
301 Dim OptionBase As Long: OptionBase = 1
302 Dim filteredStmts As New Collection
303 For i = 1 To stmtsAST.Count
304     Dim stmt As Map: Set stmt = stmtsAST(i)
305     If stmt.GetValue("type") = "OptionBase" Then
306         OptionBase = CLng(stmt.GetValue("value"))
307     Else
308         filteredStmts.Add stmt
309     End If
310 Next i
311 Set stmtsAST = filteredStmts
312 Dim pIndex As Long
313 Dim normalizedStmts As Collection
314 ' Normalize assignment-like expression-statements into proper Assign AST nodes
315 Set normalizedStmts = NormalizeAssignsInStmts(stmtsAST)
316 ' Normalize compound assignments (+=, -=, etc.)
317 Set normalizedStmts = NormalizeCompoundAssigns(normalizedStmts)
318 pIndex = GLOBALS_.gPrograms.Count + 1
319 GLOBALS_.gPrograms.Add Array(progName, normalizedStmts, progScope.Raw, OptionBase)
320 GLOBALS_.ProgramCache.Add src, pIndex
321 CompileProgram = pIndex
322 End Function
323
324 ' Compile a function body (AST) and register it as separate program
325 Private Function CompileFunctionAST(fname As String, argList As Collection, bodyTokens
326 As Collection) As Long
327     Dim funcScope As ScopeStack
328     Dim i As Long
329     Dim bodyStmtsAST As Collection
330     Set funcScope = New ScopeStack
331     funcScope.Push
332     For i = 1 To argList.Count
333         funcScope.SetValue argList(i), Empty
334     Next i
335     Dim idx As Long
336     Set bodyStmtsAST = ParseTokensToAST(bodyTokens)
337     ' Normalize assignments inside function body as well
338     Set bodyStmtsAST = NormalizeAssignsInStmts(bodyStmtsAST)

```

```

338     idx = GLOBALS_.gPrograms.Count + 1
339     GLOBALS_.gPrograms.Add Array(fname, bodyStmtsAST, funcScope.Raw)
340     CompileFunctionAST = idx
341 End Function
342
343 ' Parse a list of tokens representing a block into AST statements (returns Collection of
Map nodes)
344 Private Function ParseTokensToAST(toks As Collection) As Collection
345     Dim out As Collection
346     Dim stmtTokens As Collection
347     Set out = New Collection
348     Dim i As Long: i = 1
349     Dim closedTopLevelBlock As Boolean
350
351 ParseMain:
352     Do While i <= toks.Count
353         If toks(i)(0) = "COMMENT" Then
354             i = i + 1
355             GoTo ParseMain
356         End If
357         Set stmtTokens = New Collection
358         Dim braceDepth As Long: braceDepth = 0
359         Dim parenDepth As Long: parenDepth = 0
360         Dim bracketDepth As Long: bracketDepth = 0
361
362 ParseInner:
363         Do While i <= toks.Count
364             If toks(i)(0) = "COMMENT" Then
365                 i = i + 1
366                 GoTo ParseInner
367             End If
368             If toks(i)(0) = "PAREN" Then
369                 Select Case toks(i)(1)
370                     Case "{"
371                         braceDepth = braceDepth + 1
372                     Case "}"
373                         ' decrement only if we have an inner brace to close
374                         If braceDepth > 0 Then
375                             braceDepth = braceDepth - 1
376                         End If
377                         ' If we've just closed a top-level block (braceDepth now 0),
378                         ' that usually signals the end of a statement (e.g. end of an
if/for/while block).
379                         ' Finalize the current statement tokens so adjacent statements
are parsed separately.
380                         If braceDepth = 0 Then
381                             ' include this '}' token into the current stmtTokens and
finish the statement
382                             stmtTokens.Add toks(i)
383                             i = i + 1
384                             ' mark that we closed a top-level block for follow-up checks
385                             closedTopLevelBlock = True
386                             Exit Do
387                         End If
388                     Case "("
389                         parenDepth = parenDepth + 1
390                     Case ")"
391                         If parenDepth > 0 Then
392                             parenDepth = parenDepth - 1
393                         End If
394                     Case "["
395                         bracketDepth = bracketDepth + 1
396                     Case "]"
397                         If bracketDepth > 0 Then
398                             bracketDepth = bracketDepth - 1
399                         End If
400                 End Select
401             End If
402

```

```

403         ' Only a semicolon (;) is a top-level statement separator.
404         If toks(i)(0) = "SEP" And toks(i)(1) = ";" And braceDepth = 0 And parenDepth
         = 0 And bracketDepth = 0 Then
405             i = i + 1
406             Exit Do
407         End If
408
409         ' default: append current token and continue
410         stmtTokens.Add toks(i)
411         i = i + 1
412     Loop
413
414     If stmtTokens.Count > 0 Then
415         Dim tmpNode As Map
416         Set tmpNode = ParseStatementTokensToAST(stmtTokens)
417         If Not tmpNode Is Nothing Then
418             ' If we closed a top-level block and the next token is not a semicolon,
419             ' warn the user (do not silently swallow).
420             If closedTopLevelBlock Then
421                 If i <= toks.Count Then
422                     If Not (toks(i)(0) = "SEP" And toks(i)(1) = ";") Then
423                         ' If the next token is not a comment and not end-of-input,
424                         log warning.
425                         If Not (toks(i)(0) = "COMMENT") Then
426                             On Error Resume Next
427                             ' best-effort: add to runtime log if present, and print
428                             to Immediate
429                             If Not GLOBALS_.gRuntimeLog Is Nothing Then
430                                 GLOBALS_.gRuntimeLog.Add "COMPILER: missing ';' after
431                                 block near token index " & CStr(i)
432                             On Error GoTo 0
433                         End If
434                     Else
435                         ' consume explicit semicolon
436                         i = i + 1
437                     End If
438                 End If
439                 closedTopLevelBlock = False
440             End If
441             out.Add tmpNode.Clone
442         End If
443     End If
444     GoTo ParseMain
445 Loop
446
447 Set ParseTokensToAST = out
448 End Function
449
450 ' Parse a single statement token collection into an AST node (Map)
451 Private Function ParseStatementTokensToAST(stmtTokens As Collection) As Map
452     If stmtTokens.Count = 0 Then
453         Set ParseStatementTokensToAST = Nothing
454         Exit Function
455     End If
456
457     ' quick check for print
458     If stmtTokens.Count >= 2 Then
459         If stmtTokens(1)(0) = "IDENT" And LCase$(stmtTokens(1)(1)) = "print" Then
460             Dim k As Long: k = 2
461             Do While k <= stmtTokens.Count And Not (stmtTokens(k)(0) = "PAREN" And
462                 stmtTokens(k)(1) = "(")
463                 k = k + 1
464             Loop
465             If k <= stmtTokens.Count Then
466                 Dim opened As Long: opened = 1
467                 k = k + 1
468                 Dim innerToks As Collection
469                 Set innerToks = New Collection
470                 Do While k <= stmtTokens.Count And opened > 0

```

```

466         If stmtTokens(k)(0) = "PAREN" Then
467             If stmtTokens(k)(1) = "(" Then
468                 opened = opened + 1
469             ElseIf stmtTokens(k)(1) = ")" Then
470                 opened = opened - 1
471             End If
472         End If
473         If opened > 0 Then innerToks.Add stmtTokens(k)
474         k = k + 1
475     Loop
476     Dim args As Collection: Set args = ParseArgsTokensToExprNodes(innerToks)
477     Dim node As Map: Set node = MakeNode("Print")
478     node.SetValue "args", args
479     Set ParseStatementTokensToAST = node
480     Exit Function
481 End If
482 End If
483 End If
484
485 ' keywords
486 If stmtTokens(1)(0) = "IDENT" Then
487     Dim kw As String: kw = LCase$(stmtTokens(1)(1))
488     Select Case kw
489         Case "if"
490             Set ParseStatementTokensToAST = ParseIfAST(stmtTokens)
491             Exit Function
492         Case "for"
493             Set ParseStatementTokensToAST = ParseForAST(stmtTokens)
494             Exit Function
495         Case "while"
496             Set ParseStatementTokensToAST = ParseWhileAST(stmtTokens)
497             Exit Function
498         Case "try"
499             Set ParseStatementTokensToAST = ParseTryCatchAST(stmtTokens)
500             Exit Function
501         Case "switch"
502             Set ParseStatementTokensToAST = ParseSwitchAST(stmtTokens)
503             Exit Function
504         Case "break"
505             Dim b As Map: Set b = MakeNode("Break")
506             Set ParseStatementTokensToAST = b
507             Exit Function
508         Case "continue"
509             Dim c As Map: Set c = MakeNode("Continue")
510             Set ParseStatementTokensToAST = c
511             Exit Function
512         Case "return"
513             Dim rv As Map: Set rv = MakeNode("Return")
514             If stmtTokens.Count >= 2 Then
515                 Dim rhs As New Collection, m As Long
516                 For m = 2 To stmtTokens.Count: rhs.Add stmtTokens(m): Next m
517                 Dim rexr As Map: Set rexr = ParseExprTokensToNode(rhs)
518                 rv.SetValue "expr", rexr
519             Else
520                 rv.SetValue "expr", Nothing
521             End If
522             Set ParseStatementTokensToAST = rv
523             Exit Function
524         Case "option"
525             If stmtTokens.Count >= 3 And LCase$(stmtTokens(2)(1)) = "base" Then
526                 If stmtTokens.Count >= 4 And stmtTokens(3)(0) = "NUMBER" Then
527                     Dim baseVal As Long: baseVal = CLng(stmtTokens(3)(1))
528                     If baseVal = 0 Or baseVal = 1 Then
529                         ' Set as program flag - return special node
530                         Dim optNode As Map: Set optNode = MakeNode("OptionBase")
531                         optNode.SetValue "value", baseVal
532                         Set ParseStatementTokensToAST = optNode
533                         Exit Function
534                     End If

```

```

535         End If
536     End If
537 End Select
538 End If
539
540 ' assignment: find top-level "=" operator position
541 Dim assignPos As Long: assignPos = 0
542 Dim ii As Long
543 Dim depthB As Long, depthP As Long, depthBr As Long
544 For ii = 1 To stmtTokens.Count
545     If stmtTokens(ii)(0) = "PAREN" Then
546         Select Case stmtTokens(ii)(1)
547             Case "{"
548                 depthB = depthB + 1
549             Case "}"
550                 If depthB > 0 Then depthB = depthB - 1
551             Case "("
552                 depthP = depthP + 1
553             Case ")"
554                 If depthP > 0 Then depthP = depthP - 1
555             Case "["
556                 depthBr = depthBr + 1
557             Case "]"
558                 If depthBr > 0 Then depthBr = depthBr - 1
559         End Select
560     End If
561     If stmtTokens(ii)(0) = "OP" And stmtTokens(ii)(1) = "=" And depthB = 0 And
    depthP = 0 And depthBr = 0 Then
562         assignPos = ii
563     Exit For
564 End If
565 Next ii
566
567 If assignPos > 0 Then
568     Dim lhsT As Collection, rhsT As Collection, t As Long
569     Set lhsT = New Collection: Set rhsT = New Collection
570     For t = 1 To assignPos - 1: lhsT.Add stmtTokens(t): Next t
571     For t = assignPos + 1 To stmtTokens.Count: rhsT.Add stmtTokens(t): Next t
572     Dim lhsNode As Map
573     If lhsT.Count = 1 And lhsT(1)(0) = "IDENT" Then
574         Dim nm As String: nm = lhsT(1)(1)
575         If InStr(nm, "[") > 0 Then
576             Dim baseName As String: baseName = left$(nm, InStr(nm, "[") - 1)
577             Dim idxStr As String: idxStr = Mid$(nm, InStr(nm, "[") + 1, Len(nm) -
    InStr(nm, "[") - 1)
578             Dim idxNode As Map: Set idxNode = ParseExprFromStringToNode(idxStr)
579             Set lhsNode = MakeNode("Index")
580             Dim tmpVar As Map: Set tmpVar = MakeNode("Variable")
581             tmpVar.SetValue "name", baseName
582             lhsNode.SetValue "base", tmpVar
583             lhsNode.SetValue "index", idxNode
584         Else
585             Set lhsNode = MakeNode("Variable")
586             lhsNode.SetValue "name", nm
587         End If
588     Else
589         Set lhsNode = ParseExprTokensToNode(lhsT)
590     End If
591     Dim rhsNode As Map: Set rhsNode = ParseExprTokensToNode(rhsT)
592     Dim asn As Map: Set asn = MakeNode("Assign")
593     asn.SetValue "left", lhsNode
594     asn.SetValue "right", rhsNode
595     Set ParseStatementTokensToAST = asn
596     Exit Function
597 End If
598
599 ' fallback: expression statement
600 Dim exprNode As Map
601 Set exprNode = ParseExprTokensToNode(stmtTokens)

```

```

602     Dim es As Map: Set es = MakeNode("ExprStmt")
603     es.SetValue "expr", exprNode
604     Set ParseStatementTokensToAST = es
605 End Function
606
607 ' -----
608 ' Expression parsing -> Expr AST nodes
609 ' -----
610
611 Private Function ParseArgsTokensToExprNodes(argTokens As Collection) As Collection
612     Dim out As New Collection
613     If argTokens.Count = 0 Then
614         Set ParseArgsTokensToExprNodes = out
615         Exit Function
616     End If
617     Dim i As Long: i = 1
618     Dim braceDepth As Long: braceDepth = 0
619     Dim parenDepth As Long: parenDepth = 0
620     Dim bracketDepth As Long: bracketDepth = 0
621     Dim cur As New Collection
622 ParseArgsMain:
623     Do While i <= argTokens.Count
624         If argTokens(i)(0) = "PAREN" Then
625             Select Case argTokens(i)(1)
626                 Case "{"
627                     braceDepth = braceDepth + 1
628                 Case "}"
629                     If braceDepth > 0 Then braceDepth = braceDepth - 1
630                 Case "("
631                     parenDepth = parenDepth + 1
632                 Case ")"
633                     If parenDepth > 0 Then parenDepth = parenDepth - 1
634                 Case "["
635                     bracketDepth = bracketDepth + 1
636                 Case "]"
637                     If bracketDepth > 0 Then bracketDepth = bracketDepth - 1
638             End Select
639         End If
640         ' Argument/element separator: comma only (semicolon is NOT an argument separator)
641         If argTokens(i)(0) = "SEP" And argTokens(i)(1) = "," And braceDepth = 0 And
        parenDepth = 0 And bracketDepth = 0 Then
642             If cur.Count > 0 Then out.Add ParseExprTokensToNode(cur)
643             Set cur = New Collection
644             i = i + 1
645             GoTo ParseArgsMain
646         End If
647
648         cur.Add argTokens(i)
649         i = i + 1
650     Loop
651     If cur.Count > 0 Then out.Add ParseExprTokensToNode(cur)
652     Set ParseArgsTokensToExprNodes = out
653 End Function
654
655 Private Function ParseExprFromStringToNode(exprStr As String) As Map
656     Dim p As Parser
657     Set p = New Parser
658     Dim toks As Collection
659     Set toks = p.Tokenize(exprStr)
660     Set ParseExprFromStringToNode = ParseExprTokensToNode(toks)
661 End Function
662
663 Private Function ParseExprTokensToNode(toks As Collection) As Map
664     Dim n As Long: n = toks.Count
665     If n = 0 Then
666         Set ParseExprTokensToNode = MakeNode("Literal"): ParseExprTokensToNode.SetValue
        "value", Empty
667     Exit Function
668 End If

```

```

669
670     Dim types() As String, vals() As Variant, i As Long
671     ReDim types(0 To n - 1)
672     ReDim vals(0 To n - 1)
673     For i = 1 To n
674         types(i - 1) = toks(i)(0)
675         vals(i - 1) = toks(i)(1)
676     Next i
677
678     Dim idx As Long: idx = 0
679     Set ParseExprTokensToNode = ParseTernaryNode(types, vals, n, idx)
680 End Function
681
682 ' Recursive descent building nodes
683
684 Private Function ParseTernaryNode(types() As String, vals() As Variant, n As Long, ByRef
idx As Long) As Map
685     ' parse cond ? trueExpr : falseExpr (right-assoc)
686     Dim cond As Map
687     Set cond = ParseLogicalOrNode(types, vals, n, idx)
688     If idx < n Then
689         If types(idx) = "OP" And vals(idx) = "?" Then
690             idx = idx + 1
691             Dim trueExpr As Map
692             Set trueExpr = ParseTernaryNode(types, vals, n, idx) ' right-assoc: allow
nested ternary
693             ' expect ':'
694             If idx < n And types(idx) = "OP" And vals(idx) = ":" Then
695                 idx = idx + 1
696             Else
697                 err.Raise vbObjectError + 8001, "Compiler.ParseTernaryNode", "Expected
':' in ternary expression"
698             End If
699             Dim falseExpr As Map
700             Set falseExpr = ParseTernaryNode(types, vals, n, idx)
701             Dim tn As New Map
702             tn.Add "type", "Ternary"
703             tn.SetValue "cond", cond
704             tn.SetValue "trueExpr", trueExpr
705             tn.SetValue "falseExpr", falseExpr
706             Set ParseTernaryNode = tn
707             Exit Function
708         End If
709     End If
710     Set ParseTernaryNode = cond
711 End Function
712
713 Private Function ParseLogicalOrNode(types() As String, vals() As Variant, n As Long,
ByRef idx As Long) As Map
714     Dim left_ As Map
715     Dim right_ As Map
716     Dim node As Map
717     Dim op As String
718     Set left_ = ParseLogicalAndNode(types, vals, n, idx)
719     Do While idx < n
720         If Not (types(idx) = "OP" And vals(idx) = "||") Then Exit Do
721         op = vals(idx)
722         idx = idx + 1
723         Set right_ = ParseLogicalAndNode(types, vals, n, idx)
724         Set node = MakeNode("Binary")
725         node.SetValue "op", op
726         node.SetValue "left", left_.Clone
727         node.SetValue "right", right_.Clone
728         Set left_ = node
729     Loop
730     Set ParseLogicalOrNode = left_.Clone
731     Set left_ = Nothing
732     Set right_ = Nothing
733     Set node = Nothing

```

```

734 End Function
735
736 Private Function ParseLogicalAndNode(types() As String, vals() As Variant, n As Long,
ByRef idx As Long) As Map
737     Dim left_ As Map
738     Dim right_ As Map
739     Dim node As Map
740     Set left_ = ParseEqualityNode(types, vals, n, idx)
741     Do While idx < n
742         If Not (types(idx) = "OP" And vals(idx) = "&&") Then Exit Do
743         idx = idx + 1
744         Set right_ = ParseEqualityNode(types, vals, n, idx)
745         Set node = MakeNode("Binary")
746         node.SetValue "op", "&&"
747         node.SetValue "left", left_.Clone
748         node.SetValue "right", right_.Clone
749         Set left_ = node
750     Loop
751     Set ParseLogicalAndNode = left_.Clone
752     Set left_ = Nothing
753     Set right_ = Nothing
754     Set node = Nothing
755 End Function
756
757 Private Function ParseEqualityNode(types() As String, vals() As Variant, n As Long,
ByRef idx As Long) As Map
758     Dim left_ As Map
759     Dim right_ As Map
760     Dim node As Map
761     Dim op As String
762     Set left_ = ParseRelationalNode(types, vals, n, idx)
763     Do While idx < n
764         If Not (types(idx) = "OP" And (vals(idx) = "==" Or vals(idx) = "=" Or vals(idx) =
765             = "!=")) Then Exit Do
766         op = vals(idx)
767         idx = idx + 1
768         Set right_ = ParseRelationalNode(types, vals, n, idx)
769         Set node = MakeNode("Binary")
770         node.SetValue "op", op
771         node.SetValue "left", left_.Clone
772         node.SetValue "right", right_.Clone
773         Set left_ = node
774     Loop
775     Set ParseEqualityNode = left_.Clone
776     Set left_ = Nothing
777     Set right_ = Nothing
778     Set node = Nothing
779 End Function
780
781 Private Function ParseRelationalNode(types() As String, vals() As Variant, n As Long,
ByRef idx As Long) As Map
782     Dim left_ As Map
783     Dim right_ As Map
784     Dim node As Map
785     Dim op As String
786     Set left_ = ParseAddNode(types, vals, n, idx)
787     Do While idx < n
788         If Not (types(idx) = "OP" And (vals(idx) = "<" Or vals(idx) = ">" Or vals(idx) =
789             "<=" Or vals(idx) = ">=")) Then Exit Do
790         op = vals(idx)
791         idx = idx + 1
792         Set right_ = ParseAddNode(types, vals, n, idx)
793         Set node = MakeNode("Binary")
794         node.SetValue "op", op
795         node.SetValue "left", left_.Clone
796         node.SetValue "right", right_.Clone
797         Set left_ = node
798     Loop
799     Set ParseRelationalNode = left_.Clone

```

```

798     Set left_ = Nothing
799     Set right_ = Nothing
800     Set node = Nothing
801 End Function
802
803 Private Function ParseAddNode(types() As String, vals() As Variant, n As Long, ByRef idx
As Long) As Map
804     Dim left_ As Map
805     Dim op As String
806     Dim right_ As Map
807     Dim node As Map
808     Set left_ = ParseMulNode(types, vals, n, idx)
809     Do While idx < n
810         If Not (types(idx) = "OP" And (vals(idx) = "+" Or vals(idx) = "-")) Then Exit Do
811         op = vals(idx)
812         idx = idx + 1
813         Set right_ = ParseMulNode(types, vals, n, idx)
814         Set node = MakeNode("Binary")
815         node.SetValue "op", op
816         node.SetValue "left", left_.Clone
817         node.SetValue "right", right_.Clone
818         Set left_ = node
819     Loop
820     Set ParseAddNode = left_.Clone
821     Set left_ = Nothing
822     Set right_ = Nothing
823     Set node = Nothing
824 End Function
825
826 Private Function ParseMulNode(types() As String, vals() As Variant, n As Long, ByRef idx
As Long) As Map
827     Dim left_ As Map
828     Dim right_ As Map
829     Dim node As Map
830     Dim op As String
831     Set left_ = ParsePowNode(types, vals, n, idx)
832     Do While idx < n
833         If Not (types(idx) = "OP" And (vals(idx) = "*" Or vals(idx) = "/" Or vals(idx) =
"%")) Then Exit Do
834         op = vals(idx)
835         idx = idx + 1
836         Set right_ = ParsePowNode(types, vals, n, idx)
837         Set node = MakeNode("Binary")
838         node.SetValue "op", op
839         node.SetValue "left", left_.Clone
840         node.SetValue "right", right_.Clone
841         Set left_ = node
842     Loop
843     Set ParseMulNode = left_.Clone
844     Set left_ = Nothing
845     Set right_ = Nothing
846     Set node = Nothing
847 End Function
848
849 Private Function ParsePowNode(types() As String, vals() As Variant, n As Long, ByRef idx
As Long) As Map
850     Dim left_ As Map
851     Dim right_ As Map
852     Dim node As Map
853     Set left_ = ParseUnaryNode(types, vals, n, idx)
854     Do While idx < n
855         ' Right-associative exponentiation:
856         ' if we see '^' after the left operand, parse the RHS with ParsePow
857         ' so a ^ b ^ c => a ^ (b ^ c)
858         If Not (types(idx) = "OP" And vals(idx) = "^") Then Exit Do
859         idx = idx + 1
860         ' parse the right-hand side with ParsePow to ensure right-assoc
861         Set right_ = ParsePowNode(types, vals, n, idx)
862         Set node = MakeNode("Binary")

```

```

863         node.SetValue "op", "^"
864         node.SetValue "left", left_.Clone
865         node.SetValue "right", right_.Clone
866         Set left_ = node
867     Loop
868     Set ParsePowNode = left_.Clone
869     Set left_ = Nothing
870     Set right_ = Nothing
871     Set node = Nothing
872 End Function
873
874 Private Function ParseUnaryNode(types() As String, vals() As Variant, n As Long, ByRef
idx As Long) As Map
875     If idx < n And types(idx) = "OP" Then
876         Dim op As String: op = vals(idx)
877         If op = "!" Or op = "-" Then
878             idx = idx + 1
879             Dim v As Map: Set v = ParseUnaryNode(types, vals, n, idx)
880             Dim node As Map: Set node = MakeNode("Unary")
881             node.SetValue "op", op
882             node.SetValue "expr", v
883             Set ParseUnaryNode = node
884             Exit Function
885         End If
886     End If
887     Set ParseUnaryNode = ParsePrimaryNode(types, vals, n, idx)
888 End Function
889
890 Private Function ParsePrimaryNode(types() As String, vals() As Variant, n As Long, ByRef
idx As Long) As Map
891     If idx >= n Then
892         Set ParsePrimaryNode = MakeNode("Literal")
893         ParsePrimaryNode.SetValue "value", Empty
894         Exit Function
895     End If
896     Dim t As String: t = types(idx)
897     Dim v As Variant: v = vals(idx)
898
899     Select Case t
900     Case "NUMBER"
901         Dim nNode As Map: Set nNode = MakeNode("Literal")
902         nNode.SetValue "value", CDBl(v)
903         idx = idx + 1
904         Set ParsePrimaryNode = nNode
905         Exit Function
906     Case "STRING"
907         Dim sNode As Map: Set sNode = MakeNode("Literal")
908         sNode.SetValue "value", v
909         idx = idx + 1
910         Set ParsePrimaryNode = sNode
911         Exit Function
912     Case "VBEXPR"
913         ' Special token produced by the parser for @(...) or direct VBAexpressions
block.
914         ' Create a VBExpr node with the raw expression string (to be evaluated by
VBAexpressions at runtime).
915         Dim vbNode As Map: Set vbNode = MakeNode("VBExpr")
916         vbNode.SetValue "expr", CStr(v)
917         idx = idx + 1
918         Set ParsePrimaryNode = vbNode
919         Exit Function
920     Case "IDENT"
921         Dim name As String: name = v
922         Dim idxNode As Map
923         Dim idxExpr As Map
924         If LCase$(name) = "true" Then
925             Dim bt As Map: Set bt = MakeNode("Literal")
926             bt.SetValue "value", True
927             idx = idx + 1

```

```

928         Set ParsePrimaryNode = bt
929         Exit Function
930     ElseIf LCase$(name) = "false" Then
931         Dim bf As Map: Set bf = MakeNode("Literal")
932         bf.SetValue "value", False
933         idx = idx + 1
934         Set ParsePrimaryNode = bf
935         Exit Function
936 End If
937
938 ' Expression-level anonymous function literal: fun (p1, p2) { ... }
939 ' This reuses the same 'fun' token used for top-level function declarations
but emits
940 ' a FuncLiteral node usable inside expressions.
941 If LCase$(name) = "fun" Then
942     ' advance past 'fun'
943     idx = idx + 1
944     ' expect parameter list
945     Dim params As Collection
946     Set params = New Collection
947     If idx < n And types(idx) = "PAREN" And vals(idx) = "(" Then
948         idx = idx + 1
949         Do While idx < n And Not (types(idx) = "PAREN" And vals(idx) = ")")
950             If types(idx) = "IDENT" Then
951                 params.Add CStr(vals(idx))
952                 idx = idx + 1
953                 If idx < n And types(idx) = "SEP" And vals(idx) = "," Then
954                     idx = idx + 1
955                 ElseIf types(idx) = "SEP" And vals(idx) = "," Then
956                     idx = idx + 1
957                 Else
958                     ' skip unexpected tokens until ')'
959                     idx = idx + 1
960                 End If
961             Loop
962             If idx < n And types(idx) = "PAREN" And vals(idx) = ")" Then idx =
idx + 1
963         End If
964     ' parse body { ... }
965     If idx < n And types(idx) = "PAREN" And vals(idx) = "{" Then
966         idx = idx + 1
967         Dim depthBody As Long: depthBody = 0
968         Dim bodyTok As Collection
969         Set bodyTok = New Collection
970         Do While idx < n
971             If types(idx) = "PAREN" Then
972                 If vals(idx) = "{" Then
973                     depthBody = depthBody + 1
974                 ElseIf vals(idx) = "}" Then
975                     If depthBody = 0 Then
976                         Exit Do
977                     Else
978                         depthBody = depthBody - 1
979                     End If
980                 End If
981             End If
982             bodyTok.Add Array(types(idx), vals(idx))
983             idx = idx + 1
984         Loop
985         ' consume closing '}'
986         If idx < n And types(idx) = "PAREN" And vals(idx) = "}" Then idx =
idx + 1
987         Dim bodyStmts As Collection: Set bodyStmts =
ParseTokensToAST(bodyTok)
988         Dim fnode As Map: Set fnode = MakeNode("FuncLiteral")
989         fnode.SetValue "params", params
990         fnode.SetValue "body", bodyStmts
991         Set ParsePrimaryNode = fnode.Clone

```

```

992         Exit Function
993     Else
994         ' no body found -> emit empty function literal
995         Dim fnode2 As Map: Set fnode2 = MakeNode("FuncLiteral")
996         fnode2.SetValue "params", params
997         Dim emptyCol As New Collection
998         fnode2.SetValue "body", emptyCol
999         Set ParsePrimaryNode = fnode2
1000     Exit Function
1001 End If
1002 End If
1003
1004 Dim lenCall As Map
1005 Dim builtinLen As Map
1006 Dim al As Collection
1007 ' Handle collapsed identifiers with array/index and optional trailing
1008 ' ".length"
1009 ' Examples:
1010 '   a[3]
1011 '   a[3].length
1012 If InStr(name, "[") > 0 Then
1013     Dim base As String: base = left$(name, InStr(name, "[") - 1)
1014     ' Find matching closing bracket for the first '[' (supports nested
1015     ' brackets)
1016     Dim posOpen As Long: posOpen = InStr(name, "[")
1017     Dim posClose As Long: posClose = 0
1018     Dim depthBr As Long: depthBr = 0
1019     Dim iCh As Long
1020     For iCh = posOpen To Len(name)
1021         Dim ch As String: ch = Mid$(name, iCh, 1)
1022         If ch = "[" Then
1023             depthBr = depthBr + 1
1024         ElseIf ch = "]" Then
1025             depthBr = depthBr - 1
1026             If depthBr = 0 Then
1027                 posClose = iCh
1028                 Exit For
1029             End If
1030         End If
1031     Next iCh
1032     If posClose = 0 Then
1033         err.Raise vbObjectError + 8002, "Compiler.ParsePrimary", "Invalid
1034         collapsed identifier: missing ']' "
1035     End If
1036     Dim idxStr As String: idxStr = Mid$(name, posOpen + 1, posClose -
1037     posOpen - 1)
1038     Dim idxNodeFromStr As Map: Set idxNodeFromStr =
1039     ParseExprFromStringToNode(idxStr)
1040     Dim inNode As Map: Set inNode = MakeNode("Index")
1041     Dim vn As Map: Set vn = MakeNode("Variable")
1042     vn.SetValue "name", base
1043     inNode.SetValue "base", vn
1044     inNode.SetValue "index", idxNodeFromStr
1045     idx = idx + 1
1046     ' If next tokens are ". length" treat as builtin length call on the
1047     indexed result.
1048     If idx < n Then
1049         If types(idx) = "SYM" And vals(idx) = "." Then
1050             If (idx + 1) < n And types(idx + 1) = "IDENT" And
1051             LCase$(CStr(vals(idx + 1))) = "length" Then
1052                 ' consume '.' and 'length'
1053                 idx = idx + 2
1054                 Set lenCall = MakeNode("Call")
1055                 ' set both 'callee' and 'name' for maximum compatibility
1056                 with VM variants
1057                 Set builtinLen = MakeNode("Variable")
1058                 builtinLen.SetValue "name", ".__len__"
1059                 lenCall.SetValue "callee", builtinLen
1060                 lenCall.SetValue "name", ".__len__"

```

```

1053             Set al = New Collection
1054             al.Add inNode
1055             lenCall.SetValue "args", al
1056             Set ParsePrimaryNode = lenCall
1057             Exit Function
1058         End If
1059     End If
1060 End If
1061 Set ParsePrimaryNode = inNode
1062 Exit Function
1063 End If
1064 ' if collapsed dotted/indexed form => convert into AST nodes
1065 If InStr(name, ".") Then
1066     Dim complexNode As Map
1067     Set complexNode = ParseCollapsedIdentToNode(name)
1068     If Not complexNode Is Nothing Then
1069         idx = idx + 1
1070         ' return currentNode (no postfix loop needed because we've already
            built the chain)
1071         Set ParsePrimaryNode = complexNode
1072         Exit Function
1073     End If
1074 End If
1075 ' create initial variable node for an IDENT and then apply postfix operators:
1076 Dim currentNode As Map: Set currentNode = MakeNode("Variable")
1077 currentNode.SetValue "name", name
1078 idx = idx + 1
1079
1080 ' Postfix loop: handle .prop, [...], func calls (...) as postfixes chaining
    onto currentNode
1081 Do While idx < n
1082     ' member access a.b
1083     If types(idx) = "SYM" And vals(idx) = "." Then
1084         idx = idx + 1
1085         If idx < n And types(idx) = "IDENT" Then
1086             Dim mem As Map: Set mem = MakeNode("Member")
1087             mem.SetValue "base", currentNode
1088             mem.SetValue "prop", CStr(vals(idx))
1089             Set currentNode = mem.Clone
1090             idx = idx + 1
1091             ' continue loop
1092         Else
1093             ' invalid member access; stop postfixing
1094             Exit Do
1095         End If
1096     ' index access a[expr]
1097     ElseIf types(idx) = "PAREN" And vals(idx) = "[" Then
1098         idx = idx + 1
1099         Dim idxTok As Collection
1100         Set idxTok = New Collection
1101         Dim depthIdx As Long: depthIdx = 0
1102         Do While idx < n
1103             If types(idx) = "PAREN" Then
1104                 If vals(idx) = "[" Then
1105                     depthIdx = depthIdx + 1
1106                 ElseIf vals(idx) = "]" Then
1107                     If depthIdx = 0 Then
1108                         Exit Do
1109                     Else
1110                         depthIdx = depthIdx - 1
1111                     End If
1112                 End If
1113             End If
1114             idxTok.Add Array(types(idx), vals(idx))
1115             idx = idx + 1
1116         Loop
1117         Dim idxExprNode As Map: Set idxExprNode =
            ParseExprTokensToNode(idxTok)
1118         If idx < n And types(idx) = "PAREN" And vals(idx) = "]" Then idx =

```

```

1119         idx + 1
1120         Dim indexNode As Map: Set indexNode = MakeNode("Index")
1121         indexNode.SetValue "base", currentNode
1122         indexNode.SetValue "index", idxExprNode
1123         Set currentNode = indexNode.Clone
1124     ' function / method call: ( arglist )
1125     ElseIf types(idx) = "PAREN" And vals(idx) = "(" Then
1126         ' parse args with nested depth counters (borrowed pattern from
1127         previous call parsing logic)
1128         idx = idx + 1
1129         Dim argNodes As Collection
1130         Dim argTok As Collection
1131         Dim argTokDepthParen As Long
1132         Dim argTokDepthBr As Long
1133         Dim argTokDepthB As Long
1134         Set argNodes = New Collection
1135         Do
1136             Set argTok = New Collection
1137             argTokDepthParen = 0
1138             argTokDepthBr = 0
1139             argTokDepthB = 0
1140             Do While idx < n
1141                 If types(idx) = "PAREN" Then
1142                     Select Case CStr(vals(idx))
1143                     Case "("
1144                         argTokDepthParen = argTokDepthParen + 1
1145                     Case ")"
1146                         If argTokDepthParen = 0 Then Exit Do Else
1147                             argTokDepthParen = argTokDepthParen - 1
1148                     Case "["
1149                         argTokDepthBr = argTokDepthBr + 1
1150                     Case "]"
1151                         If argTokDepthBr = 0 Then Exit Do Else
1152                             argTokDepthBr = argTokDepthBr - 1
1153                     Case "{"
1154                         argTokDepthB = argTokDepthB + 1
1155                     Case "}"
1156                         If argTokDepthB = 0 Then Exit Do Else
1157                             argTokDepthB = argTokDepthB - 1
1158                     End Select
1159                 ElseIf types(idx) = "SEP" And vals(idx) = "," And
1160                     argTokDepthParen = 0 And argTokDepthBr = 0 And argTokDepthB
1161                     = 0 Then
1162                     Exit Do
1163                 End If
1164                 argTok.Add Array(types(idx), vals(idx))
1165                 idx = idx + 1
1166             Loop
1167             If argTok.Count > 0 Then
1168                 argNodes.Add ParseExprTokensToNode(argTok)
1169             Else
1170                 Dim litEmpty As Map: Set litEmpty = MakeNode("Literal")
1171                 litEmpty.SetValue "value", Empty
1172                 argNodes.Add litEmpty.Clone
1173             End If
1174             If idx < n Then
1175                 If types(idx) = "SEP" And vals(idx) = "," Then
1176                     idx = idx + 1
1177                     ' continue parsing next arg
1178                 ElseIf types(idx) = "PAREN" And vals(idx) = ")" Then
1179                     idx = idx + 1
1180                     Exit Do
1181                 Else
1182                     ' unexpected token -> stop args parsing
1183                     Exit Do
1184                 End If
1185             Else
1186                 Exit Do
1187             End If
1188         End If

```

```

1181         Loop
1182         Dim callNode As Map: Set callNode = MakeNode("Call")
1183         callNode.SetValue "callee", currentNode
1184         callNode.SetValue "args", argNodes
1185         Set currentNode = callNode.Clone
1186     Else
1187         Exit Do
1188     End If
1189 Loop
1190 Set ParsePrimaryNode = currentNode.Clone
1191 Exit Function
1192 Case "PAREN"
1193     If v = "(" Then
1194         idx = idx + 1
1195         Dim innerNode As Map: Set innerNode = ParseLogicalOrNode(types, vals, n,
1196             idx)
1197         If idx < n And types(idx) = "PAREN" And vals(idx) = ")" Then idx = idx +
1198             1
1199         Set ParsePrimaryNode = innerNode.Clone
1200         Exit Function
1201     ElseIf v = "[" Then
1202         idx = idx + 1
1203         Dim arrList As Collection
1204         Dim elemTok As Collection
1205         Set arrList = New Collection
1206         Do While idx < n And Not (types(idx) = "PAREN" And vals(idx) = "]")
1207             Set elemTok = New Collection
1208             ' Collect element tokens until a comma (element separator) or
1209             closing bracket
1210             Do While idx < n And Not ((types(idx) = "SEP" And vals(idx) = ",")
1211                 Or (types(idx) = "PAREN" And vals(idx) = "]"))
1212                 elemTok.Add Array(types(idx), vals(idx))
1213                 idx = idx + 1
1214             Loop
1215             arrList.Add ParseExprTokensToNode(elemTok)
1216             If idx < n And types(idx) = "SEP" And vals(idx) = "," Then idx = idx
1217                 + 1
1218         Loop
1219         If idx < n And types(idx) = "PAREN" And vals(idx) = "]" Then idx = idx +
1220             1
1221         Dim arrNode As Map: Set arrNode = MakeNode("Array")
1222         arrNode.SetValue "items", arrList
1223         Set ParsePrimaryNode = arrNode.Clone
1224         Exit Function
1225     ElseIf v = "{" Then
1226         ' object literal: { key: value, key2: value2 }
1227         idx = idx + 1
1228         Dim objItems As Collection
1229         Dim pair As Collection
1230         Set objItems = New Collection
1231         ' key can be IDENT or STRING
1232         Dim keyTok As String
1233         Do While idx < n And Not (types(idx) = "PAREN" And vals(idx) = "}")
1234             If types(idx) = "IDENT" Or types(idx) = "STRING" Then
1235                 keyTok = CStr(vals(idx))
1236                 idx = idx + 1
1237             Else
1238                 err.Raise vbObjectError + 8002, "Compiler.ParsePrimary",
1239                     "Invalid object key"
1240             End If
1241             ' expect ':'
1242             If idx < n And types(idx) = "OP" And vals(idx) = ":" Then
1243                 idx = idx + 1
1244             Else
1245                 err.Raise vbObjectError + 8003, "Compiler.ParsePrimary",
1246                     "Expected ':' after object key"
1247             End If
1248             ' parse value expression
1249             Dim valNode As Map

```

```

1242         Set valNode = ParseTernaryNode(types, vals, n, idx)
1243         ' store pair (key, node)
1244         Set pair = New Collection
1245         pair.Add keyTok
1246         pair.Add valNode.Clone
1247         objItems.Add pair
1248         ' optional comma
1249         If idx < n And types(idx) = "SEP" And vals(idx) = "," Then idx = idx
            + 1
1250     Loop
1251     If idx < n And types(idx) = "PAREN" And vals(idx) = "}" Then idx = idx +
        1
1252     Dim onode As Map: Set onode = MakeNode("Object")
1253     onode.SetValue "items", objItems
1254     Set ParsePrimaryNode = onode.Clone
1255     Exit Function
1256 End If
1257 End Select
1258
1259 Dim defN As Map: Set defN = MakeNode("Literal")
1260 defN.SetValue "value", Empty
1261 Set ParsePrimaryNode = defN
1262 End Function
1263
1264 ' -----
1265 ' AST builders for control structures
1266 ' -----
1267
1268 Private Function ParseIfAST(stmtTokens As Collection) As Map
1269     Dim node As Map: Set node = MakeNode("If")
1270     Dim i As Long: i = 2
1271     Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
        stmtTokens(i)(1) = "(")
1272         i = i + 1
1273     Loop
1274     If i > stmtTokens.Count Then Set ParseIfAST = node: Exit Function
1275     i = i + 1
1276     Dim condTokens As Collection
1277     Set condTokens = New Collection
1278     Dim condDepth As Long: condDepth = 0
1279     Do While i <= stmtTokens.Count
1280         If stmtTokens(i)(0) = "PAREN" Then
1281             If stmtTokens(i)(1) = "(" Then
1282                 condDepth = condDepth + 1
1283             ElseIf stmtTokens(i)(1) = ")" Then
1284                 If condDepth = 0 Then
1285                     Exit Do
1286                 Else
1287                     condDepth = condDepth - 1
1288                 End If
1289             End If
1290         End If
1291         condTokens.Add stmtTokens(i)
1292         i = i + 1
1293     Loop
1294     node.SetValue "cond", ParseExprTokensToNode(condTokens)
1295
1296     Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
        stmtTokens(i)(1) = "{")
1297         i = i + 1
1298     Loop
1299     If i > stmtTokens.Count Then Set ParseIfAST = node.Clone: GoTo clean_
1300     Dim j As Long: j = i + 1
1301     Dim depth As Long: depth = 0
1302     Dim thenTokens As Collection
1303     Set thenTokens = New Collection
1304     Do While j <= stmtTokens.Count
1305         If stmtTokens(j)(0) = "PAREN" Then
1306             If stmtTokens(j)(1) = "{" Then

```

```

1307         depth = depth + 1
1308     ElseIf stmtTokens(j)(1) = "}" Then
1309         If depth = 0 Then
1310             Exit Do
1311         Else
1312             depth = depth - 1
1313         End If
1314     End If
1315 End If
1316 thenTokens.Add stmtTokens(j)
1317 j = j + 1
1318 Loop
1319 node.SetValue "then", ParseTokensToAST(thenTokens)
1320
1321 Dim pos As Long: pos = j + 1
1322 Dim elseifConds As Collection, elseifBlocks As Collection
1323 Dim hasElse As Boolean: hasElse = False
1324 Dim tname As String
1325 Dim eCondTok As Collection
1326 Dim eBlockTok As Collection
1327 Dim elseTok As Collection
1328 Dim eDepth As Long
1329 Set elseifConds = New Collection
1330 Set elseifBlocks = New Collection
1331 Do While pos <= stmtTokens.Count
1332     If stmtTokens(pos)(0) = "COMMENT" Then
1333         pos = pos + 1
1334         GoTo IfParseNext
1335     End If
1336     If stmtTokens(pos)(0) = "IDENT" Then
1337         tname = LCase$(stmtTokens(pos)(1))
1338         If tname = "elseif" Then
1339             pos = pos + 1
1340             Do While pos <= stmtTokens.Count And Not (stmtTokens(pos)(0) = "PAREN"
1341                 And stmtTokens(pos)(1) = "(")
1342                 pos = pos + 1
1343             Loop
1344             If pos > stmtTokens.Count Then Exit Do
1345             pos = pos + 1
1346             eDepth = 0
1347             Set eCondTok = New Collection
1348             Do While pos <= stmtTokens.Count
1349                 If stmtTokens(pos)(0) = "PAREN" Then
1350                     If stmtTokens(pos)(1) = "(" Then
1351                         eDepth = eDepth + 1
1352                     ElseIf stmtTokens(pos)(1) = ")" Then
1353                         If eDepth = 0 Then
1354                             Exit Do
1355                         Else
1356                             eDepth = eDepth - 1
1357                         End If
1358                     End If
1359                 End If
1360                 eCondTok.Add stmtTokens(pos)
1361                 pos = pos + 1
1362             Loop
1363             Do While pos <= stmtTokens.Count And Not (stmtTokens(pos)(0) = "PAREN"
1364                 And stmtTokens(pos)(1) = "{")
1365                 pos = pos + 1
1366             Loop
1367             If pos > stmtTokens.Count Then Exit Do
1368             pos = pos + 1
1369             Set eBlockTok = New Collection
1370             depth = 0
1371             Do While pos <= stmtTokens.Count
1372                 If stmtTokens(pos)(0) = "PAREN" Then
1373                     If stmtTokens(pos)(1) = "{" Then
1374                         depth = depth + 1
1375                     ElseIf stmtTokens(pos)(1) = "}" Then
1376                         If depth = 0 Then
1377                             Exit Do
1378                         Else
1379                             depth = depth - 1
1380                         End If
1381                     End If
1382                 End If
1383                 eBlockTok.Add stmtTokens(pos)
1384                 pos = pos + 1
1385             Loop
1386             If eCondTok.Count > 0 Then
1387                 elseifConds.Add eCondTok
1388             End If
1389             If eBlockTok.Count > 0 Then
1390                 elseifBlocks.Add eBlockTok
1391             End If
1392             pos = pos + 1
1393         End If
1394     End If
1395 End While
1396 node.SetValue "elseif", ParseTokensToAST(elseifConds, elseifBlocks, hasElse, tname)
1397
1398 Dim stmtTok As Collection
1399 Set stmtTok = New Collection
1400 Do While pos <= stmtTokens.Count
1401     If stmtTokens(pos)(0) = "COMMENT" Then
1402         pos = pos + 1
1403     Else
1404         stmtTok.Add stmtTokens(pos)
1405         pos = pos + 1
1406     End If
1407 End While
1408 node.SetValue "stmt", ParseTokensToAST(stmtTok)
1409 End Sub

```

```

1374             If depth = 0 Then
1375                 Exit Do
1376             Else
1377                 depth = depth - 1
1378             End If
1379         End If
1380     End If
1381     eBlockTok.Add stmtTokens(pos)
1382     pos = pos + 1
1383     Loop
1384     elseifConds.Add ParseExprTokensToNode(eCondTok)
1385     elseifBlocks.Add ParseTokensToAST(eBlockTok)
1386     pos = pos + 1
1387     GoTo IfParseNext
1388 ElseIf tname = "else" Then
1389     pos = pos + 1
1390     Do While pos <= stmtTokens.Count And Not (stmtTokens(pos)(0) = "PAREN"
1391         And stmtTokens(pos)(1) = "{")
1392         pos = pos + 1
1393     Loop
1394     If pos > stmtTokens.Count Then Exit Do
1395     pos = pos + 1
1396     Set elseTok = New Collection
1397     depth = 0
1398     Do While pos <= stmtTokens.Count
1399         If stmtTokens(pos)(0) = "PAREN" Then
1400             If stmtTokens(pos)(1) = "{" Then
1401                 depth = depth + 1
1402             ElseIf stmtTokens(pos)(1) = "}" Then
1403                 If depth = 0 Then
1404                     Exit Do
1405                 Else
1406                     depth = depth - 1
1407                 End If
1408             End If
1409             elseTok.Add stmtTokens(pos)
1410             pos = pos + 1
1411         Loop
1412         hasElse = True
1413         node.SetValue "else", ParseTokensToAST(elseTok)
1414         Exit Do
1415     Else
1416         Exit Do
1417     End If
1418 Else
1419     Exit Do
1420 End If
1421 IfParseNext:
1422     Loop
1423
1424     node.SetValue "elseif_conds", CloneCollectionOfVariants(elseifConds)
1425     node.SetValue "elseif_blocks", CloneCollectionOfVariants(elseifBlocks)
1426     node.SetValue "hasElse", hasElse
1427     Set ParseIfAST = node
1428 clean_:
1429     Set node = Nothing
1430     Set eCondTok = Nothing
1431     Set eBlockTok = Nothing
1432     Set elseTok = Nothing
1433     Set elseifConds = Nothing
1434     Set elseifBlocks = Nothing
1435 End Function
1436
1437 Private Function ParseForAST(stmtTokens As Collection) As Map
1438     Dim node As Map: Set node = MakeNode("For")
1439     Dim initTok As Collection
1440     Dim condTok As Collection
1441     Dim stepTok As Collection

```

```

1442 Dim bodyToks As Collection
1443 Dim i As Long: i = 2
1444 Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
    stmtTokens(i)(1) = "(")
1445     i = i + 1
1446 Loop
1447 If i > stmtTokens.Count Then Set ParseForAST = node: Exit Function
1448 i = i + 1
1449 Set initTok = New Collection
1450 Dim hdrParen As Long, hdrBracket As Long, hdrBrace As Long
1451 hdrParen = 0: hdrBracket = 0: hdrBrace = 0
1452 Do While i <= stmtTokens.Count
1453     If stmtTokens(i)(0) = "PAREN" Then
1454         Select Case stmtTokens(i)(1)
1455             Case "("
1456                 hdrParen = hdrParen + 1
1457             Case ")"
1458                 If hdrParen > 0 Then
1459                     hdrParen = hdrParen - 1
1460                 Else
1461                     Exit Do
1462                 End If
1463             Case "["
1464                 hdrBracket = hdrBracket + 1
1465             Case "]"
1466                 If hdrBracket > 0 Then hdrBracket = hdrBracket - 1
1467             Case "{"
1468                 hdrBrace = hdrBrace + 1
1469             Case "}"
1470                 If hdrBrace > 0 Then hdrBrace = hdrBrace - 1
1471         End Select
1472     End If
1473     If hdrParen = 0 And hdrBracket = 0 And hdrBrace = 0 And stmtTokens(i)(0) = "SEP"
    And _
1474         (stmtTokens(i)(1) = "," Or stmtTokens(i)(1) = ";") Then
1475         i = i + 1
1476         Exit Do
1477     End If
1478     initTok.Add stmtTokens(i)
1479     i = i + 1
1480 Loop
1481 node.SetValue "init", ParseExprTokensToNode(initTok)
1482
1483 Set condTok = New Collection
1484 hdrParen = 0: hdrBracket = 0: hdrBrace = 0
1485 Do While i <= stmtTokens.Count
1486     If stmtTokens(i)(0) = "PAREN" Then
1487         Select Case stmtTokens(i)(1)
1488             Case "("
1489                 hdrParen = hdrParen + 1
1490             Case ")"
1491                 If hdrParen > 0 Then
1492                     hdrParen = hdrParen - 1
1493                 Else
1494                     Exit Do
1495                 End If
1496             Case "["
1497                 hdrBracket = hdrBracket + 1
1498             Case "]"
1499                 If hdrBracket > 0 Then hdrBracket = hdrBracket - 1
1500             Case "{"
1501                 hdrBrace = hdrBrace + 1
1502             Case "}"
1503                 If hdrBrace > 0 Then hdrBrace = hdrBrace - 1
1504         End Select
1505     End If
1506     If hdrParen = 0 And hdrBracket = 0 And hdrBrace = 0 And stmtTokens(i)(0) = "SEP"
    And _
1507         (stmtTokens(i)(1) = "," Or stmtTokens(i)(1) = ";") Then

```

```

1508         i = i + 1
1509         Exit Do
1510     End If
1511     condTok.Add stmtTokens(i)
1512     i = i + 1
1513 Loop
1514 node.SetValue "cond", ParseExprTokensToNode(condTok)
1515
1516 Set stepTok = New Collection
1517 hdrParen = 0: hdrBracket = 0: hdrBrace = 0
1518 Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
stmtTokens(i)(1) = ")")
1519     If stmtTokens(i)(0) = "PAREN" Then
1520         Select Case stmtTokens(i)(1)
1521             Case "("
1522                 hdrParen = hdrParen + 1
1523             Case ")"
1524                 If hdrParen > 0 Then hdrParen = hdrParen - 1
1525             Case "["
1526                 hdrBracket = hdrBracket + 1
1527             Case "]"
1528                 If hdrBracket > 0 Then hdrBracket = hdrBracket - 1
1529             Case "{"
1530                 hdrBrace = hdrBrace + 1
1531             Case "}"
1532                 If hdrBrace > 0 Then hdrBrace = hdrBrace - 1
1533         End Select
1534     End If
1535     stepTok.Add stmtTokens(i)
1536     i = i + 1
1537 Loop
1538 node.SetValue "step", ParseExprTokensToNode(stepTok)
1539
1540 Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
stmtTokens(i)(1) = "{")
1541     i = i + 1
1542 Loop
1543 If i > stmtTokens.Count Then Set ParseForAST = node.Clone: GoTo clean_
1544 Dim j As Long: j = i + 1
1545 Dim depth As Long: depth = 0
1546 Set bodyToks = New Collection
1547 Do While j <= stmtTokens.Count
1548     If stmtTokens(j)(0) = "PAREN" Then
1549         If stmtTokens(j)(1) = "{" Then
1550             depth = depth + 1
1551         ElseIf stmtTokens(j)(1) = "}" Then
1552             If depth = 0 Then
1553                 Exit Do
1554             Else
1555                 depth = depth - 1
1556             End If
1557         End If
1558     End If
1559     bodyToks.Add stmtTokens(j)
1560     j = j + 1
1561 Loop
1562 node.SetValue "body", ParseTokensToAST(bodyToks)
1563 Set ParseForAST = node.Clone
1564 clean_:
1565 Set node = Nothing
1566 Set bodyToks = Nothing
1567 Set stepTok = Nothing
1568 Set condTok = Nothing
1569 Set initTok = Nothing
1570 End Function
1571
1572 Private Function ParseWhileAST(stmtTokens As Collection) As Map
1573     Dim node As Map: Set node = MakeNode("While")
1574     Dim i As Long: i = 2

```

```

1575 Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
1576     stmtTokens(i)(1) = "(")
1577     i = i + 1
1578 Loop
1579 If i > stmtTokens.Count Then Set ParseWhileAST = node: Exit Function
1580 i = i + 1
1581 Dim condTok As Collection
1582 Set condTok = New Collection
1583 Dim depthP As Long: depthP = 0
1584 Do While i <= stmtTokens.Count
1585     If stmtTokens(i)(0) = "PAREN" Then
1586         If stmtTokens(i)(1) = "(" Then
1587             depthP = depthP + 1
1588         ElseIf stmtTokens(i)(1) = ")" Then
1589             If depthP = 0 Then
1590                 Exit Do
1591             Else
1592                 depthP = depthP - 1
1593             End If
1594         End If
1595     End If
1596     condTok.Add stmtTokens(i)
1597     i = i + 1
1598 Loop
1599 node.SetValue "cond", ParseExprTokensToNode(condTok)
1600 Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
1601     stmtTokens(i)(1) = "{")
1602     i = i + 1
1603 Loop
1604 If i > stmtTokens.Count Then Set ParseWhileAST = node: Exit Function
1605 Dim j As Long: j = i + 1
1606 Dim depth As Long: depth = 0
1607 Dim bodyToks As Collection
1608 Set bodyToks = New Collection
1609 Do While j <= stmtTokens.Count
1610     If stmtTokens(j)(0) = "PAREN" Then
1611         If stmtTokens(j)(1) = "{" Then
1612             depth = depth + 1
1613         ElseIf stmtTokens(j)(1) = "}" Then
1614             If depth = 0 Then
1615                 Exit Do
1616             Else
1617                 depth = depth - 1
1618             End If
1619         End If
1620     End If
1621     bodyToks.Add stmtTokens(j)
1622     j = j + 1
1623 Loop
1624 node.SetValue "body", ParseTokensToAST(bodyToks)
1625 Set ParseWhileAST = node
1626 End Function
1627 Private Function ParseTryCatchAST(stmtTokens As Collection) As Map
1628     Dim node As Map: Set node = MakeNode("TryCatch")
1629     Dim i As Long: i = 2
1630     Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
1631         stmtTokens(i)(1) = "{")
1632         i = i + 1
1633 Loop
1634 If i > stmtTokens.Count Then Set ParseTryCatchAST = node: Exit Function
1635 Dim j As Long: j = i + 1
1636 Dim depth As Long: depth = 0
1637 Dim tryTok As Collection
1638 Set tryTok = New Collection
1639 Do While j <= stmtTokens.Count
1640     If stmtTokens(j)(0) = "PAREN" Then
1641         If stmtTokens(j)(1) = "{" Then

```

```

1641         depth = depth + 1
1642     ElseIf stmtTokens(j)(1) = "}" Then
1643         If depth = 0 Then
1644             Exit Do
1645         Else
1646             depth = depth - 1
1647         End If
1648     End If
1649 End If
1650 tryTok.Add stmtTokens(j)
1651 j = j + 1
1652 Loop
1653 node.SetValue "try", ParseTokensToAST(tryTok)
1654
1655 Dim k As Long: k = j + 1
1656 Dim catchTok As Collection
1657 Set catchTok = New Collection
1658 Do While k <= stmtTokens.Count
1659     If stmtTokens(k)(0) = "IDENT" And LCase$(stmtTokens(k)(1)) = "catch" Then
1660         Dim kk As Long: kk = k + 1
1661         Do While kk <= stmtTokens.Count And Not (stmtTokens(kk)(0) = "PAREN" And
1662             stmtTokens(kk)(1) = "{")
1663             kk = kk + 1
1664         Loop
1665         If kk <= stmtTokens.Count Then
1666             kk = kk + 1
1667             Dim depth2 As Long: depth2 = 0
1668             Do While kk <= stmtTokens.Count
1669                 If stmtTokens(kk)(0) = "PAREN" Then
1670                     If stmtTokens(kk)(1) = "{" Then
1671                         depth2 = depth2 + 1
1672                     ElseIf stmtTokens(kk)(1) = "}" Then
1673                         If depth2 = 0 Then
1674                             Exit Do
1675                         Else
1676                             depth2 = depth2 - 1
1677                         End If
1678                     End If
1679                 End If
1680                 catchTok.Add stmtTokens(kk)
1681                 kk = kk + 1
1682             Loop
1683         End If
1684     End If
1685     k = k + 1
1686 Loop
1687 node.SetValue "catch", ParseTokensToAST(catchTok)
1688 Set ParseTryCatchAST = node
1689 End Function
1690
1691 Private Function ParseSwitchAST(stmtTokens As Collection) As Map
1692     Dim node As Map: Set node = MakeNode("Switch")
1693     Dim i As Long: i = 2
1694     Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
1695         stmtTokens(i)(1) = "(")
1696         i = i + 1
1697     Loop
1698     If i > stmtTokens.Count Then Set ParseSwitchAST = node: Exit Function
1699     i = i + 1
1700     Dim condTok As New Collection
1701     Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
1702         stmtTokens(i)(1) = ")")
1703         condTok.Add stmtTokens(i)
1704         i = i + 1
1705     Loop
1706     node.SetValue "expr", ParseExprTokensToNode(condTok)
1707     Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And

```

```

1707     stmtTokens(i)(1) = "{"
1708     i = i + 1
1709 Loop
1709 If i > stmtTokens.Count Then Set ParseSwitchAST = node: Exit Function
1710 i = i + 1
1711 Dim cases As Collection
1712 Set cases = New Collection
1713 Dim defaultBlock As Collection
1714 Dim valTok As Collection
1715 Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
stmtTokens(i)(1) = "}")
1716     If stmtTokens(i)(0) = "IDENT" And LCase$(stmtTokens(i)(1)) = "case" Then
1717         i = i + 1
1718         Set valTok = New Collection
1719         Dim cvParen As Long: cvParen = 0
1720         Dim cvBracket As Long: cvBracket = 0
1721         Dim cvBrace As Long: cvBrace = 0
1722         Do While i <= stmtTokens.Count
1723             If stmtTokens(i)(0) = "PAREN" Then
1724                 Select Case stmtTokens(i)(1)
1725                     Case "("
1726                         cvParen = cvParen + 1
1727                     Case ")"
1728                         If cvParen > 0 Then cvParen = cvParen - 1
1729                     Case "["
1730                         cvBracket = cvBracket + 1
1731                     Case "]"
1732                         If cvBracket > 0 Then cvBracket = cvBracket - 1
1733                     Case "{"
1734                         If cvParen = 0 And cvBracket = 0 Then
1735                             Exit Do
1736                         Else
1737                             cvBrace = cvBrace + 1
1738                         End If
1739                     Case "}"
1740                         If cvBrace > 0 Then cvBrace = cvBrace - 1
1741                 End Select
1742             End If
1743             valTok.Add stmtTokens(i)
1744             i = i + 1
1745         Loop
1746         Dim caseValNode As Map: Set caseValNode = ParseExprTokensToNode(valTok)
1747         If i <= stmtTokens.Count And stmtTokens(i)(0) = "PAREN" And stmtTokens(i)(1)
= "{" Then
1748             i = i + 1
1749             Dim depth As Long: depth = 0
1750             Dim blockTok As Collection
1751             Dim pair As Collection
1752             Dim caseBlock As Collection
1753             Set blockTok = New Collection
1754             Do While i <= stmtTokens.Count
1755                 If stmtTokens(i)(0) = "PAREN" Then
1756                     If stmtTokens(i)(1) = "{" Then
1757                         depth = depth + 1
1758                     ElseIf stmtTokens(i)(1) = "}" Then
1759                         If depth = 0 Then
1760                             Exit Do
1761                         Else
1762                             depth = depth - 1
1763                         End If
1764                     End If
1765                 End If
1766                 blockTok.Add stmtTokens(i)
1767                 i = i + 1
1768             Loop
1769             Set caseBlock = ParseTokensToAST(blockTok)
1770             Set pair = New Collection
1771             pair.Add caseValNode
1772             pair.Add caseBlock

```

```

1773         cases.Add pair
1774         i = i + 1
1775     End If
1776     ElseIf stmtTokens(i)(0) = "IDENT" And LCase$(stmtTokens(i)(1)) = "default" Then
1777         i = i + 1
1778         Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
1779             stmtTokens(i)(1) = "{")
1780             i = i + 1
1781         Loop
1782         If i <= stmtTokens.Count And stmtTokens(i)(0) = "PAREN" And stmtTokens(i)(1)
1783             = "{" Then
1784             i = i + 1
1785             Dim depth2 As Long: depth2 = 0
1786             Dim defTok As New Collection
1787             Do While i <= stmtTokens.Count
1788                 If stmtTokens(i)(0) = "PAREN" Then
1789                     If stmtTokens(i)(1) = "{" Then
1790                         depth2 = depth2 + 1
1791                     ElseIf stmtTokens(i)(1) = "}" Then
1792                         If depth2 = 0 Then
1793                             Exit Do
1794                         Else
1795                             depth2 = depth2 - 1
1796                         End If
1797                     End If
1798                 End If
1799                 defTok.Add stmtTokens(i)
1800                 i = i + 1
1801             Loop
1802             Set defaultBlock = ParseTokensToAST(defTok)
1803             i = i + 1
1804         End If
1805     Else
1806         i = i + 1
1807     End If
1808 Loop
1809 node.SetValue "cases", cases
1810 node.SetValue "default", defaultBlock
1811 Set ParseSwitchAST = node
1812 End Function
1813
1814 ' Compiler AST normalization pass
1815 ' Converts expression statements like (Binary op "=" ...) into Assign nodes
1816 ' when the left-hand side is a valid assignment target (Variable or Index).
1817 ' Also normalizes `for` init/step and recursively visits blocks.
1818
1819 Private Function IsAssignableNode(n As Map) As Boolean
1820     If n Is Nothing Then
1821         IsAssignableNode = False: Exit Function
1822     End If
1823     Dim t As String: t = n.GetValue("type")
1824     ' Accept variable, indexed access (arr[i]) and member access (obj.prop)
1825     If t = "Variable" Or t = "Index" Or t = "Member" Then
1826         IsAssignableNode = True
1827     Else
1828         IsAssignableNode = False
1829     End If
1830 End Function
1831
1832 Private Function NormalizeExprNodeRecursive(expr As Map) As Map
1833     ' Recursively walk expression nodes and normalize inner expressions as needed.
1834     If expr Is Nothing Then
1835         Set NormalizeExprNodeRecursive = Nothing: Exit Function
1836     End If
1837     Dim t As String: t = expr.GetValue("type")
1838     Select Case t
1839         Case "Binary"
1840             Dim left_ As Map: Set left_ =

```

```

1840     NormalizeExprNodeRecursive(expr.GetValue("left"))
1841     Dim right_ As Map: Set right_ =
1842     NormalizeExprNodeRecursive(expr.GetValue("right"))
1843     ' create new Binary node only if children changed, otherwise return original
1844     If (Not left_ Is expr.GetValue("left")) Or (Not right_ Is
1845     expr.GetValue("right")) Then
1846         Dim nb As New Map
1847         nb.Add "type", "Binary"
1848         nb.SetValue "op", expr.GetValue("op")
1849         nb.SetValue "left", IIf(left_ Is Nothing, expr.GetValue("left"), left_)
1850         nb.SetValue "right", IIf(right_ Is Nothing, expr.GetValue("right"),
1851         right_)
1852         Set NormalizeExprNodeRecursive = nb
1853         Exit Function
1854     End If
1855     Set NormalizeExprNodeRecursive = expr
1856     Exit Function
1857 Case "Unary"
1858     Dim rec As Map: Set rec = NormalizeExprNodeRecursive(expr.GetValue("expr"))
1859     If Not rec Is expr.GetValue("expr") Then
1860         Dim nu As New Map
1861         nu.Add "type", "Unary"
1862         nu.SetValue "op", expr.GetValue("op")
1863         nu.SetValue "expr", rec
1864         Set NormalizeExprNodeRecursive = nu
1865         Exit Function
1866     End If
1867     Set NormalizeExprNodeRecursive = expr
1868     Exit Function
1869 Case "Call"
1870     ' normalize args
1871     Dim args As Collection: Set args = expr.GetValue("args")
1872     If Not args Is Nothing Then
1873         Dim newArgs As New Collection
1874         Dim changed As Boolean: changed = False
1875         Dim i As Long
1876         For i = 1 To args.Count
1877             Dim ae As Map: Set ae = NormalizeExprNodeRecursive(args(i))
1878             If Not ae Is args(i) Then changed = True: newArgs.Add ae Else
1879             newArgs.Add args(i)
1880         Next i
1881         If changed Then
1882             Dim nc As New Map
1883             nc.Add "type", "Call"
1884             nc.SetValue "name", expr.GetValue("name")
1885             nc.SetValue "args", newArgs
1886             Set NormalizeExprNodeRecursive = nc
1887             Exit Function
1888         End If
1889     End If
1890     Set NormalizeExprNodeRecursive = expr
1891     Exit Function
1892 Case "Index"
1893     Dim b As Map: Set b = NormalizeExprNodeRecursive(expr.GetValue("base"))
1894     Dim idx As Map: Set idx = NormalizeExprNodeRecursive(expr.GetValue("index"))
1895     If Not b Is expr.GetValue("base") Or Not idx Is expr.GetValue("index") Then
1896         Dim ni As New Map
1897         ni.Add "type", "Index"
1898         ni.SetValue "base", IIf(b Is Nothing, expr.GetValue("base"), b)
1899         ni.SetValue "index", IIf(idx Is Nothing, expr.GetValue("index"), idx)
1900         Set NormalizeExprNodeRecursive = ni
1901         Exit Function
1902     End If
1903     Set NormalizeExprNodeRecursive = expr
1904     Exit Function
1905 Case "Array"
1906     Dim items As Collection: Set items = expr.GetValue("items")
1907     If Not items Is Nothing Then
1908         Dim newit As New Collection

```

```

1904         Dim ch As Boolean: ch = False
1905         Dim ii As Long
1906         For ii = 1 To items.Count
1907             Dim ei As Map: Set ei = NormalizeExprNodeRecursive(items(ii))
1908             If Not ei Is items(ii) Then ch = True: newit.Add ei Else newit.Add
                items(ii)
1909         Next ii
1910         If ch Then
1911             Dim na As New Map
1912             na.Add "type", "Array"
1913             na.SetValue "items", newit
1914             Set NormalizeExprNodeRecursive = na
1915             Exit Function
1916         End If
1917     End If
1918     Set NormalizeExprNodeRecursive = expr
1919     Exit Function
1920 Case Else
1921     Set NormalizeExprNodeRecursive = expr
1922     Exit Function
1923 End Select
1924 End Function
1925
1926 Private Function NormalizeNodeRecursive(node As Map) As Map
1927     ' Normalize a statement node and recursively process inner blocks.
1928     If node Is Nothing Then
1929         Set NormalizeNodeRecursive = Nothing: Exit Function
1930     End If
1931     Dim expr As Map
1932     Dim left_ As Map
1933     Dim a As Map
1934     Dim r As Map
1935     Dim newExpr As Map
1936     Dim ns As Map
1937     Dim cond As Map
1938     Dim thenBlk As Collection
1939     Dim outThen As Collection
1940     Dim elseifConds As Collection
1941     Dim elseifBlocks As Collection
1942     Dim outElseIfConds As Collection
1943     Dim outElseIfBlocks As Collection
1944     Dim eb As Collection
1945     Dim nb As Collection
1946     Dim elseBlk As Collection
1947     Dim outElse As Collection
1948     Dim nif As Map
1949     Dim initNode As Map
1950     Dim condNode As Map
1951     Dim stepNode As Map
1952     Dim body As Collection
1953     Dim nInit As Map, nCond As Map, nStep As Map
1954     Dim na As Map
1955     Dim L As Map
1956     Dim e As Map
1957     Dim L2 As Map
1958     Dim na2 As Map
1959     Dim SL As Map
1960     Dim na3 As Map
1961     Dim newBody As Collection
1962     Dim nf As Map
1963     Dim cnd As Map
1964     Dim bdy As Collection
1965     Dim nbdy As Collection
1966     Dim nw As Map
1967     Dim tryBlk As Collection
1968     Dim catchBlk As Collection
1969     Dim nt As Collection, nc As Collection
1970     Dim ntc As Map
1971     Dim cases As Collection

```

```

1972 Dim ncases As Collection
1973 Dim pair As Collection
1974 Dim caseExpr As Map
1975 Dim blockStmts As Collection
1976 Dim newPair As Collection
1977 Dim def As Collection
1978 Dim ndef As Collection
1979 Dim nsw As Map
1980 Dim leftA As Map
1981 Dim rightA As Map
1982 Dim naNode As Map
1983 Dim tp As String: tp = node.GetValue("type")
1984 Select Case tp
1985     Case "ExprStmt"
1986         Set expr = node.GetValue("expr")
1987         ' If expression is a Binary "=" with assignable LHS, convert to Assign node.
1988         If Not expr Is Nothing Then
1989             If expr.GetValue("type") = "Binary" Then
1990                 If CStr(expr.GetValue("op")) = "=" Then
1991                     Set left_ = expr.GetValue("left")
1992                     If Not left_ Is Nothing Then
1993                         If IsAssignableNode(left_) Then
1994                             a.Add "type", "Assign"
1995                             a.SetValue "left", left_
1996                             ' right: normalize recursively as expression
1997                             Set r =
1998                                 NormalizeExprNodeRecursive(expr.GetValue("right"))
1999                             If r Is Nothing Then Set r = expr.GetValue("right")
2000                             a.SetValue "right", r
2001                             Set NormalizeNodeRecursive = a
2002                             Exit Function
2003                         End If
2004                     End If
2005                 End If
2006             End If
2007             ' otherwise just normalize inner expression (if any)
2008             If Not expr Is Nothing Then
2009                 Set newExpr = NormalizeExprNodeRecursive(expr)
2010                 If Not newExpr Is expr Then
2011                     Set ns = New Map
2012                     ns.Add "type", "ExprStmt"
2013                     ns.SetValue "expr", newExpr
2014                     Set NormalizeNodeRecursive = ns
2015                     Exit Function
2016                 End If
2017             End If
2018             Set NormalizeNodeRecursive = node
2019             Exit Function
2020
2021     Case "If"
2022         Set cond = NormalizeExprNodeRecursive(node.GetValue("cond"))
2023         Set thenBlk = node.GetValue("then")
2024         Dim i As Long
2025         If Not thenBlk Is Nothing Then
2026             Set outThen = New Collection
2027             For i = 1 To thenBlk.Count
2028                 outThen.Add NormalizeNodeRecursive(thenBlk(i))
2029             Next i
2030         End If
2031         Set elseifConds = node.GetValue("elseif_conds")
2032         Set elseifBlocks = node.GetValue("elseif_blocks")
2033         Set outElseIfConds = New Collection
2034         Set outElseIfBlocks = New Collection
2035         If Not elseifConds Is Nothing Then
2036             For i = 1 To elseifConds.Count
2037                 outElseIfConds.Add NormalizeExprNodeRecursive(elseifConds(i))
2038             Next i
2039         End If

```

```

2040     If Not elseifBlocks Is Nothing Then
2041         For i = 1 To elseifBlocks.Count
2042             Set eb = elseifBlocks(i)
2043             Set nb = New Collection
2044             Dim j As Long
2045             For j = 1 To eb.Count
2046                 nb.Add NormalizeNodeRecursive(eb(j))
2047             Next j
2048             outElseIfBlocks.Add nb
2049         Next i
2050     End If
2051     Set elseBlk = node.GetValue("else")
2052     If Not elseBlk Is Nothing Then
2053         Set outElse = New Collection
2054         For i = 1 To elseBlk.Count
2055             outElse.Add NormalizeNodeRecursive(elseBlk(i))
2056         Next i
2057     End If
2058     Set nif = New Map
2059     nif.Add "type", "If"
2060     nif.SetValue "cond", IIf(cond Is Nothing, node.GetValue("cond"), cond)
2061     nif.SetValue "then", outThen
2062     nif.SetValue "elseif_conds", outElseIfConds
2063     nif.SetValue "elseif_blocks", outElseIfBlocks
2064     nif.SetValue "hasElse", node.GetValue("hasElse")
2065     If Not outElse Is Nothing Then nif.SetValue "else", outElse
2066     Set NormalizeNodeRecursive = nif
2067     Exit Function
2068
2069 Case "For"
2070     ' normalize init/cond/step and body
2071     Set initNode = node.GetValue("init")
2072     Set condNode = node.GetValue("cond")
2073     Set stepNode = node.GetValue("step")
2074     Set body = node.GetValue("body")
2075     If Not initNode Is Nothing Then
2076         If initNode.GetValue("type") = "Binary" And
2077             CStr(initNode.GetValue("op")) = "=" Then
2078             Set L = initNode.GetValue("left")
2079             If Not L Is Nothing And IsAssignableNode(L) Then
2080                 Set na = New Map
2081                 na.Add "type", "Assign"
2082                 na.SetValue "left", L
2083                 na.SetValue "right",
2084                     NormalizeExprNodeRecursive(initNode.GetValue("right"))
2085                 Set nInit = na
2086             Else
2087                 Set nInit = NormalizeExprNodeRecursive(initNode)
2088             End If
2089         ElseIf initNode.GetValue("type") = "ExprStmt" Then
2090             ' exprstmt could wrap a binary expression
2091             Set e = initNode.GetValue("expr")
2092             If Not e Is Nothing And e.GetValue("type") = "Binary" And
2093                 CStr(e.GetValue("op")) = "=" Then
2094                 Set L2 = e.GetValue("left")
2095                 If Not L2 Is Nothing And IsAssignableNode(L2) Then
2096                     Set na2 = New Map
2097                     na2.Add "type", "Assign"
2098                     na2.SetValue "left", L2
2099                     na2.SetValue "right",
2100                         NormalizeExprNodeRecursive(e.GetValue("right"))
2101                     Set nInit = na2
2102                 Else
2103                     Set nInit = NormalizeExprNodeRecursive(e)
2104                 End If
2105             Else
2106                 Set nInit = NormalizeExprNodeRecursive(e)
2107             End If
2108         Else
2109             Set nInit = NormalizeExprNodeRecursive(e)
2110         End If
2111     Else
2112         Set nInit = NormalizeExprNodeRecursive(e)
2113     End If
2114 Else

```

```

2105         Set nInit = NormalizeExprNodeRecursive(initNode)
2106     End If
2107 End If
2108 If Not condNode Is Nothing Then Set nCond =
NormalizeExprNodeRecursive(condNode)
2109 If Not stepNode Is Nothing Then
2110     If stepNode.GetValue("type") = "Binary" And
CStr(stepNode.GetValue("op")) = "=" Then
2111         Set SL = stepNode.GetValue("left")
2112         If Not SL Is Nothing And IsAssignableNode(SL) Then
2113             Set na3 = New Map
2114             na3.Add "type", "Assign"
2115             na3.SetValue "left", SL
2116             na3.SetValue "right",
NormalizeExprNodeRecursive(stepNode.GetValue("right"))
2117             Set nStep = na3
2118         Else
2119             Set nStep = NormalizeExprNodeRecursive(stepNode)
2120         End If
2121     Else
2122         Set nStep = NormalizeExprNodeRecursive(stepNode)
2123     End If
2124 End If
2125 Set newBody = New Collection
2126 If Not body Is Nothing Then
2127     Dim bidx As Long
2128     For bidx = 1 To body.Count
2129         newBody.Add NormalizeNodeRecursive(body(bidx))
2130     Next bidx
2131 End If
2132 Set nf = New Map
2133 nf.Add "type", "For"
2134 If Not nInit Is Nothing Then nf.SetValue "init", nInit
2135 If Not nCond Is Nothing Then nf.SetValue "cond", nCond
2136 If Not nStep Is Nothing Then nf.SetValue "step", nStep
2137 nf.SetValue "body", newBody
2138 Set NormalizeNodeRecursive = nf
2139 Exit Function
2140
2141 Case "While"
2142     Set cnd = NormalizeExprNodeRecursive(node.GetValue("cond"))
2143     Set bdy = node.GetValue("body")
2144     Set nbdy = New Collection
2145     If Not bdy Is Nothing Then
2146         Dim bi As Long
2147         For bi = 1 To bdy.Count
2148             nbdy.Add NormalizeNodeRecursive(bdy(bi))
2149         Next bi
2150     End If
2151     Set nw = New Map
2152     nw.Add "type", "While"
2153     nw.SetValue "cond", IIf(cnd Is Nothing, node.GetValue("cond"), cnd)
2154     nw.SetValue "body", nbdy
2155     Set NormalizeNodeRecursive = nw
2156     Exit Function
2157
2158 Case "TryCatch"
2159     Set tryBlk = node.GetValue("try")
2160     Set catchBlk = node.GetValue("catch")
2161     Set nt = New Collection: Set nc = New Collection
2162     Dim ti As Long
2163     If Not tryBlk Is Nothing Then
2164         For ti = 1 To tryBlk.Count: nt.Add NormalizeNodeRecursive(tryBlk(ti)):
Next ti
2165     End If
2166     If Not catchBlk Is Nothing Then
2167         For ti = 1 To catchBlk.Count: nc.Add
NormalizeNodeRecursive(catchBlk(ti)): Next ti
2168     End If

```

```

2169         Set ntc = New Map
2170         ntc.Add "type", "TryCatch"
2171         ntc.SetValue "try", nt
2172         ntc.SetValue "catch", nc
2173         Set NormalizeNodeRecursive = ntc
2174         Exit Function
2175
2176     Case "Switch"
2177         Set cases = node.GetValue("cases")
2178         Set ncases = New Collection
2179         If Not cases Is Nothing Then
2180             Dim ci As Long
2181             For ci = 1 To cases.Count
2182                 Set pair = cases(ci)
2183                 Set caseExpr = NormalizeExprNodeRecursive(pair(1))
2184                 Set blockStmts = pair(2)
2185                 Set nb = New Collection
2186                 Dim bi2 As Long
2187                 For bi2 = 1 To blockStmts.Count
2188                     nb.Add NormalizeNodeRecursive(blockStmts(bi2))
2189                 Next bi2
2190                 Set newPair = New Collection
2191                 newPair.Add caseExpr
2192                 newPair.Add nb
2193                 ncases.Add newPair
2194             Next ci
2195         End If
2196         Set def = node.GetValue("default")
2197         If Not def Is Nothing Then
2198             Set ndef = New Collection
2199             Dim di As Long
2200             For di = 1 To def.Count: ndef.Add NormalizeNodeRecursive(def(di)): Next
                di
2201         End If
2202         Set nsw = New Map
2203         nsw.Add "type", "Switch"
2204         nsw.SetValue "expr", NormalizeExprNodeRecursive(node.GetValue("expr"))
2205         nsw.SetValue "cases", ncases
2206         If Not ndef Is Nothing Then nsw.SetValue "default", ndef
2207         Set NormalizeNodeRecursive = nsw
2208         Exit Function
2209
2210     Case Else
2211         ' Assign, Print, Return, Break, Continue, etc. often have expr fields to
        normalize.
2212         If node.GetValue("type") = "Assign" Then
2213             Set leftA = node.GetValue("left")
2214             Set rightA = NormalizeExprNodeRecursive(node.GetValue("right"))
2215             Set naNode = New Map
2216             naNode.Add "type", "Assign"
2217             naNode.SetValue "left", leftA
2218             naNode.SetValue "right", IIf(rightA Is Nothing, node.GetValue("right"),
                rightA)
2219             Set NormalizeNodeRecursive = naNode
2220             Exit Function
2221         End If
2222         Set NormalizeNodeRecursive = node
2223         Exit Function
2224     End Select
2225 End Function
2226
2227 Private Function NormalizeAssignsInStmts(stmts As Collection) As Collection
2228     Dim out As New Collection
2229     If stmts Is Nothing Then
2230         Set NormalizeAssignsInStmts = out: Exit Function
2231     End If
2232     Dim i As Long
2233     Dim n As Map
2234     For i = 1 To stmts.Count

```

```

2235         Set n = stmts(i)
2236         out.Add NormalizeNodeRecursive(n)
2237     Next i
2238     Set NormalizeAssignsInStmts = out
2239 End Function
2240
2241 ' Normalize compound assignments like a += b -> Assign(a, Binary(a, +, b))
2242 Private Function NormalizeCompoundAssigns(stmts As Collection) As Collection
2243     Dim out As New Collection
2244     If stmts Is Nothing Then
2245         Set NormalizeCompoundAssigns = out: Exit Function
2246     End If
2247     Dim i As Long
2248     Dim e As Map
2249     Dim n As Map
2250     Dim op As String
2251     Dim left_ As Map
2252     Dim right_ As Map
2253     Dim bin As Map
2254     Dim baseOp As String
2255     Dim a As Map
2256     For i = 1 To stmts.Count
2257         Set n = stmts(i)
2258         If Not n Is Nothing And n.GetValue("type") = "ExprStmt" Then
2259             Set e = n.GetValue("expr")
2260             If Not e Is Nothing And e.GetValue("type") = "Binary" Then
2261                 op = CStr(e.GetValue("op"))
2262                 Select Case op
2263                     Case "+=", "-=", "*=", "/=", "%=", "^="
2264                         ' build Assign node: left = left <op> right
2265                         Set left_ = e.GetValue("left")
2266                         Set right_ = e.GetValue("right")
2267                         Set bin = New Map
2268                         bin.Add "type", "Binary"
2269                         baseOp = left$(op, Len(op) - 1)
2270                         bin.SetValue "op", baseOp
2271                         ' left needs to be used as a copy for the RHS binary left
2272                         operand
2273                         bin.SetValue "left", left_
2274                         bin.SetValue "right", right_
2275                         Set a = New Map
2276                         a.Add "type", "Assign"
2277                         a.SetValue "left", left_
2278                         a.SetValue "right", bin
2279                         out.Add a
2280                         GoTo NormNext
2281                     End Select
2282                 End If
2283             ' default: pass-through (but recursively normalize inner blocks too if desired)
2284             out.Add n
2285 NormNext:
2286         Next i
2287         Set NormalizeCompoundAssigns = out
2288     End Function
2289 ' -----
2290 ' End Compiler class
2291 ' -----
2292
2293 ' -----
2294 ' VM class
2295 ' -----
2296 ' Class Module: VM (AST executor)
2297 Option Explicit
2298 Private Const VTP_MAX_DEPTH As Long = 8 ' safety depth limit
2299 Private Const VTP_MAX_ITEMS_INLINE As Long = 8 ' prefer inline for small containers
2300 Private VERBOSE_ As Boolean
2301 Private GLOBALS_ As Globals
2302 Private OUTPUT_ As Variant

```

```

2303
2304 ' Executes AST nodes produced by Compiler (AST).
2305 ' Uses Map node types and ScopeStack. Logs to gRuntimeLog.
2306
2307 Public Sub SetGlobals(aGlobals As Globals)
2308     Set GLOBALS_ = aGlobals
2309 End Sub
2310 Public Property Get OUTPUT_() As Variant
2311     vAssignment OUTPUT_, OUTPUT__
2312 End Property
2313 Public Property Get Verbose() As Boolean
2314     Verbose = VERBOSE_
2315 End Property
2316 Public Property Let Verbose(aValue As Boolean)
2317     VERBOSE_ = aValue
2318 End Property
2319
2320 ' Node helpers (Map-based)
2321 Private Function MakeNode(nodeType As String) As Map
2322     Dim m As New Map
2323     m.Add "type", nodeType
2324     Set MakeNode = m
2325 End Function
2326
2327 Public Sub RunProgramByIndex(idx As Long)
2328     GLOBALS_.ASF_InitGlobals
2329     If idx < 1 Or idx > GLOBALS_.gPrograms.Count Then Exit Sub
2330     Dim p As Variant: p = GLOBALS_.gPrograms(idx)
2331     Dim progName As String: progName = p(0)
2332     Dim stmts As Collection: Set stmts = p(1)
2333     Dim rawScope As Collection: Set rawScope = p(2)
2334     Dim OptionBase As Long: OptionBase = IIf(UBound(p) >= 3, p(3), 1)
2335     Dim progScope As New ScopeStack
2336     progScope.LoadRaw rawScope
2337     progScope.SetValue "__option_base", OptionBase
2338     progScope.Push
2339     If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "RUN Program: " & progName
2340     Dim i As Long
2341     For i = 1 To stmts.Count
2342         Dim ctrl As String
2343         ctrl = ExecuteStmtNode(stmts(i), progScope)
2344         If ctrl = "RETURN" Then
2345             vAssignment OUTPUT__, progScope.GetValue("__return")
2346             Exit For
2347         End If
2348         If ctrl = "ERR" Then Exit For
2349     Next i
2350     progScope.Pop
2351 End Sub
2352
2353 ' Execute a statement node (Map). Return control signals: "", "BREAK", "CONTINUE",
2354 "RETURN", "ERR"
2355 Private Function ExecuteStmtNode(node As Map, progScope As ScopeStack) As String
2356     On Error GoTo ErrHandler
2357     Dim tp As String: tp = node.GetValue("type")
2358     Dim rval As Variant
2359     Dim i As Long
2360     Select Case tp
2361         Case "Print"
2362             Dim args As Collection: Set args = node.GetValue("args")
2363             Dim outParts As New Collection
2364             For i = 1 To args.Count
2365                 Dim v As Variant: vAssignment v, EvalExprNode(args(i), progScope)
2366                 outParts.Add ValueToStringForPrint(v)
2367             Next i
2368             Dim sb As String: sb = ""
2369             For i = 1 To outParts.Count
2370                 If i > 1 Then sb = sb & ", "
2371                 sb = sb & outParts(i)

```

```

2371         Next i
2372         If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "PRINT:" & sb
2373         Debug.Print sb
2374         ExecuteStmtNode = ""
2375         Exit Function
2376
2377     Case "Assign"
2378         Dim left As Map: Set left = node.GetValue("left")
2379         Dim right As Map: Set right = node.GetValue("right")
2380         vAssignment rval, EvalExprNode(right, progScope)
2381         HandleAssignment left, rval, progScope
2382         If left.GetValue("type") = "Variable" Then
2383             Dim lName As String: lName = left.GetValue("name")
2384         End If
2385         ExecuteStmtNode = ""
2386         Exit Function
2387
2388     Case "ExprStmt"
2389         Dim res As Variant: res = EvalExprNode(node.GetValue("expr"), progScope)
2390         ExecuteStmtNode = ""
2391         Exit Function
2392
2393     Case "If"
2394         ExecuteStmtNode = ExecIfNode(node, progScope)
2395         Exit Function
2396
2397     Case "For"
2398         ExecuteStmtNode = ExecForNode(node, progScope)
2399         Exit Function
2400
2401     Case "While"
2402         ExecuteStmtNode = ExecWhileNode(node, progScope)
2403         Exit Function
2404
2405     Case "Break"
2406         ExecuteStmtNode = "BREAK": Exit Function
2407     Case "Continue"
2408         ExecuteStmtNode = "CONTINUE": Exit Function
2409
2410     Case "Return"
2411         Dim rex As Map: Set rex = node.GetValue("expr")
2412         If Not rex Is Nothing Then vAssignment rval, EvalExprNode(rex, progScope)
2413         Else rval = Empty
2414         progScope.SetValue "__return", rval
2415         ExecuteStmtNode = "RETURN": Exit Function
2416
2417     Case "TryCatch"
2418         ExecuteStmtNode = ExecTryCatchNode(node, progScope)
2419         Exit Function
2420
2421     Case "Switch"
2422         ExecuteStmtNode = ExecSwitchNode(node, progScope)
2423         Exit Function
2424
2425     Case Else
2426         ' unknown node type
2427         If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "Unknown statement node: " & tp
2428         ExecuteStmtNode = ""
2429         Exit Function
2430 End Select
2431
2432 ErrHandler:
2433     If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "VM statement error: " & err.Description
2434     err.Clear
2435     ExecuteStmtNode = "ERR"
2436 End Function
2437
2438 ' Reuse your AssignToArray logic
2439 Private Sub AssignToArray(arrName As String, idxV As Variant, val As Variant, progScope

```

```

As ScopeStack)
2439     Dim arr As Variant
2440     Dim pos As Long
2441     Dim ub As Long
2442     Dim lb As Long
2443     Dim OptionBase As Long: OptionBase = progScope.GetValue("__option_base")
2444
2445     arr = progScope.GetValue(arrName)
2446     pos = CLng(idxV)
2447     If pos < OptionBase Then
2448         err.Raise vbObjectError + 5001, "VM.AssignToArray", "Invalid array index (must
            be >=1)"
2449     End If
2450     If Not IsArray(arr) Then
2451         If IsEmpty(arr) Then
2452             arr = Array()
2453         Else
2454             err.Raise vbObjectError + 5002, "VM.AssignToArray", "Not an array"
2455         End If
2456     End If
2457     ub = -1
2458     If IsArray(arr) Then
2459         ub = UBound(arr)
2460     End If
2461     If ub < 0 Then
2462         ReDim arr(1 To pos)
2463     Else
2464         lb = LBound(arr)
2465         If pos > (ub - lb + 1) Then
2466             ReDim Preserve arr(lb To lb + (pos - OptionBase))
2467         End If
2468     End If
2469     arr(LBound(arr) + (pos - OptionBase)) = val
2470     progScope.SetValue arrName, arr
2471 End Sub
2472
2473 Private Function EvalMemberNode(node As Map, progScope As ScopeStack) As Variant
2474     Dim baseVal As Variant
2475     vAssignment baseVal, EvalExprNode(node.GetValue("base"), progScope)
2476     If TypeName(baseVal) <> "Map" Then err.Raise vbObjectError + 5007,
        "VM.EvalMemberNode", "Cannot access property on non-map"
2477     vAssignment EvalMemberNode, baseVal.GetValue(node.GetValue("prop"))
2478 End Function
2479
2480 Private Function EvalIndexNode(node As Map, progScope As ScopeStack) As Variant
2481     Dim baseVal As Variant
2482     vAssignment baseVal, EvalExprNode(node.GetValue("base"), progScope)
2483     Dim idxV As Variant: vAssignment idxV, EvalExprNode(node.GetValue("index"),
        progScope)
2484     Dim pos As Long: pos = CLng(idxV)
2485     Dim OptionBase As Long: OptionBase = progScope.GetValue("__option_base")
2486     If pos < OptionBase Then err.Raise vbObjectError + 5001, "VM.EvalIndexNode",
        "Invalid index (must >= base)"
2487     If IsArray(baseVal) Then
2488         Dim lb As Long: lb = LBound(baseVal)
2489         Dim ub As Long: ub = UBound(baseVal)
2490         Dim adjustedPos As Long: adjustedPos = lb + (pos - OptionBase)
2491         If adjustedPos > ub Then err.Raise vbObjectError + 5005, "VM.EvalIndexNode",
            "Index out of bounds"
2492         vAssignment EvalIndexNode, baseVal(adjustedPos)
2493     ElseIf TypeName(baseVal) = "Map" Then
2494         vAssignment EvalIndexNode, baseVal.GetValue(pos)
2495     Else
2496         err.Raise vbObjectError + 5006, "VM.EvalIndexNode", "Cannot index
            non-array/non-map"
2497     End If
2498 End Function
2499
2500 Private Function EvalArrayNode(node As Map, progScope As ScopeStack) As Variant

```

```

2501 Dim items As Collection: Set items = node.GetValue("items")
2502 Dim arr As Variant
2503 Dim OptionBase As Long: OptionBase = progScope.GetValue("__option_base")
2504 If items Is Nothing Or items.Count = 0 Then
2505     ReDim arr(OptionBase To OptionBase - 1)
2506 Else
2507     ReDim arr(OptionBase To OptionBase + items.Count - 1)
2508     Dim ii As Long
2509     For ii = 1 To items.Count
2510         vAssignment arr(OptionBase + ii - 1), EvalExprNode(items(ii), progScope)
2511     Next ii
2512 End If
2513 vAssignment EvalArrayNode, arr
2514 End Function
2515
2516 Private Function ParsePath(ByVal path As String) As Collection
2517     Dim col As New Collection
2518     Dim pos As Long: pos = 1
2519     Dim lenP As Long: lenP = Len(path)
2520     Dim currentPart As String
2521     While pos <= lenP
2522         Dim ch As String: ch = Mid(path, pos, 1)
2523         If ch = "." Then
2524             If currentPart <> "" Then col.Add currentPart
2525             currentPart = ""
2526             pos = pos + 1
2527         ElseIf ch = "[" Then
2528             If currentPart <> "" Then col.Add currentPart
2529             currentPart = ""
2530             Dim startB As Long: startB = pos
2531             pos = pos + 1
2532             While pos <= lenP And Mid(path, pos, 1) <> "]"
2533                 pos = pos + 1
2534             Wend
2535             If pos > lenP Then err.Raise vbObjectError + 5003, "VM.ParsePath",
                "Unmatched bracket in path"
2536             currentPart = Mid(path, startB, pos - startB + 1)
2537             col.Add currentPart
2538             currentPart = ""
2539             pos = pos + 1
2540         Else
2541             currentPart = currentPart & ch
2542             pos = pos + 1
2543         End If
2544     Wend
2545     If currentPart <> "" Then col.Add currentPart
2546     Set ParsePath = col
2547 End Function
2548
2549 Private Function ResolvePath(node As Map, progScope As ScopeStack, Optional forAssign As
Boolean = False) As Map
2550     Dim res As New Map
2551     Dim tp As String: tp = node.GetValue("type")
2552     If tp = "Variable" Then
2553         If (InStr(node.GetValue("name"), ".") > 0 Or InStr(node.GetValue("name"), "[") >
0) Then
2554             ' Fallback for flat names (if Compiler didn't nest)
2555             Dim path As String: path = node.GetValue("name")
2556             Dim parts As Collection: Set parts = ParsePath(path) ' Add ParsePath from
previous patch
2557             Dim current As Variant
2558             vAssignment current, progScope.GetValue(CStr(parts(1)))
2559             Dim ii As Long
2560             For ii = 2 To parts.Count - IIf(forAssign, 1, 0)
2561                 Dim part As String: part = parts(ii)
2562                 If left(part, 1) = "[" Then
2563                     Dim indexStr As String: indexStr = Mid(part, 2, Len(part) - 2)
2564                     Dim indexV As Variant: indexV = EvalVBExpressionWithScope(indexStr,
progScope) ' Twist: use VBEXPR for fallback computed

```

```

2565         current = GetIndexedValue(current, indexV, progScope)
2566     Else
2567         If TypeName(current) = "Map" Then
2568             vAssignment current, current.GetValue(part)
2569         Else
2570             err.Raise vbObjectError + 5007, "VM.ResolvePath", "Cannot access
                property on non-map"
2571         End If
2572     End If
2573 Next ii
2574 If forAssign Then
2575     res.SetValue "container", current
2576     res.SetValue "key", parts(parts.Count)
2577     res.SetValue "kind", IIf(left(parts(parts.Count), 1) = "[", "index",
        "prop")
2578 Else
2579     res.SetValue "value", current
2580 End If
2581 Else
2582     If forAssign Then
2583         res.SetValue "container", progScope
2584         res.SetValue "key", node.GetValue("name")
2585         res.SetValue "kind", "scopeVar"
2586     Else
2587         res.SetValue "value", progScope.GetValue(node.GetValue("name"))
2588     End If
2589 End If
2590 Else
2591     ' Nested nodes (preferred)
2592     Select Case tp
2593     Case "Member"
2594         Dim baseRes As Map: Set baseRes = ResolvePath(node.GetValue("base"),
            progScope, forAssign)
2595         If forAssign Then
2596             res.SetValue "container", baseRes("container")
2597             res.SetValue "container", baseRes
2598             res.SetValue "key", node.GetValue("prop")
2599             res.SetValue "kind", "prop"
2600         Else
2601             res.SetValue "value", EvalMemberNode(node, progScope)
2602         End If
2603     Case "Index"
2604         Dim baseRes2 As Map: Set baseRes2 = ResolvePath(node.GetValue("base"),
            progScope, forAssign)
2605         If forAssign Then
2606             res.SetValue "container", baseRes2("container")
2607             res.SetValue "container", baseRes2
2608             res.SetValue "key", EvalExprNode(node.GetValue("index"), progScope)
2609             res.SetValue "kind", "index"
2610         Else
2611             res.SetValue "value", EvalIndexNode(node, progScope)
2612         End If
2613     End Select
2614 End If
2615 Set ResolvePath = res
2616 End Function
2617
2618 Private Function GetIndexedValue(baseVal As Variant, idxV As Variant, progScope As
ScopeStack) As Variant
2619     Dim pos As Long: pos = CLng(idxV)
2620     Dim OptionBase As Long: OptionBase = progScope.GetValue("__option_base")
2621     If pos < OptionBase Then err.Raise vbObjectError + 5001, "VM.GetIndexedValue",
        "Invalid index"
2622     If IsArray(baseVal) Then
2623         Dim lb As Long: lb = LBound(baseVal)
2624         Dim adjustedPos As Long: adjustedPos = lb + (pos - OptionBase)
2625         If adjustedPos > UBound(baseVal) Then err.Raise vbObjectError + 5005,
            "VM.GetIndexedValue", "Index out of bounds"
2626         vAssignment GetIndexedValue, baseVal(adjustedPos)

```

```

2627     ElseIf TypeName(baseVal) = "Map" Then
2628         vAssignment GetIndexedValue, baseVal.GetValue(pos)
2629     Else
2630         err.Raise vbObjectError + 5006, "VM.GetIndexedValue", "Cannot index
            non-array/non-map"
2631     End If
2632 End Function
2633
2634 Private Function ResolveLValue(node As Map, progScope As ScopeStack) As Map
2635     Set ResolveLValue = ResolvePath(node, progScope, True)
2636     If node.GetValue("type") = "Index" Then
2637         If node.GetValue("index").GetValue("type") <> "Literal" Then
2638             ResolveLValue.Add "computed", True ' Twist: flag for lazy
2639         End If
2640     End If
2641 End Function
2642
2643 ' -----
2644 ' ExecIfNode: Evaluate condition, then execute proper block
2645 ' -----
2646 Private Function ExecIfNode(node As Map, progScope As ScopeStack) As String
2647     Dim condNode As Map: Set condNode = node.GetValue("cond")
2648     If IsTruthy(EvalExprNode(condNode, progScope)) Then
2649         Dim thenStmts As Collection: Set thenStmts = node.GetValue("then")
2650         Dim si As Long
2651         For si = 1 To thenStmts.Count
2652             Dim ctrl As String: ctrl = ExecuteStmtNode(thenStmts(si), progScope)
2653             If ctrl <> "" Then ExecIfNode = ctrl: Exit Function
2654         Next si
2655         ExecIfNode = ""
2656     Exit Function
2657 End If
2658 Dim elseifConds As Collection: Set elseifConds = node.GetValue("elseif_conds")
2659 Dim elseifBlocks As Collection: Set elseifBlocks = node.GetValue("elseif_blocks")
2660 Dim idx As Long
2661 For idx = 1 To elseifConds.Count
2662     If IsTruthy(EvalExprNode(elseifConds(idx), progScope)) Then
2663         Dim bl As Collection: Set bl = elseifBlocks(idx)
2664         Dim si2 As Long
2665         For si2 = 1 To bl.Count
2666             Dim ctrl2 As String: ctrl2 = ExecuteStmtNode(bl(si2), progScope)
2667             If ctrl2 <> "" Then ExecIfNode = ctrl2: Exit Function
2668         Next si2
2669         ExecIfNode = ""
2670     Exit Function
2671 End If
2672 Next idx
2673 If node.GetValue("hasElse") Then
2674     Dim els As Collection: Set els = node.GetValue("else")
2675     Dim ee As Long
2676     For ee = 1 To els.Count
2677         Dim ctrl3 As String: ctrl3 = ExecuteStmtNode(els(ee), progScope)
2678         If ctrl3 <> "" Then ExecIfNode = ctrl3: Exit Function
2679     Next ee
2680 End If
2681 ExecIfNode = ""
2682 End Function
2683
2684 ' -----
2685 ' ExecForNode
2686 ' -----
2687 Private Function ExecForNode(node As Map, progScope As ScopeStack) As String
2688     Dim initNode As Map: Set initNode = node.GetValue("init")
2689     Dim condNode As Map: Set condNode = node.GetValue("cond")
2690     Dim stepNode As Map: Set stepNode = node.GetValue("step")
2691     Dim body As Collection: Set body = node.GetValue("body")
2692     Dim condOk As Boolean
2693
2694     If Not initNode Is Nothing Then

```

```

2695     Dim initType As String
2696     initType = initNode.GetValue("type")
2697     Select Case initType
2698         Case "Assign", "Print", "If", "For", "While", "TryCatch", "Switch",
2699             "Return", "Break", "Continue", "ExprStmt"
2700             ' Already a statement node – execute directly so side-effects happen.
2701             ExecuteStmtNode initNode, progScope
2702         Case Else
2703             ' Expression node – wrap into ExprStmt for side-effect evaluation.
2704             ExecuteStmtNode MakeNodeExprStmt(initNode), progScope
2705     End Select
2706 End If
2707 ForStart:
2708     condOk = True
2709     If Not condNode Is Nothing Then
2710         condOk = IsTruthy(EvalExprNode(condNode, progScope))
2711     End If
2712     If Not condOk Then GoTo ForEnd
2713
2714     Dim s As Long
2715     For s = 1 To body.Count
2716         Dim ctrl As String: ctrl = ExecuteStmtNode(body(s), progScope)
2717         If ctrl = "BREAK" Then GoTo ForEnd
2718         If ctrl = "CONTINUE" Then Exit For
2719         If ctrl = "RETURN" Or ctrl = "ERR" Then ExecForNode = ctrl: Exit Function
2720     Next s
2721
2722     If Not stepNode Is Nothing Then
2723         Dim stepType As String
2724         stepType = stepNode.GetValue("type")
2725         Select Case stepType
2726             Case "Assign", "Print", "If", "For", "While", "TryCatch", "Switch",
2727                 "Return", "Break", "Continue", "ExprStmt"
2728                 ExecuteStmtNode stepNode, progScope
2729             Case Else
2730                 ExecuteStmtNode MakeNodeExprStmt(stepNode), progScope
2731         End Select
2732     End If
2733     GoTo ForStart
2734 ForEnd:
2735     ExecForNode = ""
2736 End Function
2737
2738 ' Helper to wrap an expression node into an ExprStmt node
2739 Private Function MakeNodeExprStmt(expr As Map) As Map
2740     Dim m As Map: Set m = MakeNode("ExprStmt")
2741     m.SetValue "expr", expr
2742     Set MakeNodeExprStmt = m
2743 End Function
2744
2745 ' -----
2746 ' ExecWhileNode
2747 ' -----
2748 Private Function ExecWhileNode(node As Map, progScope As ScopeStack) As String
2749     Dim condNode As Map: Set condNode = node.GetValue("cond")
2750     Dim body As Collection: Set body = node.GetValue("body")
2751
2752 WhileStart:
2753     If Not IsTruthy(EvalExprNode(condNode, progScope)) Then GoTo WhileEnd
2754     Dim i As Long
2755     For i = 1 To body.Count
2756         Dim ctrl As String: ctrl = ExecuteStmtNode(body(i), progScope)
2757         If ctrl = "BREAK" Then GoTo WhileEnd
2758         If ctrl = "CONTINUE" Then Exit For
2759         If ctrl = "RETURN" Or ctrl = "ERR" Then ExecWhileNode = ctrl: Exit Function
2760     Next i
2761     GoTo WhileStart

```

```

2762
2763 WhileEnd:
2764     ExecWhileNode = ""
2765 End Function
2766
2767 ' -----
2768 ' ExecTryCatchNode
2769 ' -----
2770 Private Function ExecTryCatchNode(node As Map, progScope As ScopeStack) As String
2771     On Error GoTo TryErr
2772     Dim tryStmts As Collection: Set tryStmts = node.GetValue("try")
2773     Dim i As Long
2774     For i = 1 To tryStmts.Count
2775         Dim ctrl As String: ctrl = ExecuteStmtNode(tryStmts(i), progScope)
2776         If ctrl = "RETURN" Or ctrl = "ERR" Then
2777             ExecTryCatchNode = ctrl
2778             If ctrl = "ERR" Then GoTo TryErr
2779             Exit Function
2780         End If
2781     Next i
2782     ExecTryCatchNode = ""
2783     Exit Function
2784
2785 TryErr:
2786     err.Clear
2787     Dim catchStmts As Collection: Set catchStmts = node.GetValue("catch")
2788     If Not catchStmts Is Nothing Then
2789         Dim j As Long
2790         For j = 1 To catchStmts.Count
2791             Dim cctrl As String: cctrl = ExecuteStmtNode(catchStmts(j), progScope)
2792             If cctrl = "RETURN" Or cctrl = "ERR" Then ExecTryCatchNode = cctrl: Exit
2793                 Function
2794             Next j
2795         End If
2796         ExecTryCatchNode = ""
2797     End Function
2798
2799 ' -----
2800 ' ExecSwitchNode
2801 ' -----
2802 Private Function ExecSwitchNode(node As Map, progScope As ScopeStack) As String
2803     Dim switchVal As Variant: switchVal = EvalExprNode(node.GetValue("expr"), progScope)
2804     Dim cases As Collection: Set cases = node.GetValue("cases")
2805     Dim i As Long
2806     Dim match As Boolean
2807     For i = 1 To cases.Count
2808         Dim pair As Collection: Set pair = cases(i)
2809         Dim caseExpr As Map: Set caseExpr = pair(1)
2810         Dim blockStmts As Collection: Set blockStmts = pair(2)
2811         match = (EvalExprNode(caseExpr, progScope) = switchVal)
2812         If match Then
2813             Dim s As Long
2814             For s = 1 To blockStmts.Count
2815                 Dim ctrl As String: ctrl = ExecuteStmtNode(blockStmts(s), progScope)
2816                 If ctrl = "BREAK" Then ExecSwitchNode = "": Exit Function
2817                 If ctrl = "RETURN" Or ctrl = "ERR" Then ExecSwitchNode = ctrl: Exit
2818                     Function
2819             Next s
2820         End If
2821     Next i
2822     If Not match Then
2823         Dim defBlk As Collection: Set defBlk = node.GetValue("default")
2824         If Not defBlk Is Nothing Then
2825             Dim d As Long
2826             For d = 1 To defBlk.Count
2827                 Dim ctrl2 As String: ctrl2 = ExecuteStmtNode(defBlk(d), progScope)
2828                 If ctrl2 = "BREAK" Then ExecSwitchNode = "": Exit Function
2829                 If ctrl2 = "RETURN" Or ctrl2 = "ERR" Then ExecSwitchNode = ctrl2: Exit
2830                     Function

```

```

2828         Next d
2829     End If
2830 End If
2831 ExecSwitchNode = ""
2832 End Function
2833
2834 Private Sub vAssignment(ByRef var As Variant, ByRef vValue As Variant)
2835     If IsObject(vValue) Then
2836         Set var = vValue
2837     Else
2838         var = vValue
2839     End If
2840 End Sub
2841 ' -----
2842 ' Expression evaluator: Evaluate Expr AST nodes to runtime values
2843 ' -----
2844 Private Function EvalExprNode(node As Map, progScope As ScopeStack) As Variant
2845     If node Is Nothing Then EvalExprNode = Empty: Exit Function
2846     Dim tp As String: tp = node.GetValue("type")
2847     Dim items As Collection
2848     Dim tmpResult As Variant
2849     Dim pi As Long
2850     Dim funcIdx As Long
2851     Dim baseExpr As Map
2852
2853     Select Case tp
2854     Case "FuncLiteral", "AnonFunc"
2855         ' Create closure (capture env by reference for shared-write closures)
2856         Dim cparams As Collection: vAssignment cparams, node.GetValue("params")
2857         Dim cbody As Collection: vAssignment cbody, node.GetValue("body")
2858         Dim cmap As New Map
2859         Dim envCopy As ScopeStack
2860         cmap.Add "type", "Closure"
2861         cmap.SetValue "params", cparams
2862         cmap.SetValue "body", cbody
2863         ' capture current scope reference (shared-write)
2864         Set envCopy = New ScopeStack
2865         envCopy.LoadRawByRef progScope.RawByRef
2866         cmap.SetValue "env", envCopy
2867         vAssignment tmpResult, cmap
2868         GoTo exitfun
2869     Case "VBEXPR"
2870         ' Forced VBAexpression node (from @(...)). The node stores the raw VB
2871         expression string in "expr".
2872         Dim rawVB As String: rawVB = node.GetValue("expr")
2873         vAssignment tmpResult, EvalVBExpressionWithScope(rawVB, progScope)
2874         GoTo exitfun
2875     Case "Object"
2876         ' Build and return a Map containing the evaluated properties
2877         Set items = node.GetValue("items")
2878         Dim om As New Map
2879         Dim valNode As Map
2880         Dim pair As Collection
2881         Dim key As String
2882         For pi = 1 To items.Count
2883             Set pair = items(pi)
2884             key = CStr(pair(1))
2885             Set valNode = pair(2)
2886             om.SetValue key, EvalExprNode(valNode, progScope)
2887         Next pi
2888         ' Return Map object (as Variant holding the object)
2889         vAssignment tmpResult, om
2890         GoTo exitfun
2891     Case "Member"
2892         vAssignment tmpResult, EvalMemberNode(node, progScope)
2893         GoTo exitfun
2894     Case "Index"
2895         vAssignment tmpResult, EvalIndexNode(node, progScope)
2896         GoTo exitfun

```

```

2896 Case "Array"
2897     vAssignment tmpResult, EvalArrayNode(node, progScope)
2898     GoTo exitfun
2899 Case "Literal"
2900     vAssignment tmpResult, node.GetValue("value")
2901     GoTo exitfun
2902 Case "Variable"
2903     Dim vname As String: vname = CStr(node.GetValue("name"))
2904     Dim res As Map: Set res = ResolvePath(node, progScope)
2905     vAssignment tmpResult, res("value")
2906     GoTo exitfun
2907 Case "Call"
2908     ' Call can be either:
2909     ' - Call with "name" (string) and args (legacy)
2910     ' - Call with "callee" (an expression node) and args
2911     '     (new: allows variable holding closures)
2912
2913     Dim fname As String
2914     Dim hasCalleeExpr As Boolean
2915     If node.Exists("name") Then
2916         fname = CStr(node.GetValue("name"))
2917     End If
2918     If fname = "" Then
2919         ' try to derive name from callee expression when callee is a simple
2920         ' Variable node
2921         If node.Exists("callee") Then
2922             hasCalleeExpr = True
2923             Dim calleeExpr As Map: Set calleeExpr = node.GetValue("callee")
2924             If Not calleeExpr Is Nothing Then
2925                 If calleeExpr.GetValue("type") = "Variable" Then
2926                     fname = CStr(calleeExpr.GetValue("name"))
2927                 End If
2928             End If
2929         End If
2930     End If
2931     ' builtin length function
2932     If fname = ".__len__" Then
2933         Dim argn As Collection: Set argn = node.GetValue("args")
2934         Dim av As Variant: av = EvalExprNode(argn(1), progScope)
2935         If Not IsArray(av) Then
2936             vAssignment tmpResult, 0
2937         Else
2938             vAssignment tmpResult, UBound(av) - LBound(av) + 1
2939         End If
2940         GoTo exitfun
2941     End If
2942
2943     Dim argsColl As Collection: Set argsColl = node.GetValue("args")
2944     Dim evaluated As Collection
2945     Set evaluated = New Collection
2946     Dim k As Long
2947     For k = 1 To argsColl.Count
2948         evaluated.Add EvalExprNode(argsColl(k), progScope)
2949     Next k
2950
2951     If Not hasCalleeExpr Then
2952         ' named call (existing behavior + fallback to VBAexpressions)
2953         If GLOBALS_.gFuncTable.Exists(fname) Then
2954             funcIdx = CLng(GLOBALS_.gFuncTable.GetValue(fname))
2955             vAssignment tmpResult, CallFuncByIndex_AST(funcIdx, evaluated)
2956         Else
2957             ' named but not internal -> try VBAexpressions function call
2958             vAssignment tmpResult, EvalVBFunctionCall(fname, evaluated,
2959                 progScope)
2960         End If
2961     Else
2962         ' dynamic callee: evaluate callee expression -> it must yield a closure
2963         ' map or call via VB
2964         Dim calleeVal As Variant

```

```

2962 Dim hasThisVal As Boolean: hasThisVal = False
2963 Dim thisVal As Variant
2964 ' If callee is a member or index expression, compute base as `this` for
method call binding.
2965 If Not calleeExpr Is Nothing Then
2966     Dim ct As String: ct = IIf(calleeExpr.Exists("type"),
CStr(calleeExpr.GetValue("type")), "")
2967     If ct = "Member" Or ct = "Index" Then
2968         hasThisVal = True
2969         Set baseExpr = calleeExpr.GetValue("base")
2970         thisVal = EvalExprNode(baseExpr, progScope)
2971     End If
2972 End If
2973 vAssignment calleeVal, EvalExprNode(calleeExpr, progScope)
2974 If TypeName(calleeVal) = "Map" Then
2975     Dim cM As Map: Set cM = calleeVal
2976     If cM.GetValue("type") = "Closure" Then
2977         If hasThisVal Then
2978             vAssignment tmpResult, CallClosure(cM, evaluated, thisVal)
2979         Else
2980             vAssignment tmpResult, CallClosure(cM, evaluated)
2981         End If
2982     Else
2983         ' not a closure -> fail gracefully
2984         vAssignment tmpResult, Empty
2985     End If
2986 Else
2987     ' if calleeVal is string -> call via VBAexpressions
2988     If VarType(calleeVal) = vbString Then
2989         vAssignment tmpResult, EvalVBFunctionCall(CStr(calleeVal),
evaluated, progScope)
2990     Else
2991         vAssignment tmpResult, Empty
2992     End If
2993 End If
2994 End If
2995 GoTo exitfun
2996
2997 Case "Unary"
2998     Dim op As String: op = node.GetValue("op")
2999     Dim ev As Variant: ev = EvalExprNode(node.GetValue("expr"), progScope)
3000     If op = "!" Then
3001         vAssignment tmpResult, Not IsTruthy(ev)
3002     ElseIf op = "-" Then
3003         vAssignment tmpResult, -CDbl(ev)
3004     Else
3005         vAssignment tmpResult, ev
3006     End If
3007     GoTo exitfun
3008 Case "Ternary"
3009     Dim cnd As Variant: cnd = EvalExprNode(node.GetValue("cond"), progScope)
3010     If IsTruthy(cnd) Then
3011         vAssignment tmpResult, EvalExprNode(node.GetValue("trueExpr"), progScope)
3012     Else
3013         vAssignment tmpResult, EvalExprNode(node.GetValue("falseExpr"),
progScope)
3014     End If
3015     GoTo exitfun
3016 Case "Binary"
3017     Dim lop As Variant: lop = EvalExprNode(node.GetValue("left"), progScope)
3018     Dim rop As Variant
3019     Dim op2 As String: op2 = node.GetValue("op")
3020     If op2 = "&&" Then
3021         If Not IsTruthy(lop) Then vAssignment tmpResult, False: GoTo exitfun
3022         rop = EvalExprNode(node.GetValue("right"), progScope)
3023         vAssignment tmpResult, (IsTruthy(lop) And IsTruthy(rop))
3024         GoTo exitfun
3025     ElseIf op2 = "||" Then
3026         If IsTruthy(lop) Then vAssignment tmpResult, True: GoTo exitfun

```

```

3027         rop = EvalExprNode(node.GetValue("right"), progScope)
3028         vAssignment tmpResult, (IsTruthy(lop) Or IsTruthy(rop))
3029         GoTo exitfun
3030     Else
3031         rop = EvalExprNode(node.GetValue("right"), progScope)
3032         Select Case op2
3033             Case "+"
3034                 vAssignment tmpResult, SafeAdd(lop, rop)
3035             Case "-": vAssignment tmpResult, lop - rop
3036             Case "*": vAssignment tmpResult, lop * rop
3037             Case "/"
3038                 If rop = 0 Then err.Raise vbObjectError + 2001,
3039                     "VM.EvalExprNode", "Division by zero" Else vAssignment
3040                     tmpResult, lop / rop
3041             Case "%": vAssignment tmpResult, lop Mod rop
3042             Case "^": vAssignment tmpResult, lop ^ rop
3043             Case "==" , "=": vAssignment tmpResult, (lop = rop)
3044             Case "!=": vAssignment tmpResult, (lop <> rop)
3045             Case "<": vAssignment tmpResult, (lop < rop)
3046             Case ">": vAssignment tmpResult, (lop > rop)
3047             Case "<=": vAssignment tmpResult, (lop <= rop)
3048             Case ">=": vAssignment tmpResult, (lop >= rop)
3049             Case Else
3050                 vAssignment tmpResult, Empty
3051         End Select
3052         GoTo exitfun
3053     End If
3054 End Select
3055 vAssignment tmpResult, Empty
3056 exitfun:
3057 vAssignment EvalExprNode, tmpResult
3058 End Function
3059
3060 Private Function ReturnCollection(ByRef aColl As Variant) As Collection
3061     Set ReturnCollection = aColl
3062 End Function
3063 Private Function ArrayIsInit(aArray As Variant) As Boolean
3064     Dim ub As Long
3065     ArrayIsInit = True
3066     On Error GoTo err_handler
3067     ub = UBound(aArray)
3068     Exit Function
3069 err_handler:
3070     err.Clear
3071     ArrayIsInit = False
3072 End Function
3073 ' Call function program by index (AST). "args" is Collection of evaluated arg values.
3074 Private Function CallFuncByIndex_AST(funcIdx As Long, args As Collection) As Variant
3075     Dim p As Variant: p = GLOBALS_.gPrograms(funcIdx)
3076     Dim rawScope As Collection: Set rawScope = p(2)
3077     Dim callScope As New ScopeStack
3078     Dim i As Long
3079     callScope.LoadRaw rawScope
3080     callScope.Push
3081     Dim fname As String: fname = p(0)
3082     ' set params from gFuncParams
3083     If GLOBALS_.gFuncParams.Exists(fname) Then
3084         Dim pa As Variant: pa = GLOBALS_.gFuncParams.GetValue(fname)
3085         For i = LBound(pa) To UBound(pa)
3086             If i - LBound(pa) + 1 <= args.Count Then
3087                 callScope.SetValue CStr(pa(i)), args(i - LBound(pa) + 1)
3088             Else
3089                 callScope.SetValue CStr(pa(i)), Empty
3090             End If
3091         Next i
3092     End If
3093     Dim stmts As Collection: Set stmts = p(1)

```

```

3094     For i = 1 To stmts.Count
3095         Dim ctrl As String: ctrl = ExecuteStmtNode(stmts(i), callScope)
3096         If ctrl = "RETURN" Then
3097             CallFuncByIndex_AST = callScope.GetValue("__return")
3098             callScope.Pop
3099             Exit Function
3100         ElseIf ctrl = "ERR" Then
3101             err.Raise vbObjectError + 3000, "VM.CallFunc", "Error during function
              execution"
3102         End If
3103     Next i
3104     CallFuncByIndex_AST = Empty
3105     callScope.Pop
3106 End Function
3107
3108 ' -----
3109 ' Utilities adapted from your previous VM
3110 ' -----
3111 Private Function IsTruthy(v As Variant) As Boolean
3112     If IsObject(v) Then IsTruthy = Not v Is Nothing: Exit Function
3113     If IsNull(v) Then IsTruthy = False: Exit Function
3114     If IsEmpty(v) Then IsTruthy = False: Exit Function
3115     If VarType(v) = vbBoolean Then IsTruthy = CBool(v): Exit Function
3116     If IsNumeric(v) Then IsTruthy = (CDBl(v) <> 0): Exit Function
3117     If VarType(v) = vbString Then
3118         If IsBoolean(CStr(v)) Then IsTruthy = CBool(v) Else IsTruthy = (CStr(v) <> "")
3119     End If
3120     Exit Function
3121     IsTruthy = True
3122 End Function
3123
3124 Private Function IsBoolean(ByRef expression As String) As Boolean
3125     IsBoolean = (LCase(expression) = "true")
3126     If Not IsBoolean Then IsBoolean = (LCase(expression) = "false")
3127 End Function
3128
3129 Private Function SafeAdd(a As Variant, b As Variant) As Variant
3130     If IsNumeric(a) And IsNumeric(b) Then SafeAdd = a + b Else SafeAdd = CStr(a) &
      CStr(b)
3131 End Function
3132
3133 'Private Function ValueToStringForPrint(v As Variant) As String
3134 '    Dim ub As Long, s As String, i As Long
3135 '    If Not IsArray(v) Then
3136 '        If IsNull(v) Then ValueToStringForPrint = "NULL": Exit Function
3137 '        If IsEmpty(v) Then ValueToStringForPrint = "": Exit Function
3138 '        ' pretty-print Map objects as {k: v, ...}
3139 '        If TypeName(v) = "Map" Then
3140 '            Dim outMap As String: outMap = "{"
3141 '            Dim keysCol As Collection
3142 '            Set keysCol = v.keys
3143 '            Dim kk As Long
3144 '            For kk = 1 To keysCol.Count
3145 '                If kk > 1 Then outMap = outMap & ", "
3146 '                Dim kname As String: kname = CStr(keysCol(kk))
3147 '                Dim kval As Variant: kval = v.GetValue(kname)
3148 '                outMap = outMap & kname & ": " & ValueToStringForPrint(kval)
3149 '            Next kk
3150 '            outMap = outMap & "}"
3151 '            ValueToStringForPrint = outMap
3152 '            Exit Function
3153 '        End If
3154 '        ValueToStringForPrint = CStr(v): Exit Function
3155 '    End If
3156 '    ub = UBound(v)
3157 '    If ub < 0 Then ValueToStringForPrint = "[]": Exit Function
3158 '    s = "["
3159 '    For i = LBound(v) To ub
3160 '        If i > LBound(v) Then s = s & ", "

```

```

3161     '      s = s & CStr(v(i))
3162     '      Next i
3163     '      ValueToStringForPrint = s & "]"
3164 'End Function
3165 Private Function ValueToStringForPrint(v As Variant) As String
3166     Dim visited As New Collection
3167     ValueToStringForPrint = ValueToStringWithCtx(v, visited, 0)
3168 End Function
3169 ' core recursive printer with context
3170 Private Function ValueToStringWithCtx(v As Variant, visited As Collection, depth As
Long) As String
3171     On Error GoTo ErrHandler
3172
3173     ' Depth guard
3174     If depth > VTP_MAX_DEPTH Then
3175         ValueToStringWithCtx = "..."
3176         Exit Function
3177     End If
3178
3179     ' Null / Empty
3180     If IsNull(v) Then
3181         ValueToStringWithCtx = "NULL": Exit Function
3182     End If
3183     If IsEmpty(v) Then
3184         ValueToStringWithCtx = "": Exit Function
3185     End If
3186
3187     ' Scalars
3188     If VarType(v) = vbString Then
3189         ValueToStringWithCtx = CStr(v): Exit Function
3190     End If
3191     If VarType(v) = vbBoolean Then
3192         If CBool(v) Then ValueToStringWithCtx = "True" Else ValueToStringWithCtx =
"False"
3193         Exit Function
3194     End If
3195     If IsNumeric(v) Then
3196         ValueToStringWithCtx = CStr(v): Exit Function
3197     End If
3198
3199     ' Arrays (native VBA arrays)
3200     If IsArray(v) Then
3201         ValueToStringWithCtx = ArrayToString(v, visited, depth)
3202         Exit Function
3203     End If
3204
3205     ' Objects
3206     If IsObject(v) Then
3207         Dim tn As String: tn = TypeName(v)
3208         Select Case tn
3209             Case "Map"
3210                 ' cycle detection by object identity
3211                 Dim i As Long
3212                 For i = 1 To visited.Count
3213                     If visited(i) Is v Then
3214                         ValueToStringWithCtx = "[Circular]": Exit Function
3215                     End If
3216                 Next i
3217                 visited.Add v
3218                 ValueToStringWithCtx = MapToString(v, visited, depth + 1)
3219                 visited.Remove visited.Count
3220                 Exit Function
3221             Case "Collection"
3222                 ' cycle detection
3223                 Dim j As Long
3224                 For j = 1 To visited.Count
3225                     If visited(j) Is v Then
3226                         ValueToStringWithCtx = "[Circular]": Exit Function

```

```

3228             End If
3229         Next j
3230         visited.Add v
3231         ValueToStringWithCtx = CollectionToString(v, visited, depth + 1)
3232         visited.Remove visited.Count
3233         Exit Function
3234
3235     Case Else
3236         ' generic objects: try to call a ToString-like property if present, else
        fallback
3237         ValueToStringWithCtx = ObjectToString(v, visited, depth + 1)
3238         Exit Function
3239     End Select
3240 End If
3241
3242 ' Fallback
3243 ValueToStringWithCtx = CStr(v)
3244 Exit Function
3245
3246 ErrHandler:
3247 ' On any unexpected error, return a safe placeholder and continue
3248 On Error Resume Next
3249 ValueToStringWithCtx = "[error: " & err.Number & "]"
3250 err.Clear
3251 End Function
3252
3253 ' Convert Map -> string
3254 Private Function MapToString(m As Variant, visited As Collection, depth As Long) As
String
3255     On Error GoTo ErrHandler
3256     Dim keys As Collection: Set keys = m.keys
3257     Dim kcnt As Long: kcnt = keys.Count
3258
3259     If kcnt = 0 Then
3260         MapToString = "{}": Exit Function
3261     End If
3262
3263     ' For small maps and shallow depth prefer inline representation
3264     If kcnt <= VTP_MAX_ITEMS_INLINE And depth <= 2 Then
3265         Dim parts() As String
3266         ReDim parts(1 To kcnt)
3267         Dim i As Long
3268         For i = 1 To kcnt
3269             Dim key As String: key = CStr(keys(i))
3270             Dim val As Variant: val = m.GetValue(key)
3271             parts(i) = CStr(key) & ": " & ValueToStringWithCtx(val, visited, depth)
3272         Next i
3273         MapToString = "{ " & Join(parts, ", ") & " }"
3274         Exit Function
3275     End If
3276
3277     ' Multi-line pretty print
3278     Dim sb As String
3279     Dim indent As String: indent = String(depth * 2, " ")
3280     Dim innerIndent As String: innerIndent = String((depth + 1) * 2, " ")
3281     sb = "{"
3282     Dim first As Boolean: first = True
3283     Dim kk As Variant
3284     For Each kk In keys
3285         If Not first Then sb = sb & vbCrLf
3286         sb = sb & innerIndent & CStr(kk) & ": " &
ValueToStringWithCtx(m.GetValue(CStr(kk)), visited, depth + 1)
3287         first = False
3288     Next kk
3289     sb = sb & vbCrLf & indent & "}"
3290     MapToString = sb
3291     Exit Function
3292
3293 ErrHandler:

```

```

3294     MapToString = "{<error>}"
3295     err.Clear
3296 End Function
3297
3298 ' Convert Collection -> string (treat as list)
3299 Private Function CollectionToString(col As Variant, visited As Collection, depth As
Long) As String
3300     On Error GoTo ErrHandler
3301     Dim n As Long: n = col.Count
3302     If n = 0 Then CollectionToString = "[]": Exit Function
3303     If n <= VTP_MAX_ITEMS_INLINE And depth <= 2 Then
3304         Dim tmp() As String: ReDim tmp(1 To n)
3305         Dim ii As Long
3306         For ii = 1 To n
3307             tmp(ii) = ValueToStringWithCtx(col(ii), visited, depth)
3308         Next ii
3309         CollectionToString = "[ " & Join(tmp, ", ") & " ]"
3310         Exit Function
3311     End If
3312
3313     Dim sb As String: sb = "["
3314     Dim i As Long
3315     Dim indent As String: indent = String((depth + 1) * 2, " ")
3316     For i = 1 To n
3317         If i > 1 Then sb = sb & vbCrLf
3318         sb = sb & indent & ValueToStringWithCtx(col(i), visited, depth + 1)
3319     Next i
3320     sb = sb & vbCrLf & String(depth * 2, " ") & "]"
3321     CollectionToString = sb
3322     Exit Function
3323
3324 ErrHandler:
3325     CollectionToString = "[<error>]"
3326     err.Clear
3327 End Function
3328
3329 ' Convert native VBA array -> string
3330 Private Function ArrayToString(arr As Variant, visited As Collection, depth As Long) As
String
3331     On Error GoTo ErrHandler
3332     Dim lb As Long, ub As Long
3333     lb = LBound(arr): ub = UBound(arr)
3334     Dim n As Long: n = ub - lb + 1
3335     If n <= 0 Then ArrayToString = "[]": Exit Function
3336     If n <= VTP_MAX_ITEMS_INLINE And depth <= 2 Then
3337         Dim tmp() As String: ReDim tmp(1 To n)
3338         Dim i As Long
3339         For i = lb To ub
3340             tmp(i - lb + 1) = ValueToStringWithCtx(arr(i), visited, depth)
3341         Next i
3342         ArrayToString = "[ " & Join(tmp, ", ") & " ]"
3343         Exit Function
3344     End If
3345
3346     Dim sb As String: sb = "["
3347     Dim indent As String: indent = String((depth + 1) * 2, " ")
3348     Dim ii As Long
3349     For ii = lb To ub
3350         If ii > lb Then sb = sb & vbCrLf
3351         sb = sb & indent & ValueToStringWithCtx(arr(ii), visited, depth + 1)
3352     Next ii
3353     sb = sb & vbCrLf & String(depth * 2, " ") & "]"
3354     ArrayToString = sb
3355     Exit Function
3356
3357 ErrHandler:
3358     ArrayToString = "[<error>]"
3359     err.Clear
3360 End Function

```

```

3361
3362 ' Generic object to string fallback:
3363 ' - If the object is a Map-like (has Keys and GetValue), will attempt to treat it as Map.
3364 ' - Else TypeName + simple to-string
3365 Private Function ObjectToString(obj As Variant, visited As Collection, depth As Long) As
String
3366     On Error GoTo Fallback
3367     Dim tn As String: tn = TypeName(obj)
3368
3369     ' Attempt Map-like duck typing: presence of Keys and GetValue
3370     ' (use On Error to bail out if methods missing)
3371     Dim dummy As Collection
3372     Dim tryKeys As Collection
3373     On Error GoTo Fallback2
3374     Set tryKeys = obj.keys
3375     ' if successful, treat as Map
3376     Dim i As Long
3377     For i = 1 To visited.Count
3378         If visited(i) Is obj Then
3379             ObjectToString = "[Circular]": Exit Function
3380         End If
3381     Next i
3382     visited.Add obj
3383     ObjectToString = MapToString(obj, visited, depth)
3384     visited.Remove visited.Count
3385     Exit Function
3386
3387 Fallback2:
3388     ' Not a Map-like object: try default string
3389     On Error GoTo Fallback
3390     ObjectToString = "<" & tn & ">"
3391     Exit Function
3392
3393 Fallback:
3394     ObjectToString = "<object>"
3395     err.Clear
3396 End Function
3397
3398 ' Utility to escape short strings for printing if you want (optional)
3399 Private Function EscapeStringForPrint(s As String) As String
3400     ' currently returns s raw; adapt if you want quoted output
3401     EscapeStringForPrint = s
3402 End Function
3403 ' VBAexpressions integration helpers
3404 ' -----
3405 ' Evaluate a raw VBAexpressions expression string using a VBAexpressions instance,
3406 ' seeding it with the current ASF scope variables so VB expressions can reference ASF
variables.
3407 Private Function EvalVBExpressionWithScope(expr As String, progScope As ScopeStack) As
Variant
3408     On Error GoTo ErrHandler
3409     Dim exprEval As VBAexpressions
3410     Set exprEval = New VBAexpressions
3411
3412     ' Create expression in evaluator
3413     exprEval.Create expr
3414
3415     ' Inject variables from progScope (shadowing: global frames first, then locals)
3416     Dim frame As Variant
3417     Dim m As Map
3418     Dim keyCol As Collection
3419     Dim key As Variant
3420     For Each frame In progScope.Raw
3421         Set m = frame
3422         Set keyCol = m.keys
3423         For Each key In keyCol
3424             exprEval.VarValue(CStr(key)) = m.GetValue(CStr(key))
3425         Next key
3426     Next frame

```

```

3427
3428     ' Evaluate
3429     exprEval.Eval
3430     If exprEval.ErrorType = 0 Then
3431         EvalVBExpressionWithScope = exprEval.result
3432     Else
3433         ' On error, raise to caller; the try/catch at Exec layer can handle it
3434         err.Raise vbObjectError + 7001, "VM.EvalVBExpressionWithScope", "VBAexpressions
            eval error"
3435     End If
3436     Exit Function
3437 ErrHandler:
3438     ' convert to runtime log and return Empty
3439     If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "VBAexpr error: " & err.Description
3440     err.Clear
3441     EvalVBExpressionWithScope = Empty
3442 End Function
3443
3444     ' Evaluate a function call in VBAexpressions. Args is a Collection of evaluated values.
3445     ' We create a temporary variable for each argument inside the VBAexpressions environment
3446     ' to avoid needing to serialize complex values into textual literals.
3447 Private Function EvalVBFunctionCall(fname As String, args As Collection, progScope As
ScopeStack) As Variant
3448     On Error GoTo ErrHandler
3449     Dim exprEval As VBAexpressions
3450     Set exprEval = New VBAexpressions
3451
3452     ' Seed evaluator with ASF scope variables
3453     Dim frame As Variant
3454     Dim m As Map
3455     Dim keyCol As Collection
3456     Dim key As Variant
3457     For Each frame In progScope.Raw
3458         Set m = frame
3459         Set keyCol = m.keys
3460         For Each key In keyCol
3461             exprEval.VarValue(CStr(key)) = m.GetValue(CStr(key))
3462         Next key
3463     Next frame
3464
3465     ' Inject arguments as temporary variables: __asf_vbarg_1, __asf_vbarg_2, ...
3466     Dim i As Long
3467     Dim tmpNames As New Collection
3468     For i = 1 To args.Count
3469         Dim tname As String: tname = "__asf_vbarg_" & CStr(i)
3470         tmpNames.Add tname
3471         exprEval.VarValue(tname) = args(i)
3472     Next i
3473
3474     ' build call string referencing temp names
3475     Dim callStr As String: callStr = fname & "("
3476     For i = 1 To tmpNames.Count
3477         If i > 1 Then callStr = callStr & ","
3478         callStr = callStr & tmpNames(i)
3479     Next i
3480     callStr = callStr & ")"
3481
3482     ' Evaluate
3483     exprEval.Create callStr
3484     exprEval.Eval
3485     If exprEval.ErrorType = 0 Then
3486         EvalVBFunctionCall = exprEval.result
3487     Else
3488         err.Raise vbObjectError + 7002, "VM.EvalVBFunctionCall", "VBAexpressions
            function call error"
3489     End If
3490     Exit Function
3491 ErrHandler:
3492     If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "VBAexpr function-call error: " &

```

```

err.Description
3493 err.Clear
3494 EvalVBFunctionCall = Empty
3495 End Function
3496
3497 ' -----
3498 ' Closure / function-value support (runtime helpers)
3499 ' -----
3500
3501 ' Create and call closures (closureMap is a Map with keys:
3502 ' "type" = "Closure", "params" = Collection, "body" = Collection (stmts), "env" =
ScopeStack)
3503 Private Function CallClosure(closureMap As Map, evaluatedArgs As Collection, Optional
thisVal As Variant) As Variant
3504     On Error GoTo ErrHandler
3505     Dim env As ScopeStack
3506     Set env = closureMap.GetValue("env") ' shared reference (shared-write semantics)
3507
3508     ' push a new frame for this call
3509     env.Push
3510     Dim i As Long
3511     Dim pa As Variant
3512     Dim elm As Variant
3513     Dim fc As Boolean
3514
3515     vAssignment pa, closureMap.GetValue("params")
3516     If IsObject(pa) Then
3517         fc = (Not pa Is Nothing)
3518     Else
3519         fc = Not IsEmpty(pa)
3520     End If
3521     If fc Then
3522         For Each elm In pa
3523             i = i + 1
3524             If i <= evaluatedArgs.Count Then
3525                 env.SetValue CStr(elm), evaluatedArgs(i)
3526             Else
3527                 env.SetValue CStr(elm), Empty
3528             End If
3529         Next elm
3530     End If
3531
3532     ' set 'this' if provided
3533     If Not IsMissing(thisVal) Then
3534         env.SetValue "this", thisVal
3535     End If
3536
3537     ' execute body
3538     Dim stmts As Collection: Set stmts = closureMap.GetValue("body")
3539     Dim ctrl As String
3540     Dim stmtIdx As Long
3541     For stmtIdx = 1 To stmts.Count
3542         ctrl = ExecuteStmtNode(stmts(stmtIdx), env)
3543         If ctrl = "RETURN" Then
3544             vAssignment CallClosure, env.GetValue("__return")
3545             env.Pop
3546             Exit Function
3547         ElseIf ctrl = "ERR" Then
3548             err.Raise vbObjectError + 8001, "VM.CallClosure", "Error during closure
execution"
3549         End If
3550     Next stmtIdx
3551
3552     ' normal return -> Empty
3553     env.Pop
3554     CallClosure = Empty
3555     Exit Function
3556 ErrHandler:
3557     If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "CallClosure error: " & err.Description

```

```

3558     err.Clear
3559     CallClosure = Empty
3560 End Function
3561 ' ----- LValue resolution and helpers -----
3562
3563 Private Sub HandleAssignment(left As Map, ByVal rval As Variant, progScope As ScopeStack)
3564     Dim lval As Map: Set lval = ResolveLValue(left, progScope)
3565     Dim key As Variant: vAssignment key, lval("key")
3566     If lval.Exists("computed") Then vAssignment key,
3567     EvalExprNode(left.GetValue("index"), progScope) ' Lazy compute
3568     Dim container As Variant: vAssignment container, lval("container")
3569     Select Case lval("kind")
3570     Case "scopeVar"
3571         progScope.SetValue CStr(key), rval
3572     Case "prop"
3573         If TypeName(container) <> "Map" Then err.Raise vbObjectError + 5007,
3574         "VM.HandleAssignment", "Cannot set property on non-map"
3575         container.SetValue CStr(key), rval
3576     Case "index"
3577         Dim pos As Long: pos = CLng(key)
3578         Dim OptionBase As Long: OptionBase = progScope.GetValue("__option_base")
3579         If pos < OptionBase Then err.Raise vbObjectError + 5001,
3580         "VM.HandleAssignment", "Invalid index"
3581         If IsArray(container) Then
3582             Dim lb As Long: lb = LBound(container)
3583             Dim ub As Long: ub = UBound(container)
3584             Dim adjustedPos As Long: adjustedPos = lb + (pos - OptionBase)
3585             If adjustedPos > ub Then ReDim Preserve container(lb To lb + (pos -
3586             OptionBase))
3587             vAssignment container(adjustedPos), rval
3588         ElseIf TypeName(container) = "Map" Then
3589             Dim innerC As Map
3590             Dim passKey As String
3591             passKey = container.GetValue("key")
3592             Set innerC = container.GetValue("container")
3593             vAssignment innerC.GetValue(passKey)(pos), rval
3594         Else
3595             err.Raise vbObjectError + 5002, "VM.HandleAssignment", "Cannot index
3596             non-array/non-map for set"
3597         End If
3598     End Select
3599 End Sub
3600
3601 Sub AssignToArrayByName(arrVarName As String, idx As Long, newValue As Variant,
3602 progScope As ScopeStack)
3603     Dim arrVal As Variant: arrVal = progScope.GetValue(arrVarName)
3604     If IsEmpty(arrVal) Then
3605         Dim tmpA() As Variant
3606         ReDim tmpA(1 To 0)
3607         arrVal = tmpA
3608     End If
3609     Call AssignToArrayValueInPlace(arrVal, idx, newValue)
3610     progScope.SetValue arrVarName, arrVal
3611 End Sub
3612
3613 Sub AssignToArrayValueInPlace(ByRef arrVal As Variant, idx As Long, newValue As Variant)
3614     If Not IsArray(arrVal) Then
3615         Dim na() As Variant
3616         ReDim na(1 To idx)
3617         na(idx) = newValue
3618         arrVal = na
3619         Exit Sub
3620     End If
3621     Dim lb As Long: lb = LBound(arrVal)
3622     Dim ub As Long: ub = UBound(arrVal)
3623     Dim pos As Long: pos = lb + idx - 1
3624     If pos < lb Then
3625         err.Raise vbObjectError + 9120, "VM.AssignToArrayValueInPlace", "Index out of
3626         bounds (below LBound)"
3627     End If

```

```

3620     End If
3621     If pos > ub Then
3622         ReDim Preserve arrVal(lb To pos)
3623     End If
3624     arrVal(pos) = newValue
3625 End Sub
3626 '-----
3627 ' GetElementMapFromIndexNode
3628 ' Given an Index AST node, resolve the array container and return the actual
3629 ' element Map stored at the index (creating a Map element if empty).
3630 '-----
3631 Function GetElementMapFromIndexNode(indexNode As Map, progScope As ScopeStack) As Map
3632     Dim resolved As Variant
3633     resolved = ResolveLValue(indexNode, progScope)
3634     If IsEmpty(resolved) Then
3635         Set GetElementMapFromIndexNode = Nothing
3636         Exit Function
3637     End If
3638     Dim kind As String: kind = CStr(resolved(0))
3639     Select Case kind
3640     Case "arrayInScope"
3641         Dim arrName As String: arrName = CStr(resolved(1))
3642         Dim idx As Long: idx = CLng(resolved(2))
3643         Dim arrVal As Variant: arrVal = progScope.GetValue(arrName)
3644         If IsEmpty(arrVal) Then
3645             Dim tmpA() As Variant
3646             ReDim tmpA(1 To 0)
3647             arrVal = tmpA
3648         End If
3649         Dim lb As Long: lb = LBound(arrVal)
3650         Dim pos As Long: pos = lb + idx - 1
3651         If pos > UBound(arrVal) Then
3652             ' extend array so the slot exists
3653             ReDim Preserve arrVal(lb To pos)
3654         End If
3655         Dim elem As Variant: elem = arrVal(pos)
3656         If IsEmpty(elem) Then
3657             Dim nm As Map: Set nm = MakeNode("Map")
3658             arrVal(pos) = nm
3659             progScope.SetValue arrName, arrVal
3660             Set GetElementMapFromIndexNode = nm
3661             Exit Function
3662         ElseIf TypeName(elem) = "Map" Then
3663             Set GetElementMapFromIndexNode = elem
3664             Exit Function
3665         Else
3666             ' not a Map stored in the array slot
3667             err.Raise vbObjectError + 9021, "VM.GetElementMapFromIndexNode", "Array
                element is not an object"
3668         End If
3669     Case "arrayInMap"
3670         ' resolved shape: Array("arrayInMap", mapObj, index, propName)
3671         Dim mapObj As Map: Set mapObj = resolved(1)
3672         Dim theIdx As Long: theIdx = CLng(resolved(2))
3673         Dim propName As String: propName = CStr(resolved(3))
3674         Dim arrVal2 As Variant: arrVal2 = mapObj.GetValue(propName)
3675         If IsEmpty(arrVal2) Then
3676             Dim tmpB() As Variant
3677             ReDim tmpB(1 To 0)
3678             arrVal2 = tmpB
3679         End If
3680         Dim lb2 As Long: lb2 = LBound(arrVal2)
3681         Dim pos2 As Long: pos2 = lb2 + theIdx - 1
3682         If pos2 > UBound(arrVal2) Then
3683             ReDim Preserve arrVal2(lb2 To pos2)
3684         End If
3685         Dim elem2 As Variant: elem2 = arrVal2(pos2)
3686         If IsEmpty(elem2) Then
3687             Dim nm2 As Map: Set nm2 = MakeNode("Map")

```

```

3688         arrVal2(pos2) = nm2
3689         mapObj.SetValue propName, arrVal2
3690         Set GetElementMapFromIndexNode = nm2
3691         Exit Function
3692     ElseIf TypeName(elem2) = "Map" Then
3693         Set GetElementMapFromIndexNode = elem2
3694         Exit Function
3695     Else
3696         err.Raise vbObjectError + 9022, "VM.GetElementMapFromIndexNode", "Array
        element is not an object (map)"
3697     End If
3698     Case Else
3699         err.Raise vbObjectError + 9023, "VM.GetElementMapFromIndexNode", "Unsupported
        resolved kind for index -> " & kind
3700     End Select
3701 End Function
3702 ' ----- end LValue helpers -----
3703 ' -----
3704 ' End VM class
3705 ' -----
3706
3707

```