

```

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(ByName 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
82     ' from collapse)
83     Dim idxNode As Map: Set idxNode = ParseExprFromStringToNode(idxStr)
84     ' create Index node: base = curNode
85     Dim inNode As Map: Set inNode = MakeNode("Index")
86     inNode.SetValue "base", curNode
87     inNode.SetValue "index", idxNode
88     Set curNode = inNode
89     pos = closePos + 1
90 Loop
91
92     ' handle trailing dot (will be processed in next loop iteration)
93 ContinueLoop:
94     Loop
95
96     ' Special-case '.length' when last part is 'length':
97     ' If last node is Member with prop "length", convert into Call to builtin ".__len__"
98 If Not curNode Is Nothing Then
99     If curNode.GetValue("type") = "Member" Then
100        If LCase$(CStr(curNode.GetValue("prop"))) = "length" Then
101            Dim baseNode As Map: Set baseNode = curNode.GetValue("base")
102            Dim lenCall As Map: Set lenCall = MakeNode("Call")
103            ' builtin callee as Variable ".__len__" (VM will treat this as builtin)
104            Dim builtinLen As Map: Set builtinLen = MakeNode("Variable")
105            builtinLen.SetValue "name", ".__len__"
106            lenCall.SetValue "callee", builtinLen
107            Dim args As New Collection
108            args.Add baseNode
109            lenCall.SetValue "args", args
110            Set ParseCollapsedIdentToNode = lenCall
111            Exit Function
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
143 Long
144     Dim p As Parser
145     Dim toks As Collection
146     Dim progScope As ScopeStack
147     Dim stmtsAST As Collection
148     Dim stmtTokens As Collection
149     Dim i As Long
150     Dim node As Map
151
152     GLOBALS_.ASF_InitGlobals
153
154     If GLOBALS_.ProgramCache.Exists(src) Then
155         CompileProgram = GLOBALS_.ProgramCache.GetValue(src)
156         Exit Function
157     End If
158
159     Set p = New Parser
160     With p
161         .SetGlobals GLOBALS_
162         Set toks = .Tokenize(src)
163     End With
164
165     Set progScope = New ScopeStack
166     progScope.Push
167
168     Set stmtsAST = New Collection
169     i = 1
170
171 Compiler_MainLoop:
172     Do While i <= toks.Count
173         ' skip comments at top level
174         If toks(i)(0) = "COMMENT" Then
175             i = i + 1
176             GoTo Compiler_MainLoop
177         End If
178
179         ' Function definition at top-level
180         If toks(i)(0) = "IDENT" And LCase$(toks(i)(1)) = "fun" Then
181             i = i + 1
182             ' skip comments
183             Do While i <= toks.Count And toks(i)(0) = "COMMENT"
184                 i = i + 1
185             Loop
186             If i > toks.Count Then err.Raise vbObjectError + 1, , "Unexpected end after
187             fun"
188             Dim fname As String: fname = toks(i)(1)
189             i = i + 1
190             Do While i <= toks.Count And toks(i)(0) = "COMMENT"
191                 i = i + 1
192             Loop
193
194             Dim argList As New Collection
195             If i <= toks.Count And toks(i)(0) = "PAREN" And toks(i)(1) = "(" Then
196                 i = i + 1
197                 Do While i <= toks.Count And Not (toks(i)(0) = "PAREN" And toks(i)(1) =
198                     ")")
199                     If toks(i)(0) = "COMMENT" Then
200                         i = i + 1
201                     Else
202                         If toks(i)(0) = "IDENT" Then argList.Add toks(i)(1)
203                         i = i + 1
204                     End If
205                 Loop
206                 If i <= toks.Count And toks(i)(0) = "PAREN" And toks(i)(1) = ")" Then i

```

```

        = i + 1
204 End If

205
206 Do While i <= toks.Count And toks(i)(0) = "COMMENT"
207     i = i + 1
208 Loop
209 If i > toks.Count Or Not (toks(i)(0) = "PAREN" And toks(i)(1) = "{}") Then
210     err.Raise vbObjectError + 2, , "Expected function body"
211 End If
212 i = i + 1
213 Dim depth As Long: depth = 0
214 Dim bodyTokens As New Collection
215 Do While i <= toks.Count
216     If toks(i)(0) = "PAREN" Then
217         If toks(i)(1) = "{" Then
218             depth = depth + 1
219         ElseIf toks(i)(1) = "}" Then
220             If depth = 0 Then
221                 Exit Do
222             Else
223                 depth = depth - 1
224             End If
225         End If
226     End If
227     bodyTokens.Add toks(i)
228     i = i + 1
229 Loop
230
231 Dim funcIdx As Long
232 funcIdx = CompileFunctionAST(fname, argList, bodyTokens)
233 GLOBALS_.gFuncTable.Add fname, funcIdx
234
235 Dim pa() As String
236 If argList.Count > 0 Then ReDim pa(0 To argList.Count - 1)
237 Dim j As Long
238 For j = 1 To argList.Count: pa(j - 1) = argList(j): Next j
239 GLOBALS_.gFuncParams.Add fname, pa
240
241 If i <= toks.Count Then i = i + 1 ' skip the trailing '}' if present
242 GoTo Compiler_MainLoop
243 Else
244     ' collect a top-level statement tokens (nesting-aware) then parse to AST node
245     Dim braceDepth As Long: braceDepth = 0
246     Dim parenDepth As Long: parenDepth = 0
247     Dim bracketDepth As Long: bracketDepth = 0
248
249     Set stmtTokens = New Collection
250
251 Do While i <= toks.Count
252     If toks(i)(0) = "COMMENT" Then
253         i = i + 1
254         GoTo CollectNextTop
255     End If
256
257     If toks(i)(0) = "PAREN" Then
258         Select Case toks(i)(1)
259             Case "{"
260                 braceDepth = braceDepth + 1
261             Case "}"
262                 If braceDepth > 0 Then
263                     braceDepth = braceDepth - 1
264                 End If
265             Case "("
266                 parenDepth = parenDepth + 1
267             Case ")"
268                 If parenDepth > 0 Then
269                     parenDepth = parenDepth - 1
270                 End If
271             Case "["

```

```

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
288             stmtTokens.Add toks(i)
289             i = i + 1
290
291 CollectNextTop:
292     Loop
293
294     If stmtTokens.Count > 0 Then
295         Set node = ParseStatementTokensToAST(stmtTokens)
296         If Not node Is Nothing Then
297             ' add a clone to avoid accidental reference reuse
298             stmtsAST.Add node.Clone
299         End If
300     End If
301     GoTo Compiler_MainLoop
302
303     End If
304     Loop
305     Dim OptionBase As Long: OptionBase = 1
306     Dim filteredStmts As New Collection
307     For i = 1 To stmtsAST.Count
308         Dim stmt As Map: Set stmt = stmtsAST(i)
309         If stmt.GetValue("type") = "OptionBase" Then
310             OptionBase = CLng(stmt.GetValue("value"))
311         Else
312             filteredStmts.Add stmt
313         End If
314     Next i
315     Set stmtsAST = filteredStmts
316     Dim pIndex As Long
317     Dim normalizedStmts As Collection
318     ' Normalize assignment-like expression-statements into proper Assign AST nodes
319     Set normalizedStmts = NormalizeAssignsInStmts(stmtsAST)
320     ' Normalize compound assignments (+=, -=, etc.)
321     Set normalizedStmts = NormalizeCompoundAssigns(normalizedStmts)
322     pIndex = GLOBALS_.gPrograms.Count + 1
323     GLOBALS_.gPrograms.Add Array(progName, normalizedStmts, progScope.Raw, OptionBase)
324     GLOBALS_.ProgramCache.Add src, pIndex
325     CompileProgram = pIndex
326 End Function
327
328 ' Compile a function body (AST) and register it as separate program
329 Private Function CompileFunctionAST(fname As String, argList As Collection, bodyTokens
330 As Collection) As Long
331     Dim funcScope As ScopeStack
332     Dim i As Long
333     Dim bodyStmtsAST As Collection
334     Set funcScope = New ScopeStack
335     funcScope.Push
336     For i = 1 To argList.Count
337         funcScope.SetValue argList(i), Empty
338     Next i
339     Dim idx As Long
340     Set bodyStmtsAST = ParseTokensToAST(bodyTokens)
341     ' Normalize assignments inside function body as well
342     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
344 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
379                     ' if/for/while block).
380                     ' Finalize the current statement tokens so adjacent statements
381                     ' are parsed separately.
382                     If braceDepth = 0 Then
383                         ' include this '}' token into the current stmtTokens and
384                         ' finish the statement
385                         stmtTokens.Add toks(i)
386                         i = i + 1
387                         ' mark that we closed a top-level block for follow-up checks
388                         closedTopLevelBlock = True
389                         Exit Do
390                     End If
391                 Case "("
392                     parenDepth = parenDepth + 1
393                 Case ")"
394                     If parenDepth > 0 Then
395                         parenDepth = parenDepth - 1
396                     End If
397                 Case "["
398                     bracketDepth = bracketDepth + 1
399                 Case "]"
400                     If bracketDepth > 0 Then
401                         bracketDepth = bracketDepth - 1
402                     End If
403             End Select
404         End If

```

```

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
405         = 0 And bracketDepth = 0 Then
406             i = i + 1
407             Exit Do
408         End If
409
410         ' default: append current token and continue
411         stmtTokens.Add toks(i)
412         i = i + 1
413     Loop
414
415     If stmtTokens.Count > 0 Then
416         Dim tmpNode As Map
417         Set tmpNode = ParseStatementTokensToAST(stmtTokens)
418         If Not tmpNode Is Nothing Then
419             ' If we closed a top-level block and the next token is not a semicolon,
420             ' warn the user (do not silently swallow).
421             If closedTopLevelBlock Then
422                 If i <= toks.Count Then
423                     If Not (toks(i)(0) = "SEP" And toks(i)(1) = ";") Then
424                         ' If the next token is not a comment and not end-of-input,
425                         log warning.
426                         If Not (toks(i)(0) = "COMMENT") Then
427                             On Error Resume Next
428                             ' best-effort: add to runtime log if present, and print
429                             to Immediate
430                             If Not GLOBALS_.gRuntimeLog Is Nothing Then
431                             GLOBALS_.gRuntimeLog.Add "COMPILER: missing ';' after
432                             block near token index " & CStr(i)
433                             On Error GoTo 0
434                         End If
435                     Else
436                         ' consume explicit semicolon
437                         i = i + 1
438                     End If
439                 End If
440                 closedTopLevelBlock = False
441             End If
442             out.Add tmpNode.Clone
443         End If
444     End If
445     GoTo ParseMain
446 Loop
447
448     Set ParseTokensToAST = out
449 End Function
450
451     ' Parse a single statement token collection into an AST node (Map)
452 Private Function ParseStatementTokensToAST(stmtTokens As Collection) As Map
453     If stmtTokens.Count = 0 Then
454         Set ParseStatementTokensToAST = Nothing
455         Exit Function
456     End If
457
458     ' quick check for print
459     If stmtTokens.Count >= 2 Then
460         If stmtTokens(1)(0) = "IDENT" And LCase$(stmtTokens(1)(1)) = "print" Then
461             Dim k As Long: k = 2
462             Do While k <= stmtTokens.Count And Not (stmtTokens(k)(0) = "PAREN" And
463                 stmtTokens(k)(1) = "(")
464                 k = k + 1
465             Loop
466             If k <= stmtTokens.Count Then
467                 Dim opened As Long: opened = 1
468                 k = k + 1
469                 Dim innerToks As Collection
470                 Set innerToks = New Collection
471                 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$(stmtTokens(1)(1))
488     Select Case kw
489     Case "if"
490         Set ParseStatementTokensToAST = ParseIfAST(stmtTokens)
491         Exit Function
492     Case "for"
493         Set ParseStatementTokensToAST = ParseForAST(stmtTokens)
494         Exit Function
495     Case "while"
496         Set ParseStatementTokensToAST = ParseWhileAST(stmtTokens)
497         Exit Function
498     Case "try"
499         Set ParseStatementTokensToAST = ParseTryCatchAST(stmtTokens)
500         Exit Function
501     Case "switch"
502         Set ParseStatementTokensToAST = ParseSwitchAST(stmtTokens)
503         Exit Function
504     Case "break"
505         Dim b As Map: Set b = MakeNode("Break")
506         Set ParseStatementTokensToAST = b
507         Exit Function
508     Case "continue"
509         Dim c As Map: Set c = MakeNode("Continue")
510         Set ParseStatementTokensToAST = c
511         Exit Function
512     Case "return"
513         Dim rv As Map: Set rv = MakeNode("Return")
514         If stmtTokens.Count >= 2 Then
515             Dim rhs As New Collection, m As Long
516             For m = 2 To stmtTokens.Count: rhs.Add stmtTokens(m): Next m
517             Dim reexpr As Map: Set reexpr = ParseExprTokensToNode(rhs)
518             rv.SetValue "expr", reexpr
519         Else
520             rv.SetValue "expr", Nothing
521         End If
522         Set ParseStatementTokensToAST = rv
523         Exit Function
524     Case "option"
525         If stmtTokens.Count >= 3 And LCase$(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
562             depthP = 0 And depthBr = 0 Then
563             assignPos = ii
564             Exit For
565         End If
566     Next ii
567
568     If assignPos > 0 Then
569         Dim lhsT As Collection, rhsT As Collection, t As Long
570         Set lhsT = New Collection: Set rhsT = New Collection
571         For t = 1 To assignPos - 1: lhsT.Add stmtTokens(t): Next t
572         For t = assignPos + 1 To stmtTokens.Count: rhsT.Add stmtTokens(t): Next t
573         Dim lhsNode As Map
574         If lhsT.Count = 1 And lhsT(1)(0) = "IDENT" Then
575             Dim nm As String: nm = lhsT(1)(1)
576             If InStr(nm, "[") > 0 Then
577                 Dim baseName As String: baseName = left$(nm, InStr(nm, "[") - 1)
578                 Dim idxStr As String: idxStr = Mid$(nm, InStr(nm, "[") + 1, Len(nm) -
579                     InStr(nm, "]") - 1)
580                 Dim idxNode As Map: Set idxNode = ParseExprFromStringToNode(idxStr)
581                 Set lhsNode = MakeNode("Index")
582                 Dim tmpVar As Map: Set tmpVar = MakeNode("Variable")
583                 tmpVar.SetValue "name", baseName
584                 lhsNode.SetValue "base", tmpVar
585                 lhsNode.SetValue "index", idxNode
586             Else
587                 Set lhsNode = MakeNode("Variable")
588                 lhsNode.SetValue "name", nm
589             End If
590         Else
591             Set lhsNode = ParseExprTokensToNode(lhsT)
592         End If
593         Dim rhsNode As Map: Set rhsNode = ParseExprTokensToNode(rhsT)
594         Dim asn As Map: Set asn = MakeNode("Assign")
595         asn.SetValue "left", lhsNode
596         asn.SetValue "right", rhsNode
597         Set ParseStatementTokensToAST = asn
598         Exit Function
599     End If
600
601     ' fallback: expression statement
602     Dim exprNode As Map
603     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
642             parenDepth = 0 And bracketDepth = 0 Then
643             If cur.Count > 0 Then out.Add ParseExprTokensToNode(cur)
644             Set cur = New Collection
645             i = i + 1
646             GoTo ParseArgsMain
647         End If
648
649         cur.Add argTokens(i)
650         i = i + 1
651     Loop
652     If cur.Count > 0 Then out.Add ParseExprTokensToNode(cur)
653     Set ParseArgsTokensToExprNodes = out
654 End Function
655
656 Private Function ParseExprFromStringToNode(exprStr As String) As Map
657     Dim p As Parser
658     Set p = New Parser
659     Dim toks As Collection
660     Set toks = p.Tokenize(exprStr)
661     Set ParseExprFromStringToNode = ParseExprTokensToNode(toks)
662 End Function
663
664 Private Function ParseExprTokensToNode(toks As Collection) As Map
665     Dim n As Long: n = toks.Count
666     If n = 0 Then
667         Set ParseExprTokensToNode = MakeNode("Literal"): ParseExprTokensToNode.SetValue
668             "value", Empty
669         Exit Function
670     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
685 idx As Long) As Map
686     ' parse cond ? trueExpr : falseExpr (right-assoc)
687     Dim cond As Map
688     Set cond = ParseLogicalOrNode(types, vals, n, idx)
689     If idx < n Then
690         If types(idx) = "OP" And vals(idx) = "?" Then
691             idx = idx + 1
692             Dim trueExpr As Map
693             Set trueExpr = ParseTernaryNode(types, vals, n, idx) ' right-assoc: allow
694             nested ternary
695             ' expect ':'
696             If idx < n And types(idx) = "OP" And vals(idx) = ":" Then
697                 idx = idx + 1
698             Else
699                 err.Raise vbObjectError + 8001, "Compiler.ParseTernaryNode", "Expected
700                 ':' in ternary expression"
701             End If
702             Dim falseExpr As Map
703             Set falseExpr = ParseTernaryNode(types, vals, n, idx)
704             Dim tn As New Map
705             tn.Add "type", "Ternary"
706             tn.SetValue "cond", cond
707             tn.SetValue "trueExpr", trueExpr
708             tn.SetValue "falseExpr", falseExpr
709             Set ParseTernaryNode = tn
710             Exit Function
711         End If
712     End If
713     Set ParseTernaryNode = cond
714 End Function
715
716 Private Function ParseLogicalOrNode(types() As String, vals() As Variant, n As Long,
717 ByRef idx As Long) As Map
718     Dim left_ As Map
719     Dim right_ As Map
720     Dim node As Map
721     Dim op As String
722     Set left_ = ParseLogicalAndNode(types, vals, n, idx)
723     Do While idx < n
724         If Not (types(idx) = "OP" And vals(idx) = "||") Then Exit Do
725         op = vals(idx)
726         idx = idx + 1
727         Set right_ = ParseLogicalAndNode(types, vals, n, idx)
728         Set node = MakeNode("Binary")
729         node.SetValue "op", op
730         node.SetValue "left", left_.Clone
731         node.SetValue "right", right_.Clone
732         Set left_ = node
733     Loop
734     Set ParseLogicalOrNode = left_.Clone
735     Set left_ = Nothing
736     Set right_ = Nothing
737     Set node = Nothing

```

```

734 End Function
735
736 Private Function ParseLogicalAndNode(types() As String, vals() As Variant, n As Long,
737 ByRef idx As Long) As Map
738     Dim left_ As Map
739     Dim right_ As Map
740     Dim node As Map
741     Set left_ = ParseEqualityNode(types, vals, n, idx)
742     Do While idx < n
743         If Not (types(idx) = "OP" And vals(idx) = "&&") Then Exit Do
744         idx = idx + 1
745         Set right_ = ParseEqualityNode(types, vals, n, idx)
746         Set node = MakeNode("Binary")
747         node.SetValue "op", "&&"
748         node.SetValue "left", left_.Clone
749         node.SetValue "right", right_.Clone
750         Set left_ = node
751     Loop
752     Set ParseLogicalAndNode = left_.Clone
753     Set left_ = Nothing
754     Set right_ = Nothing
755     Set node = Nothing
756 End Function
757
758 Private Function ParseEqualityNode(types() As String, vals() As Variant, n As Long,
759 ByRef idx As Long) As Map
760     Dim left_ As Map
761     Dim right_ As Map
762     Dim node As Map
763     Dim op As String
764     Set left_ = ParseRelationalNode(types, vals, n, idx)
765     Do While idx < n
766         If Not (types(idx) = "OP" And (vals(idx) = "==" Or vals(idx) = "=" Or vals(idx)
767 = "!=")) Then Exit Do
768         op = vals(idx)
769         idx = idx + 1
770         Set right_ = ParseRelationalNode(types, vals, n, idx)
771         Set node = MakeNode("Binary")
772         node.SetValue "op", op
773         node.SetValue "left", left_.Clone
774         node.SetValue "right", right_.Clone
775         Set left_ = node
776     Loop
777     Set ParseEqualityNode = left_.Clone
778     Set left_ = Nothing
779     Set right_ = Nothing
780     Set node = Nothing
781 End Function
782
783 Private Function ParseRelationalNode(types() As String, vals() As Variant, n As Long,
784 ByRef idx As Long) As Map
785     Dim left_ As Map
786     Dim right_ As Map
787     Dim node As Map
788     Dim op As String
789     Set left_ = ParseAddNode(types, vals, n, idx)
790     Do While idx < n
791         If Not (types(idx) = "OP" And (vals(idx) = "<" Or vals(idx) = ">" Or vals(idx) =
792 "=<" Or vals(idx) = ">=")) Then Exit Do
793         op = vals(idx)
794         idx = idx + 1
795         Set right_ = ParseAddNode(types, vals, n, idx)
796         Set node = MakeNode("Binary")
797         node.SetValue "op", op
798         node.SetValue "left", left_.Clone
799         node.SetValue "right", right_.Clone
800         Set left_ = node
801     Loop
802     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
875 idx As Long) As Map
876     If idx < n And types(idx) = "OP" Then
877         Dim op As String: op = vals(idx)
878         If op = "!" Or op = "-" Then
879             idx = idx + 1
880             Dim v As Map: Set v = ParseUnaryNode(types, vals, n, idx)
881             Dim node As Map: Set node = MakeNode("Unary")
882             node.SetValue "op", op
883             node.SetValue "expr", v
884             Set ParseUnaryNode = node
885             Exit Function
886         End If
887     End If
888     Set ParseUnaryNode = ParsePrimaryNode(types, vals, n, idx)
889 End Function
890
891 Private Function ParsePrimaryNode(types() As String, vals() As Variant, n As Long, ByRef
892 idx As Long) As Map
893     If idx >= n Then
894         Set ParsePrimaryNode = MakeNode("Literal")
895         ParsePrimaryNode.SetValue "value", Empty
896         Exit Function
897     End If
898     Dim t As String: t = types(idx)
899     Dim v As Variant: v = vals(idx)
900
901     Select Case t
902         Case "NUMBER"
903             Dim nNode As Map: Set nNode = MakeNode("Literal")
904             nNode.SetValue "value", CDbl(v)
905             idx = idx + 1
906             Set ParsePrimaryNode = nNode
907             Exit Function
908         Case "STRING"
909             Dim sNode As Map: Set sNode = MakeNode("Literal")
910             sNode.SetValue "value", v
911             idx = idx + 1
912             Set ParsePrimaryNode = sNode
913             Exit Function
914         Case "VBEXPR"
915             ' Special token produced by the parser for @(...) or direct VBAexpressions
916             ' block.
917             ' Create a VBExpr node with the raw expression string (to be evaluated by
918             ' VBAexpressions at runtime).
919             Dim vbNode As Map: Set vbNode = MakeNode("VBExpr")
920             vbNode.SetValue "expr", CStr(v)
921             idx = idx + 1
922             Set ParsePrimaryNode = vbNode
923             Exit Function
924         Case "IDENT"
925             Dim name As String: name = v
926             Dim idxNode As Map
927             Dim idxExpr As Map
928             If LCase$(name) = "true" Then
929                 Dim bt As Map: Set bt = MakeNode("Literal")
930                 bt.SetValue "value", True
931                 idx = idx + 1

```

```

928     Set ParsePrimaryNode = bt
929     Exit Function
930 ElseIf LCase$(name) = "false" Then
931     Dim bf As Map: Set bf = MakeNode("Literal")
932     bf.SetValue "value", False
933     idx = idx + 1
934     Set ParsePrimaryNode = bf
935     Exit Function
936 End If
937
938     ' Expression-level anonymous function literal: fun (p1, p2) { ... }
939     ' This reuses the same 'fun' token used for top-level function declarations
940     but emits
941     ' a FuncLiteral node usable inside expressions.
942 If LCase$(name) = "fun" Then
943     ' advance past 'fun'
944     idx = idx + 1
945     ' expect parameter list
946     Dim params As Collection
947     Set params = New Collection
948     If idx < n And types(idx) = "PAREN" And vals(idx) = "(" Then
949         idx = idx + 1
950         Do While idx < n And Not (types(idx) = "PAREN" And vals(idx) = ")")
951             If types(idx) = "IDENT" Then
952                 params.Add CStr(vals(idx))
953                 idx = idx + 1
954                 If idx < n And types(idx) = "SEP" And vals(idx) = "," Then
955                     idx = idx + 1
956                     ElseIf types(idx) = "SEP" And vals(idx) = "," Then
957                         idx = idx + 1
958                     Else
959                         ' skip unexpected tokens until ')'
960                         idx = idx + 1
961                     End If
962                 Loop
963                 If idx < n And types(idx) = "PAREN" And vals(idx) = ")" Then idx =
964                 idx + 1
965             End If
966
967             ' parse body { ... }
968             If idx < n And types(idx) = "PAREN" And vals(idx) = "{" Then
969                 idx = idx + 1
970                 Dim depthBody As Long: depthBody = 0
971                 Dim bodyTok As Collection
972                 Set bodyTok = New Collection
973                 Do While idx < n
974                     If types(idx) = "PAREN" Then
975                         If vals(idx) = "{" Then
976                             depthBody = depthBody + 1
977                         ElseIf vals(idx) = "}" Then
978                             If depthBody = 0 Then
979                                 Exit Do
980                             Else
981                                 depthBody = depthBody - 1
982                             End If
983                         End If
984                     End If
985                     bodyTok.Add Array(types(idx), vals(idx))
986                     idx = idx + 1
987                 Loop
988                 ' consume closing }'
989                 If idx < n And types(idx) = "PAREN" And vals(idx) = "}" Then idx =
990                 idx + 1
991                 Dim bodyStmts As Collection: Set bodyStmts =
ParseTokensToAST(bodyTok)
992                 Dim fnode As Map: Set fnode = MakeNode("FuncLiteral")
993                 fnode.SetValue "params", params
994                 fnode.SetValue "body", bodyStmts
995                 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
1071         ' built the chain)
1072         Set ParsePrimaryNode = complexNode
1073         Exit Function
1074     End If
1075 End If
1076 ' create initial variable node for an IDENT and then apply postfix operators:
1077 Dim currentNode As Map: Set currentNode = MakeNode("Variable")
1078 currentNode.SetValue "name", name
1079 idx = idx + 1
1080
1081 ' Postfix loop: handle .prop, [...], func calls (...) as postfixes chaining
1082 onto currentNode
1083 Do While idx < n
1084     ' member access a.b
1085     If types(idx) = "SYM" And vals(idx) = "." Then
1086         idx = idx + 1
1087         If idx < n And types(idx) = "IDENT" Then
1088             Dim mem As Map: Set mem = MakeNode("Member")
1089             mem.SetValue "base", currentNode
1090             mem.SetValue "prop", CStr(vals(idx))
1091             Set currentNode = mem.Clone
1092             idx = idx + 1
1093             ' continue loop
1094         Else
1095             ' invalid member access; stop postfixing
1096             Exit Do
1097         End If
1098     ' index access a[expr]
1099     ElseIf types(idx) = "PAREN" And vals(idx) = "[" Then
1100         idx = idx + 1
1101         Dim idxTok As Collection
1102         Set idxTok = New Collection
1103         Dim depthIdx As Long: depthIdx = 0
1104         Do While idx < n
1105             If types(idx) = "PAREN" Then
1106                 If vals(idx) = "[" Then
1107                     depthIdx = depthIdx + 1
1108                 ElseIf vals(idx) = "]" Then
1109                     If depthIdx = 0 Then
1110                         Exit Do
1111                     Else
1112                         depthIdx = depthIdx - 1
1113                     End If
1114                 End If
1115                 idxTok.Add Array(types(idx), vals(idx))
1116                 idx = idx + 1
1117             Loop
1118             Dim idxExprNode As Map: Set idxExprNode =
1119             ParseExprTokensToNode(idxTok)
1120             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 Do
1189         End If
1190     End If

```

```

1181
1182     Loop
1183     Dim callNode As Map: Set callNode = MakeNode("Call")
1184     callNode.SetValue "callee", currentNode
1185     callNode.SetValue "args", argNodes
1186     Set currentNode = callNode.Clone
1187     Else
1188         Exit Do
1189     End If
1190     Loop
1191     Set ParsePrimaryNode = currentNode.Clone
1192     Exit Function
1193 Case "PAREN"
1194     If v = "(" Then
1195         idx = idx + 1
1196         Dim innerNode As Map: Set innerNode = ParseLogicalOrNode(types, vals, n,
1197         idx)
1198         If idx < n And types(idx) = "PAREN" And vals(idx) = ")" Then idx = idx +
1199         1
1200         Set ParsePrimaryNode = innerNode.Clone
1201         Exit Function
1202     ElseIf v = "[" Then
1203         idx = idx + 1
1204         Dim arrList As Collection
1205         Dim elemTok As Collection
1206         Set arrList = New Collection
1207         Do While idx < n And Not (types(idx) = "PAREN" And vals(idx) = "]")
1208             Set elemTok = New Collection
1209             ' Collect element tokens until a comma (element separator) or
1210             closing bracket
1211             Do While idx < n And Not ((types(idx) = "SEP" And vals(idx) = ",") Or
1212             (types(idx) = "PAREN" And vals(idx) = "]"))
1213                 elemTok.Add Array(types(idx), vals(idx))
1214                 idx = idx + 1
1215             Loop
1216             arrList.Add ParseExprTokensToNode(elemTok)
1217             If idx < n And types(idx) = "SEP" And vals(idx) = "," Then idx = idx +
1218             + 1
1219             Loop
1220             If idx < n And types(idx) = "PAREN" And vals(idx) = "]" Then idx = idx +
1221             1
1222             Dim arrNode As Map: Set arrNode = MakeNode("Array")
1223             arrNode.SetValue "items", arrList
1224             Set ParsePrimaryNode = arrNode.Clone
1225             Exit Function
1226     ElseIf v = "{" Then
1227         ' object literal: { key: value, key2: value2 }
1228         idx = idx + 1
1229         Dim objItems As Collection
1230         Dim pair As Collection
1231         Set objItems = New Collection
1232         ' key can be IDENT or STRING
1233         Dim keyTok As String
1234         Do While idx < n And Not (types(idx) = "PAREN" And vals(idx) = "}")
1235             If types(idx) = "IDENT" Or types(idx) = "STRING" Then
1236                 keyTok = CStr(vals(idx))
1237                 idx = idx + 1
1238             Else
1239                 err.Raise vbObjectError + 8002, "Compiler.ParsePrimary",
1240                 "Invalid object key"
1241             End If
1242             ' expect ':'
1243             If idx < n And types(idx) = "OP" And vals(idx) = ":" Then
1244                 idx = idx + 1
1245             Else
1246                 err.Raise vbObjectError + 8003, "Compiler.ParsePrimary",
1247                 "Expected ':' after object key"
1248             End If
1249             ' parse value expression
1250             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
1250         + 1
1251     Loop
1252     If idx < n And types(idx) = "PAREN" And vals(idx) = "}" Then idx = idx +
1253         1
1254     Dim onode As Map: Set onode = MakeNode("Object")
1255     onode.SetValue "items", objItems
1256     Set ParsePrimaryNode = onode.Clone
1257     Exit Function
1258 End If
1259 End Select
1260
1261 Dim defN As Map: Set defN = MakeNode("Literal")
1262 defN.SetValue "value", Empty
1263 Set ParsePrimaryNode = defN
1264 End Function
1265
1266 ' -----
1267 ' AST builders for control structures
1268 ' -----
1269
1270 Private Function ParseIfAST(stmtTokens As Collection) As Map
1271     Dim node As Map: Set node = MakeNode("If")
1272     Dim i As Long: i = 2
1273     Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
1274         stmtTokens(i)(1) = "(")
1275         i = i + 1
1276     Loop
1277     If i > stmtTokens.Count Then Set ParseIfAST = node: Exit Function
1278     i = i + 1
1279     Dim condTokens As Collection
1280     Set condTokens = New Collection
1281     Dim condDepth As Long: condDepth = 0
1282     Do While i <= stmtTokens.Count
1283         If stmtTokens(i)(0) = "PAREN" Then
1284             If stmtTokens(i)(1) = "(" Then
1285                 condDepth = condDepth + 1
1286             ElseIf stmtTokens(i)(1) = ")" Then
1287                 If condDepth = 0 Then
1288                     Exit Do
1289                 Else
1290                     condDepth = condDepth - 1
1291                 End If
1292             End If
1293         End If
1294         condTokens.Add stmtTokens(i)
1295         i = i + 1
1296     Loop
1297     node.SetValue "cond", ParseExprTokensToNode(condTokens)
1298
1299     Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
1300         stmtTokens(i)(1) = "{")
1301         i = i + 1
1302     Loop
1303     If i > stmtTokens.Count Then Set ParseIfAST = node.Clone: GoTo clean_
1304     Dim j As Long: j = i + 1
1305     Dim depth As Long: depth = 0
1306     Dim thenTokens As Collection
1307     Set thenTokens = New Collection
1308     Do While j <= stmtTokens.Count
1309         If stmtTokens(j)(0) = "PAREN" Then
1310             If stmtTokens(j)(1) = "{" Then

```



```

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
1445     stmtTokens(i)(1) = "(")
1446     i = i + 1
1447 Loop
1448 If i > stmtTokens.Count Then Set ParseForAST = node: Exit Function
1449 i = i + 1
1450 Set initTok = New Collection
1451 Dim hdrParen As Long, hdrBracket As Long, hdrBrace As Long
1452 hdrParen = 0: hdrBracket = 0: hdrBrace = 0
1453 Do While i <= stmtTokens.Count
1454     If stmtTokens(i)(0) = "PAREN" Then
1455         Select Case stmtTokens(i)(1)
1456             Case "("
1457                 hdrParen = hdrParen + 1
1458             Case ")"
1459                 If hdrParen > 0 Then
1460                     hdrParen = hdrParen - 1
1461                 Else
1462                     Exit Do
1463                 End If
1464             Case "["
1465                 hdrBracket = hdrBracket + 1
1466             Case "]"
1467                 If hdrBracket > 0 Then hdrBracket = hdrBracket - 1
1468             Case "{"
1469                 hdrBrace = hdrBrace + 1
1470             Case "}"
1471                 If hdrBrace > 0 Then hdrBrace = hdrBrace - 1
1472         End Select
1473     End If
1474     If hdrParen = 0 And hdrBracket = 0 And hdrBrace = 0 And stmtTokens(i)(0) = "SEP"
1475     And _
1476         (stmtTokens(i)(1) = "," Or stmtTokens(i)(1) = ";") Then
1477         i = i + 1
1478         Exit Do
1479     End If
1480     initTok.Add stmtTokens(i)
1481     i = i + 1
1482 Loop
1483 node.SetValue "init", ParseExprTokensToNode(initTok)

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"
1507     And _
1508         (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
1519     stmtTokens(i)(1) = ")")
1520         If stmtTokens(i)(0) = "PAREN" Then
1521             Select Case stmtTokens(i)(1)
1522                 Case "("
1523                     hdrParen = hdrParen + 1
1524                 Case ")"
1525                     If hdrParen > 0 Then hdrParen = hdrParen - 1
1526                 Case "["
1527                     hdrBracket = hdrBracket + 1
1528                 Case "]"
1529                     If hdrBracket > 0 Then hdrBracket = hdrBracket - 1
1530                 Case "{"
1531                     hdrBrace = hdrBrace + 1
1532                 Case "}"
1533                     If hdrBrace > 0 Then hdrBrace = hdrBrace - 1
1534             End Select
1535         End If
1536         stepTok.Add stmtTokens(i)
1537         i = i + 1
1538     Loop
1539     node.SetValue "step", ParseExprTokensToNode(stepTok)
1540
1541     Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
1542     stmtTokens(i)(1) = "{")
1543         i = i + 1
1544     Loop
1545     If i > stmtTokens.Count Then Set ParseForAST = node.Clone: GoTo clean_
1546     Dim j As Long: j = i + 1
1547     Dim depth As Long: depth = 0
1548     Set bodyToks = New Collection
1549     Do While j <= stmtTokens.Count
1550         If stmtTokens(j)(0) = "PAREN" Then
1551             If stmtTokens(j)(1) = "{" Then
1552                 depth = depth + 1
1553             ElseIf stmtTokens(j)(1) = "}" Then
1554                 If depth = 0 Then
1555                     Exit Do
1556                 Else
1557                     depth = depth - 1
1558                 End If
1559             End If
1560             bodyToks.Add stmtTokens(j)
1561             j = j + 1
1562     Loop
1563     node.SetValue "body", ParseTokensToAST(bodyToks)
1564     Set ParseForAST = node.Clone
1565     clean_:
1566     Set node = Nothing
1567     Set bodyToks = Nothing
1568     Set stepTok = Nothing
1569     Set condTok = Nothing
1570     Set initTok = Nothing
1571 End Function
1572
1573 Private Function ParseWhileAST(stmtTokens As Collection) As Map
1574     Dim node As Map: Set node = MakeNode("While")
1575     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         condTok.Add stmtTokens(i)
1596         i = i + 1
1597     Loop
1598     node.SetValue "cond", ParseExprTokensToNode(condTok)
1599
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         bodyToks.Add stmtTokens(j)
1621         j = j + 1
1622     Loop
1623     node.SetValue "body", ParseTokensToAST(bodyToks)
1624     Set ParseWhileAST = node
1625 End Function
1626
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                     catchTok.Add stmtTokens(kk)
1680                     kk = kk + 1
1681                 Loop
1682             End If
1683             Exit Do
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
1708     Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And

```

```

stmtTokens(i)(1) = "{"
    i = i + 1
Loop
If i > stmtTokens.Count Then Set ParseSwitchAST = node: Exit Function
i = i + 1
Dim cases As Collection
Set cases = New Collection
Dim defaultBlock As Collection
Dim valTok As Collection
Do While i <= stmtTokens.Count And Not (stmtTokens(i)(0) = "PAREN" And
stmtTokens(i)(1) = "}")
    If stmtTokens(i)(0) = "IDENT" And LCase$(stmtTokens(i)(1)) = "case" Then
        i = i + 1
        Set valTok = New Collection
        Dim cvParen As Long: cvParen = 0
        Dim cvBracket As Long: cvBracket = 0
        Dim cvBrace As Long: cvBrace = 0
        Do While i <= stmtTokens.Count
            If stmtTokens(i)(0) = "PAREN" Then
                Select Case stmtTokens(i)(1)
                    Case "("
                        cvParen = cvParen + 1
                    Case ")"
                        If cvParen > 0 Then cvParen = cvParen - 1
                    Case "["
                        cvBracket = cvBracket + 1
                    Case "]"
                        If cvBracket > 0 Then cvBracket = cvBracket - 1
                    Case "{"
                        If cvParen = 0 And cvBracket = 0 Then
                            Exit Do
                        Else
                            cvBrace = cvBrace + 1
                        End If
                    Case "}"
                        If cvBrace > 0 Then cvBrace = cvBrace - 1
                End Select
            End If
            valTok.Add stmtTokens(i)
            i = i + 1
        Loop
        Dim caseValNode As Map: Set caseValNode = ParseExprTokensToNode(valTok)
        If i <= stmtTokens.Count And stmtTokens(i)(0) = "PAREN" And stmtTokens(i)(1) =
= "{" Then
            i = i + 1
            Dim depth As Long: depth = 0
            Dim blockTok As Collection
            Dim pair As Collection
            Dim caseBlock As Collection
            Set blockTok = New Collection
            Do While i <= stmtTokens.Count
                If stmtTokens(i)(0) = "PAREN" Then
                    If stmtTokens(i)(1) = "{" Then
                        depth = depth + 1
                    ElseIf stmtTokens(i)(1) = "}" Then
                        If depth = 0 Then
                            Exit Do
                        Else
                            depth = depth - 1
                        End If
                    End If
                End If
                blockTok.Add stmtTokens(i)
                i = i + 1
            Loop
            Set caseBlock = ParseTokensToAST(blockTok)
            Set pair = New Collection
            pair.Add caseValNode
            pair.Add caseBlock

```

```

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                     defTok.Add stmtTokens(i)
1799                     i = i + 1
1800                 Loop
1801                 Set defaultBlock = ParseTokensToAST(defTok)
1802                 i = i + 1
1803             End If
1804         Else
1805             i = i + 1
1806         End If
1807     Loop
1808     node.SetValue "cases", cases
1809     node.SetValue "default", defaultBlock
1810     Set ParseSwitchAST = node
1811 End Function
1812
1813 ' Compiler AST normalization pass
1814 ' Converts expression statements like (Binary op "=" ...) into Assign nodes
1815 ' when the left-hand side is a valid assignment target (Variable or Index).
1816 ' Also normalizes `for` init/step and recursively visits blocks.
1817 '
1818 Private Function IsAssignableNode(n As Map) As Boolean
1819     If n Is Nothing Then
1820         IsAssignableNode = False: Exit Function
1821     End If
1822     Dim t As String: t = n.GetValue("type")
1823     ' Accept variable, indexed access (arr[i]) and member access (obj.prop)
1824     If t = "Variable" Or t = "Index" Or t = "Member" Then
1825         IsAssignableNode = True
1826     Else
1827         IsAssignableNode = False
1828     End If
1829 End Function
1830
1831 Private Function NormalizeExprNodeRecursive(expr As Map) As Map
1832     ' Recursively walk expression nodes and normalize inner expressions as needed.
1833     If expr Is Nothing Then
1834         Set NormalizeExprNodeRecursive = Nothing: Exit Function
1835     End If
1836     Dim t As String: t = expr.GetValue("type")
1837     Select Case t
1838         Case "Binary"
1839             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
1909                     items(ii)
1910             Next ii
1911             If ch Then
1912                 Dim na As New Map
1913                     na.Add "type", "Array"
1914                     na.SetValue "items", newit
1915                     Set NormalizeExprNodeRecursive = na
1916                     Exit Function
1917             End If
1918         End If
1919         Set NormalizeExprNodeRecursive = expr
1920         Exit Function
1921     Case Else
1922         Set NormalizeExprNodeRecursive = expr
1923         Exit Function
1924     End Select
1925 End Function
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 nbry 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
2041     If Not elseifBlocks Is Nothing Then
2042         For i = 1 To elseifBlocks.Count
2043             Set eb = elseifBlocks(i)
2044             Set nb = New Collection
2045             Dim j As Long
2046             For j = 1 To eb.Count
2047                 nb.Add NormalizeNodeRecursive(eb(j))
2048             Next j
2049             outElseIfBlocks.Add nb
2050         Next i
2051     End If
2052     Set elseBlk = node.GetValue("else")
2053     If Not elseBlk Is Nothing Then
2054         Set outElse = New Collection
2055         For i = 1 To elseBlk.Count
2056             outElse.Add NormalizeNodeRecursive(elseBlk(i))
2057         Next i
2058     End If
2059     Set nif = New Map
2060     nif.Add "type", "If"
2061     nif.SetValue "cond", IIf(cond Is Nothing, node.GetValue("cond"), cond)
2062     nif.SetValue "then", outThen
2063     nif.SetValue "elseif_cons", outElseIfConds
2064     nif.SetValue "elseif_blocks", outElseIfBlocks
2065     nif.SetValue "hasElse", node.GetValue("hasElse")
2066     If Not outElse Is Nothing Then nif.SetValue "else", outElse
2067     Set NormalizeNodeRecursive = nif
2068     Exit Function
2069
2070 Case "For"
2071     ' normalize init/cond/step and body
2072     Set initNode = node.GetValue("init")
2073     Set condNode = node.GetValue("cond")
2074     Set stepNode = node.GetValue("step")
2075     Set body = node.GetValue("body")
2076     If Not initNode Is Nothing Then
2077         If initNode.GetValue("type") = "Binary" And
2078             CStr(initNode.GetValue("op")) = "=" Then
2079             Set L = initNode.GetValue("left")
2080             If Not L Is Nothing And IsAssignableNode(L) Then
2081                 Set na = New Map
2082                 na.Add "type", "Assign"
2083                 na.SetValue "left", L
2084                 na.SetValue "right",
2085                 NormalizeExprNodeRecursive(initNode.GetValue("right"))
2086                 Set nInit = na
2087             Else
2088                 Set nInit = NormalizeExprNodeRecursive(initNode)
2089             End If
2090             ElseIf initNode.GetValue("type") = "ExprStmt" Then
2091                 ' exprstmt could wrap a binary expression
2092                 Set e = initNode.GetValue("expr")
2093                 If Not e Is Nothing And e.GetValue("type") = "Binary" And
2094                     CStr(e.GetValue("op")) = "=" Then
2095                     Set L2 = e.GetValue("left")
2096                     If Not L2 Is Nothing And IsAssignableNode(L2) Then
2097                         Set na2 = New Map
2098                         na2.Add "type", "Assign"
2099                         na2.SetValue "left", L2
2100                         na2.SetValue "right",
2101                         NormalizeExprNodeRecursive(e.GetValue("right"))
2102                         Set nInit = na2
2103                     Else
2104                         Set nInit = NormalizeExprNodeRecursive(e)
2105                     End If
2106                 Else
2107                     Set nInit = NormalizeExprNodeRecursive(e)
2108                 End If
2109             Else
2110                 Set nInit = NormalizeExprNodeRecursive(e)
2111             End If
2112         Else

```

```

2105     Set nInit = NormalizeExprNodeRecursive(initNode)
2106     End If
2107 End If
2108 If Not condNode Is Nothing Then Set nCond =
2109 NormalizeExprNodeRecursive(condNode)
2110 If Not stepNode Is Nothing Then
2111     If stepNode.GetValue("type") = "Binary" And
2112     CStr(stepNode.GetValue("op")) = "=" Then
2113         Set SL = stepNode.GetValue("left")
2114         If Not SL Is Nothing And IsAssignableNode(SL) Then
2115             Set na3 = New Map
2116             na3.Add "type", "Assign"
2117             na3.SetValue "left", SL
2118             na3.SetValue "right",
2119             NormalizeExprNodeRecursive(stepNode.GetValue("right"))
2120             Set nStep = na3
2121         Else
2122             Set nStep = NormalizeExprNodeRecursive(stepNode)
2123         End If
2124     Else
2125         Set nStep = NormalizeExprNodeRecursive(stepNode)
2126     End If
2127 End If
2128 Set newBody = New Collection
2129 If Not body Is Nothing Then
2130     Dim bidx As Long
2131     For bidx = 1 To body.Count
2132         newBody.Add NormalizeNodeRecursive(body(bidx))
2133     Next bidx
2134 End If
2135 Set nf = New Map
2136 nf.Add "type", "For"
2137 If Not nInit Is Nothing Then nf.SetValue "init", nInit
2138 If Not nCond Is Nothing Then nf.SetValue "cond", nCond
2139 If Not nStep Is Nothing Then nf.SetValue "step", nStep
2140 nf.SetValue "body", newBody
2141 Set NormalizeNodeRecursive = nf
2142 Exit Function
2143
2144 Case "While"
2145     Set cnd = NormalizeExprNodeRecursive(node.GetValue("cond"))
2146     Set bdy = node.GetValue("body")
2147     Set nbdy = New Collection
2148     If Not bdy Is Nothing Then
2149         Dim bi As Long
2150         For bi = 1 To bdy.Count
2151             nbdy.Add NormalizeNodeRecursive(bdy(bi))
2152         Next bi
2153     End If
2154     Set nw = New Map
2155     nw.Add "type", "While"
2156     nw.SetValue "cond", IIf(cnd Is Nothing, node.GetValue("cond"), cnd)
2157     nw.SetValue "body", nbdy
2158     Set NormalizeNodeRecursive = nw
2159     Exit Function
2160
2161 Case "TryCatch"
2162     Set tryBlk = node.GetValue("try")
2163     Set catchBlk = node.GetValue("catch")
2164     Set nt = New Collection: Set nc = New Collection
2165     Dim ti As Long
2166     If Not tryBlk Is Nothing Then
2167         For ti = 1 To tryBlk.Count: nt.Add NormalizeNodeRecursive(tryBlk(ti)):
2168         Next ti
2169     End If
2170     If Not catchBlk Is Nothing Then
2171         For ti = 1 To catchBlk.Count: nc.Add
2172             NormalizeNodeRecursive(catchBlk(ti)): Next ti
2173     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
2201         di
2202     End If
2203     Set nsw = New Map
2204     nsw.Add "type", "Switch"
2205     nsw.SetValue "expr", NormalizeExprNodeRecursive(node.GetValue("expr"))
2206     nsw.SetValue "cases", ncases
2207     If Not ndef Is Nothing Then nsw.SetValue "default", ndef
2208     Set NormalizeNodeRecursive = nsw
2209     Exit Function
2210
2211 Case Else
2212     ' Assign, Print, Return, Break, Continue, etc. often have expr fields to
2213     ' normalize.
2214     If node.GetValue("type") = "Assign" Then
2215         Set leftA = node.GetValue("left")
2216         Set rightA = NormalizeExprNodeRecursive(node.GetValue("right"))
2217         Set naNode = New Map
2218         naNode.Add "type", "Assign"
2219         naNode.SetValue "left", leftA
2220         naNode.SetValue "right", IIf(rightA Is Nothing, node.GetValue("right"),
2221                                     rightA)
2222         Set NormalizeNodeRecursive = naNode
2223         Exit Function
2224     End If
2225     Set NormalizeNodeRecursive = node
2226     Exit Function
2227 End Select
2228 End Function
2229
2230 Private Function NormalizeAssignsInStmts(stmts As Collection) As Collection
2231     Dim out As New Collection
2232     If stmts Is Nothing Then
2233         Set NormalizeAssignsInStmts = out: Exit Function
2234     End If
2235     Dim i As Long
2236     Dim n As Map
2237     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         End If
2284         ' default: pass-through (but recursively normalize inner blocks too if desired)
2285         out.Add n
2286 NormNext:
2287     Next i
2288     Set NormalizeCompoundAssigns = out
2289 End Function
2290 ' -----
2291 ' End Compiler class
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
2431 ErrorHandler:
2432     If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "VM statement error: " & err.Description
2433     err.Clear
2434     ExecuteStmtNode = "ERR"
2435
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
2449         be >=1)"
2450     End If
2451     If Not IsArray(arr) Then
2452         If IsEmpty(arr) Then
2453             arr = Array()
2454         Else
2455             err.Raise vbObjectError + 5002, "VM.AssignToArray", "Not an array"
2456         End If
2457     End If
2458     ub = -1
2459     If IsArray(arr) Then
2460         ub = UBound(arr)
2461     End If
2462     If ub < 0 Then
2463         ReDim arr(1 To pos)
2464     Else
2465         lb = LBound(arr)
2466         If pos > (ub - lb + 1) Then
2467             ReDim Preserve arr(lb To lb + (pos - OptionBase))
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,
2477     "VM.EvalMemberNode", "Cannot access property on non-map"
2478     vAssignment EvalMemberNode, baseVal.GetValue(node.GetValue("prop"))
2479 End Function
2480
2481 Private Function EvalIndexNode(node As Map, progScope As ScopeStack) As Variant
2482     Dim baseVal As Variant
2483     vAssignment baseVal, EvalExprNode(node.GetValue("base"), progScope)
2484     Dim idxV As Variant: vAssignment idxV, EvalExprNode(node.GetValue("index"),
2485     progScope)
2486     Dim pos As Long: pos = CLng(idxV)
2487     Dim OptionBase As Long: OptionBase = progScope.GetValue("__option_base")
2488     If pos < OptionBase Then err.Raise vbObjectError + 5001, "VM.EvalIndexNode",
2489     "Invalid index (must >= base)"
2490     If IsArray(baseVal) Then
2491         Dim lb As Long: lb = LBound(baseVal)
2492         Dim ub As Long: ub = UBound(baseVal)
2493         Dim adjustedPos As Long: adjustedPos = lb + (pos - OptionBase)
2494         If adjustedPos > ub Then err.Raise vbObjectError + 5005, "VM.EvalIndexNode",
2495         "Index out of bounds"
2496         vAssignment EvalIndexNode, baseVal(adjustedPos)
2497     ElseIf TypeName(baseVal) = "Map" Then
2498         vAssignment EvalIndexNode, baseVal.GetValue(pos)
2499     Else
2500         err.Raise vbObjectError + 5006, "VM.EvalIndexNode", "Cannot index
2501         non-array/non-map"
2502     End If
2503 End Function
2504
2505 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",
2536             "Unmatched bracket in path"
2537             currentPart = Mid(path, startB, pos - startB + 1)
2538             col.Add currentPart
2539             currentPart = ""
2540             pos = pos + 1
2541         Else
2542             currentPart = currentPart & ch
2543             pos = pos + 1
2544         End If
2545     Wend
2546     If currentPart <> "" Then col.Add currentPart
2547     Set ParsePath = col
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
2571             property on non-map"
2572         End If
2573     End If
2574 Next ii
2575 If forAssign Then
2576     res.SetValue "container", current
2577     res.SetValue "key", parts(parts.Count)
2578     res.SetValue "kind", IIf(left(parts(parts.Count), 1) = "[", "index",
2579             "prop")
2580 Else
2581     res.SetValue "value", current
2582 End If
2583 Else
2584     If forAssign Then
2585         res.SetValue "container", progScope
2586         res.SetValue "key", node.GetValue("name")
2587         res.SetValue "kind", "scopeVar"
2588     Else
2589         res.SetValue "value", progScope.GetValue(node.GetValue("name"))
2590     End If
2591 End If
2592 Else
2593     ' Nested nodes (preferred)
2594     Select Case tp
2595         Case "Member"
2596             Dim baseRes As Map: Set baseRes = ResolvePath(node.GetValue("base"),
2597                     progScope, forAssign)
2598             If forAssign Then
2599                 res.SetValue "container", baseRes("container")
2600                 res.SetValue "container", baseRes
2601                 res.SetValue "key", node.GetValue("prop")
2602                 res.SetValue "kind", "prop"
2603             Else
2604                 res.SetValue "value", EvalMemberNode(node, progScope)
2605             End If
2606         Case "Index"
2607             Dim baseRes2 As Map: Set baseRes2 = ResolvePath(node.GetValue("base"),
2608                     progScope, forAssign)
2609             If forAssign Then
2610                 res.SetValue "container", baseRes2("container")
2611                 res.SetValue "container", baseRes2
2612                 res.SetValue "key", EvalExprNode(node.GetValue("index"), progScope)
2613                 res.SetValue "kind", "index"
2614             Else
2615                 res.SetValue "value", EvalIndexNode(node, progScope)
2616             End If
2617         End Select
2618     End If
2619     Set ResolvePath = res
2620 End Function
2621
2622 Private Function GetIndexedValue(baseVal As Variant, idxV As Variant, progScope As
2623 ScopeStack) As Variant
2624     Dim pos As Long: pos = CLng(idxV)
2625     Dim OptionBase As Long: OptionBase = progScope.GetValue("option_base")
2626     If pos < OptionBase Then err.Raise vbObjectError + 5001, "VM.GetIndexedValue",
2627             "Invalid index"
2628     If IsArray(baseVal) Then
2629         Dim lb As Long: lb = LBound(baseVal)
2630         Dim adjustedPos As Long: adjustedPos = lb + (pos - OptionBase)
2631         If adjustedPos > UBound(baseVal) Then err.Raise vbObjectError + 5005,
2632             "VM.GetIndexedValue", "Index out of bounds"
2633         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
2631         non-array/non-map"
2632 End If
2633 End Function
2634
2635 Private Function ResolveLValue(node As Map, progScope As ScopeStack) As Map
2636     Set ResolveLValue = ResolvePath(node, progScope, True)
2637     If node.GetValue("type") = "Index" Then
2638         If node.GetValue("index").GetValue("type") <> "Literal" Then
2639             ResolveLValue.Add "computed", True ' Twist: flag for lazy
2640         End If
2641     End If
2642 End Function
2643
2644 ' -----
2645 ' ExecIfNode: Evaluate condition, then execute proper block
2646 ' -----
2647 Private Function ExecIfNode(node As Map, progScope As ScopeStack) As String
2648     Dim condNode As Map: Set condNode = node.GetValue("cond")
2649     If IsTruthy(EvalExprNode(condNode, progScope)) Then
2650         Dim thenStmts As Collection: Set thenStmts = node.GetValue("then")
2651         Dim si As Long
2652         For si = 1 To thenStmts.Count
2653             Dim ctrl As String: ctrl = ExecuteStmtNode(thenStmts(si), progScope)
2654             If ctrl <> "" Then ExecIfNode = ctrl: Exit Function
2655         Next si
2656         ExecIfNode = ""
2657         Exit Function
2658     End If
2659     Dim elseifConds As Collection: Set elseifConds = node.GetValue("elseif_conds")
2660     Dim elseifBlocks As Collection: Set elseifBlocks = node.GetValue("elseif_blocks")
2661     Dim idx As Long
2662     For idx = 1 To elseifConds.Count
2663         If IsTruthy(EvalExprNode(elseifConds(idx), progScope)) Then
2664             Dim bl As Collection: Set bl = elseifBlocks(idx)
2665             Dim si2 As Long
2666             For si2 = 1 To bl.Count
2667                 Dim ctrl2 As String: ctrl2 = ExecuteStmtNode(bl(si2), progScope)
2668                 If ctrl2 <> "" Then ExecIfNode = ctrl2: Exit Function
2669             Next si2
2670             ExecIfNode = ""
2671             Exit Function
2672         End If
2673     Next idx
2674     If node.GetValue("hasElse") Then
2675         Dim els As Collection: Set els = node.GetValue("else")
2676         Dim ee As Long
2677         For ee = 1 To els.Count
2678             Dim ctrl3 As String: ctrl3 = ExecuteStmtNode(els(ee), progScope)
2679             If ctrl3 <> "" Then ExecIfNode = ctrl3: Exit Function
2680         Next ee
2681     End If
2682     ExecIfNode = ""
2683 End Function
2684
2685 ' -----
2686 ' ExecForNode
2687 ' -----
2688 Private Function ExecForNode(node As Map, progScope As ScopeStack) As String
2689     Dim initNode As Map: Set initNode = node.GetValue("init")
2690     Dim condNode As Map: Set condNode = node.GetValue("cond")
2691     Dim stepNode As Map: Set stepNode = node.GetValue("step")
2692     Dim body As Collection: Set body = node.GetValue("body")
2693     Dim condOk As Boolean
2694
2695     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
2708 ForStart:
2709     condOk = True
2710     If Not condNode Is Nothing Then
2711         condOk = IsTruthy(EvalExprNode(condNode, progScope))
2712     End If
2713     If Not condOk Then GoTo ForEnd
2714
2715     Dim s As Long
2716     For s = 1 To body.Count
2717         Dim ctrl As String: ctrl = ExecuteStmtNode(body(s), progScope)
2718         If ctrl = "BREAK" Then GoTo ForEnd
2719         If ctrl = "CONTINUE" Then Exit For
2720         If ctrl = "RETURN" Or ctrl = "ERR" Then ExecForNode = ctrl: Exit Function
2721     Next s
2722
2723     If Not stepNode Is Nothing Then
2724         Dim stepType As String
2725         stepType = stepNode.GetValue("type")
2726         Select Case stepType
2727             Case "Assign", "Print", "If", "For", "While", "TryCatch", "Switch",
2728                 "Return", "Break", "Continue", "ExprStmt"
2729                 ExecuteStmtNode stepNode, progScope
2730             Case Else
2731                 ExecuteStmtNode MakeNodeExprStmt(stepNode), progScope
2732         End Select
2733     End If
2734     GoTo ForStart
2735
2736 ForEnd:
2737     ExecForNode = ""
2738 End Function
2739
2740 ' Helper to wrap an expression node into an ExprStmt node
2741 Private Function MakeNodeExprStmt(expr As Map) As Map
2742     Dim m As Map: Set m = MakeNode("ExprStmt")
2743     m.SetValue "expr", expr
2744     Set MakeNodeExprStmt = m
2745 End Function
2746
2747 ' -----
2748 ' ExecWhileNode
2749 ' -----
2750 Private Function ExecWhileNode(node As Map, progScope As ScopeStack) As String
2751     Dim condNode As Map: Set condNode = node.GetValue("cond")
2752     Dim body As Collection: Set body = node.GetValue("body")
2753
2754 WhileStart:
2755     If Not IsTruthy(EvalExprNode(condNode, progScope)) Then GoTo WhileEnd
2756     Dim i As Long
2757     For i = 1 To body.Count
2758         Dim ctrl As String: ctrl = ExecuteStmtNode(body(i), progScope)
2759         If ctrl = "BREAK" Then GoTo WhileEnd
2760         If ctrl = "CONTINUE" Then Exit For
2761         If ctrl = "RETURN" Or ctrl = "ERR" Then ExecWhileNode = ctrl: Exit Function
2762     Next i
2763     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         ' builtin length function
2931         If fname = ".__len__" Then
2932             Dim argn As Collection: Set argn = node.GetValue("args")
2933             Dim av As Variant: av = EvalExprNode(argn(1), progScope)
2934             If Not IsArray(av) Then
2935                 vAssignment tmpResult, 0
2936             Else
2937                 vAssignment tmpResult, UBound(av) - LBound(av) + 1
2938             End If
2939             GoTo exitfun
2940         End If
2941
2942         Dim argsColl As Collection: Set argsColl = node.GetValue("args")
2943         Dim evaluated As Collection
2944         Set evaluated = New Collection
2945         Dim k As Long
2946         For k = 1 To argsColl.Count
2947             evaluated.Add EvalExprNode(argsColl(k), progScope)
2948         Next k
2949
2950         If Not hasCalleeExpr Then
2951             ' named call (existing behavior + fallback to VBAexpressions)
2952             If GLOBALS_.gFuncTable.Exists(fname) Then
2953                 funcIdx = CLng(GLOBALS_.gFuncTable.GetValue(fname))
2954                 vAssignment tmpResult, CallFuncByIndex_AST(funcIdx, evaluated)
2955             Else
2956                 ' named but not internal -> try VBAexpressions function call
2957                 vAssignment tmpResult, EvalVBFUNCTIONCALL(fname, evaluated,
2958                 progScope)
2959             End If
2960         Else
2961             ' dynamic callee: evaluate callee expression -> it must yield a closure
2962             ' map or call via VB
2963             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
2965     ' method call binding.
2966     If Not calleeExpr Is Nothing Then
2967         Dim ct As String: ct = IIf(calleeExpr.Exists("type"),
2968             CStr(calleeExpr.GetValue("type")), "")
2969         If ct = "Member" Or ct = "Index" Then
2970             hasThisVal = True
2971             Set baseExpr = calleeExpr.GetValue("base")
2972             thisVal = EvalExprNode(baseExpr, progScope)
2973         End If
2974     End If
2975     vAssignment calleeVal, EvalExprNode(calleeExpr, progScope)
2976     If TypeName(calleeVal) = "Map" Then
2977         Dim cM As Map: Set cM = calleeVal
2978         If cM.GetValue("type") = "Closure" Then
2979             If hasThisVal Then
2980                 vAssignment tmpResult, CallClosure(cM, evaluated, thisVal)
2981             Else
2982                 vAssignment tmpResult, CallClosure(cM, evaluated)
2983             End If
2984         Else
2985             ' not a closure -> fail gracefully
2986             vAssignment tmpResult, Empty
2987         End If
2988     Else
2989         ' if calleeVal is string -> call via VBAexpressions
2990         If VarType(calleeVal) = vbString Then
2991             vAssignment tmpResult, EvalVBFunctionCall(CStr(calleeVal),
2992                 evaluated, progScope)
2993         Else
2994             vAssignment tmpResult, Empty
2995         End If
2996     End If
2997     GoTo exitfun
2998
2999     Case "Unary"
3000         Dim op As String: op = node.GetValue("op")
3001         Dim ev As Variant: ev = EvalExprNode(node.GetValue("expr"), progScope)
3002         If op = "!" Then
3003             vAssignment tmpResult, Not IsTruthy(ev)
3004         ElseIf op = "-" Then
3005             vAssignment tmpResult, -CDbl(ev)
3006         Else
3007             vAssignment tmpResult, ev
3008         End If
3009         GoTo exitfun
3010
3011     Case "Ternary"
3012         Dim cnd As Variant: cnd = EvalExprNode(node.GetValue("cond"), progScope)
3013         If IsTruthy(cnd) Then
3014             vAssignment tmpResult, EvalExprNode(node.GetValue("trueExpr"), progScope)
3015         Else
3016             vAssignment tmpResult, EvalExprNode(node.GetValue("falseExpr"),
3017                 progScope)
3018         End If
3019         GoTo exitfun
3020
3021     Case "Binary"
3022         Dim lop As Variant: lop = EvalExprNode(node.GetValue("left"), progScope)
3023         Dim rop As Variant
3024         Dim op2 As String: op2 = node.GetValue("op")
3025         If op2 = "&&" Then
3026             If Not IsTruthy(lop) Then vAssignment tmpResult, False: GoTo exitfun
3027             rop = EvalExprNode(node.GetValue("right"), progScope)
3028             vAssignment tmpResult, (IsTruthy(lop) And IsTruthy(rop))
3029             GoTo exitfun
3030         ElseIf op2 = "||" Then
3031             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
3056     vAssignment tmpResult, Empty
3057 exitfun:
3058     vAssignment EvalExprNode, tmpResult
3059 End Function
3060
3061 Private Function ReturnCollection(ByRef aColl As Variant) As Collection
3062     Set ReturnCollection = aColl
3063 End Function
3064 Private Function ArrayIsInit(aArray As Variant) As Boolean
3065     Dim ub As Long
3066     ArrayIsInit = True
3067     On Error GoTo err_handler
3068     ub = UBound(aArray)
3069     Exit Function
3070 err_handler:
3071     err.Clear
3072     ArrayIsInit = False
3073 End Function
3074 ' Call function program by index (AST). "args" is Collection of evaluated arg values.
3075 Private Function CallFuncByIndex_AST(funcIdx As Long, args As Collection) As Variant
3076     Dim p As Variant: p = GLOBALS_.gPrograms(funcIdx)
3077     Dim rawScope As Collection: Set rawScope = p(2)
3078     Dim callScope As New ScopeStack
3079     Dim i As Long
3080     callScope.LoadRaw rawScope
3081     callScope.Push
3082     Dim fname As String: fname = p(0)
3083     ' set params from gFuncParams
3084     If GLOBALS_.gFuncParams.Exists(fname) Then
3085         Dim pa As Variant: pa = GLOBALS_.gFuncParams.GetValue(fname)
3086         For i = LBound(pa) To UBound(pa)
3087             If i - LBound(pa) + 1 <= args.Count Then
3088                 callScope.SetValue CStr(pa(i)), args(i - LBound(pa) + 1)
3089             Else
3090                 callScope.SetValue CStr(pa(i)), Empty
3091             End If
3092             Next i
3093     End If
3094
3095     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
3102             execution"
3103         End If
3104     Next i
3105     CallFuncByIndex_AST = Empty
3106     callScope.Pop
3107 End Function
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         Exit Function
3120     End If
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) &
3131     CStr(b)
3132 End Function
3133
3134 'Private Function ValueToStringForPrint(v As Variant) As String
3135 '    Dim ub As Long, s As String, i As Long
3136 '    If Not IsArray(v) Then
3137 '        If IsNull(v) Then ValueToStringForPrint = "NULL": Exit Function
3138 '        If IsEmpty(v) Then ValueToStringForPrint = "": Exit Function
3139 '        ' pretty-print Map objects as {k: v, ...}
3140 '        If TypeName(v) = "Map" Then
3141 '            Dim outMap As String: outMap = "{"
3142 '            Dim keysCol As Collection
3143 '            Set keysCol = v.keys
3144 '            Dim kk As Long
3145 '            For kk = 1 To keysCol.Count
3146 '                If kk > 1 Then outMap = outMap & ", "
3147 '                Dim kname As String: kname = CStr(keysCol(kk))
3148 '                Dim kval As Variant: kval = v.GetValue(kname)
3149 '                outMap = outMap & kname & ":" & ValueToStringForPrint(kval)
3150 '            Next kk
3151 '            outMap = outMap & "}"
3152 '            ValueToStringForPrint = outMap
3153 '            Exit Function
3154 '        End If
3155 '        ValueToStringForPrint = CStr(v): Exit Function
3156 '    End If
3157 '    ub = UBound(v)
3158 '    If ub < 0 Then ValueToStringForPrint = "[]": Exit Function
3159 '    s = "["
3160 '    For i = LBound(v) To ub
3161 '        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
3171 Long) As String
3172     On Error GoTo ErrHandler
3173
3174     ' Depth guard
3175     If depth > VTP_MAX_DEPTH Then
3176         ValueToStringWithCtx = "..."
3177         Exit Function
3178     End If
3179
3180     ' Null / Empty
3181     If IsNull(v) Then
3182         ValueToStringWithCtx = "NULL": Exit Function
3183     End If
3184     If IsEmpty(v) Then
3185         ValueToStringWithCtx = "": Exit Function
3186     End If
3187
3188     ' Scalars
3189     If VarType(v) = vbString Then
3190         ValueToStringWithCtx = CStr(v): Exit Function
3191     End If
3192     If VarType(v) = vbBoolean Then
3193         If CBool(v) Then ValueToStringWithCtx = "True" Else ValueToStringWithCtx =
3194         "False"
3195         Exit Function
3196     End If
3197     If IsNumeric(v) Then
3198         ValueToStringWithCtx = CStr(v): Exit Function
3199     End If
3200
3201     ' Arrays (native VBA arrays)
3202     If IsArray(v) Then
3203         ValueToStringWithCtx = ArrayToString(v, visited, depth)
3204         Exit Function
3205     End If
3206
3207     ' Objects
3208     If IsObject(v) Then
3209         Dim tn As String: tn = TypeName(v)
3210         Select Case tn
3211             Case "Map"
3212                 ' cycle detection by object identity
3213                 Dim i As Long
3214                 For i = 1 To visited.Count
3215                     If visited(i) Is v Then
3216                         ValueToStringWithCtx = "[Circular)": Exit Function
3217                     End If
3218                 Next i
3219                 visited.Add v
3220                 ValueToStringWithCtx = MapToString(v, visited, depth + 1)
3221                 visited.Remove visited.Count
3222                 Exit Function
3223
3224             Case "Collection"
3225                 ' cycle detection
3226                 Dim j As Long
3227                 For j = 1 To visited.Count
3228                     If visited(j) Is v Then
3229                         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
3237             ' fallback
3238             ValueToStringWithCtx = ObjectToString(v, visited, depth + 1)
3239             Exit Function
3240         End Select
3241     End If
3242
3243     ' Fallback
3244     ValueToStringWithCtx = CStr(v)
3245     Exit Function
3246
3247 ErrorHandler:
3248     ' On any unexpected error, return a safe placeholder and continue
3249     On Error Resume Next
3250     ValueToStringWithCtx = "[error: " & err.Number & "]"
3251     err.Clear
3252 End Function
3253
3254     ' Convert Map -> string
3255 Private Function MapToString(m As Variant, visited As Collection, depth As Long) As
3256     String
3257     On Error GoTo ErrHandler
3258     Dim keys As Collection: Set keys = m.keys
3259     Dim kcnt As Long: kcnt = keys.Count
3260
3261     If kcnt = 0 Then
3262         MapToString = "{}": Exit Function
3263     End If
3264
3265     ' For small maps and shallow depth prefer inline representation
3266     If kcnt <= VTP_MAX_ITEMS_INLINE And depth <= 2 Then
3267         Dim parts() As String
3268         ReDim parts(1 To kcnt)
3269         Dim i As Long
3270         For i = 1 To kcnt
3271             Dim key As String: key = CStr(keys(i))
3272             Dim val As Variant: val = m.GetValue(key)
3273             parts(i) = CStr(key) & ":" & ValueToStringWithCtx(val, visited, depth)
3274         Next i
3275         MapToString = "{" & Join(parts, ", ") & "}"
3276         Exit Function
3277     End If
3278
3279     ' Multi-line pretty print
3280     Dim sb As String
3281     Dim indent As String: indent = String(depth * 2, " ")
3282     Dim innerIndent As String: innerIndent = String((depth + 1) * 2, " ")
3283     sb = "{"
3284     Dim first As Boolean: first = True
3285     Dim kk As Variant
3286     For Each kk In keys
3287         If Not first Then sb = sb & vbCrLf
3288         sb = sb & innerIndent & CStr(kk) & ":" &
3289         ValueToStringWithCtx(m.GetValue(CStr(kk)), visited, depth + 1)
3290         first = False
3291     Next kk
3292     sb = sb & vbCrLf & indent & "}"
3293     MapToString = sb
3294     Exit Function
3295
3296 ErrorHandler:

```

```

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
3300 Long) As String
3301     On Error GoTo ErrHandler
3302     Dim n As Long: n = col.Count
3303     If n = 0 Then CollectionToString = "[]": Exit Function
3304     If n <= VTP_MAX_ITEMS_INLINE And depth <= 2 Then
3305         Dim tmp() As String: ReDim tmp(1 To n)
3306         Dim ii As Long
3307         For ii = 1 To n
3308             tmp(ii) = ValueToStringWithCtx(col(ii), visited, depth)
3309         Next ii
3310         CollectionToString = "[ " & Join(tmp, ", ") & "]"
3311         Exit Function
3312     End If
3313
3314     Dim sb As String: sb = "["
3315     Dim i As Long
3316     Dim indent As String: indent = String((depth + 1) * 2, " ")
3317     For i = 1 To n
3318         If i > 1 Then sb = sb & vbCrLf
3319         sb = sb & indent & ValueToStringWithCtx(col(i), visited, depth + 1)
3320     Next i
3321     sb = sb & vbCrLf & String(depth * 2, " ") & "]"
3322     CollectionToString = sb
3323     Exit Function
3324
3325 ErrHandler:
3326     CollectionToString = "[<error>]"
3327     err.Clear
3328 End Function
3329
3330 ' Convert native VBA array -> string
3331 Private Function ArrayToString(arr As Variant, visited As Collection, depth As Long) As
3332 String
3333     On Error GoTo ErrHandler
3334     Dim lb As Long, ub As Long
3335     lb = LBound(arr): ub = UBound(arr)
3336     Dim n As Long: n = ub - lb + 1
3337     If n <= 0 Then ArrayToString = "[]": Exit Function
3338     If n <= VTP_MAX_ITEMS_INLINE And depth <= 2 Then
3339         Dim tmp() As String: ReDim tmp(1 To n)
3340         Dim i As Long
3341         For i = lb To ub
3342             tmp(i - lb + 1) = ValueToStringWithCtx(arr(i), visited, depth)
3343         Next i
3344         ArrayToString = "[ " & Join(tmp, ", ") & "]"
3345         Exit Function
3346     End If
3347
3348     Dim sb As String: sb = "["
3349     Dim indent As String: indent = String((depth + 1) * 2, " ")
3350     Dim ii As Long
3351     For ii = lb To ub
3352         If ii > lb Then sb = sb & vbCrLf
3353         sb = sb & indent & ValueToStringWithCtx(arr(ii), visited, depth + 1)
3354     Next ii
3355     sb = sb & vbCrLf & String(depth * 2, " ") & "]"
3356     ArrayToString = sb
3357     Exit Function
3358
3359 ErrHandler:
3360     ArrayToString = "[<error>]"
3361     err.Clear
3362 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
3366 String
3367     On Error GoTo Fallback
3368     Dim tn As String: tn = TypeName(obj)
3369
3370     ' Attempt Map-like duck typing: presence of Keys and GetValue
3371     ' (use On Error to bail out if methods missing)
3372     Dim dummy As Collection
3373     Dim tryKeys As Collection
3374     On Error GoTo Fallback2
3375     Set tryKeys = obj.keys
3376     ' if successful, treat as Map
3377     Dim i As Long
3378     For i = 1 To visited.Count
3379         If visited(i) Is obj Then
3380             ObjectToString = "[Circular]": Exit Function
3381         End If
3382     Next i
3383     visited.Add obj
3384     ObjectToString = MapToString(obj, visited, depth)
3385     visited.Remove visited.Count
3386     Exit Function
3387
3388 Fallback2:
3389     ' Not a Map-like object: try default string
3390     On Error GoTo Fallback
3391     ObjectToString = "<" & tn & ">"
3392     Exit Function
3393
3394 Fallback:
3395     ObjectToString = "<object>"
3396     err.Clear
3397 End Function
3398
3399 ' Utility to escape short strings for printing if you want (optional)
3400 Private Function EscapeStringForPrint(s As String) As String
3401     ' currently returns s raw; adapt if you want quoted output
3402     EscapeStringForPrint = s
3403 End Function
3404
3405 ' VBAexpressions integration helpers
3406 ' -----
3407 ' Evaluate a raw VBAexpressions expression string using a VBAexpressions instance,
3408 ' seeding it with the current ASF scope variables so VB expressions can reference ASF
3409 variables.
3410 Private Function EvalVBExpressionWithScope(expr As String, progScope As ScopeStack) As
3411 Variant
3412     On Error GoTo ErrHandler
3413     Dim exprEval As VBAexpressions
3414     Set exprEval = New VBAexpressions
3415
3416     ' Create expression in evaluator
3417     exprEval.Create expr
3418
3419     ' Inject variables from progScope (shadowing: global frames first, then locals)
3420     Dim frame As Variant
3421     Dim m As Map
3422     Dim keyCol As Collection
3423     Dim key As Variant
3424     For Each frame In progScope.Raw
3425         Set m = frame
3426         Set keyCol = m.keys
3427         For Each key In keyCol
3428             exprEval.VarValue(CStr(key)) = m.GetValue(CStr(key))
3429         Next key
3430     Next frame

```

```

3427
3428     ' Evaluate
3429     exprEval.Eval
3430     If exprEval.ErrorType = 0 Then
3431         EvalVBEvaluationWithScope = 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.EvalVBEvaluationWithScope", "VBAexpressions
3435             eval error"
3436     End If
3437     Exit Function
3438 ErrHandler:
3439     ' convert to runtime log and return Empty
3440     If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "VBAexpr error: " & err.Description
3441     err.Clear
3442     EvalVBEvaluationWithScope = Empty
3443 End Function
3444
3445     ' Evaluate a function call in VBAexpressions. Args is a Collection of evaluated values.
3446     ' We create a temporary variable for each argument inside the VBAexpressions environment
3447     ' to avoid needing to serialize complex values into textual literals.
3448 Private Function EvalVBFunctionCall(fname As String, args As Collection, progScope As
3449 ScopeStack) As Variant
3450     On Error GoTo ErrHandler
3451     Dim exprEval As VBAexpressions
3452     Set exprEval = New VBAexpressions
3453
3454     ' Seed evaluator with ASF scope variables
3455     Dim frame As Variant
3456     Dim m As Map
3457     Dim keyCol As Collection
3458     Dim key As Variant
3459     For Each frame In progScope.Raw
3460         Set m = frame
3461         Set keyCol = m.keys
3462         For Each key In keyCol
3463             exprEval.VarValue(CStr(key)) = m.GetValue(CStr(key))
3464         Next key
3465     Next frame
3466
3467     ' Inject arguments as temporary variables: __asf_vbarg_1, __asf_vbarg_2, ...
3468     Dim i As Long
3469     Dim tmpNames As New Collection
3470     For i = 1 To args.Count
3471         Dim tname As String: tname = "__asf_vbarg_" & CStr(i)
3472         tmpNames.Add tname
3473         exprEval.VarValue(tname) = args(i)
3474     Next i
3475
3476     ' build call string referencing temp names
3477     Dim callStr As String: callStr = fname & "("
3478     For i = 1 To tmpNames.Count
3479         If i > 1 Then callStr = callStr & ","
3480         callStr = callStr & tmpNames(i)
3481     Next i
3482     callStr = callStr & ")"
3483
3484     ' Evaluate
3485     exprEval.Create callStr
3486     exprEval.Eval
3487     If exprEval.ErrorType = 0 Then
3488         EvalVBFunctionCall = exprEval.result
3489     Else
3490         err.Raise vbObjectError + 7002, "VM.EvalVBFunctionCall", "VBAexpressions
3491             function call error"
3492     End If
3493     Exit Function
3494 ErrHandler:
3495     If VERBOSE_ Then GLOBALS_.gRuntimeLog.Add "VBAexpr function-call error: " &

```

```

        err.Description
        err.Clear
        EvalVBFunctionCall = Empty
    End Function

    ' -----
    ' Closure / function-value support (runtime helpers)
    ' -----

    ' Create and call closures (closureMap is a Map with keys:
    ' "type" = "Closure", "params" = Collection, "body" = Collection (stmts), "env" =
    ScopeStack)
    Private Function CallClosure(closureMap As Map, evaluatedArgs As Collection, Optional
    thisVal As Variant) As Variant
        On Error GoTo ErrHandler
        Dim env As ScopeStack
        Set env = closureMap.GetValue("env") ' shared reference (shared-write semantics)

        ' push a new frame for this call
        env.Push
        Dim i As Long
        Dim pa As Variant
        Dim elm As Variant
        Dim fc As Boolean

        vAssignment pa, closureMap.GetValue("params")
        If IsObject(pa) Then
            fc = (Not pa Is Nothing)
        Else
            fc = Not IsEmpty(pa)
        End If
        If fc Then
            For Each elm In pa
                i = i + 1
                If i <= evaluatedArgs.Count Then
                    env.SetValue CStr(elm), evaluatedArgs(i)
                Else
                    env.SetValue CStr(elm), Empty
                End If
            Next elm
        End If

        ' set 'this' if provided
        If Not IsMissing(thisVal) Then
            env.SetValue "this", thisVal
        End If

        ' execute body
        Dim stmts As Collection: Set stmts = closureMap.GetValue("body")
        Dim ctrl As String
        Dim stmtIdx As Long
        For stmtIdx = 1 To stmts.Count
            ctrl = ExecuteStmtNode(stmts(stmtIdx), env)
            If ctrl = "RETURN" Then
                vAssignment CallClosure, env.GetValue("__return")
                env.Pop
                Exit Function
            ElseIf ctrl = "ERR" Then
                err.Raise vbObjectError + 8001, "VM.CallClosure", "Error during closure
                execution"
            End If
        Next stmtIdx

        ' normal return -> Empty
        env.Pop
        CallClosure = Empty
        Exit Function
    ErrorHandler:
        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)"

```

```

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
3668             element is not an object"
3669         End If
3670     Case "arrayInMap"
3671         ' resolved shape: Array("arrayInMap", mapObj, index, propName)
3672         Dim mapObj As Map: Set mapObj = resolved(1)
3673         Dim theIdx As Long: theIdx = CLng(resolved(2))
3674         Dim propName As String: propName = CStr(resolved(3))
3675         Dim arrVal2 As Variant: arrVal2 = mapObj.GetValue(propName)
3676         If IsEmpty(arrVal2) Then
3677             Dim tmpB() As Variant
3678             ReDim tmpB(1 To 0)
3679             arrVal2 = tmpB
3680         End If
3681         Dim lb2 As Long: lb2 = LBound(arrVal2)
3682         Dim pos2 As Long: pos2 = lb2 + theIdx - 1
3683         If pos2 > UBound(arrVal2) Then
3684             ReDim Preserve arrVal2(lb2 To pos2)
3685         End If
3686         Dim elem2 As Variant: elem2 = arrVal2(pos2)
3687         If IsEmpty(elem2) Then
3688             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
3697     element is not an object (map)"
3698   End If
3699 Case Else
3700   err.Raise vbObjectError + 9023, "VM.GetElementMapFromIndexNode", "Unsupported
3701   resolved kind for index -> " & kind
3702 End Select
3703 End Function
3704 ' ----- end LValue helpers -----
3705 ' -----
3706 ' End VM class
3707 ' -----
```