The ActionScript 3 Language Specification
Syntax

Avik Chaudhuri, Jeff Dyer, Lars Hansen, and Basil Hosmer

Adobe Systems Inc.
`{achaudhu,jodyer,lhansen,bhosmer}@adobe.com`

May 12, 2011

Adobe ActionScript Language 3 Specification Version 1.0

This user guide is protected under copyright law, furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

This guide contains links to third-party websites that are not under the control of Adobe Systems Incorporated, and Adobe Systems Incorporated is not responsible for the content on any linked site. If you access a third-party website mentioned in this guide, then you do so at your own risk. Adobe Systems Incorporated provides these links only as a convenience, and the inclusion of the link does not imply that Adobe Systems Incorporated endorses or accepts any responsibility for the content on those third-party sites. No right, license, or interest is granted in any third party technology referenced in this guide.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA.

Notice to U.S. Government End Users: The Software and Documentation are “Commercial Items,” as that term is defined at 48 C.F.R. §2.101, consisting of “Commercial Computer Software” and “Commercial Computer Software Documentation,” as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial Items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250, and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Contents

1 Interpretation

1.1 Phases

1(1) The syntax of the language defines the interpretation of a sequence of characters (the text of a program) as a syntax tree that represents a syntactically valid program in the language. This interpretation proceeds in the following steps.

1. The mutually recursive processes of *scanning* (Section 3), *parsing* (Section 4), *resolution of syntactic ambiguities* (Section 5), and *expansion of include directives* (Section 6) translate a sequence of characters (the text of a program) to an intermediate syntax tree.
2. Next, the process of *program configuration* (Section 7) prunes the intermediate syntax tree.
3. Finally, the process of *enforcement of syntactic restrictions* (Section 8) discards the pruned intermediate syntax tree unless it satisfies various conditions of syntactic validity.

1.2 Grammars

1(2) A grammar is specified by a set of *rules*. A rule defines a *nonterminal* by a set of *productions*. A production is a sequence of *terminals* and nonterminals, possibly with some side conditions.

1(3) A grammar identifies the sequences of terminals that *match* a nonterminal. A sequence of terminals A matches a nonterminal B if there is a production in the rule for the nonterminal B in the grammar that, upon substituting every nonterminal in that production with some sequence of terminals that matches it, becomes the sequence of terminals A. Furthermore, side conditions may appear in various positions in a production, and the conditions must be satisfied at those positions. In particular, side conditions may disambiguate ambiguous matches or restrict possible matches, based on context.

1(4) A *syntax tree* is an ordered tree that represents how a sequence of terminals match a nonterminal. The terminals are the leaves of the tree, and the nonterminal is the root. Furthermore, intermediate nonterminals are the internal nodes of the tree. Any subtree is a syntax tree that represents how the subsequence of terminals that are leaves of that subtree match the intermediate nonterminal that is the root of that subtree.

The children of any parent are the nonterminals and terminals that appear in some production of the parent, and they are ordered by the order in which they appear in that production from left to right.

1(5) A syntax tree A is *nested* by another syntax tree B if A is a (proper) subtree of B. A is nested by B *without crossing* a syntax tree C if A is nested by B but either A is not nested by C or C is not nested by B. In particular, if A is nested by B, then it follows that A is not B, A is nested by B without crossing A, and A is nested by B without crossing B.

1(6) An *ordered traversal* of a syntax tree is a traversal of the nodes of the tree in which a parent is visited before its children, and the children of a parent are visited in order. A node A appears *earlier* or *later* than another node B in the syntax tree if a ordered traversal visits A before or after B, respectively. By extension, a subtree A appears earlier or later than another subtree B if the root of A appears earlier or later than the root of B, respectively.

1(7) A *valid prefix* is a prefix of a sequence of terminals that matches a nonterminal. The valid prefix is *invalidated* by a terminal if appending the terminal to the sequence does not yield a valid prefix for that nonterminal.

1.3 Language Syntax

1(8) The syntax of the language is specified by the *syntactic grammar*, which in turn relies on the *lexical grammar*. The nonterminals and terminals of the syntactic grammar are *syntactic nonterminals* and *syntactic terminals*, respectively. The nonterminals and terminals of the lexical grammar are *lexical nonterminals* and *lexical terminals*, respectively.

1(9) A lexical terminal is a sequence of Unicode code units (characters, or 16-bit unsigned integers). A sequence of lexical terminals that matches the lexical nonterminal *InputElementOperator*, *InputElementOperand*, *InputElementXMLTag*, or *InputElementXMLContent* (following the rules of Section 1.4) is an *input element*. An input element that is a syntactic terminal is a *token*. Any other input element is a *token separator*.

1(10) *Scanning* is the process of matching some text (a sequence of lexical terminals) to a sequence of tokens, some of which may be separated by token separators. The tokens must be maximal, in the following sense: if both A, B, and $A\,B$ are tokens, then the text $A\,B$ is scanned as the token $A\,B$, instead of the token A followed by the token B.

1(11) *Parsing* is the process of matching a sequence of tokens to a syntactic nonterminal (satisfying any associated side conditions).

1(12) A *syntactically valid program* is a sequence of lexical terminals (the text of the program) that, upon scanning, can be parsed to the syntactic nonterminal *Program* without any remaining text.

1(13) Note that while some side conditions appear inline in the grammars of Sections 4 and 5, several other side conditions appear in Sections 7 and 8. All such conditions must be satisfied to successfully parse a program and ensure that such a program is in the language.

1.4 Input Elements

1(14) There are four distinct *parser contexts* defined by the lexical nonterminals InputElementXMLContent, InputElementXMLTag, InputElementOperator, and InputElementOperand. In a particular parser context, input elements (tokens and token separators) must match the particular lexical nonterminal that defines that parser context. The parser switches into a particular parser context before or after it matches particular nonterminals (i.e., when the current position immediately precedes or immediately succeeds some input text that matches particular nonterminals), as described below.

1. The parser switches into the parser context defined by InputElementXMLContent before and after matching a *XMLInitializer* or *XMLElementContent*.
2. The parser switches into the parser context defined by InputElementXMLTag before and after matching a *XMLTagContent*, *XMLTagName*, or *XMLAttribute*.
3. The parser is initially in the parser context defined by InputElementOperator, and switches into that parser context after matching a *PrimaryExpression*.
4. The parser switches into the parser context defined by InputElementOperand before matching a *PrimaryExpression*.

1(15) The input elements that serve as token separators are Whitespace, LineTerminator, and Comment. By separating tokens, they provide flexibility in how the text of a program is formatted. Token separators are discarded from the output of scanning (which then becomes the input of parsing), marking the locations of any LineTerminators (which are necessary to discharge some of the side conditions in Section 2.2). Note that token separators do not occur in *XMLInitializer*s (which have input elements matched by InputElementXMLTag and InputElementXMLContent), and so unlike in the other syntactic contexts, white space and line terminators that occur in these contexts are significant to the syntactic grammar.

2 Notation

2.1 Rules, Productions, Terminals, and Nonterminals

2(1) A rule spans several lines; the first line contains the nonterminal that is defined by the rule, and each remaining line contains a production for that nonterminal. Rules are separated by blank lines.

2(2) A production is a sequence of terminal and nonterminal symbols with optional side conditions at various positions in the sequence.

2(3) Names of syntactic nonterminals begin with uppercase letters and are in slanted sans serif font, e.g., *Expression*. Names of lexical nonterminals (which may also be syntactic terminals) begin with uppercase letters and are in sans serif font, e.g. NumericLiteral. Lexical terminals (which may also be syntactic terminals) represent sequences of Unicode code units that are either represented by literal characters in typewriter font, e.g. `{` or `function`, or described by Unicode categories.

2(4) Identifiers that are represented in typewriter font have special meaning in the context in which they occur in the grammar. Such identifiers may or may not be globally reserved. Globally reserved identifiers are listed in the lexical nonterminal Keyword.

2.2 Side Conditions

Side conditions rely on the following notation. ($\mathcal{X}$ is a metavariable denoting some grammatical entity).

2(5) Literal non-blank characters in a typewriter font are taken from the ISO Latin-1 character set and represent the corresponding Unicode code units.

2(6) ϵ is matched by the empty sequence.

2(7) $\mathcal{X}_{\texttt{opt}}$ is matched by either the empty sequence or a sequence that matches $\mathcal{X}$.

2(8) `U+` followed by four HexadecimalDigits (hexadecimal digits) is standard notation for a Unicode code unit.

2(9) $\langle \underline{\text{no}}\ \mathcal{X} \rangle$ requires the absence of a Unicode code unit that matches $\mathcal{X}$ immediately following the last character matched.

2(10) ⟨lookahead not $\mathcal{X}$⟩ requires that the next token not match $\mathcal{X}$.

2(11) ⟨noLineTerminator⟩ requires that no line terminator occurs between the previous token and the next token.

2(12) ⟨but not $\mathcal{X}$⟩ requires that the preceding nonterminal is not matched by a sequence of Unicode code units that matches $\mathcal{X}$.

2(13) ⟨but no embedded $\mathcal{X}$⟩ requires that the preceding nonterminal is not matched by a sequence of Unicode code units such that some subsequence of that sequence matches $\mathcal{X}$.

2(14) ⟨any Unicode $\mathcal{X}$⟩ is any Unicode code unit denoted by $\mathcal{X}$.

2(15) . . . or . . . means choice.

3 Lexical Grammar

3.1 Input Elements

InputElementXMLContent

1 XMLMarkup
2 XMLText
3 {
4 < ⟨no ? or !⟩
5 </

InputElementXMLTag [1]

6 XMLName
7 XMLAttributeValue
8 XMLWhitespace
9 =
10 {

[1] The definition of InputElementXMLTag differs from the one in ECMA-357 but is in fact correct: the lexical grammar in AS3 is tighter but does not disallow phrases that would not have been disallowed by the syntactic grammar anyway.

InputElementOperator

11 Whitespace
12 LineTerminator
13 Comment
14 IdentifierOrKeyword
15 NumericLiteral
16 StringLiteral
17 Punctuator

InputElementOperand

18 Whitespace
19 LineTerminator
20 Comment
21 IdentifierOrKeyword
22 NumericLiteral
23 StringLiteral
24 Punctuator ⟨but not / or /= or <= or < or <= or << or <<=⟩
25 RegularExpressionLiteral

26 XMLMarkup

27 < ⟨no ? or !⟩

3.2 Whitespace and Line Terminators

Whitespace [2]

28 U+0009

29 U+000B

30 U+000C

31 U+FEFF

32 ⟨any Unicode Zs⟩

[2] Any Unicode Cf can be used within comments, strings, and regular expressions. Outside of comments, strings, and regular expressions, the following three Unicode code units have the given meanings:

u200c → IdentifierPart

u200d → IdentifierPart

uFEFF → Whitespace

LineTerminator

33 U+000A

34 U+000D

35 U+2028

36 U+2029

37 U+000D U+000A

3.3 Comments

Comment

38 MultiLineComment

39 SingleLineComment

MultiLineComment

40 /* MultiLineCommentCharacters$_{\texttt{opt}}$ */

MultiLineCommentCharacters

41 SourceCharacters ⟨but no embedded * /⟩

SingleLineComment

42 // SingleLineCommentCharacters$_{\texttt{opt}}$

SingleLineCommentCharacters

43 SourceCharacters ⟨but no embedded LineTerminator⟩

SourceCharacters

44 SourceCharacter SourceCharacters$_{\texttt{opt}}$

SourceCharacter

45 ⟨any Unicode code unit⟩

3.4 Identifiers

Identifier

46 IdentifierOrKeyword ⟨but not Keyword⟩

IdentifierOrKeyword [3]

47 IdentifierStart

48 IdentifierOrKeyword IdentifierPart

[3] Unicode escape sequences may be used to spell the names of identifiers that would otherwise be keywords. This is in contrast to ECMAScript.

IdentifierStart

49 UnicodeLetter

50 $

51 _

52 \ UnicodeEscapeSequence

IdentifierPart

53 IdentifierStart

54 UnicodeCombiningMark

55 UnicodeDigit

56 UnicodeConnectorPunctuation

57 U+200C ⟨ZWNJ⟩

58 U+200D ⟨ZWJ⟩

UnicodeLetter

59 ⟨any Unicode Lu or Ll or Lt or Lm or Lo or Nl⟩

UnicodeCombiningMark

60 ⟨any Unicode Mn or Mc⟩

UnicodeDigit

61 ⟨any Unicode Nd⟩

UnicodeConnectorPunctuation

62 ⟨any Unicode Pc⟩

3.5 Keywords and Punctuators

Keyword [4]

63 `as`

64 `break`

65 `case`

66 `catch`

67 `class`

68 `const`

69 `continue`

70 `default`

71 `delete`

72 `do`

```
73   else
74   false
75   finally
76   for
77   function
78   if
79   import
80   in
81   include
82   instanceof
83   interface
84   internal
85   is
86   new
87   null
88   package
89   private
90   protected
91   public
92   return
93   super
94   switch
95   this
96   throw
97   true
98   try
99   typeof
100  use
101  var
102  void
103  while
104  with
```

4 Keywords are reserved words that have special meanings. Some Identifiers have special meanings in some syntactic contexts, but are not Keywords; such Identifiers are contextually reserved (see note for *NamespaceIdentifier*).

The following bugs were logged to track proposed additions to the list of keywords:

https://bugs.adobe.com/jira/browse/ASLSPEC-8
https://bugs.adobe.com/jira/browse/ASLSPEC-9

Punctuator

```
105    .
106    .<
107    ..
108    ...
109    !
110    !=
111    !==
112    %
113    %=
114    &
115    &=
116    &&
117 &&=
118 *
119 *=
120 +
121 +=
122 ++
123 -
124 -=
125 --
126 =
127 ==
128 ===
129 >
130 >=
131 >>
132 >>=
133 >>>
134 >>>=
135 ^
136 ^=
137 |
138 |=
139 ||
140 ||=
141 :
142 ::
143 (
144 )
145 [
146 ]
147 {
148 }
149 ~
150 ,
151 ;
152 ?
153 @
154 /
155 /=
156 <
157 <=
158 <<
159 <<=
```

3.6 Numeric Literals

NumericLiteral [5]

160 DecimalLiteral

161 HexadecimalIntegerLiteral

[5] The source character immediately following a NumericLiteral may be an IdentifierStart. This is in contrast to ECMAScript.

DecimalLiteral
162 DecimalDigits . DecimalDigits$_{opt}$ ExponentPart$_{opt}$
163 . DecimalDigits ExponentPart$_{opt}$
164 DecimalDigits ExponentPart$_{opt}$

DecimalDigits
165 DecimalDigit DecimalDigits$_{opt}$

DecimalDigit
166 0
167 1
168 2
169 3
170 4
171 5
172 6
173 7
174 8
175 9

ExponentPart
176 ExponentIndicator Sign$_{opt}$ DecimalDigits

ExponentIndicator
177 e
178 E

Sign
179 +
180 -

HexadecimalIntegerLiteral
181 0x HexadecimalDigit
182 0X HexadecimalDigit
183 HexadecimalIntegerLiteral HexadecimalDigit

HexadecimalDigit
184 0
185 1
186 2
187 3
188 4
189 5
190 6
191 7
192 8
193 9
194 a
195 b

```
196     c
197     d
198     e
199     f
200     A
201     B
202     C
203     D
204     E
205     F
```

3.7 String Literals

```
StringLiteral
206     " DoubleStringCharacters_opt "
207     ' SingleStringCharacters_opt '

DoubleStringCharacters
208     DoubleStringCharacter DoubleStringCharacters_opt

SingleStringCharacters
209     SingleStringCharacter SingleStringCharacters_opt

DoubleStringCharacter
210     SourceCharacter ⟨but not " or \ or LineTerminator⟩
211     \ EscapeSequence
212     LineContinuation

SingleStringCharacter
213     SourceCharacter ⟨but not ' or \ or LineTerminator⟩
214     \ EscapeSequence
215     LineContinuation

LineContinuation
216     \ LineTerminator

EscapeSequence [6]
217     CharacterEscapeSequence
218     0 ⟨no DecimalDigit⟩
219     HexadecimalEscapeSequence
220     UnicodeEscapeSequence

CharacterEscapeSequence
221     SingleEscapeCharacter
222     NonEscapeCharacter

SingleEscapeCharacter
```

[6] A \EscapeSequence is translated to a single Unicode code unit during lexical analysis. This means that its interpretation does not affect the lexical structure (and therefore syntax) of the program. For example, `\n` is a string character that is interpreted as a line feed. This holds for UnicodeEscapeSequence as well, e.g., `\u000A`, in contrast to Java's treatment of Unicode escape sequences, which are interpreted before lexical analysis.

223 '
224 "
225 \\
226 b
227 f
228 n
229 r
230 t
231 v

NonEscapeCharacter

232 SourceCharacter ⟨but not EscapeCharacter or LineTerminator⟩

EscapeCharacter

233 SingleEscapeCharacter
234 DecimalDigit
235 x
236 u

HexadecimalEscapeSequence

237 x HexadecimalDigit HexadecimalDigit

UnicodeEscapeSequence

238 u HexadecimalDigit HexadecimalDigit HexadecimalDigit HexadecimalDigit

3.8 Regular Expression Literals

RegularExpressionLiteral [7]

239 / RegularExpressionBody / RegularExpressionFlags$_{\text{opt}}$

[7] A RegularExpressionBody is never ϵ; instead of representing an empty regular expression, // starts a SingleLineComment. To specify an empty regular expression, use /(?:)/.

RegularExpressionBody

240 RegularExpressionFirstCharacter RegularExpressionCharacters$_{\text{opt}}$

RegularExpressionCharacters

241 RegularExpressionCharacter RegularExpressionCharacters$_{\text{opt}}$

RegularExpressionFirstCharacter

242 RegularExpressionNonTerminator ⟨but not * or \ or / or [⟩
243 RegularExpressionBackslashSequence
244 RegularExpressionClass

RegularExpressionCharacter

245 RegularExpressionNonTerminator ⟨but not \ or / or [⟩
246 RegularExpressionBackslashSequence
247 RegularExpressionClass

RegularExpressionBackslashSequence

248 \ SourceCharacter

RegularExpressionNonTerminator

249 SourceCharacter ⟨but not LineTerminator⟩

RegularExpressionClass

250 [RegularExpressionClassCharacters$_{\text{opt}}$]

RegularExpressionClassCharacters

251 RegularExpressionClassCharacter RegularExpressionClassCharacters$_{\text{opt}}$

RegularExpressionClassCharacter

252 RegularExpressionNonTerminator ⟨but not] or \⟩
253 RegularExpressionBackslashSequence

RegularExpressionFlags

254 IdentifierPart RegularExpressionFlags$_{\text{opt}}$

3.9 XML Literals

XMLMarkup

255 XMLComment
256 XMLCDATA
257 XMLPI

XMLWhitespaceCharacter

258 U+0009
259 U+000A
260 U+000D
261 U+0020

XMLWhitespace

262 XMLWhitespaceCharacter XMLWhitespace$_{\text{opt}}$

XMLText

263 SourceCharacters ⟨but no embedded { or <⟩

XMLName

264 XMLNameStart
265 XMLName XMLNamePart

XMLNameStart

266 UnicodeLetter
267 _
268 :

XMLNamePart

269 UnicodeLetter
270 UnicodeDigit
271 .

272 -

273 _

274 :

XMLComment

275 <!-- XMLCommentCharacters$_{opt}$ -->

XMLCommentCharacters

276 SourceCharacters ⟨but no embedded --⟩

XMLCDATA

277 <![CDATA[XMLCDATACharacters$_{opt}$]]>

XMLCDATACharacters

278 SourceCharacters ⟨but no embedded]]>⟩

XMLPI

279 <? XMLPICharacters$_{opt}$?>

XMLPICharacters

280 SourceCharacters ⟨but no embedded ?>⟩

XMLAttributeValue

281 " XMLDoubleStringCharacters$_{opt}$ "

282 ' XMLSingleStringCharacters$_{opt}$ '

XMLDoubleStringCharacters

283 SourceCharacters ⟨but no embedded "⟩

XMLSingleStringCharacters

284 SourceCharacters ⟨but no embedded '⟩

4 Syntactic Grammar

4.1 Names

Name

285 *UnqualifiedName*

286 *QualifiedName*

UnqualifiedName[8]

287 Identifier

[8] An *UnqualifiedName* represents an Identifier that is (implicitly) associated with a set of open namespaces in scope. The Identifier when associated with the set of open namespaces in scope becomes a *multiname*, denoting a set of *QualifiedName*s.

QualifiedName

288 *PackageName* . Identifier

289 *NamespaceExpression* :: *QualifiedNameIdentifier*

QualifiedNameIdentifier [9]

290 `*`
291 Identifier
292 *Brackets*

[9]

290: `*` represents an Identifier wildcard and matches all Identfiers that occur in the context that it occurs.

292: *Brackets* represents an *UnqualifiedName* with an *Identifier* that is computed at run time.

PropertyName [10]

293 `*`
294 *Name*
295 *XMLAttributeName*

[10]

293: ASC always scans `*` followed by `=` as the assignment operator `*=` (and never as a *PropertyName* `*` followed by the assignment operator `=`). The following bug has been logged to track this issue:

https://bugs.adobe.com/jira/browse/ASLSPEC-20

XMLAttributeName

296 `@ *`
297 `@` *Name*
298 `@` *Brackets*

NamespaceName [11]

299 *RestrictedName*

[11]

NamespaceName is called out in the grammar because it is a special use of *RestrictedName* that resolves to a namespace value at compile time.

RestrictedName

300 *UnqualifiedName*
301 *ReservedNamespace* `::` *Identifier*
302 *PackageName* `.` *Identifier*
303 *RestrictedName* `::` *Identifier*
304 `(` *RestrictedName* `)`

NamespaceExpression

305 `*`
306 *Name*
307 *ReservedNamespace*
308 *ParenExpression*

ReservedNamespace

309 `internal`
310 `private`
311 `protected`
312 `public`

4.2 Types

TypedBinding
313 Identifier
314 Identifier `:` *Type*

TypeName[12]
315 *RestrictedName*

[12] *TypeName* is called out in the grammar because it is a special use of *RestrictedName* that resolves to a type value at compile time.

Type[13]
316 `*`
317 *TypeName*
318 *Identifier TypeApplication*

[13] 318: The *Identifier* associated with the *TypeApplication* must be `Vector`. (In fact, the *TypeName* must reference the built-in definition of `Vector`.)

TypeApplication
319 `.<` *Type* `>`

4.3 Primary Expressions

ArrayInitializer
320 `[` *ArrayElements*$_{\mathtt{opt}}$ `]`

ArrayElements
321 *ArrayElement*
322 `,` *ArrayElements*$_{\mathtt{opt}}$
323 *ArrayElement* `,` *ArrayElements*$_{\mathtt{opt}}$

ArrayElement
324 *ConfigCondition*$_{\mathtt{opt}}$ *AssignmentExpression*

VectorInitializer
325 `new <` *Type* `> [` *VectorElements*$_{\mathtt{opt}}$ `]`

VectorElements[14]
326 *VectorElement*
327 *VectorElement* `,` *VectorElements*$_{\mathtt{opt}}$

[14] *VectorElements* does not allow holes (unlike *ArrayElements*), but a trailing comma is allowed.

VectorElement
328 *ConfigCondition*$_{\mathtt{opt}}$ *AssignmentExpression*

ObjectInitializer

329 { *Fields*$_{\text{opt}}$ }

Fields[15]

330 *Field*

331 *Field* , *Fields*$_{\text{opt}}$

[15] ASC currently does not allow a trailing comma in *Fields*. The following bug has been logged to track this issue:

https://bugs.adobe.com/jira/browse/ASLSPEC-18

Field

332 *ConfigCondition*$_{\text{opt}}$ *FieldName* : *AssignmentExpression*

FieldName

333 Identifier

334 StringLiteral

335 NumericLiteral

XMLInitializer

336 XMLMarkup

337 *XMLElement*

XMLElement

338 < *XMLTagContent* XMLWhitespace$_{\text{opt}}$ />

339 < *XMLTagContent* XMLWhitespace$_{\text{opt}}$ > *XMLElementContent*$_{\text{opt}}$ </ *XMLTagName* XMLWhitespace$_{\text{opt}}$ >

XMLTagContent

340 *XMLTagName* *XMLAttributes*$_{\text{opt}}$

XMLTagName

341 { *Expression* }

342 XMLName

XMLAttributes

343 XMLWhitespace { *Expression* }

344 *XMLAttribute* *XMLAttributes*$_{\text{opt}}$

XMLAttribute

345 XMLWhitespace XMLName XMLWhitespace$_{\text{opt}}$ = XMLWhitespace$_{\text{opt}}$ { *Expression* }

346 XMLWhitespace XMLName XMLWhitespace$_{\text{opt}}$ = XMLWhitespace$_{\text{opt}}$ XMLAttributeValue

XMLElementContent

347 { *Expression* } *XMLElementContent*$_{\text{opt}}$

348 XMLMarkup *XMLElementContent*$_{\text{opt}}$

349 XMLText *XMLElementContent*$_{\text{opt}}$

350 *XMLElement* *XMLElementContent*$_{\text{opt}}$

XMLListInitializer

351 < > *XMLElementContent*$_{\text{opt}}$ </ >

FunctionExpression

352 `function` Identifier$_{\texttt{opt}}$ *FunctionSignature* *FunctionBody*

FunctionSignature
353 `( )` *ResultType*$_{\texttt{opt}}$
354 `(` *Parameters* `)` *ResultType*$_{\texttt{opt}}$

Parameters
355 *RestParameter*
356 *NonRestParameters*
357 *NonRestParameters* `,` *RestParameter*

NonRestParameters
358 *Parameter*
359 *OptionalParameters*
360 *Parameter* `,` *NonRestParameters*

OptionalParameters
361 *OptionalParameter*
362 *OptionalParameter* `,` *OptionalParameters*

OptionalParameter[16]
363 *Parameter* `=` *NonAssignmentExpression*

[16]

363: *NonAssignmentExpression* must be compile-time evaluable to `true`, `false`, `null`, undefined, or a value of type `String`, `Number`, or `Namespace`.

Parameter
364 *TypedBinding*

RestParameter[17]
365 `...`
366 `...` *TypedBinding*

[17]

366: If a *RestParameter* has a *TypedBinding*, its *Type* must be `Array`. (In fact, the *Type* must reference the built-in definition of `Array`.)

ResultType
367 `: void`
368 `:` *Type*

FunctionBody
369 *Block*

PrimaryExpression[18]
370 `null`
371 `true`
372 `false`
373 `this`
374 NumericLiteral
375 StringLiteral
376 RegularExpressionLiteral

377 *ArrayInitializer*
378 *VectorInitializer*
379 *ObjectInitializer*
380 *XMLInitializer*
381 *XMLListInitializer*
382 *FunctionExpression*

[18]

373: Further syntactic restrictions on the keyword `this` appear in Section 8.8.1.

4.4 Expressions

ParenExpression
383 `(` *Expression* `)`

Arguments
384 `(` *ArgumentExpressions*$_{\texttt{opt}}$ `)`

ArgumentExpressions
385 *AssignmentExpression*
386 *ArgumentExpressions* `,` *AssignmentExpression*

PropertyOperator
387 *ReferenceOperator*
388 `.` *ParenExpression*
389 *TypeApplication*

ReferenceOperator
390 *BasicReferenceOperator*
391 `..` *PropertyName*

BasicReferenceOperator
392 `.` *PropertyName*
393 *Brackets*

Brackets
394 `[` *Expression* `]`

SuperExpression[19]
395 `super` *BasicReferenceOperator*
396 `super` *ParenExpression* *BasicReferenceOperator*

[19] Further syntactic restrictions on super expressions appear in Section 8.8.2.

MemberExpression
397 *PrimaryExpression*
398 *ParenExpression*
399 *PropertyName*
400 *SuperExpression*
401 *MemberExpression* *PropertyOperator*
402 `new` *MemberExpression* *Arguments*

```
CallExpression
403      MemberExpression Arguments
404      CallExpression Arguments
405      CallExpression PropertyOperator

LeftHandSideExpression
406      PropertyName
407      SuperExpression
408      MemberExpression ReferenceOperator
409      CallExpression ReferenceOperator

NewExpression
410      MemberExpression
411      new NewExpression

PostfixExpression
412      NewExpression
413      CallExpression
414      LeftHandSideExpression ++
415      LeftHandSideExpression --

PrefixExpression
416      PostfixExpression
417      delete PostfixExpression
418      ++ LeftHandSideExpression
419      -- LeftHandSideExpression

UnaryExpression
420      PrefixExpression
421      void UnaryExpression
422      typeof UnaryExpression
423      + UnaryExpression
424      - UnaryExpression
425      ~ UnaryExpression
426      ! UnaryExpression

MultiplicativeExpression
427      UnaryExpression
428      MultiplicativeExpression * UnaryExpression
429      MultiplicativeExpression / UnaryExpression
430      MultiplicativeExpression % UnaryExpression

AdditiveExpression
431      MultiplicativeExpression
432      AdditiveExpression + MultiplicativeExpression
433      AdditiveExpression - MultiplicativeExpression

ShiftExpression
434      AdditiveExpression
435      ShiftExpression << AdditiveExpression
```

```
436     ShiftExpression >> AdditiveExpression
437     ShiftExpression >>> AdditiveExpression

RelationalExpression
438     ShiftExpression
439     RelationalExpression < ShiftExpression
440     RelationalExpression > ShiftExpression
441     RelationalExpression <= ShiftExpression
442     RelationalExpression >= ShiftExpression
443     RelationalExpression in ShiftExpression
444     RelationalExpression as ShiftExpression
445     RelationalExpression instanceof ShiftExpression
446     RelationalExpression is ShiftExpression

EqualityExpression
447     RelationalExpression
448     EqualityExpression == RelationalExpression
449     EqualityExpression != RelationalExpression
450     EqualityExpression === RelationalExpression
451     EqualityExpression !== RelationalExpression

BitwiseANDExpression
452     EqualityExpression
453     BitwiseANDExpression & EqualityExpression

BitwiseXORExpression
454     BitwiseANDExpression
455     BitwiseXORExpression ^ BitwiseANDExpression

BitwiseORExpression
456     BitwiseXORExpression
457     BitwiseORExpression | BitwiseXORExpression

LogicalANDExpression
458     BitwiseORExpression
459     LogicalANDExpression && BitwiseORExpression

LogicalORExpression
460     LogicalANDExpression
461     LogicalORExpression || LogicalANDExpression

ConditionalExpression
462     LogicalORExpression
463     LogicalORExpression ? AssignmentExpression : AssignmentExpression

NonAssignmentExpression
464     LogicalORExpression
465     LogicalORExpression ? NonAssignmentExpression : NonAssignmentExpression

AssignmentExpression
```

466 *ConditionalExpression*
467 *LeftHandSideExpression* `=` *AssignmentExpression*
468 *LeftHandSideExpression* `*=` *AssignmentExpression*
469 *LeftHandSideExpression* `/=` *AssignmentExpression*
470 *LeftHandSideExpression* `%=` *AssignmentExpression*
471 *LeftHandSideExpression* `+=` *AssignmentExpression*
472 *LeftHandSideExpression* `-=` *AssignmentExpression*
473 *LeftHandSideExpression* `<<=` *AssignmentExpression*
474 *LeftHandSideExpression* `>>=` *AssignmentExpression*
475 *LeftHandSideExpression* `>>>=` *AssignmentExpression*
476 *LeftHandSideExpression* `&=` *AssignmentExpression*
477 *LeftHandSideExpression* `^=` *AssignmentExpression*
478 *LeftHandSideExpression* `|=` *AssignmentExpression*
479 *LeftHandSideExpression* `&&=` *AssignmentExpression*
480 *LeftHandSideExpression* `||=` *AssignmentExpression*

Expression

481 *AssignmentExpression*
482 *Expression* `,` *AssignmentExpression*

4.5 Statements

Statement

483 *BreakStatement*
484 *ContinueStatement*
485 *DefaultXMLNamespaceStatement*
486 *EmptyStatement*
487 *ExpressionStatement*
488 *ForStatement*
489 *IfStatement*
490 *LabeledStatement*
491 *MetadataStatement*
492 *ReturnStatement*
493 *SuperStatement*
494 *SwitchStatement*
495 *ThrowStatement*
496 *TryStatement*
497 *WhileStatement*
498 *DoStatement*
499 *WithStatement*

Substatement

500 *Statement*
501 *Block*
502 *VariableDefinition*

Block

503 `{` *Directives*$_{\texttt{opt}}$ `}`

InnerSubstatement[20]

504 *Substatement*

20

InnerSubstatement is defined in the grammar for the sole purpose of specifying side conditions that disambiguate various syntactic ambiguities in a context-sensitive manner specified in Section 5.

Semicolon 21

505 ;

506 *VirtualSemicolon*

507 ϵ

21

507: This rule involves a syntactic ambiguity around ϵ, which is disambiguated by a side condition specified in 5(3).

VirtualSemicolon 22

508 ϵ ⟨followed by at least one LineTerminator, and preceded by a valid prefix that is invalidated by the following token⟩

22 If the 1^{st} through the n^{th} tokens of a program can be parsed but the 1^{st} through the $n+1^{\text{th}}$ tokens cannot and there is at least one line break between the n^{th} token and the $n+1^{\text{st}}$ token, then the parser tries to parse the program again after inserting a *VirtualSemicolon* between the n^{th} and the $n+1^{\text{st}}$ tokens.

EmptyStatement

509 ;

ExpressionStatement

510 ⟨lookahead not `[` or `{` or `function`⟩ *Expression Semicolon*

LabeledStatement 23

511 Identifier : *Substatement*

23 Further syntactic restrictions on labeled statements appear in Section 8.7.2.

IfStatement 24

512 `if` *ParenExpression Substatement*

513 `if` *ParenExpression InnerSubstatement* `else` *Substatement*

24 This rule involves a syntactic ambiguity around `else`, which is disambiguated by a side condition specified in 5(1).

WithStatement

514 `with` *ParenExpression Substatement*

SwitchStatement

515 `switch` *ParenExpression Cases*

Cases

516 `{` $\textit{CaseClauses}_{\texttt{opt}}$ `}`

517 `{` $\textit{CaseClauses}_{\texttt{opt}}$ *DefaultClause* $\textit{CaseClauses}_{\texttt{opt}}$ `}`

CaseClauses

518 *CaseClause* $\textit{CaseClauses}_{\texttt{opt}}$

CaseClause

519 `case` *Expression* : $\textit{Directives}_{\texttt{opt}}$

DefaultClause

520 `default :` *Directives*$_{\texttt{opt}}$

WhileStatement

521 `while` *ParenExpression Substatement*

DoStatement

522 `do` *InnerSubstatement* `while` *ParenExpression Semicolon*

ForStatement[25]

523 `for (` *ForInitializer*$_{\texttt{opt}}$ `;` *Expression*$_{\texttt{opt}}$ `;` *Expression*$_{\texttt{opt}}$ `)` *Substatement*

524 `for (` *ForInInitializer* `in` *Expression* `)` *Substatement*

525 `for each (` *ForInInitializer* `in` *Expression* `)` *Substatement*

[25] This rule involves a syntactic ambiguity around `in`, which is disambiguated by a side condition specified in 5(2).

ForInitializer

526 *Expression*

527 *VariableDefinitionKind VariableBindings*

ForInInitializer

528 *LeftHandSideExpression*

529 *VariableDefinitionKind VariableBinding*

ContinueStatement[26]

530 `continue` *Semicolon*

531 `continue` ⟨noLineTerminator⟩ Identifier *Semicolon*

[26] Further syntactic restrictions on continue statements appear in Section 8.7.4.

BreakStatement[27]

532 `break` *Semicolon*

533 `break` ⟨noLineTerminator⟩ Identifier *Semicolon*

[27] Further syntactic restrictions on break statements appear in Section 8.7.3.

ReturnStatement[28]

534 `return` *Semicolon*

535 `return` ⟨noLineTerminator⟩ *Expression Semicolon*

[28] Further syntactic restrictions on return statements appear in Section 8.7.5.

ThrowStatement

536 `throw` ⟨noLineTerminator⟩ *Expression Semicolon*

TryStatement

537 `try` *Block CatchClauses*

538 `try` *Block* `finally` *Block*

539 `try` *Block CatchClauses* `finally` *Block*

CatchClauses

540 *CatchClause CatchClauses*$_{\texttt{opt}}$

CatchClause

541 `catch (` *TypedBinding* `)` *Block*

[29]

[29] Further syntactic restrictions on super statements appear in Section 8.7.1.

SuperStatement

542 `super` *Arguments Semicolon*

DefaultXMLNamespaceStatement

543 `default xml namespace =` *Expression Semicolon*

MetadataStatement[30]

544 `[` *ArrayElements*$_{\texttt{opt}}$ `]`

[30] A metadata statement does not have a trailing semicolon.

4.6 Definitions

AttributedDefinition

545 *ConfigCondition*$_{\texttt{opt}}$ *Attributes*$_{\texttt{opt}}$ *Definition*

ConfigCondition[31]

546 *Identifier* `::` *Identifier*

[31]

A *ConfigCondition* must resolve at parse time to a boolean value (true or false). If the value of a *ConfigCondition* is false, the *GroupDirective*, *Field*, *ArrayElement*, *VectorElement*, or *AttributedDefinition* in which it appears is erased from the program. If the value of the *ConfigCondition* is true, only the *ConfigCondition* is erased. See 7(13).

Attributes[32]

547 *ModifierAttribute* ⟨noLineTerminator⟩ *AttributesPart*$_{\texttt{opt}}$

548 *NamespaceName* ⟨noLineTerminator⟩ *AttributesPart*$_{\texttt{opt}}$

549 *ReservedNamespace AttributesPart*$_{\texttt{opt}}$

[32]

A LineTerminator is not allowed after the first attribute if it is a *ModifierAttribute* or *NamespaceName*. This is to disambiguate an *ExpressionStatement*, which may be a *Name*.

AttributesPart

550 *ModifierAttribute AttributesPart*$_{\texttt{opt}}$

551 *NamespaceAttribute AttributesPart*$_{\texttt{opt}}$

NamespaceAttribute

552 *NamespaceName*

553 *ReservedNamespace*

ModifierAttribute

554 `dynamic`

555 `final`

556 `native`

557 `override`

558 `static`
559 `virtual`

Definition

560 *VariableDefinition*
561 *NamespaceDefinition*
562 *FunctionDefinition*
563 *ClassDefinition*
564 *InterfaceDefinition*

VariableDefinition[33]

565 *VariableDefinitionKind VariableBindings Semicolon*

[33] Further syntactic restrictions on variable definitions appear in Section 8.5.

VariableDefinitionKind

566 `const`
567 `var`

VariableBindings

568 *VariableBinding*
569 *VariableBindings* `,` *VariableBinding*

VariableBinding

570 *TypedBinding VariableInitialization*$_{\texttt{opt}}$

VariableInitialization

571 `=` *AssignmentExpression*

NamespaceDefinition[34]

572 `namespace` ⟨noLineTerminator⟩ *NamespaceIdentifier NamespaceInitialization*$_{\texttt{opt}}$ *Semicolon*

[34] Further syntactic restrictions on namespace definitions appear in Section 8.6.

NamespaceIdentifier[35]

573 Identifier ⟨but not `config` or `dynamic` or `final` or `namespace` or `native` or `override` or `static` or `virtual`⟩

[35]

Identifiers that are contextually reserved cannot be used to define namespaces. This is to avoid ambiguities that would occur if they were used as attributes in *AttributedDefinition*s.

NamespaceInitialization

574 `=` *NamespaceName*
575 `=` StringLiteral

FunctionDefinition[36]

576 `function` *AccessorKind*$_{\texttt{opt}}$ Identifier *FunctionSignature OptionalFunctionBody*

[36] Further syntactic restrictions on function definitions appear in Section 8.4.

AccessorKind

577 `get`
578 `set`

OptionalFunctionBody

579 *FunctionBody*
580 *Semicolon*

ClassDefinition[37]

581 `class` Identifier *ClassInheritance*$_{\texttt{opt}}$ *ClassBody*

[37] Further syntactic restrictions on class definitions appear in Section 8.2.

ClassInheritance

582 `extends` *TypeName*
583 `implements` *TypeNames*
584 `extends` *TypeName* `implements` *TypeNames*

TypeNames

585 *TypeName*
586 *TypeNames* `,` *TypeName*

ClassBody

587 *Block*

InterfaceDefinition[38]

588 `interface` Identifier *InterfaceInheritance*$_{\texttt{opt}}$ *InterfaceBody*

[38] Further syntactic restrictions on interface definitions appear in Section 8.3.

InterfaceInheritance

589 `extends` *TypeNames*

InterfaceBody

590 *Block*

4.7 Directives

Directives

591 *Directive* *Directives*$_{\texttt{opt}}$

Directive

592 *IncludeDirective*
593 *ConfigNamespaceDirective*
594 *PackageDirective*
595 *ImportDirective*
596 *UseDirective*
597 *GroupDirective*
598 *AttributedDefinition*
599 *Statement*

IncludeDirective

600 `include` StringLiteral *Semicolon*

ConfigNamespaceDirective[39]

601 `config` ⟨noLineTerminator⟩ `namespace` Identifier *Semicolon*

[39] Further syntactic restrictions on program configuration constructs appear in Section 7.1.

PackageDirective[40]

602 `package` *PackageName*$_{\texttt{opt}}$ *Block*

[40] Further syntactic restrictions on package directives appear in Section 8.1.

PackageName[41]

603 Identifier

604 *PackageName* `.` Identifier

[41]

A Whitespace or LineTerminator is allowed around a `.` in a *PackageName*. For example, the following is a syntactically valid program:

```
package a .
        b
{    }
```

The resulting *PackageName* value is equivalent to a *PackageName* without any intervening Whitespace and LineTerminators.

ImportDirective

605 `import` *PackageName* `. *` *Semicolon*

606 `import` *PackageName* `.` Identifier *Semicolon*

UseDirective

607 `use namespace` *NamespaceName Semicolon*

GroupDirective

608 *ConfigCondition*$_{\texttt{opt}}$ *Group*

Group

609 `{` *Directives*$_{\texttt{opt}}$ `}`

Program

610 *Directives*$_{\texttt{opt}}$

5 Resolution of Syntactic Ambiguities

Definition 5.1 (Braced syntax tree)**.** A *braced* syntax tree is a *Block*, *Group*, *Cases*, or *ObjectInitializer*.

Definition 5.2 (Bracketed syntax tree)**.** A *bracketed* syntax tree is a *ArrayInitializer*, *VectorInitializer*, or *Brackets*, or *MetadataStatement*.

Definition 5.3 (Parenthesized syntax tree)**.** A *parenthesized* syntax tree is a *ParenExpression*, *Arguments*, or *Parameters*.

Definition 5.4 (Exposed)**.** A syntax tree A is *exposed* in a syntax tree B if A is either B or is nested by B without crossing a braced syntax tree, bracketed syntax tree, parenthesized syntax tree, or an *InnerSubstatement*.

The following rules disambiguate ambiguities in the syntactic grammar.

5(1) Any exposed *IfStatement* in the *InnerSubstatement* of another *IfStatement* must be of the form

> `if` *ParenExpression InnerSubstatement* `else` *Substatement*

5(2) Any exposed *RelationalExpression* in a *ForBinding* or a *ForInitializer* must not be of the form

> *RelationalExpression* `in` *ShiftExpression*

5(3) Any *Directive* that has a trailing *Semicolon* that is ϵ must be followed by `}` or the end of input, unless it is exposed in an *InnerSubstatement*.

Semicolon insertion in AS3 is more lenient than in ES5. In particular, AS3 allows these two cases that are not allowed in ES5:

```
do x++ while (x < 10);       // ES5 would require a ; after x++
if (x > 10) x++ else y++;    // ES5 would require a ; after x++
```

5(4) **Examples.** The following examples show how 5(1), 5(2), and 5(3) are applied to disambiguate various syntactic ambiguities.

```
if (true) if (false) { } else { print("...") }                            // "..."
if (true) { if (false) { } } else { print("...") }                        // no operation

for (var i = -1 in [];;) { }                                               // syntax error
for (var i = (-1 in []);;) { }                                             // infinite loop

for (var i = -1 in [], a = [false,true]) { print(a[i]) }                  // "false" "true"

for each (var x = true in [true] in true ? [true] : [false]) { print(x) }     // "false"
for each (var x = (true in [true]) in true ? [true] : [false]) { print(x) }   // "true"

do print("...") while (false) print(false) while (false) print(true)         // syntax error
do print("...") while (false); print(false); while (false) print(true)       // "..." "false"
do { print("..."); while (false) print(true) } while (false); print(true)    // "..." "true"
```

6 Expansion of Include Directives

6(1) Expansion of include directives is the process of replacing *IncludeDirective*s in a program with syntax trees corresponding to the included files.

6(2) The text of a file included by an *IncludeDirective* must match *Directives*$_{\texttt{opt}}$, deriving a syntax tree that replaces the *IncludeDirective*.

6(3) The syntax tree that replaces an *IncludeDirective* is derived by scanning, parsing, resolution of syntactic ambiguities, and (recursive) expansion of include directives, but no later phases. In particular, this means that program configuration, enforcement of syntactic restrictions, and so on are done only after all include directives have been expanded.

7 Program Configuration

7(1) A *Program* is treated as if it had a *ConfigNamespaceDirective* with Identifier `CONFIG` before its *Directive*s.

Definition 7.1 (Configuration namespace)**.** A *configuration namespace* is an *UnqualifiedName* A that is defined by a *ConfigNamespaceDirective* whose Identifier is A.

Definition 7.2 (Configuration name)**.** A *configuration name* is a *QualifiedName* A :: B that is defined by a *VariableDefinition* whose *VariableDefinitionKind* is `const`, whose *NamespaceAttribute* is a configuration namespace A, and whose Identifier is B.

7(2) Program configuration is the process of evaluating configuration names and erasing various constructs based on their values. This process is done before any later syntactic transformation (such as enforcement of syntactic restrictions). It transforms a valid program such that it does not have any configuration namespaces and configuration names.

7.1 Syntactic Restrictions on Configuration Constructs

Definition 7.3 (Global context)**.** A syntax tree is in a *global context* if it is nested by a *Program* without crossing a *Block*, or nested by the *Block* of a *PackageDirective* without crossing another *Block*.

7(3) The following syntactic restrictions ensure that configuration names can be evaluated by a simple depth-first traversal of the *Program*, and definitions of configuration namespaces and configuration names can be safely erased after replacing configuration names by their values.

7(4) A configuration namespace definition may appear only in a global context.

7(5) A configuration name definition may appear only in a global context.

7(6) There must be a unique definition for every configuration namespace, and a unique definition for every configuration name.

7(7) A configuration namespace may appear only as the *NamespaceAttribute* in a configuration name definition, or as the *NamespaceExpression* in a configuration name.

7(8) For any configuration name A::B, the definition of the configuration namespace A must appear earlier than the definition of the configuration name A::B.

7(9) The definition of a configuration name must have an initializer that is a *NonAssignmentExpression* in which: any *UnaryExpression* must be a boolean literal, numeric literal, string literal, `null`, or configuration name; and any Punctuator must be `!`, `||`, `&&`, `!=`, `==`, `!==`, `===`, `+`, `-`, `*`, `/`, `%`, `<`, `<=`, `>`, `>=`, `<<`, `>>`, `>>>`, `&`, `|`, `^`, or `?...:`.

7(10) The definition of a configuration name A::B must appear earlier than any use of A::B in the program.

7(11) A *ConfigCondition* must be a configuration name.

7.2 Evaluation of Configuration Names

Definition 7.4 (Metadata association)**.** A *MetadataStatement* is associated with an *AttributedDefinition* if the *MetadataStatement* is immediately followed by that *AttributedDefinition*, or is immediately followed by another *MetadataStatement* that is associated with that *AttributedDefinition*.

7(12) Configuration names are evaluated in the order in which they are defined in the program. The value of a configuration name A::B is the value of the *NonAssignmentExpression* in the definition of A::B. The evaluation semantics is exactly that of constant expressions, as defined elsewhere.

7(13) A *ConfigCondition* must have the value of either true or false. A *ConfigCondition* whose value is true is erased from the syntax tree. In contrast, a *ConfigCondition* whose value is false causes the *Field*, *ArrayElement*, *VectorElement*, *GroupDirective*, or *AttributedDefinition* that it is part of, as well as any *MetadataStatement* that is associated with such a *AttributedDefinition*, to be erased from the syntax tree.

7(14) Any other configuration name in the program is replaced by its value.

7(15) Finally, the definitions of any configuration namespaces and configuration names are erased from the syntax tree.

8 Enforcement of Syntactic Restrictions

Definition 8.1 (Class context)**.** A syntax tree is in a *class context* if it is nested by the *Block* of a *ClassBody* without crossing another *Block*.

Definition 8.2 (Interface context)**.** A syntax tree is in an *interface context* if it is nested by the *Block* of a *InterfaceBody* without crossing another *Block*.

Definition 8.3 (Constructor)**.** A *constructor* is a *FunctionDefinition* that is in a class context, and whose name has an identifier that matches the identifier of that class.

Definition 8.4 (Getter/Setter)**.** A *getter* is a *FunctionDefinition* whose *AccessorKind* is `get`. A *setter* is a *FunctionDefinition* whose *AccessorKind* is `set`.

Definition 8.5 (Result type)**.** The *result type* of a *FunctionExpression* or *FunctionDefinition* that has a *ResultType* is that *ResultType*. The *result type* of a *FunctionExpression* or *FunctionDefinition* that does not have a *ResultType* is `*`.

Definition 8.6 (Mark)**.** Any *NamespaceAttribute* or *ModifierAttribute* of an *AttributedDefinition* *marks* the *Definition* of that *AttributedDefinition*.

Definition 8.7 (Bodyless)**.** A *FunctionDefinition* is *bodyless* if its *OptionalFunctionBody* is *Semicolon*.

Definition 8.8 (Return expression)**.** A *FunctionExpression* or *FunctionDefinition* has a *return expression* if a *ReturnStatement* is nested by it without crossing another *FunctionBody*, and the *ReturnStatement* has an *Expression*.

The following side conditions must be satisfied to ensure that a syntax tree is in the language.

8.1 Package Directives

8(1) A *PackageDirective* may appear only in a global context.

8(2) A *PackageDirective* must not be nested by another *PackageDirective*.

8.2 Class Definitions

8(3) A *ClassDefinition* may appear only in a global context.

8.2.1 Namespace Attributes of Class Definitions

8(4) The only *NamespaceAttribute*s that may mark a *ClassDefinition* are `public` and `internal`.

8(5) The only *NamespaceAttribute* that may mark a *ClassDefinition* in a global context that is not nested by a *PackageDirective* is `internal`.

8(6) At most one *NamespaceAttribute* may mark a *ClassDefinition* and that *NamespaceAttribute* must not mark the *ClassDefinition* more than once.

8.2.2 Modifier Attributes of Class Definitions

8(7) The only *ModifierAttribute*s that may mark a *ClassDefinition* are `dynamic` and `final`.

8(8) A particular *ModifierAttribute* must not mark a *ClassDefinition* more than once.

8.3 Interface Definitions

8(9) An *InterfaceDefinition* may appear only in a global context.

8.3.1 Namespace Attributes of Interface Definitions

8(10) The only *NamespaceAttribute*s that may mark a *InterfaceDefinition* are `public` and `internal`.

8(11) The only *NamespaceAttribute* that may mark an *InterfaceDefinition* in a global context that is not nested by a *PackageDirective* is `internal`.

8(12) At most one *NamespaceAttribute* may mark a *InterfaceDefinition* and that *NamespaceAttribute* must not mark the *InterfaceDefinition* more than once.

8.3.2 Modifier Attributes of Interface Definitions

8(13) No *ModifierAttribute* may mark a *InterfaceDefinition*.

8.4 Function Definitions

8.4.1 Namespace Attributes of Function Definitions

8(14) A *NamespaceAttribute* may mark a *FunctionDefinition* only if it is in a class context or in a global context.

8(15) The only *NamespaceAttribute* that may mark a *FunctionDefinition* in a global context that is not nested by a *PackageDirective* is `internal`.

8(16) The only *NamespaceAttribute*s that may mark a *FunctionDefinition* in a global context nested by a *PackageDirective* are `public` and `internal`.

8(17) At most one *NamespaceAttribute* may mark a *FunctionDefinition* and that *NamespaceAttribute* must not mark the *FunctionDefinition* more than once.

8.4.2 Modifier Attributes of Function Definitions

8(18) The only *ModifierAttribute*s that may mark a *FunctionDefinition* are `final`, `override`, `virtual`, `static`, and `native`.

8(19) The *ModifierAttribute*s `final`, `override`, `virtual`, and `static` may mark a *FunctionDefinition* only if it is in a class context.

8(20) The *ModifierAttribute* `native` may mark a *FunctionDefinition* only in a global context or a class context.

8(21) A *FunctionDefinition* that is marked `static` must not be marked by `final`, `override`, or `virtual`.

8(22) A *FunctionDefinition* must not be marked by both `final` and `virtual`.

8(23) A particular *ModifierAttribute* must not mark a *FunctionDefinition* more than once.

8.4.3 Function Bodies of Function Definitions

8(24) A *FunctionDefinition* must be bodyless if it is in an interface context or is marked `native`.

8(25) A *FunctionDefinition* must not be bodyless unless is in an interface context, is marked marked `native`, or is a constructor.

8.4.4 Getters and Setters

8(26) A getter or setter may appear only in a global context, class context, or interface context.

8(27) The *FunctionSignature* of a getter must not have *Parameters*.

8(28) The *FunctionSignature* of a setter must have *Parameters*, which must be exactly one *Parameter*.

8(29) The result type of a setter must be `void` or `*`.

8.4.5 Constructors

8(30) A constructor must not be a getter or setter.

8(31) The only *NamespaceAttribute* that may mark a constructor is `public`.

8(32) No *ModifierAttribute* may mark a constructor.

8(33) The result type of a constructor must be `void` or `*`.

8.5 Variable Definitions

8(34) A *VariableDefinition* must not appear in an interface context.

8.5.1 Namespace Attributes of Variable Definitions

8(35) A *NamespaceAttribute* may mark a *VariableDefinition* only if it is in a class context or in a global context.

8(36) The only *NamespaceAttribute* that may mark a *VariableDefinition* in a global context that is not nested by a *PackageDirective* is `internal`.

8(37) The only *NamespaceAttribute*s that may mark a *VariableDefinition* in a global context nested by a *PackageDirective* are `public` and `internal`.

8(38) At most one *NamespaceAttribute* may mark a *VariableDefinition*, and that *NamespaceAttribute* must not mark the *VariableDefinition* more than once.

8.5.2 Modifier Attributes of Variable Definitions

8(39) The only *ModifierAttribute* that may mark a *VariableDefinition* is `static`.

8(40) The *ModifierAttribute* `static` may mark a *VariableDefinition* only if it is in a class context.

8(41) A particular *ModifierAttribute* must not mark a *VariableDefinition* more than once.

8.6 Namespace Definitions

8(42) A *NamespaceDefinition* must not appear in an interface context.

8.6.1 Namespace Attributes of Namespace Definitions

8(43) A *NamespaceAttribute* may mark a *NamespaceDefinition* only in a class context or in a global context.

8(44) The only *NamespaceAttribute* that may mark a *NamespaceDefinition* in a global context that is not nested by a *PackageDirective* is `internal`.

8(45) The only *NamespaceAttribute*s that may mark a *NamespaceDefinition* in a global context nested by a *PackageDirective* are `public` and `internal`.

8(46) At most one *NamespaceAttribute* may mark a *NamespaceDefinition*, and that *NamespaceAttribute* must not mark the *NamespaceDefinition* more than once.

8.6.2 Modifier Attributes of Namespace Definitions

8(47) No *ModifierAttribute* may mark a *NamespaceDefinition*.

8.7 Statements

8(48) A *Statement* must not appear in an interface context.

8.7.1 Super Statements

8(49) A *SuperStatement* must be nested by a constructor without crossing a *FunctionBody*.

8.7.2 Labeled Statements

8(50) The label of a *LabeledStatement* must not be the label of another *LabeledStatement* that nests it without crossing a *FunctionBody*.

8.7.3 Break Statements

8(51) A *BreakStatement* must be nested by a *WhileStatement*, a *DoStatement*, *ForStatement*, a *SwitchStatement*, or a *LabeledStatement* without crossing a *FunctionBody*.

8(52) A *BreakStatement* must carry a label if it is not nested by a *WhileStatement*, a *DoStatement*, a *ForStatement*, or a *SwitchStatement*.

8(53) The label of a *BreakStatement* must be the label of a *LabeledStatement* that nests it without crossing a *FunctionBody*.

8.7.4 Continue Statements

8(54) A *ContinueStatement* must be nested by a *WhileStatement*, a *DoStatement*, or a *ForStatement* without crossing a *FunctionBody*.

8(55) The label of a *ContinueStatement* must be the label of a *LabeledStatement* that nests it without crossing a *FunctionBody*.

8.7.5 Return Statements

8(56) A *ReturnStatement* must be nested by a *FunctionExpression* or *FunctionDefinition*.

8(57) A *FunctionExpression* or *FunctionDefinition* must have a return expression if its result type is not `void` or `*`.

8(58) A *FunctionExpression* or *FunctionDefinition* must not have a return expression if its result type is `void`.

8.8 Expressions

8.8.1 The Keyword `this`

8(59) The keyword `this` must be nested by a *FunctionExpression*, a *VariableDefinition* or *FunctionDefinition* that is not marked `static`, or by a *Statement* in a global context.

8.8.2 Super Expressions

8(60) A *SuperExpression* must be nested by a *VariableDefinition* or *FunctionDefinition* that is in a class context and is not marked `static`, without crossing a *FunctionBody*.