Dart Programming Language Grammar

Version GIT-HEAD

2018-01-21

variableDeclaration:
declaredIdentifier (‘, ’ identifier)*
;

declaredIdentifier:
metadata finalConstVarOrType identifier
;

finalConstVarOrType:
final type? |
const type? |
varOrType
;

varOrType:
var |
type
;

initializedVariableDeclaration:
declaredIdentifier (‘=’ expression)? (‘, ’ initializedIdentifier)*
;

initializedIdentifier:
identifier (‘=’ expression)?
;

initializedIdentifierList:
initializedIdentifier (‘, ’ initializedIdentifier)*
;

functionSignature:
metadata returnType? identifier formalParameterPart
;

formalParameterPart:
typeParameters? formalParameterList
;

returnType:
void |
type
;

functionBody:
  async? '=>' expression ';' |
  (**async** | **async*** | **sync***)? block
;

block:
  '{' statements '}'
;
formalParameterList:
  '(' ')' |
  '(' normalFormalParameters ','? ')' |
  '(' normalFormalParameters ',' optionalFormalParameters ')' |
  '(' optionalFormalParameters ')'
;

normalFormalParameters:
  normalFormalParameter (',' normalFormalParameter)*
;

optionalFormalParameters:
  optionalPositionalFormalParameters |
  namedFormalParameters
;

optionalPositionalFormalParameters:
  '[' defaultFormalParameter (',' defaultFormalParameter)* ','? ']'
;

namedFormalParameters:
  '{' defaultNamedParameter (',' defaultNamedParameter)* ','? '}'
;
normalFormalParameter:
  functionFormalParameter |
  fieldFormalParameter |
  simpleFormalParameter
;

functionFormalParameter:
  metadata **covariant**? returnType? identifier
      formalParameterPart
;

simpleFormalParameter:
  metadata **covariant**? finalConstVarOrType? identifier |

;

fieldFormalParameter:
metadata finalConstVarOrType? **this** '.' identifier
formalParameterPart?
;
defaultFormalParameter:
normalFormalParameter ('=' expression)?
;

defaultNamedParameter:
normalFormalParameter ('=' expression)? |
normalFormalParameter (':' expression)?
;
classDefinition:
metadata **abstract**? **class** identifier typeParameters?
(superclass mixins?)? interfaces?
'{' (metadata classMemberDefinition)* '}' |
metadata **abstract**? **class** mixinApplicationClass
;

mixins:
with typeList
;

classMemberDefinition:
declaration ';' |
methodSignature functionBody
;

methodSignature:
constructorSignature initializers? |
factoryConstructorSignature |
static? functionSignature |
static? getterSignature |
static? setterSignature |
operatorSignature
;

declaration:
constantConstructorSignature (redirection | initializers)? |
constructorSignature (redirection | initializers)? |
external constantConstructorSignature |

external constructorSignature |
((**external static**?))? getterSignature |
((**external static**?))? setterSignature |
external? operatorSignature |
((**external static**?))? functionSignature |
static (**final** | **const**) type? staticFinalDeclarationList |
final type? initializedIdentifierList |
(**static** | **covariant**)? (**var** | type) initializedIdentifierList
;

staticFinalDeclarationList:
staticFinalDeclaration (',' staticFinalDeclaration)*
;

staticFinalDeclaration:
identifier '=' expression
;

operatorSignature:
returnType? **operator** operator formalParameterList
;

operator:
'~' |
binaryOperator |
'[]' |
'[]='
;

binaryOperator:
multiplicativeOperator |
additiveOperator |
shiftOperator |
relationalOperator |
'==' |
bitwiseOperator
;

getterSignature:
returnType? **get** identifier
;

setterSignature:
returnType? **set** identifier formalParameterList
;

constructorSignature:
identifier ('.' identifier)? formalParameterList
;

redirection:
  ‘:’ **this** (‘.’ identifier)? arguments
;
initializers:
  ‘:’ initializerListEntry (‘,’ initializerListEntry)*
;

initializerListEntry:
  super arguments |
  super ‘.’ identifier arguments |
  fieldInitializer |
  assertion
;

fieldInitializer:
  (**this** ‘.’)? identifier ‘=’ conditionalExpression cascadeSection*
;
factoryConstructorSignature:
  factory identifier (‘.’ identifier)? formalParameterList
;
redirectingFactoryConstructorSignature:
  const? **factory** identifier (‘.’ identifier)? formalParameterList
      ‘=’ type (‘.’ identifier)?
;
constantConstructorSignature:
  const qualified formalParameterList
;
superclass:
  extends type
;
interfaces:
  implements typeList
;
mixinApplicationClass:
  identifier typeParameters? ‘=’ mixinApplication ‘;’
;

mixinApplication:
  type mixins interfaces?
;
enumType:
  metadata **enum** id ‘{’ id [‘,’ id]* [‘,’] ‘}’
;
typeParameter:
  metadata identifier (**extends** type)?

;

typeParameters:
‘<’ typeParameter (‘,’ typeParameter)* ‘>’
;
metadata:
(‘@’ qualified (‘.’ identifier)? (arguments)?)*
;
expression:
assignableExpression assignmentOperator expression |
conditionalExpression cascadeSection* |
throwExpression
;

expressionWithoutCascade:
assignableExpression assignmentOperator
expressionWithoutCascade |
conditionalExpression |
throwExpressionWithoutCascade
;

expressionList:
expression (‘,’ expression)*
;
primary:
thisExpression |
super unconditionalAssignableSelector |
functionExpression |
literal |
identifier |
newExpression |
constObjectExpression |
‘(’ expression ‘)’
;
literal:
nullLiteral |
booleanLiteral |
numericLiteral |
stringLiteral |
symbolLiteral |
mapLiteral |
listLiteral
;
nullLiteral:
null

```
;
numericLiteral:
  NUMBER |
  HEX_NUMBER
;

NUMBER:
  DIGIT+ ('.' DIGIT+)? EXPONENT? |
  '.' DIGIT+ EXPONENT?
;

EXPONENT:
  ('e' | 'E') ('+' | '-')? DIGIT+
;

HEX_NUMBER:
  '0x' HEX_DIGIT+ |
  '0X' HEX_DIGIT+
;

HEX_DIGIT:
  'a'..'f' |
  'A'..'F' |
  DIGIT
;
booleanLiteral:
  true |
  false
;
stringLiteral:
  (multilineString | singleLineString)+
;
singleLineString:
  '"' stringContentDQ* '"' |
  '' stringContentSQ* '' |
  'r'' (~( '' | NEWLINE ))* '' |
  'r"' (~( '"' | NEWLINE ))* '"'
;
multilineString:
  '"""' stringContentTDQ* '"""' |
  '''' stringContentTSQ* '''' |
  'r"""' (~ '"""')* '"""' |
  'r'''' (~ '''')* ''''
;
```

ESCAPE_SEQUENCE:
'\n' |
'\r' |
'\f' |
'\b' |
'\t' |
'\v' |
'\x' HEX_DIGIT HEX_DIGIT |
'\u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT |
'\u{' HEX_DIGIT_SEQUENCE '}'
;

HEX_DIGIT_SEQUENCE:
HEX_DIGIT HEX_DIGIT? HEX_DIGIT?
HEX_DIGIT? HEX_DIGIT? HEX_DIGIT?
;

stringContentDQ:
~('\' | '"' | '$' | NEWLINE) |
'\' ~(NEWLINE) |
stringInterpolation
;

stringContentSQ:
~('\' | ''' | '$' | NEWLINE) |
'\' ~(NEWLINE) |
stringInterpolation
;

stringContentTDQ:
~('\' | '"""' | '$') |
stringInterpolation
;

stringContentTSQ:
~('\' | ''''' | '$') |
stringInterpolation
;

NEWLINE:
\n |
\r |
\r\n
;

stringInterpolation:

```
  '$' IDENTIFIER_NO_DOLLAR |
  '${' expression '}'
 ;
symbolLiteral:
  '#' (operator | (identifier ('.' identifier)*))
 ;
listLiteral:
  const? typeArguments? '[' (expressionList ','?)? ']'
 ;
mapLiteral:
  const? typeArguments?
      '{' (mapLiteralEntry (',' mapLiteralEntry)* ','?)? '}'
 ;

mapLiteralEntry:
  expression ':' expression
 ;
throwExpression:
  throw expression
 ;

throwExpressionWithoutCascade:
  throw expressionWithoutCascade
 ;
functionExpression:
  formalParameterPart functionBody
 ;
thisExpression:
  this
 ;
newExpression:
  new type ('.' identifier)? arguments
 ;
constObjectExpression:
  const type ('.' identifier)? arguments
 ;
arguments:
  '(' (argumentList ','?)? ')'
 ;

argumentList:
  namedArgument (',' namedArgument)* |
  expressionList (',' namedArgument)*
 ;
```

namedArgument:
label expression
;
cascadeSection:
'..' (cascadeSelector argumentPart*)
(assignableSelector argumentPart*)*
(assignmentOperator expressionWithoutCascade)?
;

cascadeSelector:
'[' expression ']' |
identifier
;

argumentPart:
typeArguments? arguments
;
assignmentOperator:
'=' |
compoundAssignmentOperator
;
compoundAssignmentOperator:
'*=' |
'/=' |
'~/=' |
'%=' |
'+=' |
'-=' |
'<<=' |
'>>=' |
'>>>=' |
'&=' |
'^=' |
'|=' |
'??='
;
conditionalExpression:
ifNullExpression
('?' expressionWithoutCascade ':' expressionWithoutCascade)?
;
ifNullExpression:
logicalOrExpression ('??' logicalOrExpression)*
;
logicalOrExpression:
logicalAndExpression ('||' logicalAndExpression)*

```
;

logicalAndExpression:
  equalityExpression ('&&' equalityExpression)*
;
equalityExpression:
  relationalExpression (equalityOperator relationalExpression)? |
  super equalityOperator relationalExpression
;

equalityOperator:
  '==' |
  '!='
;
relationalExpression:
  bitwiseOrExpression (typeTest | typeCast |
      relationalOperator bitwiseOrExpression)? |
  super relationalOperator bitwiseOrExpression
;

relationalOperator:
  '>=' |
  '>' |
  '<=' |
  '<'
;
bitwiseOrExpression:
  bitwiseXorExpression ('|' bitwiseXorExpression)* |
  super ('|' bitwiseXorExpression)+
;

bitwiseXorExpression:
  bitwiseAndExpression ('^' bitwiseAndExpression)* |
  super ('^' bitwiseAndExpression)+
;

bitwiseAndExpression:
  shiftExpression ('&' shiftExpression)* |
  super ('&' shiftExpression)+
;

bitwiseOperator:
  '&' |
  '^' |
```

```
  '|'
;
shiftExpression:
  additiveExpression (shiftOperator additiveExpression)* |
  super (shiftOperator additiveExpression)+
;

shiftOperator:
  '<<' |
  '>>' |
  '>>>'
;
additiveExpression:
  multiplicativeExpression
      (additiveOperator multiplicativeExpression)* |
  super (additiveOperator multiplicativeExpression)+
;

additiveOperator:
  '+' |
  '-'
;
multiplicativeExpression:
  unaryExpression (multiplicativeOperator unaryExpression)* |
  super (multiplicativeOperator unaryExpression)+
;

multiplicativeOperator:
  '*' |
  '/' |
  '%' |
  '~/'
;
unaryExpression:
  prefixOperator unaryExpression |
  awaitExpression |
  postfixExpression |
  (minusOperator | tildeOperator) super |
  incrementOperator assignableExpression
;

prefixOperator:
  minusOperator |
  negationOperator |
  tildeOperator
```

;

minusOperator:
'-'
;

negationOperator:
'!'
;

tildeOperator:
'~'
;
awaitExpression:
await unaryExpression
;
postfixExpression:
assignableExpression postfixOperator |
primary selector*
;

postfixOperator:
incrementOperator
;

selector:
assignableSelector |
argumentPart
;

incrementOperator:
'++' |
'--'
;
assignableExpression:
primary (argumentPart* assignableSelector)+ |
super unconditionalAssignableSelector |
identifier
;

unconditionalAssignableSelector:
'[' expression ']' |
'.' identifier

;

assignableSelector:
unconditionalAssignableSelector |
'?.' identifier
;
identifier:
IDENTIFIER
;

IDENTIFIER_NO_DOLLAR:
IDENTIFIER_START_NO_DOLLAR
IDENTIFIER_PART_NO_DOLLAR*
;

IDENTIFIER:
IDENTIFIER_START IDENTIFIER_PART*
;

BUILT_IN_IDENTIFIER:
abstract |
as |
covariant |
deferred |
dynamic |
export |
external |
factory |
get |
implements |
import |
library |
operator |
part |
set |
static |
typedef
;

IDENTIFIER_START:
IDENTIFIER_START_NO_DOLLAR |
'$'
;

IDENTIFIER_START_NO_DOLLAR:
LETTER |
'_'
;

IDENTIFIER_PART_NO_DOLLAR:
IDENTIFIER_START_NO_DOLLAR |
DIGIT
;

IDENTIFIER_PART:
IDENTIFIER_START |
DIGIT
;

qualified:
identifier ('.' identifier)?
;
typeTest:
isOperator type
;

isOperator:
is '!'?
;
typeCast:
asOperator type
;

asOperator:
as
;
statements:
statement*
;

statement:
label* nonLabelledStatement
;

nonLabelledStatement:
block |
localVariableDeclaration |
forStatement |

```
  whileStatement |
  doStatement |
  switchStatement |
  ifStatement |
  rethrowStatement |
  tryStatement |
  breakStatement |
  continueStatement |
  returnStatement |
  yieldStatement |
  yieldEachStatement |
  expressionStatement |
  assertStatement |
  localFunctionDeclaration
 ;
expressionStatement:
  expression? ';'
 ;
localVariableDeclaration:
  initializedVariableDeclaration ';'
 ;
localFunctionDeclaration:
  functionSignature functionBody
 ;
ifStatement:
  if '(' expression ')' statement ( else statement)?
 ;
forStatement:
  await? for '(' forLoopParts ')' statement
 ;

forLoopParts:
  forInitializerStatement expression? ';' expressionList? |
  declaredIdentifier in expression |
  identifier in expression
 ;

forInitializerStatement:
  localVariableDeclaration |
  expression? ';'
 ;
whileStatement:
  while '(' expression ')' statement
 ;
doStatement:
```

do statement **while** '(' expression ')' ';'
;
switchStatement:
switch '(' expression ')' '{' switchCase* defaultCase? '}'
;

switchCase:
label* **case** expression ':' statements
;

defaultCase:
label* **default** ':' statements
;
rethrowStatement:
rethrow ';'
;
tryStatement:
try block (onPart+ finallyPart? | finallyPart)
;

onPart:
catchPart block |
on type catchPart? block
;

catchPart:
catch '(' identifier (',' identifier)? ')'
;

finallyPart:
finally block
;
returnStatement:
return expression? ';'
;
label:
identifier ':'
;
breakStatement:
break identifier? ';'
;
continueStatement:
continue identifier? ';'
;

yieldStatement:
  yield expression ';'
;
yieldEachStatement:
  yield* expression ';'
;
assertStatement:
  assertion ';'
;

assertion:
  assert '(' expression (',' expression)? ','? ')'
;
topLevelDefinition:
  classDefinition |
  enumType |
  typeAlias |
  external? functionSignature ';' |
  external? getterSignature ';' |
  external? setterSignature ';' |
  functionSignature functionBody |
  returnType? **get** identifier functionBody |
  returnType? **set** identifier formalParameterList functionBody |
  (**final** | **const**) type? staticFinalDeclarationList ';' |
  variableDeclaration ';'
;

getOrSet:
  get |
  set
;

libraryDefinition:
  scriptTag? libraryName? importOrExport* partDirective*
      topLevelDefinition*
;

scriptTag:
  '#!' (~NEWLINE)* NEWLINE
;

libraryName:
  metadata **library** identifier ('.' identifier)* ';'
;

importOrExport:
  libraryImport |
  libraryExport
;
libraryImport:
  metadata importSpecification
;

importSpecification:
  import uri (**as** identifier)? combinator* ';' |
  import uri **deferred as** identifier combinator* ';'
;

combinator:
  show identifierList |
  hide identifierList
;

identifierList:
  identifier (, identifier)*
;
libraryExport:
  metadata **export** uri combinator* ';'
;
partDirective:
  metadata **part** uri ';'
;

partHeader:
  metadata **part of** identifier ('.' identifier)* ';'
;

partDeclaration:
  partHeader topLevelDefinition* EOF
;
uri:
  stringLiteral
;
type:
  typeName typeArguments?
;

typeName:
  qualified
;

typeArguments:
  '<' typeList '>'
;

typeList:
  type (',' type)*
;
typeAlias:
  metadata **typedef** typeAliasBody
;

typeAliasBody:
  functionTypeAlias
;

functionTypeAlias:
  functionPrefix typeParameters? formalParameterList ';'
;

functionPrefix:
  returnType? identifier
;
LETTER:
  'a' .. 'z' |
  'A' ..'Z'
;

DIGIT:
  '0' .. '9'
;

WHITESPACE:
  ('\t' | ' ' | NEWLINE)+
;
SINGLE_LINE_COMMENT:
  '//' ~(NEWLINE)* (NEWLINE)?
;

MULTI_LINE_COMMENT:
  '/*' (MULTI_LINE_COMMENT | ~ '*/')* '*/'

;