

INCLUDED	item is used
MODIFIED	item is used and modified
RESTRICTED	item is used but not in this context
EXCLUDED	item is not used
IGNORED	item is ignored

```
/**
 * Kotlin syntax grammar in ANTLR4 notation
 */
```

```
parser grammar KotlinParser;
```

```
options { tokenVocab = KotlinLexer; }
```

```
// SECTION: general
```

```
kotlinFile
    : shebangLine? NL* fileAnnotation* packageHeader
    importList topLevelObject* EOF
    ;
```

```
script
    : shebangLine? NL* fileAnnotation* packageHeader
    importList (statement semi)* EOF
    ;
```

```
shebangLine
    : ShebangLine NL+
    ;
```

```
fileAnnotation
    : (AT_NO_WS | AT_PRE_WS) FILE NL* COLON NL*
    (LSQUARE unescapedAnnotation+ RSQUARE | unescapedAnnotation) NL*
    ;
```

```
packageHeader
    : (PACKAGE identifier semi)?
    ;
```

```
importList
    : importHeader*
    ;
```

```
importHeader
    : IMPORT identifier (DOT MULT | importAlias)? semi?
    ;
```

```
importAlias
    : AS simpleIdentifier
    ;
```

```
topLevelObject
    : declaration semis?
    ;
```

```
typeAlias
: modifiers? TYPE_ALIAS NL* simpleIdentifier
(NL* typeParameters)? NL* ASSIGNMENT NL* type
;
```

```
declaration
: classDeclaration
| objectDeclaration
| functionDeclaration
| propertyDeclaration
| typeAlias
;
```

// SECTION: classes

```
classDeclaration
: modifiers? (CLASS | (FUN NL*)? INTERFACE) NL* simpleIdentifier
(NL* typeParameters)? (NL* primaryConstructor)?
(NL* COLON NL* delegationSpecifiers)?
(NL* typeConstraints)?
(NL* classBody | NL* enumClassBody)?
;
```

```
primaryConstructor
: (modifiers? CONSTRUCTOR NL*)? classParameters
;
```

```
classBody
: LCURL NL* classMemberDeclarations NL* RCURL
;
```

```
classParameters
: LPAREN NL* (classParameter (NL* COMMA NL* classParameter)*
(NL* COMMA)? )? NL* RPAREN
;
```

```
classParameter
: modifiers? (VAL | VAR)? NL* simpleIdentifier COLON NL* type
(NL* ASSIGNMENT NL* expression)?
;
```

```
delegationSpecifiers
: annotatedDelegationSpecifier (NL* COMMA NL* annotatedDelegationSpecifier)*
;
```

```
delegationSpecifier
: constructorInvocation
| explicitDelegation
| userType
| functionType
;
```

```
constructorInvocation
: userType valueArguments
;
```

```
annotatedDelegationSpecifier
: annotation* NL* delegationSpecifier
;
```

```
explicitDelegation
: (userType | functionType) NL* BY NL* expression
;
```

```
typeParameters
: LANGLE NL* typeParameter (NL* COMMA NL* typeParameter)*
(NL* COMMA)? NL* RANGLE
;
```

```
typeParameter
: typeParameterModifiers? NL* simpleIdentifier (NL* COLON NL* type)?
;
```

```
typeConstraints
: WHERE NL* typeConstraint (NL* COMMA NL* typeConstraint)*
;
```

```
typeConstraint
: annotation* simpleIdentifier NL* COLON NL* type
;
```

// SECTION: classMembers

```
classMemberDeclarations
: (classMemberDeclaration semis?)*
;
```

```
classMemberDeclaration
: declaration
| companionObject
| anonymousInitializer
| secondaryConstructor
;
```

```
anonymousInitializer
: INIT NL* block
;
```

```
companionObject
: modifiers? COMPANION NL* OBJECT
(NL* simpleIdentifier)?
(NL* COLON NL* delegationSpecifiers)?
(NL* classBody)?
;
```

```
functionValueParameters
: LPAREN NL* (functionValueParameter (NL* COMMA NL* functionValueParameter)*
(NL* COMMA)?)? NL* RPAREN
;
```

```
functionValueParameter
: parameterModifiers? parameter (NL* ASSIGNMENT NL* expression)?
;
```

```
functionDeclaration
: modifiers?
  FUN (NL* typeParameters)? (NL* receiverType NL* DOT)? NL* simpleIdentifier
  NL* functionValueParameters
  (NL* COLON NL* type)?
  (NL* typeConstraints)?
  (NL* functionBody)?
;
```

```
functionBody
: block
| ASSIGNMENT NL* expression
;
```

```
variableDeclaration
: annotation* NL* simpleIdentifier (NL* COLON NL* type)?
;
```

```
multiVariableDeclaration
: LPAREN NL* variableDeclaration (NL* COMMA NL* variableDeclaration)* (NL*
COMMA)? NL* RPAREN
;
```

```
propertyDeclaration
: modifiers? (VAL | VAR)
  (NL* typeParameters)?
  (NL* receiverType NL* DOT)?
  (NL* (multiVariableDeclaration | variableDeclaration))
  (NL* typeConstraints)?
  (NL* (ASSIGNMENT NL* expression | propertyDelegate))?
  (NL+ SEMICOLON)? NL*
  (getter? (NL* semi? setter)? | setter? (NL* semi? getter)?)
;
```

```
propertyDelegate
: BY NL* expression
;
```

```
getter
: modifiers? GET
| modifiers? GET NL* LPAREN NL* RPAREN (NL* COLON NL* type)? NL* functionBody
;
```

```
setter
: modifiers? SET
| modifiers? SET NL* LPAREN NL* parameterWithOptionalType
  (NL* COMMA)? NL* RPAREN (NL* COLON NL* type)? NL* functionBody
;
```

```
parametersWithOptionalType
: LPAREN NL*
```

```
(parameterWithOptionalType (NL* COMMA NL* parameterWithOptionalType)*  
(NL* COMMA)?)? NL* RPAREN  
;
```

```
parameterWithOptionalType  
: parameterModifiers? simpleIdentifier NL* (COLON NL* type)?  
;
```

```
parameter  
: simpleIdentifier NL* COLON NL* type  
;
```

```
objectDeclaration  
: modifiers? OBJECT  
NL* simpleIdentifier  
(NL* COLON NL* delegationSpecifiers)?  
(NL* classBody)?  
;
```

```
secondaryConstructor  
: modifiers? CONSTRUCTOR NL* functionValueParameters  
(NL* COLON NL* constructorDelegationCall)? NL* block?  
;
```

```
constructorDelegationCall  
: THIS NL* valueArguments  
| SUPER NL* valueArguments  
;
```

// SECTION: enumClasses

```
enumClassBody  
: LCURL NL* enumEntries? (NL* SEMICOLON NL* classMemberDeclarations)? NL* RCURL  
;
```

```
enumEntries  
: enumEntry (NL* COMMA NL* enumEntry)* NL* COMMA?  
;
```

```
enumEntry  
: (modifiers NL*)? simpleIdentifier (NL* valueArguments)? (NL* classBody)?  
;
```

// SECTION: types

```
type  
: typeModifiers?  
( parenthesizedType  
| nullableType  
| typeReference  
| functionType)  
;
```

```
typeReference  
: userType
```

```
| DYNAMIC  
;
```

```
nullableType  
: (typeReference | parenthesizedType) NL* quest+  
;
```

```
quest  
: QUEST_NO_WS  
| QUEST_WS  
;
```

```
userType  
: simpleUserType (NL* DOT NL* simpleUserType)*  
;
```

```
simpleUserType  
: simpleIdentifier (NL* typeArguments)?  
;
```

```
typeProjection  
: typeProjectionModifiers? type | MULT  
;
```

```
typeProjectionModifiers  
: typeProjectionModifier+  
;
```

```
typeProjectionModifier  
: varianceModifier NL*  
| annotation  
;
```

```
functionType  
: (receiverType NL* DOT NL*)? functionTypeParameters NL* ARROW NL* type  
;
```

```
functionTypeParameters  
: LPAREN NL* (parameter | type)? (NL* COMMA NL* (parameter | type))*  
(NL* COMMA)? NL* RPAREN  
;
```

```
parenthesizedType  
: LPAREN NL* type NL* RPAREN  
;
```

```
receiverType  
: typeModifiers?  
( parenthesizedType  
| nullableType  
| typeReference)  
;
```

```
parenthesizedUserType  
: LPAREN NL* userType NL* RPAREN
```

```
| LPAREN NL* parenthesizedUserType NL* RPAREN  
;
```

// SECTION: statements

```
statements  
: (statement (semis statement)*)? semis?  
;
```

```
statement  
: ((label | annotation)*  
  ( declaration  
  | assignment  
  | loopStatement  
  | expression))  
;
```

```
label  
: simpleIdentifier (AT_NO_WS | AT_POST_WS) NL*  
;
```

```
controlStructureBody  
: block  
| statement  
;
```

```
block  
: LCURL NL* statements NL* RCURL  
;
```

```
loopStatement  
: forStatement  
| whileStatement  
| doWhileStatement  
;
```

```
forStatement  
: FOR NL* LPAREN annotation* (variableDeclaration | multiVariableDeclaration)  
  IN expression RPAREN NL* controlStructureBody?  
;
```

```
whileStatement  
: WHILE NL* LPAREN expression RPAREN NL* controlStructureBody  
| WHILE NL* LPAREN expression RPAREN NL* SEMICOLON  
;
```

```
doWhileStatement  
: DO NL* controlStructureBody? NL* WHILE NL* LPAREN expression RPAREN  
;
```

```
assignment  
: directlyAssignableExpression ASSIGNMENT NL* expression  
| assignableExpression assignmentAndOperator NL* expression  
;
```

```
semi
: (SEMICOLON | NL) NL*
| EOF;
```

```
semis
: (SEMICOLON | NL)+
| EOF
;
```

// SECTION: expressions

```
expression
: disjunction
;
```

```
disjunction
: conjunction (NL* DISJ NL* conjunction)*
;
```

```
conjunction
: equality (NL* CONJ NL* equality)*
;
```

```
equality
: comparison (equalityOperator NL* comparison)*
;
```

```
comparison
: infixOperation (comparisonOperator NL* infixOperation)?
;
```

```
infixOperation
: elvisExpression (inOperator NL* elvisExpression | isOperator NL* type)*
;
```

```
elvisExpression
: infixFunctionCall (NL* elvis NL* infixFunctionCall)*
;
```

```
elvis
: QUEST_NO_WS COLON
;
```

```
infixFunctionCall
: rangeExpression (simpleIdentifier NL* rangeExpression)*
;
```

```
rangeExpression
: additiveExpression (RANGE NL* additiveExpression)*
;
```

```
additiveExpression
: multiplicativeExpression (additiveOperator NL* multiplicativeExpression)*
;
```



```
multiplicativeExpression
: asExpression (multiplicativeOperator NL* asExpression)*
;
```

```
asExpression
: comparisonWithLiteralRightSide (NL* asOperator NL* type)?
;
```

```
comparisonWithLiteralRightSide
: prefixUnaryExpression (NL* LANGLE NL* literalConstant NL* RANGLE NL*
(expression | parenthesizedExpression))*
;
```

```
prefixUnaryExpression
: unaryPrefix* postfixUnaryExpression
;
```

```
unaryPrefix
: annotation
| label
| prefixUnaryOperator NL*
;
```

```
postfixUnaryExpression
: primaryExpression
| primaryExpression postfixUnarySuffix+
;
```

```
postfixUnarySuffix
: postfixUnaryOperator
| typeArguments
| callSuffix
| indexingSuffix
| navigationSuffix
;
```

```
directlyAssignableExpression
: postfixUnaryExpression assignableSuffix
| simpleIdentifier
| parenthesizedDirectlyAssignableExpression
;
```

```
parenthesizedDirectlyAssignableExpression
: LPAREN NL* directlyAssignableExpression NL* RPAREN
;
```

```
assignableExpression
: prefixUnaryExpression | parenthesizedAssignableExpression
;
```

```
parenthesizedAssignableExpression
: LPAREN NL* assignableExpression NL* RPAREN
;
```

```
assignableSuffix
```

```
: typeArguments
| indexingSuffix
| navigationSuffix
;
```

```
indexingSuffix
: LSQUARE NL* expression (NL* COMMA NL* expression)* (NL* COMMA)? NL* RSQUARE
;
```

```
navigationSuffix
: NL* memberAccessOperator NL*
  (simpleIdentifier | parenthesizedExpression | CLASS)
;
```

```
callSuffix
: typeArguments? valueArguments? annotatedLambda
| typeArguments? valueArguments
;
```

```
annotatedLambda
: annotation* label? NL* lambdaLiteral
;
```

```
typeArguments
: LANGLE NL* typeProjection (NL* COMMA NL* typeProjection)*
  (NL* COMMA)? NL* RANGLE
;
```

```
valueArguments
: LPAREN NL* RPAREN
| LPAREN NL* valueArgument (NL* COMMA NL* valueArgument)*
  (NL* COMMA)? NL* RPAREN
;
```

```
valueArgument
: annotation? NL* (simpleIdentifier NL* ASSIGNMENT NL*)? MULT? NL* expression
;
```

```
primaryExpression
: parenthesizedExpression
| simpleIdentifier
| literalConstant
| stringLiteral
| callableReference
| functionLiteral
| objectLiteral
| collectionLiteral
| thisExpression
| superExpression
| ifExpression
| whenExpression
| tryExpression
| jumpExpression
;
```

```
parenthesizedExpression
: LPAREN NL* expression NL* RPAREN
;
```

```
collectionLiteral
: LSQUARE NL* expression (NL* COMMA NL* expression)* (NL* COMMA)? NL* RSQUARE
| LSQUARE NL* RSQUARE
;
```

```
literalConstant
: BooleanLiteral
| IntegerLiteral
| HexLiteral
| BinLiteral
| CharacterLiteral
| RealLiteral
| NullLiteral
| LongLiteral
| UnsignedLiteral
;
```

```
stringLiteral
: lineStringLiteral
| multiLineStringLiteral
;
```

```
lineStringLiteral
: QUOTE_OPEN (lineStringContent | lineStringExpression)* QUOTE_CLOSE
;
```

```
multiLineStringLiteral
: TRIPLE_QUOTE_OPEN
(multiLineStringContent | multiLineStringExpression | MultiLineStringQuote)*
TRIPLE_QUOTE_CLOSE
;
```

```
lineStringContent
: LineStrText
| LineStrEscapedChar
| LineStrRef
;
```

```
lineStringExpression
: LineStrExprStart expression RCURL
;
```

```
multiLineStringContent
: MultiLineStrText
| MultiLineStringQuote
| MultiLineStrRef
;
```

```
multiLineStringExpression
: MultiLineStrExprStart NL* expression NL* RCURL
;
```

```
lambdaLiteral
  : LCURL NL* statements NL* RCURL
  | LCURL NL* lambdaParameters? NL* ARROW NL* statements NL* RCURL
  ;
```

```
lambdaParameters
  : lambdaParameter (NL* COMMA NL* lambdaParameter)* (NL* COMMA)?
  ;
```

```
lambdaParameter
  : variableDeclaration
  | multiVariableDeclaration (NL* COLON NL* type)?
  ;
```

```
anonymousFunction
  : FUN
  (NL* type NL* DOT)?
  NL* parametersWithOptionalType
  (NL* COLON NL* type)?
  (NL* typeConstraints)?
  (NL* functionBody)?
  ;
```

```
functionLiteral
  : lambdaLiteral
  | anonymousFunction
  ;
```

```
objectLiteral
  : OBJECT NL* COLON NL* delegationSpecifiers NL* classBody
  | OBJECT NL* classBody
  ;
```

```
thisExpression
  : THIS
  | THIS_AT
  ;
```

```
superExpression
  : SUPER (LANGLE NL* type NL* RANGLE)? (AT_NO_WS simpleIdentifier)?
  | SUPER_AT
  ;
```

```
ifExpression
  : IF NL* LPAREN NL* expression NL* RPAREN NL*
  (controlStructureBody | SEMICOLON)
  | IF NL* LPAREN NL* expression NL* RPAREN NL*
  controlStructureBody? NL* SEMICOLON? NL* ELSE NL*
  (controlStructureBody | SEMICOLON)
  ;
```

```
whenSubject
  : LPAREN (annotation* NL* VAL NL* variableDeclaration NL* ASSIGNMENT NL*)?
  expression RPAREN
```

```
;
```

```
whenExpression
```

```
: WHEN NL* whenSubject? NL* LCURL NL* (whenEntry NL*)* NL* RCURL  
;
```

```
whenEntry
```

```
: whenCondition (NL* COMMA NL* whenCondition)* (NL* COMMA)?  
  NL* ARROW NL* controlStructureBody semi?  
  | ELSE NL* ARROW NL* controlStructureBody semi?  
;
```

```
whenCondition
```

```
: expression  
  | rangeTest  
  | typeTest  
;
```

```
rangeTest
```

```
: inOperator NL* expression  
;
```

```
typeTest
```

```
: isOperator NL* type  
;
```

```
tryExpression
```

```
: TRY NL* block ((NL* catchBlock)+ (NL* finallyBlock)? | NL* finallyBlock)  
;
```

```
catchBlock
```

```
: CATCH NL* LPAREN annotation* simpleIdentifier COLON type  
  (NL* COMMA)? RPAREN NL* block  
;
```

```
finallyBlock
```

```
: FINALLY NL* block  
;
```

```
jumpExpression
```

```
: THROW NL* expression  
  | (RETURN | RETURN_AT) expression?  
  | CONTINUE | CONTINUE_AT  
  | BREAK | BREAK_AT  
;
```

```
callableReference
```

```
: (receiverType? NL* COLONCOLON NL* (simpleIdentifier | CLASS))  
;
```

```
assignmentAndOperator
```

```
: ADD_ASSIGNMENT  
  | SUB_ASSIGNMENT  
  | MULT_ASSIGNMENT  
  | DIV_ASSIGNMENT
```

```
    | MOD_ASSIGNMENT
    ;
```

```
equalityOperator
    : EXCL_EQ
    | EXCL_EQEQ
    | EQEQ
    | EQEQEQ
    ;
```

```
comparisonOperator
    : LANGLE
    | RANGLE
    | LE
    | GE
    ;
```

```
inOperator
    : IN | NOT_IN
    ;
```

```
isOperator
    : IS | NOT_IS
    ;
```

```
additiveOperator
    : ADD | SUB
    ;
```

```
multiplicativeOperator
    : MULT
    | DIV
    | MOD
    ;
```

```
asOperator
    : AS
    | AS_SAFE
    ;
```

```
prefixUnaryOperator
    : INCR
    | DECR
    | SUB
    | ADD
    | excl
    ;
```

```
postfixUnaryOperator
    : INCR
    | DECR
    | EXCL_NO_WS excl
    ;
```

```
excl
```

```
: EXCL_NO_WS  
| EXCL_WS  
;
```

```
memberAccessOperator  
: DOT | safeNav | COLONCOLON  
;
```

```
safeNav  
: QUEST_NO_WS DOT  
;
```

```
// SECTION: modifiers
```

```
modifiers  
: (annotation | modifier)+  
| ((AT_NO_WS | AT_PRE_WS) unescapedAnnotation NL* | modifier NL*)+  
;
```

```
parameterModifiers  
: (annotation | parameterModifier)+  
;
```

```
modifier  
: (classModifier  
| memberModifier  
| visibilityModifier  
| functionModifier  
| propertyModifier  
| inheritanceModifier  
| parameterModifier  
| platformModifier) NL*  
| ENUM  
| OVERRIDE  
| PUBLIC  
| PRIVATE  
| INTERNAL  
| PROTECTED  
| FINAL  
| OPEN  
;
```

```
typeModifiers  
: typeModifier+  
;
```

```
typeModifier  
: annotation | SUSPEND NL*  
;
```

```
classModifier  
: ENUM  
| SEALED  
| ANNOTATION  
| DATA
```

```
    | INNER  
    ;
```

```
memberModifier  
    : OVERRIDE  
    | LATEINIT  
    ;
```

```
visibilityModifier  
    : PUBLIC  
    | PRIVATE  
    | INTERNAL  
    | PROTECTED  
    ;
```

```
varianceModifier  
    : IN  
    | OUT  
    ;
```

```
typeParameterModifiers  
    : typeParameterModifier+  
    ;
```

```
typeParameterModifier  
    : reificationModifier NL*  
    | varianceModifier NL*  
    | annotation  
    ;
```

```
functionModifier  
    : TAILREC  
    | OPERATOR  
    | INFIX  
    | INLINE  
    | EXTERNAL  
    | SUSPEND  
    ;
```

```
propertyModifier  
    : CONST  
    ;
```

```
inheritanceModifier  
    : ABSTRACT  
    | FINAL  
    | OPEN  
    ;
```

```
parameterModifier  
    : VARARG  
    | NOINLINE  
    | CROSSINLINE  
    ;
```



```
reificationModifier
: REIFIED
;
```

```
platformModifier
: EXPECT
| ACTUAL
;
```

// SECTION: annotations

```
annotation
: (singleAnnotation | multiAnnotation) NL*
;
```

```
singleAnnotation
: annotationUseSiteTarget NL* unescapedAnnotation
| (AT_NO_WS | AT_PRE_WS) unescapedAnnotation
;
```

```
multiAnnotation
: annotationUseSiteTarget NL* LSQUARE unescapedAnnotation+ RSQUARE
| (AT_NO_WS | AT_PRE_WS) LSQUARE unescapedAnnotation+ RSQUARE
;
```

```
annotationUseSiteTarget
: (AT_NO_WS | AT_PRE_WS)
(FIELD | PROPERTY | GET | SET | RECEIVER | PARAM | SETPARAM | DELEGATE)
NL* COLON
;
```

```
unescapedAnnotation
: constructorInvocation
| userType
;
```

// SECTION: identifiers

```
simpleIdentifier
: Identifier
| ABSTRACT
| ANNOTATION
| BY
| CATCH
| COMPANION
| CONSTRUCTOR
| CROSSINLINE
| DATA
| DYNAMIC
| ENUM
| EXTERNAL
| FINAL
| FINALLY
| GET
| IMPORT
```

```
INFIX
INIT
INLINE
INNER
INTERNAL
LATEINIT
NOINLINE
OPEN
OPERATOR
OUT
OVERRIDE
PRIVATE
PROTECTED
PUBLIC
REIFIED
SEALED
TAILREC
SET
VARARG
WHERE
FIELD
PROPERTY
RECEIVER
PARAM
SETPARAM
DELEGATE
FILE
EXPECT
ACTUAL
CONST
SUSPEND
```

```
;
```

```
identifier
: simpleIdentifier (NL* DOT simpleIdentifier)*
;
```

```
/**
 * Kotlin lexical grammar in ANTLR4 notation
 */
```

```
lexer grammar KotlinLexer;
```

```
// SECTION: lexicalGeneral
```

```
ShebangLine
: '#' ~[\r\n]*
;
```

```
DelimitedComment
: '/*' ( DelimitedComment | . )*? '*/' -> channel(HIDDEN)
;
```

```
LineComment
: '//' ~[\r\n]* -> channel(HIDDEN)
```

```
;

```

```
WS
```

```
: [\u0020\u0009\u000C] -> channel(HIDDEN)
;
```

```
NL: '\n' | '\r' '\n?';
```

```
fragment Hidden: DelimitedComment | LineComment | WS;
```

```
// SECTION: separatorsAndOperations
```

```
RESERVED: '...';
```

```
DOT: '.';
```

```
COMMA: ',';
```

```
LPAREN: '(' -> pushMode(Inside);
```

```
RPAREN: ')';
```

```
LSQUARE: '[' -> pushMode(Inside);
```

```
RSQUARE: ']';
```

```
LCURL: '{' -> pushMode(DEFAULT_MODE);
```

```
RCURL: '}' { if (!_modeStack.isEmpty()) { popMode(); } };
```

```
MULT: '*';
```

```
MOD: '%';
```

```
DIV: '/';
```

```
ADD: '+';
```

```
SUB: '-';
```

```
INCR: '++';
```

```
DECR: '--';
```

```
CONJ: '&';
```

```
DISJ: '||';
```

```
EXCL_WS: '!' Hidden;
```

```
EXCL_NO_WS: '!';
```

```
COLON: ':';
```

```
SEMICOLON: ';';
```

```
ASSIGNMENT: '=';
```

```
ADD_ASSIGNMENT: '+=';
```

```
SUB_ASSIGNMENT: '-=';
```

```
MULT_ASSIGNMENT: '*=';
```

```
DIV_ASSIGNMENT: '/=';
```

```
MOD_ASSIGNMENT: '%=';
```

```
ARROW: '->';
```

```
DOUBLE_ARROW: '=>';
```

```
RANGE: '..';
```

```
COLONCOLON: '::';
```

```
DOUBLE_SEMICOLON: ';;';
```

```
HASH: '#';
```

```
AT_NO_WS: '@';
```

```
AT_POST_WS: '@' (Hidden | NL);
```

```
AT_PRE_WS: (Hidden | NL) '@';
```

```
AT_BOTH_WS: (Hidden | NL) '@' (Hidden | NL);
```

```
QUEST_WS: '?' Hidden;
```

```
QUEST_NO_WS: '?';
```

```
LANGLE: '<';
```

```
RANGLE: '>';
```

```
LE: '<=';
```

```
GE: '>=';
EXCL_EQ: '!=';
EXCL_EQEQ: '!==';
AS_SAFE: 'as?';
EQEQ: '==';
EQEQEQ: '===';
SINGLE_QUOTE: '\'';
```

```
// SECTION: keywords
```

```
RETURN_AT: 'return@' Identifier;
CONTINUE_AT: 'continue@' Identifier;
BREAK_AT: 'break@' Identifier;
```

```
THIS_AT: 'this@' Identifier;
SUPER_AT: 'super@' Identifier;
```

```
FILE: 'file';
FIELD: 'field';
PROPERTY: 'property';
GET: 'get';
SET: 'set';
RECEIVER: 'receiver';
PARAM: 'param';
SETPARAM: 'setparam';
DELEGATE: 'delegate';
```

```
PACKAGE: 'package';
IMPORT: 'import';
CLASS: 'class';
INTERFACE: 'interface';
FUN: 'fun';
OBJECT: 'object';
VAL: 'val';
VAR: 'var';
TYPE_ALIAS: 'typealias';
CONSTRUCTOR: 'constructor';
BY: 'by';
COMPANION: 'companion';
INIT: 'init';
THIS: 'this';
SUPER: 'super';
TYPEOF: 'typeof';
WHERE: 'where';
IF: 'if';
ELSE: 'else';
WHEN: 'when';
TRY: 'try';
CATCH: 'catch';
FINALLY: 'finally';
FOR: 'for';
DO: 'do';
WHILE: 'while';
THROW: 'throw';
RETURN: 'return';
```

```
CONTINUE: 'continue';
BREAK: 'break';
AS: 'as';
IS: 'is';
IN: 'in';
NOT_IS: '!is' (Hidden | NL);
NOT_IN: '!in' (Hidden | NL);
OUT: 'out';
DYNAMIC: 'dynamic';
```

// SECTION: lexicalModifiers

```
PUBLIC: 'public';
PRIVATE: 'private';
PROTECTED: 'protected';
INTERNAL: 'internal';
ENUM: 'enum';
SEALED: 'sealed';
ANNOTATION: 'annotation';
DATA: 'data';
INNER: 'inner';
TAILREC: 'tailrec';
OPERATOR: 'operator';
INLINE: 'inline';
INFIX: 'infix';
EXTERNAL: 'external';
SUSPEND: 'suspend';
OVERRIDE: 'override';
ABSTRACT: 'abstract';
FINAL: 'final';
OPEN: 'open';
CONST: 'const';
LATEINIT: 'lateinit';
VARARG: 'vararg';
NOINLINE: 'noinline';
CROSSINLINE: 'crossinline';
REIFIED: 'reified';
EXPECT: 'expect';
ACTUAL: 'actual';
```

// SECTION: literals

```
fragment DecDigit: '0'..'9';
fragment DecDigitNoZero: '1'..'9';
fragment DecDigitOrSeparator: DecDigit | '_';
```

```
fragment DecDigits
    : DecDigit DecDigitOrSeparator* DecDigit
    | DecDigit
    ;
```

```
fragment DoubleExponent: [eE] [+~]? DecDigits;
```

```
RealLiteral
    : FloatLiteral
```

```
    | DoubleLiteral
    ;
```

```
FloatLiteral
    : DoubleLiteral [fF]
    | DecDigits [fF]
    ;
```

```
DoubleLiteral
    : DecDigits? '.' DecDigits DoubleExponent?
    | DecDigits DoubleExponent
    ;
```

```
IntegerLiteral
    : DecDigitNoZero DecDigitOrSeparator* DecDigit
    | DecDigit
    ;
```

```
fragment HexDigit: [0-9a-fA-F];
fragment HexDigitOrSeparator: HexDigit | '_';
```

```
HexLiteral
    : '0' [xX] HexDigit HexDigitOrSeparator* HexDigit
    | '0' [xX] HexDigit
    ;
```

```
fragment BinDigit: [01];
fragment BinDigitOrSeparator: BinDigit | '_';
```

```
BinLiteral
    : '0' [bB] BinDigit BinDigitOrSeparator* BinDigit
    | '0' [bB] BinDigit
    ;
```

```
UnsignedLiteral
    : (IntegerLiteral | HexLiteral | BinLiteral) [uU] 'L'?
    ;
```

```
LongLiteral
    : (IntegerLiteral | HexLiteral | BinLiteral) 'L'
    ;
```

```
BooleanLiteral: 'true' | 'false';
```

```
NullLiteral: 'null';
```

```
CharacterLiteral
    : '\'' (EscapeSeq | ~[\n\r'\\"']) '\''
    ;
```

```
// SECTION: lexicalIdentifiers
```

```
Identifier
    : [_a-zA-Z] [_a-zA-Z0-9]*
    ;
```

IdentifierOrSoftKey

: Identifier
ABSTRACT
ANNOTATION
BY
CATCH
COMPANION
CONSTRUCTOR
CROSSINLINE
DATA
DYNAMIC
ENUM
EXTERNAL
FINAL
FINALLY
IMPORT
INFIX
INIT
INLINE
INNER
INTERNAL
LATEINIT
NOINLINE
OPEN
OPERATOR
OUT
OVERRIDE
PRIVATE
PROTECTED
PUBLIC
REIFIED
SEALED
TAILREC
VARARG
WHERE
GET
SET
FIELD
PROPERTY
RECEIVER
PARAM
SETPARAM
DELEGATE
FILE
EXPECT
ACTUAL
CONST
SUSPEND

;

FieldIdentifier

: '\$' IdentifierOrSoftKey

;

```
fragment EscapedIdentifier
: '\\\' ('t' | 'b' | 'r' | 'n' | '\\\' | '\"' | '\\\' | '$')
;
```

```
fragment EscapeSeq
| EscapedIdentifier
;
```

// SECTION: strings

```
QUOTE_OPEN
: '\"' -> pushMode(LineStyle)
;
```

```
TRIPLE_QUOTE_OPEN
: '\"' -> pushMode(MultiLineStyle)
;
```

mode LineStyle;

```
QUOTE_CLOSE
: '\"' -> popMode
;
```

```
LineStyleRef
: FieldIdentifier
;
```

```
LineStyleText
: ~('\\\' | '\"' | '$')+ | '$'
;
```

```
LineStyleEscapedChar
: EscapedIdentifier
;
```

```
LineStyleExprStart
: '${' -> pushMode(DEFAULT_MODE)
;
```

mode MultiLineStyle;

```
TRIPLE_QUOTE_CLOSE
: MultiLineStyleQuote? '\"' -> popMode
;
```

```
MultiLineStyleQuote
: '\"'+
;
```

```
MultiLineStyleRef
: FieldIdentifier
;
```

```
MultiLineStyleText
```



```
      : ~('"' | '$')+ | '$'  
      ;
```

```
MultiLineStrExprStart  
      : '${' -> pushMode(DEFAULT_MODE)  
      ;
```