Kotlin language specification

Contents

Chapter 1

Glossary

w.r.t.:: with respect to

Chapter 2

Kotlin/Core

Introduction

Here be dragons...

Syntax

Grammar

Lexical grammar

Character classes

***LF*:** *<unicode character Line Feed U+000A>*

***CR*:**
<unicode character Carriage Return U+000D>

***WS*:**
<one of the following characters: SPACE U+0020, TAB U+0009, Form Feed U+000C>

***Underscore*:**
<unicode character Low Line U+005F>

***Letter*:**
<any unicode character from classes Ll, Lm, Lo, Lt, Lu or Nl>

***UnicodeDigit*:**
<any unicode character from class Nd>

***LineCharacter*:**
<any unicode character excluding LF and CR>

***BinaryDigit*:**
'0' | '1'

***DecimalDigit*:**
'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

***HexDigit*:**
DecimalDigit
| 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
| 'a' | 'b' | 'c' | 'd' | 'e' | 'f'

Keywords and operators

***Operator*:**
'.' | ',' | '(' | ')' | '[' | ']' | '@[' | '{' | '}' | '*' | '%' | '/' | '+'
| '-' | '++' | '--'
| '&&' | '||' | '!' | '!!' | ':' | ';' | '=' | '+=' | '-=' | '*=' | '/=' |
'%=' | '->' | '=>'
| '..' | '::' | '?::' | ';;' | '#' | '@' | '?' | '?:' | '<' | '>' | '\m' |
'>=' | '!=' | '!=='
| '==' | '===' | ''' | '"' | '"""'
***SoftKeyword*:**
'public' | 'private' | 'protected' | 'internal'
| 'enum' | 'sealed' | 'annotation' | 'data' | 'inner'
| 'tailrec' | 'operator' | 'inline' | 'infix' | 'external'
| 'suspend' | 'override' | 'abstract' | 'final' | 'open'
| 'const' | 'lateinit' | 'vararg' | 'noinline' | 'crossinline'
| 'reified' | 'expect' | 'actual'
***Keyword*:**
'package' | 'import' | 'class' | 'interface'
| 'fun' | 'object' | 'val' | 'var' | 'typealias'
| 'constructor' | 'by' | 'companion' | 'init'
| 'this' | 'super' | 'typeof' | 'where'
| 'if' | 'else' | 'when' | 'try' | 'catch'
| 'finally' | 'for' | 'do' | 'while' | 'throw'
| 'return' | 'continue' | 'break' | 'as'
| 'is' | 'in' | '!is' | '!in' | 'out'
| 'get' | 'set' | 'dynamic' | '@file'
| '@field' | '@property' | '@get' | '@set'
| '@receiver' | '@param' | '@setparam' | '@delegate'

Whitespace and comments

***NL*:** *LF* | *CR* [*LF*]

***ShebangLine*:**
'#!' {*LineCharacter*}

***LineComment*:**
'//' {*LineCharacter*}

***DelimitedComment*:**
'/*' {*DelimitedComment* | <any character>} '*/'

Number literals

***RealLiteral*:**
FloatLiteral | *DoubleLiteral*

***FloatLiteral*:**
DoubleLiteral ('f' | 'F') | *DecDigits* ('f' | 'F')

***DoubleLiteral*:**
[*DecDigits*] '.' *DecDigits* [*DoubleExponent*] | *DecDigits DoubleExponent*

***LongLiteral*:**
(*IntegerLiteral* | *HexLiteral* | *BinLiteral*) 'L'

***IntegerLiteral*:**
DecDigitNoZero {*DecDigitOrSeparator*} *DecDigit* | *DecDigit*

***HexLiteral*:**
'0' ('x'|'X') *HexDigit* {*HexDigitOrSeparator*} *HexDigit*
| '0' ('x'|'X') *HexDigit*

***BinLiteral*:**
'0' ('b'|'B') *BinDigit* {*BinDigitOrSeparator*} *BinDigit*
| '0' ('b'|'B') *BinDigit*

***DecDigitNoZero*:**
DecDigit - '0'

***DecDigitOrSeparator*:**
DecDigit | *Underscore*

***HexDigitOrSeparator*:**
HexDigit | *Underscore*

***BinDigitOrSeparator*:**
BinDigit | *Underscore*

***DecDigits*:**
DecDigit {*DecDigitOrSeparator*} *DecDigit* | *DecDigit*

***BooleanLiteral*:**
'true' | 'false'

***NullLiteral*:**
    `'null'`

Identifiers

***Identifier*:**
    (*Letter* | *Underscore*) {*Letter* | *Underscore* | *UnicodeDigit*}
    | `'`` {*EscapedIdentifierCharacter*} `'``

***EscapedIdentifierCharacter*:**
    <any character except CR, LF, '`', '[', ']', '<' or '>'>

***IdentifierOrSoftKey*:**
    Identifier | *SoftKeyword*

***AtIdentifier*:**
    `'@'` *IdentifierOrSoftKey*

***IdentifierAt*:**
    IdentifierOrSoftKey `'@'`

String literals

Syntax literals are fully defined in syntax grammar due to the complex nature of string interpolation

***CharacterLiteral*:**
    `'''` (*EscapeSeq* | *<any character except CR, LF, ''' and '\'>*) `'''`

***EscapeSeq*:**
    UnicodeCharacterLiteral | *EscapedCharacter*

***UnicodeCharacterLiteral*:**
    `'\' 'u'` *HexDigit HexDigit HexDigit HexDigit*

***EscapedCharacter*:**
    `'\'` (`'t'` | `'b'` | `'r'` | `'n'` | `'` | `'"'` | `'\'` | `'$'`)

***FieldIdentifier*:**
    `'$'` *IdentifierOrSoftKey*

***LineStrRef*:**
    FieldIdentifier

***LineStrEscapedChar*:**
    EscapedCharacter | *UnicodeCharacterLiteral*

***LineStrExprStart*:**
    `'${'`

***MultiLineStringQuote*:**
    `'"'` {`'"'`}

***MultiLineStrRef*:**
FieldIdentifier

***MultiLineStrText*:**
{<any character except `'"'` and `'$'`} | `'$'`

***MultiLineStrExprStart*:**
`'${'`

Misc

***EOF*:**
<end of input>

TODO: redo all the lexical grammar, right now it is a hand-written mess

Syntax grammar

***kotlinFile*:**
[*shebangLine*]
{*NL*}
{*fileAnnotation*}
packageHeader
importList
{*topLevelObject*}
EOF

***script*:**
[*shebangLine*]
{*NL*}
{*fileAnnotation*}
packageHeader
importList
{*statement semi*}
EOF

***shebangLine*:**
ShebangLine (*NL* {*NL*})

***fileAnnotation*:**
`'@file'`
{*NL*}
`':'`
{*NL*}
((`'['` (*unescapedAnnotation* {*unescapedAnnotation*}) `']'`) | *unescapedAnnotation*)
{*NL*}

***packageHeader*:**
    ['package' *identifier* [*semi*]]

***importList*:**
    {*importHeader*}

***importHeader*:**
    'import' *identifier* [('.' '*') | *importAlias*] [*semi*]

***importAlias*:**
    'as' *simpleIdentifier*

***topLevelObject*:**
    declaration [*semis*]

***typeAlias*:**
    [*modifiers*]
    'typealias'
    {*NL*}
    simpleIdentifier
    [{*NL*} *typeParameters*]
    {*NL*}
    '='
    {*NL*}
    type

***declaration*:**
    classDeclaration
    | *objectDeclaration*
    | *functionDeclaration*
    | *propertyDeclaration*
    | *typeAlias*

***classDeclaration*:**
    [*modifiers*]
    ('class' | 'interface')
    {*NL*}
    simpleIdentifier
    [{*NL*} *typeParameters*]
    [{*NL*} *primaryConstructor*]
    [{*NL*} ':' {*NL*} *delegationSpecifiers*]
    [{*NL*} *typeConstraints*]
    [({*NL*} *classBody*) | ({*NL*} *enumClassBody*)]

***primaryConstructor*:**
    [[*modifiers*] 'constructor' {*NL*}] *classParameters*

***classBody*:**
    '{'
    {*NL*}

classMemberDeclarations
{*NL*}
`'}'`

***classParameters*:**
`'('`
{*NL*}
[*classParameter* {{*NL*} `','` {*NL*} *classParameter*}]
{*NL*}
`')'`

***classParameter*:**
[*modifiers*]
[`'val'` | `'var'`]
{*NL*}
simpleIdentifier
`':'`
{*NL*}
type
[{*NL*} `'='` {*NL*} *expression*]

***delegationSpecifiers*:**
annotatedDelegationSpecifier {{*NL*} `','` {*NL*} *annotatedDelegationSpecifier*}

***delegationSpecifier*:**
constructorInvocation
| *explicitDelegation*
| *userType*
| *functionType*

***constructorInvocation*:**
userType valueArguments

***annotatedDelegationSpecifier*:**
{*annotation*} {*NL*} *delegationSpecifier*

***explicitDelegation*:**
(*userType* | *functionType*)
{*NL*}
`'by'`
{*NL*}
expression

***typeParameters*:**
`'<'`
{*NL*}
typeParameter
{{*NL*} `','` {*NL*} *typeParameter*}

{*NL*}
'>'

***typeParameter*:**
[*typeParameterModifiers*] {*NL*} *simpleIdentifier* [{*NL*} ':' {*NL*} *type*]

***typeConstraints*:**
'where' {*NL*} *typeConstraint* {{*NL*} ',' {*NL*} *typeConstraint*}

***typeConstraint*:**
{*annotation*}
simpleIdentifier
{*NL*}
':'
{*NL*}
type

***classMemberDeclarations*:**
{*classMemberDeclaration* [*semis*]}

***classMemberDeclaration*:**
declaration
| *companionObject*
| *anonymousInitializer*
| *secondaryConstructor*

***anonymousInitializer*:**
'init' {*NL*} *block*

***companionObject*:**
[*modifiers*]
'companion'
{*NL*}
'object'
[{*NL*} *simpleIdentifier*]
[{*NL*} ':' {*NL*} *delegationSpecifiers*]
[{*NL*} *classBody*]

***functionValueParameters*:**
'('
{*NL*}
[*functionValueParameter* {{*NL*} ',' {*NL*} *functionValueParameter*}]
{*NL*}
')'

***functionValueParameter*:**
[*modifiers*] *parameter* [{*NL*} '=' {*NL*} *expression*]

***functionDeclaration*:**
[*modifiers*]
'fun'

[{*NL*} *typeParameters*]
[{*NL*} *receiverType* {*NL*} `'.'`]
{*NL*}
simpleIdentifier
{*NL*}
functionValueParameters
[{*NL*} `':'` {*NL*} *type*]
[{*NL*} *typeConstraints*]
[{*NL*} *functionBody*]

***functionBody*:**
block
| (`'='` {*NL*} *expression*)

***variableDeclaration*:**
{*annotation*} {*NL*} *simpleIdentifier* [{*NL*} `':'` {*NL*} *type*]

***multiVariableDeclaration*:**
`'('`
{*NL*}
variableDeclaration
{{*NL*} `','` {*NL*} *variableDeclaration*}
{*NL*}
`')'`

***propertyDeclaration*:**
[*modifiers*]
(`'val'` | `'var'`)
[{*NL*} *typeParameters*]
[{*NL*} *receiverType* {*NL*} `'.'`]
({*NL*} (*multiVariableDeclaration* | *variableDeclaration*))
[{*NL*} *typeConstraints*]
[{*NL*} ((`'='` {*NL*} *expression*) | *propertyDelegate*)]
[(*NL* {*NL*}) `';'`]
{*NL*}
(([*getter*] [{*NL*} [*semi*] *setter*]) | ([*setter*] [{*NL*} [*semi*] *getter*]))

***propertyDelegate*:**
`'by'` {*NL*} *expression*

***getter*:**
([*modifiers*] `'get'`)
| ([*modifiers*] `'get'` {*NL*} `'('` {*NL*} `')'` [{*NL*} `':'` {*NL*} *type*] {*NL*} *functionBody*)

***setter*:**
([*modifiers*] `'set'`)
| ([*modifiers*] `'set'` {*NL*} `'('` {*annotation* | *parameterModifier*} *setterParameter* `')'` [{*NL*} `':'` {*NL*} *type*] {*NL*} *functionBody*)

***setterParameter*:**
simpleIdentifier {*NL*} [`':'` {*NL*} *type*]

***parameter*:**
simpleIdentifier
{*NL*}
`':'`
{*NL*}
type

***objectDeclaration*:**
[*modifiers*]
`'object'`
{*NL*}
simpleIdentifier
[{*NL*} `':'` {*NL*} *delegationSpecifiers*]
[{*NL*} *classBody*]

***secondaryConstructor*:**
[*modifiers*]
`'constructor'`
{*NL*}
functionValueParameters
[{*NL*} `':'` {*NL*} *constructorDelegationCall*]
{*NL*}
[*block*]

***constructorDelegationCall*:**
(`'this'` {*NL*} *valueArguments*)
| (`'super'` {*NL*} *valueArguments*)

***enumClassBody*:**
`'{'`
{*NL*}
[*enumEntries*]
[{*NL*} `';'` {*NL*} *classMemberDeclarations*]
{*NL*}
`'}'`

***enumEntries*:**
enumEntry {{*NL*} `','` {*NL*} *enumEntry*} {*NL*} [`','`]

***enumEntry*:**
[*modifiers* {*NL*}] *simpleIdentifier* [{*NL*} *valueArguments*] [{*NL*} *classBody*]

***type*:**
[*typeModifiers*] (*parenthesizedType* | *nullableType* | *typeReference* | *functionType*)

***typeReference*:**
userType
| `'dynamic'`

***nullableType*:**
(*typeReference* | *parenthesizedType*) {*NL*} (*quest* {*quest*})

***quest*:**
QUEST_NO_WS
| *QUEST_WS*

***userType*:**
simpleUserType {{*NL*} `'.'` {*NL*} *simpleUserType*}

***simpleUserType*:**
simpleIdentifier [{*NL*} *typeArguments*]

***typeProjection*:**
([*typeProjectionModifiers*] *type*)
| `'*'`

***typeProjectionModifiers*:**
typeProjectionModifier {*typeProjectionModifier*}

***typeProjectionModifier*:**
(*varianceModifier* {*NL*})
| *annotation*

***functionType*:**
[*receiverType* {*NL*} `'.'` {*NL*}]
functionTypeParameters
{*NL*}
`'->'`
{*NL*}
type

***functionTypeParameters*:**
`'('`
{*NL*}
[*parameter* | *type*]
{{*NL*} `','` {*NL*} (*parameter* | *type*)}
{*NL*}
`')'`

***parenthesizedType*:**
`'('`
{*NL*}
type
{*NL*}
`')'`

***receiverType*:**
[*typeModifiers*] (*parenthesizedType* | *nullableType* | *typeReference*)

***parenthesizedUserType*:**
('(' {*NL*} *userType* {*NL*} ')')
| ('(' {*NL*} *parenthesizedUserType* {*NL*} ')')

***statements*:**
[*statement* {*semis statement*} [*semis*]]

***statement*:**
{*label* | *annotation*} (*declaration* | *assignment* | *loopStatement* | *expression*)

***label*:**
IdentifierAt {*NL*}

***controlStructureBody*:**
block
| *statement*

***block*:**
'{'
{*NL*}
statements
{*NL*}
'}'

***loopStatement*:**
forStatement
| *whileStatement*
| *doWhileStatement*

***forStatement*:**
'for'
{*NL*}
'('
{*annotation*}
(*variableDeclaration* | *multiVariableDeclaration*)
IN
expression
')'
{*NL*}
[*controlStructureBody*]

***whileStatement*:**
('while' {*NL*} '(' *expression* ')' {*NL*} *controlStructureBody*)
| ('while' {*NL*} '(' *expression* ')' {*NL*} ';')

***doWhileStatement*:**
'do'
{*NL*}

[*controlStructureBody*]
{*NL*}
`'while'`
{*NL*}
`'('`
expression
`')'`

***assignment*:**
(*directlyAssignableExpression* `'='` {*NL*} *expression*)
| (*assignableExpression assignmentAndOperator* {*NL*} *expression*)

***semi*:**
((`';'` | *NL*) {*NL*})
| *EOF*

***semis*:**
(`';'` | *NL* {`';'` | *NL*})
| *EOF*

***expression*:**
disjunction

***disjunction*:**
conjunction {{*NL*} `'||'` {*NL*} *conjunction*}

***conjunction*:**
equality {{*NL*} `'&&'` {*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 `':'`

***infixFunctionCall*:**
rangeExpression {*simpleIdentifier* {*NL*} *rangeExpression*}

***rangeExpression*:**
additiveExpression {`'..'` {*NL*} *additiveExpression*}

***additiveExpression*:**
multiplicativeExpression {*additiveOperator* {*NL*} *multiplicativeExpression*}

***multiplicativeExpression*:**
asExpression {*multiplicativeOperator* {*NL*} *asExpression*}

***asExpression*:**
prefixUnaryExpression [{*NL*} *asOperator* {*NL*} *type*]

***prefixUnaryExpression*:**
{*unaryPrefix*} *postfixUnaryExpression*

***unaryPrefix*:**
annotation
| *label*
| (*prefixUnaryOperator* {*NL*})

***postfixUnaryExpression*:**
primaryExpression
| (*primaryExpression* (*postfixUnarySuffix* {*postfixUnarySuffix*}))

***postfixUnarySuffix*:**
postfixUnaryOperator
| *typeArguments*
| *callSuffix*
| *indexingSuffix*
| *navigationSuffix*

***directlyAssignableExpression*:**
(*postfixUnaryExpression assignableSuffix*)
| *simpleIdentifier*

***assignableExpression*:**
prefixUnaryExpression

***assignableSuffix*:**
typeArguments
| *indexingSuffix*
| *navigationSuffix*

***indexingSuffix*:**
`'['`
{*NL*}
expression
{{*NL*} `','` {*NL*} *expression*}
{*NL*}
`']'`

***navigationSuffix*:**
{*NL*} *memberAccessOperator* {*NL*} (*simpleIdentifier* | *parenthesizedExpression* | `'class'`)

***callSuffix*:**
([*typeArguments*] [*valueArguments*] *annotatedLambda*)

| ([*typeArguments*] *valueArguments*)

***annotatedLambda*:**
{*annotation*} [*label*] {*NL*} *lambdaLiteral*

***typeArguments*:**
'<'
{*NL*}
typeProjection
{{*NL*} ',' {*NL*} *typeProjection*}
{*NL*}
'>'

***valueArguments*:**
('(' {*NL*} ')')
| ('(' {*NL*} *valueArgument* {{*NL*} ',' {*NL*} *valueArgument*} {*NL*} ')')

***valueArgument*:**
[*annotation*]
{*NL*}
[*simpleIdentifier* {*NL*} '=' {*NL*}]
['*']
{*NL*}
expression

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

***parenthesizedExpression*:**
'('
{*NL*}
expression
{*NL*}
')'

***collectionLiteral*:**
('[' {*NL*} *expression* {{*NL*} ',' {*NL*} *expression*} {*NL*} ']')
| ('[' {*NL*} ']')

***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 '}'

***multiLineStringContent*:**
MultiLineStrText
| *MultiLineStringQuote*
| *MultiLineStrRef*

***multiLineStringExpression*:**
MultiLineStrExprStart
{*NL*}
expression
{*NL*}
'}'

***lambdaLiteral*:**
('{' {*NL*} *statements* {*NL*} '}')
| ('{' {*NL*} [*lambdaParameters*] {*NL*} '->' {*NL*} *statements* {*NL*} '}')

***lambdaParameters*:**
lambdaParameter {{*NL*} `','` {*NL*} *lambdaParameter*}

***lambdaParameter*:**
variableDeclaration
| (*multiVariableDeclaration* [{*NL*} `':'` {*NL*} *type*])

***anonymousFunction*:**
`'fun'`
[{*NL*} *type* {*NL*} `'.'`]
{*NL*}
functionValueParameters
[{*NL*} `':'` {*NL*} *type*]
[{*NL*} *typeConstraints*]
[{*NL*} *functionBody*]

***functionLiteral*:**
lambdaLiteral
| *anonymousFunction*

***objectLiteral*:**
(`'object'` {*NL*} `':'` {*NL*} *delegationSpecifiers* {*NL*} *classBody*)
| (`'object'` {*NL*} *classBody*)

***thisExpression*:**
`'this'`
| *THIS_AT*

***superExpression*:**
(`'super'` [`'<'` {*NL*} *type* {*NL*} `'>'`] [`'@'` *simpleIdentifier*])
| *SUPER_AT*

***ifExpression*:**
(`'if'` {*NL*} `'('` {*NL*} *expression* {*NL*} `')'` {*NL*} (*controlStructureBody* | `';'`))
| (`'if'` {*NL*} `'('` {*NL*} *expression* {*NL*} `')'` {*NL*} [*controlStructureBody*]
{*NL*} [`';'`] {*NL*} `'else'` {*NL*} (*controlStructureBody* | `';'`))

***whenExpression*:**
`'when'`
{*NL*}
[`'('` *expression* `')'`]
{*NL*}
`'{'`
{*NL*}
{*whenEntry* {*NL*}}
{*NL*}
`'}'`

***whenEntry*:**
(*whenCondition* {{*NL*} `','` {*NL*} *whenCondition*} {*NL*} `'->'` {*NL*} *controlStructureBody* [*semi*])
| (`'else'` {*NL*} `'->'` {*NL*} *controlStructureBody* [*semi*])

***whenCondition*:**
expression
| *rangeTest*
| *typeTest*

***rangeTest*:**
inOperator {*NL*} *expression*

***typeTest*:**
isOperator {*NL*} *type*

***tryExpression*:**
`'try'` {*NL*} *block* ((({*NL*} *catchBlock* {{*NL*} *catchBlock*}) [{*NL*} *finallyBlock*]) | ({*NL*} *finallyBlock*))

***catchBlock*:**
`'catch'`
{*NL*}
`'('`
{*annotation*}
simpleIdentifier
`':'`
type
`')'`
{*NL*}
block

***finallyBlock*:**
`'finally'` {*NL*} *block*

***jumpExpression*:**
(`'throw'` {*NL*} *expression*)
| ((`'return'` | *RETURN_AT*) [*expression*])
| `'continue'`
| *CONTINUE_AT*
| `'break'`
| *BREAK_AT*

***callableReference*:**
[*receiverType*]
{*NL*}
`'::'`
{*NL*}
(*simpleIdentifier* | `'class'`)

assignmentAndOperator:
    '+='
    | '-='
    | '*='
    | '/='
    | '%='

equalityOperator:
    '!='
    | '!=='
    | '=='
    | '==='

comparisonOperator:
    '<'
    | '>'
    | '<='
    | '>='

inOperator:
    'in'
    | *NOT_IN*

isOperator:
    'is'
    | *NOT_IS*

additiveOperator:
    '+'
    | '-'

multiplicativeOperator:
    '*'
    | '/'
    | '%'

asOperator:
    'as'
    | 'as?'

prefixUnaryOperator:
    '++'
    | '--'
    | '-'
    | '+'
    | *excl*

postfixUnaryOperator:
    '++'

| '--'
| (*EXCL_NO_WS excl*)

***excl*:**
EXCL_NO_WS
| *EXCL_WS*

***memberAccessOperator*:**
'.'
| *safeNav*
| '::'

***safeNav*:**
QUEST_NO_WS '.'

***modifiers*:**
annotation | *modifier* {*annotation* | *modifier*}

***modifier*:**
(*classModifier* | *memberModifier* | *visibilityModifier* | *functionModifier* | *propertyModifier* | *inheritanceModifier* | *parameterModifier* | *platformModifier*) {*NL*}

***typeModifiers*:**
typeModifier {*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*}

***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'`

***annotation*:**
(*singleAnnotation* | *multiAnnotation*) {*NL*}

***singleAnnotation*:**
(*annotationUseSiteTarget* {*NL*} *unescapedAnnotation*)
| (`'@'` *unescapedAnnotation*)

***multiAnnotation*:**
(*annotationUseSiteTarget* {*NL*} `'['` (*unescapedAnnotation* {*unescapedAnnotation*}) `']'`)
| (`'@'` `'['` (*unescapedAnnotation* {*unescapedAnnotation*}) `']'`)

***annotationUseSiteTarget*:**
`'@'` (`'field'` | `'property'` | `'get'` | `'set'` | `'receiver'` | `'param'` | `'setparam'` | `'delegate'`) {*NL*} `':'`

***unescapedAnnotation*:**
constructorInvocation
| *userType*

***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'
| 'expect'
| 'actual'
| 'const'
| 'suspend'

***identifier*:**
simpleIdentifier {{*NL*} `'.'` *simpleIdentifier*}

Type system

TODO(Add examples)

TODO(Add grammar snippets?)

Glossary

T Type
T!! Non-nullable type
T? Nullable type
$\{T\}$ Universe of all possible types
$\{T!!\}$
Universe of non-nullable types
$\{T?\}$
Universe of nullable types
Γ Type context
$A <: B$
A is a subtype of B
$A <:> B$
A and B are not related w.r.t. subtyping
Type constructor
An abstract type with one or more type parameters, which must be instantiated before use
Parameterized type
A concrete type, which is the result of type constructor instantiation
Type parameter
Formal type argument of a type constructor
Type argument
Actual type argument in a parameterized type
$T[A_1, \ldots, A_n]$
The result of type constructor T instantiation with type arguments A_i
$T[\sigma]$ The result of type constructor $T(F_1, \ldots, F_n)$ instantiation with the assumed substitution $\sigma : F_1 = A_1, \ldots, F_n = A_n$
σT The result of type substitution in type T w.r.t. substitution σ
$K_T(F, A)$
Captured type from the type capturing of type parameter F and type argument A in parameterized type T

$T\langle K_1, \ldots, K_n \rangle$
: The result of type capturing for parameterized type T with *captured* types K_i

$A \,\&\, B$
: Intersection type of A and B

$A|B$ Union type of A and B

`GLB` Greatest lower bound

`LUB` Least upper bound

TODO(Not everything is in the glossary, make some criteria of what goes where)

TODO(Cleanup glossary)

Introduction

Similarly to most other programming languages, Kotlin operates on data in the form of *values* or *objects*, which have *types* — descriptions of what is the expected behaviour and possible values for their datum. An empty value is represented by a special `null` object; most operations with it result in runtime errors or exceptions.

Kotlin has a type system with the following main properties.

- Hybrid static and gradual type checking
- Null safety
- No unsafe implicit conversions
- Unified top and bottom types
- Nominal subtyping with bounded parametric polymorphism and mixed-site variance

TODO(static type checking, gradual type checking)

Null safety is enforced by having two type universes: *nullable* (with nullable types $T?$) and *non-nullable* (with non-nullable types $T!!$). A value of any non-nullable type cannot contain `null`, meaning all operations within the non-nullable type universe are safe w.r.t. empty values, i.e., should never result in a runtime error caused by `null`.

Implicit conversions between types in Kotlin are limited to safe upcasts w.r.t. subtyping, meaning all other (unsafe) conversions must be explicit, done via either a conversion function or an explicit cast. However, Kotlin also supports smart casts — a special kind of implicit conversions which are safe w.r.t. program control- and data-flow, which are covered in more detail here.

The unified supertype type for all types in Kotlin is `kotlin.Any`?, a nullable version of [`kotlin.Any`][kotlin.Any]. The unified subtype type for all types in Kotlin is [`kotlin.Nothing`][kotlin.Nothing].

Kotlin uses nominal subtyping, meaning subtyping relation is defined when a type is declared, with bounded parametric polymorphism, implemented as generics via parameterized types. Subtyping between these parameterized types is defined through mixed-site variance.

Type kinds

For the purposes of this section, we establish the following type kinds — different flavours of types which exist in the Kotlin type system.

- Built-in types
- Classifier types
- Type parameters
- Function types
- Array types
- Flexible types
- Nullable types
- Intersection types
- Union types
- TODO(Error / invalid types)

We distinguish between *concrete* and *abstract* types. Concrete types are types which are assignable to values. Abstract types need to be instantiated as concrete types before they can be used as types for values.

> Note: for brevity, we omit specifying that a type is concrete. All types not described as abstract are implicitly concrete.

We further distinguish *concrete* types between *class* and *interface* types; as Kotlin is a language with single inheritance, sometimes it is important to discriminate between these kinds of types. Any given concrete type may be either a class or an interface type, but never both.

We also distinguish between *denotable* and *non-denotable* types. The former are types which are expressible in Kotlin and can be written by the end-user. The latter are special types which are *not* expressible in Kotlin and are used internally by the compiler.

Built-in types

Kotlin type system uses the following built-in types, which have special semantics and representation (or lack thereof).

`kotlin.Any`

`kotlin.Any` is the unified supertype ($\top$) for $\{T!!\}$, i.e., all non-nullable types are subtypes of `kotlin.Any`, either explicitly, implicitly, or by subtyping relation.

TODO(`kotlin.Any` members?)

`kotlin.Nothing`

`kotlin.Nothing` is the unified subtype ($\bot$) for $\{T\}$, i.e., `kotlin.Nothing` is a subtype of all well-formed Kotlin types, including user-defined ones. This makes it an uninhabited type (as it is impossible for anything to be, for example, a function and an integer at the same time), meaning instances of this type can never exist at runtime; subsequently, there is no way to create an instance of `kotlin.Nothing` in Kotlin.

As the evaluation of an expression with `kotlin.Nothing` type can never complete normally, it is used to mark special situations, such as

- non-terminating expressions
- exceptional control flow
- control flow transfer

Additional details about how `kotlin.Nothing` should be processed are available here.

`kotlin.Unit`

`kotlin.Unit` is a unit type, i.e., a type with only one value `kotlin.Unit`; all values of type `kotlin.Unit` should reference the same underlying `kotlin.Unit` object.

TODO(Compare to `void`?)

`kotlin.Function`

`kotlin.Function`(R) is the unified supertype of all function types. It is parameterized over function return type `R`.

Classifier types

Classifier types represent regular types which are declared as [classes][Classes], [interfaces][Interfaces] or [objects][Objects]. As Kotlin supports generics, there are two variants of classifier types: simple and parameterized.

Simple classifier types

A simple classifier type

$$T : S_1, \ldots, S_m$$

consists of

- type name T
- (optional) list of supertypes $S_1, \ldots, S_m$

To represent a well-formed simple classifier type, $T : S_1, \ldots, S_m$ should satisfy the following conditions.

- T is a valid type name
- $\forall i \in [1, m] : S_i$ must be concrete, non-nullable, well-formed type

  Example:

  ```
  // A well-formed type with no supertypes
  interface Base

  // A well-formed type with a single supertype Base
  interface Derived : Base

  // An ill-formed type,
  // as nullable type cannot be a supertype
  interface Invalid : Base?
  ```

  Note: for the purpose of different type system examples, we assume the presence of the following well-formed concrete types:

  - class `String`
  - interface `Number`
  - class `Int` <: `Number`
  - class `Double` <: `Number`

Parameterized classifier types

A classifier type constructor

$$T(F_1, \ldots, F_n) : S_1, \ldots, S_m$$

describes an abstract type and consists of

- type name T
- type parameters $F_1, \ldots, F_n$
- (optional) list of supertypes $S_1, \ldots, S_m$

To represent a well-formed type constructor, $T(F_1, \ldots, F_n) : S_1, \ldots, S_m$ should satisfy the following conditions.

- T is a valid type name
- $\forall i \in [1, n] : F_i$ must be well-formed type parameter
- $\forall j \in [1, m] : S_j$ must be concrete, non-nullable, well-formed type

To instantiate a type constructor, one provides it with type arguments, creating a concrete parameterized classifier type

$$T[A_1, \ldots, A_n]$$

which consists of

- type constructor T
- type arguments $A_1, \ldots, A_n$

To represent a well-formed parameterized type, $T[A_1, \ldots, A_n]$ should satisfy the following conditions.

- T is a well-formed type constructor with n type parameters
- $\forall i \in [1, n] : A_i$ must be well-formed concrete type
- $\forall i \in [1, n] : K_T(F_i, A_i)$ is a well-formed captured type, where K is a type capturing operator

  Example:

```
// A well-formed PACT with no supertypes
// A and B are unbounded type parameters
interface Generic<A, B>

// A well-formed PACT with a single iPACT supertype
// Int and String are well-formed concrete types
interface ConcreteDerived<P, Q> : Generic<Int, String>

// A well-formed PACT with a single iPACT supertype
// P and Q are type parameters of GenericDerived,
//   used as type arguments of Generic
interface GenericDerived<P, Q> : Generic<P, Q>
```

```
// An ill-formed PACT,
//   as an abstract type Generic
//   cannot be used as a supertype
interface Invalid<P> : Generic


// A well-formed PACT with no supertypes
// out A is a projected type parameter
interface Out<out A>


// A well-formed PACT with no supertypes
// S : Number is a bounded type parameter
// (S <: Number)
interface NumberWrapper<S : Number>

// A well-formed type with a single iPACT supertype
// NumberWrapper<Int> is well-formed,
//   as Int <: Number
interface IntWrapper : NumberWrapper<Int>

// An ill-formed type,
//   as NumberWrapper<String> is an ill-formed iPACT
//   (String <:> Number)
interface InvalidWrapper : NumberWrapper<String>
```

Type parameters

Type parameters are a special kind of types, which are introduced by type constructors. They are considered well-formed concrete types only in the type context of their declaring type constructor.

When creating a parameterized type from a type constructor, its type parameters with their respective type arguments go through capturing and create *captured* types, which follow special rules described in more detail below.

Type parameters may be either unbounded or bounded. By default, a type parameter F is unbounded, which is the same as saying it is a bounded type parameter of the form $F <: \texttt{kotlin.Any?}$.

A bounded type parameter additionally specify upper type bounds for the type parameter and is defined as $F <: B_1, \ldots, B_n$, where B_i is an i-th upper bound on type parameter F.

To represent a well-formed bounded type parameter of type constructor T, $F <: B_1, \ldots, B_n$ should satisfy either of the following sets of conditions.

- Bounded type parameter with regular bounds:
  - F is a type parameter of PACT T
  - $\forall i \in [1, n] : B_i$ must be concrete, non-type-parameter, well-formed type
  - No more than one of B_i may be a class type

  Note: the last condition is a nod to the single inheritance nature of Kotlin; as any type may be a subtype of no more than one class type, it makes no sense to support several class type bounds. For any two class types, either these types are in a subtyping relation (and you should use the more specific type in the bounded type parameter), or they are unrelated (and the bounded type parameter is empty).

- Bounded type parameter with type parameter bound:
  - F is a type parameter of PACT T
  - $i = 1$ (i.e., there is a single upper bound)
  - B_1 must be well-formed type parameter

From the definition, it follows $F <: B_1, \ldots, B_n$ can be represented as $K <: U$ where $U = B_1 \,\&\, \ldots \,\&\, B_n$.

Mixed-site variance

To implement subtyping between parameterized types, Kotlin uses *mixed-site variance* — a combination of declaration- and use-site variance, which is easier to understand and reason about, compared to wildcards from Java. Mixed-site variance means you can specify, whether you want your parameterized type to be co-, contra- or invariant on some type parameter, both in type parameter (declaration-site) and type argument (use-site).

> Info: *variance* is a way of describing how subtyping works for *variant* parameterized types. With declaration-site variance, for two types $A <: B$, subtyping between `T<A>` and `T<B>` depends on the variance of type parameter F of some type constructor T.
>
> - if F is covariant (`out` F), `T<A> <: T<B>`
> - if F is contravariant(`in` F), `T<A> :> T<B>`
> - if F is invariant (default), `T<A> <:> T<B>`
>
> Use-site variance allows the user to change the type variance of an *invariant* type parameter by specifying it on the corresponding type argument. `out` A means covariant type argument, `in` A means contravariant type argument; for two types $A <: B$ and an invariant type parameter F of some type constructor T, subtyping for use-site variance has the following rules.
>
> - `T<out A> <: T<out B>`
> - `T<in A> :> T<in B>`
> - `T<A> <: T<out A>`

- `T<A> <: T<in A>`
- `T<in A> <:> T<out A>`

Note: Kotlin does not support specifying both co- and contravariance at the same time, i.e., it is impossible to have `T<in A out B>` neither on declaration- nor on use-site.

For further discussion about mixed-site variance and its practical applications, we readdress you to subtyping and generics.

TODO(Fix formatting here)

Declaration-site variance

A type parameter F may be invariant, covariant or contravariant.

By default, all type parameters are invariant.

To specify a covariant type parameter, it is marked as `out` F. To specify a contravariant type parameter, it is marked as `in` F.

The variance information is used by subtyping and for checking allowed operations on values of co- and contravariant type parameters.

Important: declaration-site variance can be used only when declaring types, e.g., type parameters of functions cannot be variant.

Example:

```
// A type constructor with an invariant type parameter
interface Invariant<A>
// A type constructor with a covariant type parameter
interface Out<out A>
// A type constructor with a contravariant type parameter
interface In<in A>

fun testInvariant() {
    var invInt: Invariant<Int> = ...
    var invNumber: Invariant<Number> = ...

    if (random) invInt = invNumber // ERROR
    else invNumber = invInt // ERROR

    // Invariant type parameters do not create subtyping
}

fun testOut() {
    var outInt: Out<Int> = ...
    var outNumber: Out<Number> = ...
```

```
    if (random) outInt = outNumber // ERROR
    else outNumber = outInt // OK

    // Covariant type parameters create "same-way" subtyping
    //   Int <: Number => Out<Int> <: Out<Number>
    // (more specific type Out<Int> can be assigned
    //  to a less specific type Out<Number>)
}

fun testIn() {
    var inInt: In<Int> = ...
    var inNumber: In<Number> = ...

    if (random) inInt = inNumber // OK
    else inNumber = inInt // ERROR

    // Contravariant type parameters create "opposite-way" subtyping
    //   Int <: Number => In<Int> :> In<Number>
    // (more specific type In<Number> can be assigned
    //  to a less specific type In<Int>)
}
```

Use-site variance

Kotlin also supports use-site variance, by specifying the variance for type arguments. Similarly to type parameters, one can have type arguments being co-, contra- or invariant.

By default, all type arguments are invariant.

To specify a covariant type argument, it is marked as `out` A. To specify a contravariant type argument, it is marked as `in` A.

> Note: in some cases, Kotlin prohibits certain combinations of declaration- and use-site variance, i.e., which type arguments can be used in which type parameters. These rules are covered in more detail [here][TODO()].

In case one cannot specify any well-formed type argument, but still needs to use a parameterized type in a type-safe way, one may use *bivariant* type argument $\star$, which is roughly equivalent to a combination of `out kotlin.Any?` and `in kotlin.Nothing` (for further details, see subtyping and generics).

TODO(Specify how this combination of co- and contravariant parameters works from the practical PoV)

Important: use-site variance cannot be used when declaring a supertype.

Example:

```
// A type constructor with an invariant type parameter
interface Inv<A>

fun test() {
    var invInt: Inv<Int> = ...
    var invNumber: Inv<Number> = ...
    var outInt: Inv<out Int> = ...
    var outNumber: Inv<out Number> = ...
    var inInt: Inv<in Int> = ...
    var inNumber: Inv<in Number> = ...

    when (random) {
        1 -> {
            inInt = invInt    // OK
            // T<in Int> :> T<Int>

            inInt = invNumber // OK
            // T<in Int> :> T<in Number> :> T<Number>
        }
        2 -> {
            outNumber = invInt    // OK
            // T<out Number> :> T<out Int> :> T<Int>

            outNumber = invNumber // OK
            // T<out Number> :> T<Number>
        }
        3 -> {
            invInt = inInt  // ERROR
            invInt = outInt // ERROR
            // It is invalid to assign less specific type
            // to a more specific one
            //   T<Int> <: T<in Int>
            //   T<Int> <: T<out Int>
        }
        4 -> {
            inInt = outInt    // ERROR
            inInt = outNumber // ERROR
            // types with co- and contravariant type parameters
            // are not connected by subtyping
            //   T<in Int> <:> T<out Int>
        }
    }
```

```
}
```

Type capturing

Type capturing (similarly to Java capture conversion) is used when instantiating type constructors; it creates *abstract captured* types based on the type information of both type parameters and arguments, which present a unified view on the resulting types and simplifies further reasoning.

The reasoning behind type capturing is closely related to variant parameterized types being a form of *bounded existential types*; e.g., `A<out T>` may be loosely considered as the following existential type: $\exists X : X <: T.\texttt{A} < \texttt{X} >$. Informally, a bounded existential type describes a *set* of possible types, which satisfy its bound constraints. Before such a type can be used, it needs to be *opened* (or *unpacked*): existentially quantified type variables are lifted to fresh type variables with corresponding bounds. We call these type variables *captured* types.

For a given type constructor $T(F_1, \ldots, F_n) : S_1, \ldots, S_m$, its instance $T[\sigma]$ uses the following rules to create captured type K_i from the type parameter F_i and type argument A_i.

> Note: **All** applicable rules are used to create the resulting constraint set.

- For a covariant type parameter `out` F_i, if A_i is an ill-formed type or a contravariant type argument, K_i is an ill-formed type. Otherwise, $K_i <: A_i$.
- For a contravariant type parameter `in` F_i, if A_i is an ill-formed type or a covariant type argument, K_i is an ill-formed type. Otherwise, $K_i :> A_i$.
- For a bounded type parameter $F_i <: B_1, \ldots, B_m$, if $\exists j \in [1, m] : \neg(A_i <: B_j)$, K_i is an ill-formed type. Otherwise, $\forall j \in [1, m] : K_i <: \sigma B_j$.
- For a covariant type argument `out` A_i, if F_i is a contravariant type parameter, K_i is an ill-formed type. Otherwise, $K_i <: A_i$.
- For a contravariant type argument `in` A_i, if F_i is a covariant type parameter, K_i is an ill-formed type. Otherwise, $K_i :> A_i$.
- For a bivariant type argument $\star$, `kotlin.Nothing` $<: K_i <:$ `kotlin.Any`?.
- Otherwise, $K_i \equiv A_i$.

By construction, every captured type K has the following form:

$$\{L_1 <: K, \ldots, L_p <: K, K <: U_1, \ldots, K <: U_q\}$$

which can be represented as

$$L <: K <: U$$

where $L = L_1 | \ldots | L_p$ and $U = U_1 \,\&\, \ldots \,\&\, U_q$.

> Note: as every captured type corresponds to a fresh type variable, two different captured types K_i and K_j which describe the same set of possible types (i.e., their constraint sets are equal) are *not* considered equal. However, in some cases type inference may approximate a captured type K to a concrete type $K^{\approx}$; in our case, it would be that $K_i^{\approx} \equiv K_j^{\approx}$.

TODO(Need to think more about this part)

Function types

Kotlin has first-order functions; e.g., it supports function types, which describe the argument and return types of its corresponding function.

A function type FT

$$FT(A_1, \ldots, A_n) \to R$$

consists of

- argument types A_i
- return type R

and may be considered the following instantiation of a special type constructor $FunctionN(\texttt{in}\ P_1, \ldots, \texttt{in}\ P_n, \texttt{out}\ RT)$

$$FT(A_1, \ldots, A_n) \to R \equiv FunctionN[A_1, \ldots, A_n, R]$$

These $FunctionN$ types follow the rules of regular type constructors and parameterized types w.r.t. subtyping.

A function type with receiver FTR

$$FTR(TH, A_1, \ldots, A_n) \to R$$

consists of

- receiver type TH
- argument types A_i
- return type R

From the type system's point of view, it is equivalent to the following function type

$$FTR(TH, A_1, \ldots, A_n) \to R \equiv FT(TH, A_1, \ldots, A_n) \to R$$

i.e., receiver is considered as yet another argument of its function type.

> Note: this means that, for example, these two types are equivalent
>
> - `Int.(Int) -> String`
> - `(Int, Int) -> String`

Furthermore, all function types $FunctionN$ are subtypes of a general argument-agnostic type [`kotlin.Function`][kotlin.Function] for the purpose of unification.

> Note: a compiler implementation may consider a function type $FunctionN$ to have additional supertypes, if it is necessary.

TODO(We already have `kotlin.Function` settled in this spec earlier. The reason for this is that overloading needs it)

> Example:
>
> ```
> // A function of type Function1<Number, Number>
> //   or (Number) -> Number
> fun foo(i: Number): Number = ...
>
> // A valid assignment w.r.t. function type variance
> // Function1<in Int, out Any> :> Function1<in Number, out Number>
> val fooRef: (Int) -> Any = ::foo
>
> // A function with receiver of type Function1<Number, Number>
> //   or Number.() -> Number
> fun Number.bar(): Number = ...
>
> // A valid assignment w.r.t. function type variance
> // Receiver is just yet another function argument
> // Function1<in Int, out Any> :> Function1<in Number, out Number>
> val barRef: (Int) -> Any = Number::bar
> ```

Array types

Kotlin arrays are represented as a parameterized type `kotlin.Array`(T), where T is the type of the stored elements, which supports `get`/`set` operations. The `kotlin.Array`(T) type follows the rules of regular type constructors and parameterized types w.r.t. subtyping.

> Note: unlike Java, arrays in Kotlin are declared as invariant. To use them in a co- or contravariant way, one should use use-site variance.

In addition to the general `kotlin.Array`(T) type, Kotlin also has the following specialized array types:

- `DoubleArray` (for `kotlin.Array`$(Double)$)
- `FloatArray` (for `kotlin.Array`$(Float)$)
- `LongArray` (for `kotlin.Array`$(Long)$)
- `IntArray` (for `kotlin.Array`(Int))
- `ShortArray` (for `kotlin.Array`$(Short)$)
- `ByteArray` (for `kotlin.Array`$(Byte)$)
- `CharArray` (for `kotlin.Array`$(Char)$)
- `BooleanArray` (for `kotlin.Array`$(Boolean)$)

These array types structurally match the corresponding `kotlin.Array`(T) type; i.e., `IntArray` has the same methods and properties as `kotlin.Array`(Int). However, they are **not** related by subtyping; meaning one cannot pass a `BooleanArray` argument to a function expecting an `kotlin.Array`$(Boolean)$.

> Note: the presence of such specialized types allows the compiler to perform additional array-related optimizations.

Array type specialization `ATS`(T) is a transformation of a generic `kotlin.Array`(T) type to a corresponding specialized version, which works as follows.

- if `kotlin.Array`(T) has a specialized version `TArray`, `ATS`(`kotlin.Array`$(T)) = TArray$
- if `kotlin.Array`(T) does not have a specialized version, `ATS`(`kotlin.Array`$(T))$ = `kotlin.Array`(T)

`ATS` takes an important part in how variable length parameters are handled.

Flexible types

Kotlin, being a multi-platform language, needs to support transparent interoperability with platform-dependent code. However, this presents a problem in that some platforms may not support null safety the way Kotlin does. To deal with this, Kotlin supports *gradual typing* in the form of flexible types.

A flexible type represents a range of possible types between type L (lower bound) and type U (upper bound), written as $(L..U)$. One should note flexible types

are *non-denotable*, i.e., one cannot explicitly declare a variable with flexible type, these types are created by the type system when needed.

To represent a well-formed flexible type, $(L..U)$ should satisfy the following conditions.

- L and U are well-formed concrete types
- $L <: U$
- $\neg(L <: U)$
- L and U are **not** flexible types (but may contain other flexible types as some of their type arguments)

As the name suggests, flexible types are flexible — a value of type $(L..U)$ can be used in any context, where one of the possible types between L and U is needed (for more details, see subtyping rules for flexible types). However, the actual type will be a specific type between L and U, thus making the substitution possibly unsafe, which is why Kotlin generates dynamic assertions, when it is impossible to prove statically the safety of flexible type use.

TODO(Details of assertion generation?)

Dynamic type

Kotlin includes a special *dynamic* type, which is a flexible type (`kotlin.Nothing..kotlin.Any?`). By definition, this type represents *any* possible Kotlin type, and may be used to support interoperability with dynamically typed libraries, platforms or languages.

TODO(We should reconsider defining `dynamic` as a flexible type, cause it doesn't behave like one in many situations)

Platform types

The main use cases for flexible types are *platform types* — types which the Kotlin compiler uses, when interoperating with code written for another platform (e.g., Java). In this case all types on the interoperability boundary are subject to *flexibilization* — the process of converting a platform-specific type to a Kotlin-compatible flexible type.

For further details on how *flexibilization* is done, see:

- [Platform types for Java][TODO(need a way to have same section names in different parts of the spec)]

  Important: platform types should not be confused with *multi-platform projects* — another Kotlin feature targeted at supporting platform interop.

Nullable types

Kotlin supports null safety by having two type universes — nullable and non-nullable. All classifier type declarations, built-in or user-defined, create non-nullable types, i.e., types which cannot hold `null` value at runtime.

To specify a nullable version of type T, one needs to use $T?$ as a type. Redundant nullability specifiers are ignored — $T?? \equiv T?$.

> Note: informally, question mark means "$T?$ may hold values of type T or value `null`"

To represent a well-formed nullable type, $T?$ should satisfy the following conditions.

- T is a well-formed concrete type

If an operation is safe regardless of absence or presence of `null`, e.g., assignment of one nullable value to another, it can be used as-is for nullable types. For operations on $T?$ which may violate null safety, e.g., access to a property, one has the following null-safe options:

1. Use safe operations
   - safe call
2. Downcast from $T?$ to $T!!$
   - unsafe cast
   - type check combined with smart casts
   - null check combined with smart casts
   - not-null assertion operator
3. Supply a default value to use if `null` is present
   - elvis operator

Intersection types

Intersection types are special *non-denotable* types used to express the fact that a value belongs to *all* of *several* types at the same time.

Intersection type of two types A and B is denoted $A \,\&\, B$ and is equivalent to the greatest lower bound of its components $\texttt{GLB}(A, B)$. Thus, the normalization procedure for `GLB` may be used to *normalize* an intersection type.

> Note: this means intersection types are commutative and associative (following the GLB properties); e.g., $A \,\&\, B$ is the same type as $B \,\&\, A$, and $A \,\&(B \,\&\, C)$ is the same type as $A \,\&\, B \,\&\, C$.

> Note: for presentation purposes, we will henceforth order intersection type operands lexicographically based on their notation.

When needed, the compiler may *approximate* an intersection type to a *denotable concrete* type using type approximation.

One of the main uses of intersection types are smart casts.

Integer literal types

TODO(Think this through)

An integer literal type containing types $T_1, \ldots, T_N$, denoted $\texttt{LTS}(T_1, \ldots, T_N)$ is a special *non-denotable* type designed for integer literals. Each type $T_1, \ldots, T_N$ must be one of the built-in integer types

Integer literal types are the types of integer literals.

TODO(Consult with the team)

Union types

> Important: Kotlin does **not** have union types in its type system. However, they make reasoning about several type system features easier. Therefore, we decided to include a brief intro to the union types here.

Union types are special *non-denotable* types used to express the fact that a value belongs to *one* of *several* possible types.

Union type of two types A and B is denoted $A|B$ and is equivalent to the least upper bound of its components $\texttt{LUB}(A, B)$. Thus, the normalization procedure for `LUB` may be used to *normalize* a union type.

Moreover, as union types are *not* used in Kotlin, the compiler always *decays* a union type to a *non-union* type using type approximation.

Type context

TODO(Type contexts and their relation to scopes) TODO(Inner vs nested type contexts)

Subtyping

TODO(Need to change the way we think about subtyping)

Kotlin uses the classic notion of *subtyping* as *substitutability* — if S is a subtype of T (denoted as $S <: T$), values of type S can be safely used where values of type T are expected. The subtyping relation $<:$ is:

- reflexive ($A <: A$)
- transitive ($A <: B \land B <: C \Rightarrow A <: C$)

Two types A and B are *equivalent* ($A \equiv B$), iff $A <: B \land B <: A$. Due to the presence of flexible types, this relation is **not** transitive (see here for more details).

Subtyping rules

Subtyping for non-nullable, concrete types uses the following rules.

- $\forall T$: `kotlin.Nothing` $<: T <:$ `kotlin.Any`
- For any simple classifier type $T : S_1, \ldots, S_m$ it is true that $\forall i \in [1, m] : T <: S_i$
- For any parameterized type $\widehat{T} = T[\sigma] : S_1, \ldots, S_m$ it is true that $\forall i \in [1, m] : \widehat{T} <: \sigma S_i$
- For any two parameterized types $\widehat{T}$ and $\widehat{T'}$ with captured type arguments K_i and K'_i it is true that $\widehat{T} <: \widehat{T'}$ if $\forall i \in [1, n] : K_i <: K'_i$

Subtyping for non-nullable, abstract types uses the following rules.

- $\forall T$: `kotlin.Nothing` $<: T <:$ `kotlin.Any`
- For any type constructor $\widehat{T} = T(F_1, \ldots, F_n) : S_1, \ldots, S_m$ it is true that $\forall i \in [1, m] : \widehat{T} <: S_i$
- For any two type constructors $\widehat{T}$ and $\widehat{T'}$ with type parameters F_i and F'_i it is true that $\widehat{T} <: \widehat{T'}$ if $\forall i \in [1, n] : F_i <: F'_i$

Subtyping for type parameters uses the following rules.

- $\forall F$: `kotlin.Nothing` $<: F <:$ `kotlin.Any`?
- For any two type parameters F and F', it is true that $F <: F'$, if all of the following hold
  - variance of F matches variance of F'
    - `out` matches `out`
    - `in` matches `in`
    - `inv` matches any variance
  - for $F <: B$ and $F' <: B'$, $B <: B'$

Subtyping for captured types uses the following rules.

- $\forall K$: kotlin.Nothing $<: K <:$ kotlin.Any?
- For any two captured types $L <: K <: U$ and $L' <: K' <: U'$, it is true that $K <: K'$ if $L' <: L$ and $U <: U'$

Subtyping for nullable types is checked separately and uses a special set of rules which are described here.

Subtyping for flexible types

Flexible types (being flexible) follow a simple subtyping relation with other inflexible types. Let T, A, B, L, U be inflexible types.

- $L <: T \Rightarrow (L..U) <: T$
- $T <: U \Rightarrow T <: (L..U)$

This captures the notion of flexible type $(L..U)$ as something which may be used in place of any type in between L and U. If we are to extend this idea to subtyping between *two* flexible types, we get the following definition.

- $L <: B \Rightarrow (L..U) <: (A..B)$

This is the most extensive definition possible, which, unfortunately, makes the type equivalence relation non-transitive. Let A, B be two *different* types, for which $A <: B$. The following relations hold:

- $A <: (A..B) \wedge (A..B) <: A \Rightarrow A \equiv (A..B)$
- $B <: (A..B) \wedge (A..B) <: B \Rightarrow B \equiv (A..B)$

However, $A \not\equiv B$.

Subtyping for intersection types

Intersection types introduce several new rules for subtyping. Let A, B, C, D be non-nullable types.

- $A \,\&\, B <: A$
- $A \,\&\, B <: B$
- $A <: C \wedge B <: D \Rightarrow A \,\&\, B <: C \,\&\, D$

Moreover, any type T with supertypes $S_1, \ldots, S_N$ is also a subtype of $S_1 \,\&\, \ldots \,\&\, S_N$.

Subtyping for integer literal types

Every integer literal type is equivalent with w.r.t. subtyping, meaning that for any sets $T_1, \ldots, T_K$ and $U_1, \ldots, U_N$ of builtin integer types:

- $\texttt{LTS}(T_1, \ldots, T_K) <: \texttt{LTS}(U_1, \ldots, U_N)$
- $\texttt{LTS}(U_1, \ldots, U_K) <: \texttt{LTS}(T_1, \ldots, T_K)$
- $\forall T_i \in \{T_1, \ldots, T_K\}. T_i <: \texttt{LTS}(T_1, \ldots, T_K)$
- $\forall T_i \in \{T_1, \ldots, T_K\}. \texttt{LTS}(T_1, \ldots, T_K) <: T_i$

Subtyping for nullable types

TODO(Why can't we just say that $\forall T : T <: T?$ and $\forall T : T!! <: T$ and be done with it?)

Subtyping for two possibly nullable types A and B is defined via *two* relations, both of which must hold.

- Regular subtyping $<:$ for non-nullable types $A!!$ and $B!!$
- Subtyping by nullability $\overset{null}{<:}$

Subtyping by nullability $\overset{null}{<:}$ for two possibly nullable types A and B uses the following rules.

- $A!! \overset{null}{<:} B$
- $A \overset{null}{<:} B$ if $\exists T!! : A <: T!!$
- $A \overset{null}{<:} B?$
- $A \overset{null}{<:} B$ if $\nexists T!! : B <: T!!$

TODO(How the existence check works)

Generics

TODO(How are generics different from type parameters? Or are we going to get into deep technical detail?)

Upper and lower bounds

A type U is an *upper bound* of types A and B if $A <: U$ and $B <: U$. A type L is a *lower bound* of types A and B if $L <: A$ and $L <: B$.

> Note: as the type system of Kotlin is bounded by definition (the upper bound of all types is `kotlin.Any`?, and the lower bound of all types is `kotlin.Nothing`), any two types have at least one lower bound and at least one upper bound.

Least upper bound

The *least upper bound* $\texttt{LUB}(A, B)$ of types A and B is an upper bound U of A and B such that there is no other upper bound of these types which is less by subtyping relation than U.

> Note: `LUB` is commutative, i.e., $\texttt{LUB}(A, B) = \texttt{LUB}(B, A)$. This property is used in the subsequent description, e.g., other properties of `LUB` are defined only for a specific order of the arguments. Definitions following from commutativity of `LUB` are implied.

$\texttt{LUB}(A, B)$ has the following properties, which may be used to *normalize* it. This normalization procedure, if finite, creates a *canonical* representation of LUB.

- $\texttt{LUB}(A, A) = A$
- if $A <: B$, $\texttt{LUB}(A, B) = B$
- if A is nullable, $\texttt{LUB}(A, B)$ is also nullable
- if both A and B are nullable, $\texttt{LUB}(A, B) = \texttt{LUB}(A!!, B!!)$?
- if A is nullable and B is not, $\texttt{LUB}(A, B) = \texttt{LUB}(A!!, B)$?
- if $A = T\langle K_{A,1}, \ldots, K_{A,n}\rangle$ and $B = T\langle K_{B,1}, \ldots, K_{B,n}\rangle$, $\texttt{LUB}(A, B) = T\langle \phi(K_{A,1}, K_{B,1}), \ldots, \phi(K_{A,n}, K_{B,n})\rangle$, where $\phi(X, Y)$ is defined as follows:
  - $\phi(\texttt{inv}\ X, \texttt{inv}\ X) = X$
  - $\phi(\texttt{out}\ X, \texttt{out}\ Y) = \texttt{out}\ \texttt{LUB}(X, Y)$
  - $\phi(\texttt{out}\ X, \texttt{inv}\ Y) = \phi(\texttt{out}\ X, \texttt{out}\ Y)$
  - $\phi(\texttt{out}\ X, \texttt{in}\ Y) = \star$
  - $\phi(\texttt{inv}\ X, \texttt{out}\ Y) = \phi(\texttt{out}\ X, \texttt{out}\ Y)$
  - $\phi(\texttt{inv}\ X, \texttt{inv}\ Y) = \phi(\texttt{out}\ X, \texttt{out}\ Y)$
  - $\phi(\texttt{inv}\ X, \texttt{in}\ Y) = \phi(\texttt{out}\ X, \texttt{out kotlin.Any?}) = \texttt{out kotlin.Any?}$
  - $\phi(\texttt{in}\ X, \texttt{out}\ Y) = \star$
  - $\phi(\texttt{in}\ X, \texttt{inv}\ Y) = \phi(\texttt{out kotlin.Any?}, \texttt{out}\ Y) = \texttt{out kotlin.Any?}$
  - $\phi(\texttt{in}\ X, \texttt{in}\ Y) = \texttt{in}\ \texttt{GLB}(X, Y)$
  - TODO(we may also choose the `in` projection for `inv` parameters, do we wanna do it though?)
- if $A = (L_A..U_A)$ and $B = (L_B..U_B)$, $\texttt{LUB}(A, B) = (\texttt{LUB}(L_A, L_B)..\texttt{LUB}(U_A, U_B))$
- if $A = (L_A..U_A)$ and B is not flexible, $\texttt{LUB}(A, B) = (\texttt{LUB}(L_A, B)..\texttt{LUB}(U_A, B))$

TODO(prettify formatting)

TODO(actual algorithm for computing LUB)

TODO(LUB for 3+ types)

TODO(what do we do if this procedure loops?)

TODO(Why do we need union types again?)

Greatest lower bound

The *greatest lower bound* $\texttt{GLB}(A, B)$ of types A and B is a lower bound L of A and B such that there is no other lower bound of these types which is greater by subtyping relation than L.

> Note: enumerating all subtypes of a given type is impossible in general, but in the presence of intersection types, $GLB(A, B) \equiv A \,\&\, B$.

TODO(It's not if types are related)

> Note: `GLB` is commutative, i.e., $\texttt{GLB}(A, B) = \texttt{GLB}(B, A)$. This property is used in the subsequent description, e.g., other properties of `GLB` are defined only for a specific order of the arguments. Definitions following from commutativity of `GLB` are implied.

$\texttt{GLB}(A, B)$ has the following properties, which may be used to *normalize* it. This normalization procedure, if finite, creates a *canonical* representation of GLB.

- $\texttt{GLB}(A, A) = A$
- if $A <: B$, $\texttt{GLB}(A, B) = A$
- if A is non-nullable, $\texttt{GLB}(A, B)$ is also non-nullable
- if both A and B are nullable, $\texttt{GLB}(A, B) = \texttt{GLB}(A!!, B!!)$?
- if A is nullable and B is not, $\texttt{GLB}(A, B) = \texttt{GLB}(A!!, B)$
- if $A = T\langle K_{A,1}, \ldots, K_{A,n}\rangle$ and $B = T\langle K_{B,1}, \ldots, K_{B,n}\rangle$, $\texttt{GLB}(A, B) = T\langle \phi(K_{A,1}, K_{B,1}), \ldots, \phi(K_{A,n}, K_{B,n})\rangle$, where $\phi(X, Y)$ is defined as follows:
  - $\phi(\texttt{inv}\ X, \texttt{inv}\ X) = X$
  - $\phi(\texttt{out}\ X, \texttt{out}\ Y) = \texttt{out}\ \texttt{GLB}(X, Y)$
  - $\phi(\texttt{out}\ X, \texttt{inv}\ Y) = \phi(\texttt{out}\ X, \texttt{out}\ Y)$
  - $\phi(\texttt{out}\ X, \texttt{in}\ Y) = \star$
  - $\phi(\texttt{inv}\ X, \texttt{out}\ Y) = \phi(\texttt{out}\ X, \texttt{out}\ Y)$
  - $\phi(\texttt{inv}\ X, \texttt{inv}\ Y) = \phi(\texttt{out}\ X, \texttt{out}\ Y)$
  - $\phi(\texttt{inv}\ X, \texttt{in}\ Y) = \phi(\texttt{out}\ X, \texttt{out kotlin.Any?}) = \texttt{out kotlin.Any?}$
  - $\phi(\texttt{in}\ X, \texttt{out}\ Y) = \star$

  - $\phi(\texttt{in}\ X, \texttt{inv}\ Y) = \phi(\texttt{out kotlin.Any?}, \texttt{out}\ Y) = \texttt{out kotlin.Any?}$
  - $\phi(\texttt{in}\ X, \texttt{in}\ Y) = \texttt{in LUB}(X, Y)$
  - TODO(we may also choose the `in` projection for `inv` parameters, do we wanna do it though?)

- if $A = (L_A..U_A)$ and $B = (L_B..U_B)$, $\texttt{GLB}(A, B) = (\texttt{GLB}(L_A, L_B)..\texttt{GLB}(U_A, U_B))$
- if $A = (L_A..U_A)$ and B is not flexible, $\texttt{GLB}(A, B) = (\texttt{GLB}(L_A, B)..\texttt{GLB}(U_A, B))$

TODO(prettify formatting)

TODO(actual algorithm for computing GLB)

TODO(GLB for 3+ types)

TODO(what do we do if this procedure loops?)

Type approximation

TODO()

References

1. Ross Tate. “Mixed-site variance.” FOOL, 2013.
2. Ross Tate, Alan Leung, and Sorin Lerner. “Taming wildcards in Java’s type system.” PLDI, 2011.

TODO(the big TODO for the whole chapter: we need to clearly decide what kind of type system we want to specify: an algo-driven ts vs a full declarational ts, operation-based or relation-based. An example of the second distinction would be difference between $(A?)!!$ and $((A!!)?)!!$. Are they the same type? Are they different, but equivalent? Same goes for $(A..B)?$ vs $(A?..B?)$ and such.)

TODO(another big question is: do we want to formally prove all the different thing here?)

Built-in classifier types

- TODO: Move the whole section to type system?
- TODO: Move `kotlin.Unit` here?
- TODO: `Appendable/StringBuilder`? depends on how we plan to approach the interpolation expansion
- TODO: `{Builtin}Array` types?

As well as the types defined in the type system section, Kotlin defines several built-in classifier types that are important for the rest of this document. These have their own declarations in the standard library, but have special semantics in Kotlin.

> Note: this is not meant to declare all the types available in the standard library, for this please refer to the standard library documentation . (TODO: link?)

`kotlin.Boolean`

`kotlin.Boolean` is the boolean logic type of Kotlin, representing the value that may be either `true` or `false`. It is the type of boolean literals as well as the type returned or expected by some built-in Kotlin operators. For other traits of this type (such as the classes it inherits from, interfaces it may inherit from and its member functions) please refer to the standard library specification.

Built-in integer types

There are several built-in class types that represent signed integer numbers of different bit size. Kotlin does not have a built-in infinite-length integer number class. Kotlin also does not currently define any built-in unsigned integer number types . The signed integer number types are: (TODO: Kotlin 1.3 does)

- `kotlin.Int`
- `kotlin.Short`
- `kotlin.Byte`
- `kotlin.Long`

These types may or may not have different runtime representation. See your platform reference for details.

`kotlin.Int` is the type of integer numbers that is required to be able to hold at least the values in the range from -2^{31} to $2^{31}-1$. If an arithmetic operation on `kotlin.Int` results in arithmetic overflow or underflow, the result is undefined.

`kotlin.Short` is the type of integer numbers that is required to be able to hold at least the values in the range from -2^{15} to $2^{15}-1$. If an arithmetic operation on `kotlin.Short` results in arithmetic overflow or underflow, the result is undefined.

`kotlin.Byte` is the type of integer numbers that is required to be able to hold at least the values in the range from -2^{7} to $2^{7}-1$. If an arithmetic operation on `kotlin.Byte` results in arithmetic overflow or underflow, the result is undefined.

`kotlin.Long` is the type of integer numbers that is required to be able to hold at least the values in the range from -2^{63} to $2^{63}-1$. If an arithmetic operation on `kotlin.Long` results in arithmetic overflow or underflow, the result is undefined.

For other traits of these types (such as the classes they inherit from, interfaces they may inherit from and their member functions) please refer to the standard library specification.

Built-in floating point arithmetic types

There are two built-in class types that represent floating-point numbers: `kotlin.Float` and `kotlin.Double`. These types may or may not have different runtime representations. See your platform reference for details.

(TODO: link)

(TODO: link)

`kotlin.Float` is the type of floating-point number that is able to contain all the numbers as a IEEE754 single-precision binary floating number with the same precision. `kotlin.Double` is the type of floating-point number that is able to contain all the numbers as a IEEE754 double-precision binary floating number with the same precision.

TODO: or do they?

Platform specification may give a more thorough information on how these types are represented on a particular platform. For other traits of these types (such as the classes they inherit from, interfaces they may inherit from and their member functions) please refer to the standard library specification.

TODO: FP semantics are pretty hard, how much of that we want to put here?

`kotlin.Char`

`kotlin.Char` is the built-in class type that represents a single unicode symbol in UTF-16 character encoding. It is the type of character literals. For other traits of this type (such as the classes it inherits from, interfaces it may inherit from and its member functions) please refer to the standard library specification.

(TODO: link)

TODO: UTF-16 or UCS-2?

`kotlin.String`

`kotlin.String` is the built-in class type that represents a sequence of unicode symbol in UTF-16 character encoding. It is the type of the result of string interpolation. For other traits of this type (such as the classes it inherits from, interfaces it may inherit from and its member functions) please refer to the standard library specification.

(TODO: link)

TODO: UTF-16 or UCS-2?

Runtime type information

The *runtime type information* (RTTI) is the information about Kotlin types of values available from these values at runtime. RTTI affects the semantics of certain expressions, changing their evaluation depending on the amount of RTTI available for particular values, implementation, and platform:

- The type checking operator
- The cast expression, expecially the `as?` operator
- [Class literals][class literal] and the values they evaluate to

Runtime types are particular instances of RTTI for a particular value at runtime. These model a subset of the Kotlin type system. Namely, the runtime types are limited to classifier types, function types and a special case of `kotlin.Nothing?` which is the type of `null` reference and the only nullable runtime type. This includes the classifier types created by anonymous object literals. There is a slight distinction between a Kotlin type system type and its runtime counterpart:

- On some platforms, some particular types may have the same runtime type representation. This means that checking or casting values of these types works the same way as if they were the same type
- Generic types with the same classifier are not required to have different runtime representations. One cannot generally rely on them having the

> same representation outside of a particular platform. Platform specifications must clarify whether some or all types on these platforms have this feature.

RTTI is also the source of information for platform-specific *reflection* facilities in the standard library.

The types actual values may have are limited to class and object types and function types as well as `kotlin.Nothing?` for the `null` reference. `kotlin.Nothing` (not to be confused with its nullable variant `kotlin.Nothing?`) is special in the way that this type is never encountered as a runtime type even though it may have a platform-specific representation. The reason for this is that this type is used to signify non-existent values.

Runtime-available types

Runtime-available types are the types that can be guaranteed (during compilation) to have a concrete *runtime* counterpart. These include all the runtime types, their nullable variants as well as [`reified` type parameters][Reified type parameters], that are guaranteed to inline to a runtime type during type parameter substitution. Only runtime-available types may be passed (implicitly or explicitly) as substitutions to reified type paramters, used for type checks and safe casts. During these operations, the nullability of the type is checked using reference-equality to `null`, while the rest is performed by accessing the runtime type of a value and comparing it to the supplied runtime-available type.

For all generic types that are not expected to have RTTI for their generic arguments, only "raw" variants of generic types (denoted in code using the star-projected type notation or a special parameter-less notation) are runtime-available.

(TODO: link?)

> Note: one may say that classifier generics are *partially* runtime available due to them having information about only the classifier part of the type

Exception types must be runtime-available to enable type checks that the `catch` clause of `try`-expression performs.

Only non-nullable runtime types may be used in `class` literal expressions. These include reified type parameters with non-nullable upper bounds, as well as all classifier and function types.

TODO(Anything else?)

Scopes and identifiers

All the program code in Kotlin is logically divided into *scopes*. A scope is a syntactically-delimited region of code that constitutes a context in which entities and their names can be introduced. Scopes are nested, with entities introduced in outer scopes also available in the inner scopes. The top level of a Kotlin file is also a scope, containing all the scopes within the file.

All the scopes are divided into two categories: declaration scopes and statement scopes. These two kinds of scopes differ in how the identifiers in code refer to the values definied in the scopes.

Declaration scopes include:

- The top level scope of a normal Kotlin file (not script file);
- The bodies of classifier declarations;
- The bodies of object literals;
- TODO(Anything else?)

Statement scopes include:

- The top level scope of a Kotlin script file;
- Various scopes produced by control structure bodies of different expressions;
- The bodies of function declarations;
- The bodies of anonymous function literals;
- The bodies of getters and setters of properties;
- The bodies of constructors;
- The bodies of instance initialization blocks in class declarations;
- TODO(Anything else?)

All the declarations in a particular scope introduce new *bindings* of identifiers in this scope to their respective entities in the program. These entities may be types or values, where values may refer to objects, functions or properties (that may be delegated). Top-level scopes additionally allow to introduce such bindings using `import` directive from other top-level scopes.

In most situations, it is not allowed to bind several values to the same identifier in the same scope, but it is allowed to bind a value to an identifier already available in the scope through outer scopes or imports. An exception to this rule are function declarations, that, in addition to identifier bound to, also may differ by signature and allow definining several functions with the same name in the same scope. When calling functions a process called overloading resolution takes

(TODO: what's a signature?)

places that allows differentiating such functions. Overloading resolution also applies to properties if they are used as functions through `invoke`-convention, but it does not mean several properties with the same name may be defined in the same scope.

The main difference between declaration scopes and statement scopes is that names in the statement scope are bound in the order their declarations appear in it. It is not allowed to access a value through an identifier in the code that (syntactically) precedes the binding itself. On the contrary, in declaration scopes it is fully allowed, although initialization cycles may occur and need to be detected by the compiler. It also means that the statement scopes nested inside declaration scopes may access values declared after itself in the declaration scopes, but any values defined inside the statement scope must be accessed only after they are declared.

Example:

- In declaration scope:

```
// x refers to the property defined below even if there is another property
// called x in outer scope or imported
fun foo() { return x + 2; }
val x = 3;
```

- In statement scope:

```
// x either refers to other property defined in some outer scope or imported
// or it is a compile-time error
fun foo() { return x + 2; }
val x = 3;
```

  Note: please note that all the above is primarily applied to declarations, because declaration scopes do not allow standalone statements to appear in them

- TODO(qualified names?)
- TODO(extensions?)
- TODO(receivers)
- TODO(rewrite expressions and statements as references to this part)
- TODO(identifier lifetime & such)

Packages and imports

Any Kotlin project is structured into **packages**. A package may contain one or more Kotlin files and each file is related to the corresponding package using the *package header*. A file may contain only one (or zero) package headers, meaning that each file belongs to exactly one package.

***packageHeader*:**
['package' *identifier* [*semi*]]

Note: an absence of a package header in a file means that is belongs to the special *root package*

Note: Packages are different from modules. A module may contain many packages, while a single package can be spread across several modules.

The name of a package is a dot (.)-separated sequence of identifiers, introducing a package hierarchy. Unlike Java and some other languages, Kotlin does not restrict the package hierarchy to correspond directly to the folder structure of the project.

Note: this means that the hierarchy itself is only notational, not affecting the code in any way. It is strongly recommended, however, that the folder structure of the project does correspond to the package hierarchy.

Importing

Program entities declared in one package may be freely used in any file in the same package with the only restriction being module boundaries. In order to use an entity from a file belonging to a different package, the programmer must use *import directives*.

***importList*:**
{*importHeader*}

***importHeader*:**
'import' *identifier* [('.' '*') | *importAlias*] [*semi*]

***importAlias*:**
'as' *simpleIdentifier*

An import directive contains dot-separated *path* to an entity, as well as the name of the entity itself (the last argument of the navigation dot operator). A path may include not only the package the import is importing from, but also an object or a type (referring to companion object of this type). Any named declaration within that scope (that is, top-level scope of all files in the package or, in the object case, the object declararion scope) may be imported using their

names. There are two special kinds of imports: star-imports ending in an asterisk (`*`) and renaming imports employing the use of `as` operator. Star-imports import all the named entities inside the corresponding scope, but have weaker priority during resolution of functions and properties. Renaming imports work just as regular imports, but introduce the entity into current file with a name different from the name it has at declaration site.

Imports are file-based, meaning that if an entity is introduced into file A.kt belonging to package `kotlinx.foo`, it does not introduce this entity to all other files belonging to `kotlinx.foo`.

There are some packages that have all their entities *implicitly imported* into any Kotlin file, meaning one can access this entity without explicitly using import directives. One may, however, import this entities explicitly if they choose to. These are the following packages of the standard library:

- `kotlin`
- `kotlin.annotation`
- `kotlin.collections`
- `kotlin.comparisons`
- `kotlin.io`
- `kotlin.ranges`
- `kotlin.sequences`
- `kotlin.text`
- `kotlin.math`

Platform implementations may introduce additional implicitly imported packages, for example, adding standard platform functionality into Kotlin code.

> Note: an example of this would be `java.lang` package implicitly imported on the jvm platform

Importing certain entities may be disallowed by their [visibility modifiers][Visibility].

TODO(Clarify all this)

Modules

TODO(Here be The dragons)

Overloadable operators

TODO(rename this and all the refs to smth)

Some syntax forms in Kotlin are defined *by convention*, meaning that their semantics are defined through syntactic expansion of current syntax form into another syntax form. The expansion of a particular syntax form is a different piece of code usually defined in the terms of operator functions. Operator functions are function that are declared with a special keyword `operator` and are not different from normal functions when called normally, but allow themselves to be employed by syntactic expansion. Different platforms may add other criteria on whether a function may be considered a suitable candidate for operator convention.

Particular cases of definition by convention include:

- Arithmetic and comparison operators;
- Operator-form assignments;
- For-loop statements;
- Delegated properties;
- TODO(anything else?)

There are several common points among all the syntax forms defined using this mechanism:

- The expansions are hygenic: if they seemingly introduce new identifiers that were not present in original syntax, all such identifiers are not accessible outside the expansion and cannot clash with any other declarations in the program;
- The expressions captured by an expansion are using call-by-need evaluation strategy, meaning that they are evaluated only once during first usage specified in the expansion even if the expansion itself has more than one usage of such an expression;
- An expansion may lead to another expansion, following the same rules;
- All the new call expressions that are produced by expansion are only allowed to use operator functions.

For example, take the following declarations:

```
class A {
    operator fun inc(): A { ... }
}

object B {
    operator fun get(i: Int): A { ... }
    operator fun set(i: Int, value: A) { ... }
}

object C {
```

```
    operator fun get(i: Int): B { ... }
}
```

The expression `C[0][0]++` is expanded (see the Expressions section for details) using the following rules:

- First, the increment operator is expanded, resulting in:

  `C[0][0] = C[0][0].inc()`

- Second, the assigment to an indexing expression (produced by the previous expansion) is expanded, resulting in:

  `C[0].set(C[0].get(0).inc())`

- Third, the [indexing expression][Indexing expression] is expanded, resulting in:

  `C.get(0).set(C.get(0).get(0).inc())`

Although the resulting expression contains several invocations of the subexpression `C.get(0)`, it is evaluated only once, making this code roughly equivalent to:

```
val $tmp = C.get(0)
$tmp.set($tmp.get(0).inc())
```

TODO()

Declarations

Glossary

Entity
: A distinguishable part of a program

Path
: A sequence of names which identifies a program entity

Identifiers, names and paths

TODO(Explain paths)

Introduction

TODO(Examples)

Declarations in Kotlin are used to introduce entities (values, types, etc.); most declarations are *named*, i.e. they also assign an identifier to their own entity, however, some declarations may be *anonymous*.

Every declaration is accessible in a particular *scope*, which is dependent both on where the declaration is located and on the declaration itself.

Classifier declaration

***classDeclaration*:**
[*modifiers*]
('`class`' | '`interface`')
{*NL*}
simpleIdentifier
[{*NL*} *typeParameters*]
[{*NL*} *primaryConstructor*]
[{*NL*} '`:`' {*NL*} *delegationSpecifiers*]
[{*NL*} *typeConstraints*]
[({*NL*} *classBody*) | ({*NL*} *enumClassBody*)]

***objectDeclaration*:**
[*modifiers*]
'`object`'
{*NL*}
simpleIdentifier
[{*NL*} '`:`' {*NL*} *delegationSpecifiers*]
[{*NL*} *classBody*]

Classifier declarations introduce new types to the program, of the forms described here. There are three kinds of classifier declarations:

- class declarations;
- interface declarations;
- object declarations.

Class declaration

A simple class declaration consists of the following parts.

- name c;
- primary constructor declaration *ptor*;
- supertype specifiers $S_1, \ldots, S_s$;

- body b, which may include the following:
  - secondary constructor declarations $stor_1, \ldots, stor_c$;
  - instance initialization block $init$;
  - property declarations $prop_1, \ldots, prop_p$;
  - function declarations $md_1, \ldots, md_m$;
  - companion object declaration $companionObj$;
  - nested classifier declarations $nested$.

and creates a simple classifier type $c : S_1, \ldots, S_s$.

Supertype specifiers are used to create inheritance relation between the declared type and the specified supertype. You can use classes and interfaces as supertypes, but not objects.

It is allowed to inherit from a single class only, i.e., multiple class inheritance is not supported. Multiple interface inheritance is allowed.

Instance initialization block describes a block of code which should be executed during object creation.

Property and function declarations in the class body introduce their respective entities in this class' scope, meaning they are available only on an entity of the corresponding class.

Companion object declaration `companion object CO { ... }` for class `C` introduces an object, which is available under this class' name or under the path `C.CO`. Companion object name may be omitted, in which case it is considered to be equal to `Companion`.

Nested classifier declarations introduce new classifiers, available under this class' path for all nested classifiers except for inner classes. Inner classes are available only on the corresponding class' entities. Further details are available [here][Inner and nested classes].

TODO(Examples)

A parameterized class declaration consists of the following parts.

- name c
- type parameter list $T_1, \ldots, T_m$
- primary constructor declaration $ptor$
- supertype specifiers $S_1, \ldots, S_s$
- body b, which may include the following
  - secondary constructor declarations $stor_1, \ldots, stor_c$
  - instance initialization block $init$
  - property declarations $prop_1, \ldots, prop_p$
  - function declarations $md_1, \ldots, md_m$
  - companion object declaration $companionObj$
  - nested classifier declarations $nested$

and extends the rules for a simple class declaration w.r.t. type parameter list. Further details are described here.

Constructor declaration

There are two types of class constructors in Kotlin: primary and secondary.

A primary constructor is a concise way of describing class properties together with constructor parameters, and has the following form

$$ptor : (p_1, \ldots, p_n)$$

where each of p_i may be one of the following:

- regular constructor parameter $name : type$;
- read-only property constructor parameter `val`$name : type$;
- mutable property constructor parameter `val`$name : type$.

Property constructor parameters, together with being regular constructor parameters, also declare class properties of the same name and type. One can consider them to have the following syntactic expansion.

```
class Foo(i: Int, val d: Double, var s: String) : Super(i, d, s) {}

class Foo(i: Int, d_: Double, s_: String) : Super(i, d_, s_) {
  val d = d_
  var s = s_
}
```

When accessing property constructor parameters inside the class body, one works with their corresponding properties; however, when accessing them in the supertype specifier list (e.g., as an argument to a superclass constructor invocation), we see them as actual parameters, which cannot be changed.

If a class declaration has a primary constructor and also includes a class supertype specifier, that specifier must represent a valid invocation of the supertype constructor.

A secondary constructor describes an alternative way of creating a class instance and has only regular constructor parameters. If a class has a primary constructor, any secondary constructor must delegate to either the primary constructor or to another secondary constructor via `this(...)`.

If a class does not have a primary constructor, its secondary constructors must delegate to either the superclass constructor via `super(...)` (if the superclass is present in the supertype specifier list) or to another secondary constructor via `this(...)`. If the only superclass is `kotlin.Any`, delegation is optional.

In all cases, it is forbidden if two or more secondary constructors form a delegation loop.

TODO(elaborate this `this(...)` and `super(...)` business)

TODO(default values in constructors???)

Nested and inner classifiers

If a classifier declaration ND is *nested* in another classifier declaration PD, it creates a nested classifier type — a classifier type available under the path $PD.ND$. In all other aspects, nested classifiers are equivalent to regular ones.

Inner classes are a special kind of nested classifiers, which introduce types of objects associated (linked) with other (parent) objects. An inner class declaration ID nested in another classifier declaration PD may reference an *object* of type ID associated with it.

This association happens when instantiating an object of type ID, as its constructor may be invoked only when a receiver of type PD is available, and this receiver becomes associated with the new instantiated object of type ID.

TODO(...)

Inheritance delegation

In a classifier (an object or a class) C declaration any supertype I inheritance may be *delegated to* an arbitrary value v if:

- The supertype I is an interface type;
- v has type T such that $T <: I$.

The inheritance delegation uses a syntax similar to [property delegation][Property delegation] using the `by` keyword, but is specified in the classifier declaration header and is a very different concept. If inherited using delegation, each method M of I (whether they have a default implementation or not) is delegated to the corresponding method of v as if it was overriden in C with all the parameter values directly passed to the corresponding method in v, unless the body of C itself has a suitable override of M (see the method overriding section).

(TODO: link)

The particular means on how v is stored inside the classifier object is platform-defined.

Due to the initialization order of a classifier object, the expression used to construct v can not access any of the classifier object properties or methods excluding the parameters of the primary constructor.

TODO(. . .)

Data class declaration

A data class *dataClass* is a special kind of class, which represents a product type constructed from a number of data properties $(dp_1, \ldots, dp_m)$, described in its primary constructor. As such, it allows Kotlin to reduce the boilerplate and generate a number of additional data-relevant functions. Each one of these functions is generated if and only if a matching signature function is not present in the class body.

- `equals() / hashCode() / toString()` functions compliant with their contracts:
  - `equals(that)` returns true iff:
    - `that` has the same runtime type as `this`;
    - `this.prop.equals(that.prop)` returns `true` for every data property `prop`;
  - `hashCode()` returns different numbers for objects `A` and `B` if they do not equal by the generated `equals`;
  - `toString` returns a string representations which is guaranteed to include the class name along with all the data properties' string representations.
  - TODO(Be more specific?).
- A `copy()` function for shallow object copying with the following properties:
  - It has the same number of parameters as the primary constructor with the same names and types;
  - It calls the primary constructor with the corresponding parameters at the corresponding positions;
  - It has defaults for all the parameters defaulting to the value of the corresponding property in `this` object.
- A number of `componentN()` functions for destructive declaration:
  - For the data property at position N (**starting with 1**), the generated `component`N function has the same type as this property and returns the value of this property;
  - It has an `operator` modifier, allowing it to be used in [destructuring declarations][Destructuring declaration];
  - The number of these functions is the same as the number of data properties.

These generated declarations of `equals`, `hashCode` and `toString` may be overriden the same way they may be overriden in normal classes. The overriding version is preferred, as normally. In addition, for every other function, if any of the base types provide an open function with a matching signature, it is

automatically overriden by the generated function as if it was generated with an `override` modifier.

> Note: base classes may also have functions that are either not open or have a conflicting signature with the same function name. As expected, these cases result in override or overload conflicts the same way they would do with a normal class declaration.

All these functions consider only data properties $\{dp_i\}$; e.g., your data class may include regular property declarations in its body, however, they will *not* be considered in the `equals()` implementation or have a `componentN()` generated for them.

Data classes have the following restrictions:

- Data classes are final and cannot be inherited from;
- Data classes must have a primary constructor with only property constructor parameters, which become data properties for the data class;
- There must be at least one data property in the primary constructor.

Data class generation

TODO(Do we really need this?)

TODO(A more detailed explaination)

Enum class declaration

TODO(grammar reference)

TODO(Use “enumeration” instead of “enum”?)

Enum class is a special kind of class with the following properties:

- It has a number of predefined values that are declared in the class itself (*enum entries*);
- No other values of this class can be constructed;
- It implicitly inherits the built-in class `kotlin.Enum` (and cannot have any other base classes);
- It it implicitly final and cannot be inherited from;
- It has special syntax to accommodate for the properties described above.

Enum class body uses special kind of syntax (see grammar) to declare enum entries in addition to all other declarations inside the class body. Enum entries

have their own bodies that may contain their own declarations, similar to object declarations.

> Note: an enum class can have zero enum entries. This makes objects of this class impossible to construct.

In addition to this, every enum class has an implicit companion object declaration with the following member functions (in addition to the ones the object declaration specified explicitly has):

- `valueOf(value: String)` returning an object corresponding to the entry with the name equal to `value` parameter of the call;
- `values()` returning an array of all the possible enum values. Every invocation of this function returns a new array to disallow changing its contents.

  > Note: Kotlin standard library introduces another function to access all enum values for a specific enum class called `kotlin.enumValues<T>`. Please refer to the standard library documentation for details.

TODO(`kotlin.Comparable` generation?)

TODO(. . .)

Annotation class declaration

Annotations class is a special kind of class that is used to declare annotations. Annotation classes have the following properties:

- They cannot have any secondary constructors;
- All the primary constructor parameters must use the property syntax;
- They implicitly inherit `kotlin.Annotation` class (and cannot have any other base classes);
- They cannot implement interfaces;
- They are implicitly final and cannot be inherited from;
- They may not have any member functions, properties not declared in the primary constructor or any overriding declarations;
- They cannot have companion objects;
- They cannot have nested classes;
- The types of primary constructor parameters are limited to:
  - `kotlin.String`;
  - `kotlin.KClass`;
  - Built-in number types;
  - Other annotation types;
  - Arrays of any other allowed type.

Annotation classes cannot be constructed directly, but their primary constructors are used when specifying code annotations for other entities.

TODO(...)

Interface declaration

Interfaces differ from classes in that they cannot be directly instantiated in the program, they are meant as a way of describing a contract which should be satisfied by the interface's subtypes. In other aspects they are similar to classes, therefore we shall specify their declarations by specifying their differences from class declarations.

- An interface cannot have a class as its supertype;
- An interface cannot have a constructor;
- Interface properties cannot have initializers or backing fields;
- An interface cannot have inner classes (but can have nested classes and companion objects);
- An interface and all its members are implicitly open;
- All interface member properties and functions are implicitly public;
  - Trying to declare a non-public member property or function in an interface is an error.

TODO(Something else?)

Object declaration

Object declarations are used to support a singleton pattern and, thus, do two things at the same time. One, they (just like class declarations) introduce a new type to the program. Two, they create a singleton-like object of that type.

TODO(do we really need this ironic-ish statement about doing two things at the same time?)

Similarly to interfaces, we shall specify object declarations by highlighting their differences from class declarations.

- An object type cannot be used as a supertype for other types;
- An object cannot have a constructor;
- An object cannot have a companion object;
- An object may not have inner classes;
- An object cannot be parameterized, i.e., cannot have type parameters.

TODO(Something else?)

> Note: this section is about declaration of *named* objects. Kotlin also has a concept of *anonymous* objects, or object literals, which are similar to their named counterparts, but are expressions rather than declarations and, as such, are described in the corresponding section.

Classifier initialization

When creating a class or object instance via one of its constructors *ctor*, it is initialized in a particular order, which we describe here.

First, a supertype constructor corresponding to *ctor* is called with its respective parameters.

- If *ctor* is a primary constructor, a corresponding supertype constructor is the one from the supertype specifier list;
- If *ctor* is a secondary constructor, a corresponding supertype constructor is the one ending the constructor delegation chain of *ctor*;
- If an explicit supertype constructor is not available, `Any()` is implicitly used.

After the supertype initialization is done, we continue the initialization by processing each inner declaration in its body, *in the order of their inclusion in the body*. If any initialization step creates a loop, it is considered an undefined behavior.

When a classifier type is initialized using a particular secondary constructor *ctor* delegated to primary constructor *pctor* which, in turn, is delegated to the superclass constructor *sctor*, the following happens, in this order:

- *pctor* is invoked using the specified parameters, initializing all the properties declared by its property parameters in the order they appear in the constructor declaration;
- The superclass object (if any) is initialized as if created by invoking *sctor* with the specified parameters;
- Interface delegation expressions (if any) are invoked and the result of each is stored in the object to allow for interface delegation, in the order of appearance of delegation declarations in the classifier header;
- All the properties' initialization code as well as all the initialization blocks in the class body get initialized in the order of appearance in the class body;
- *ctor* body is invoked using the specified parameters.

  > Note: this means that if an `init`-block appears between two property declarations in the class body, its body is invoked between the initialization code of these two properties.

This order stays the same if any of the entities involved are omitted, omitting the corresponding step (e.g. if there is no primary constructor, it is not invoked,

and if the object is created using primary constructor, the body of the secondary one is not invoked, etc.), but performing all others. If any of the properties of the object are accessed before they are initialized in this order (for example, if a method called in an initialization block accesses a property that is mention after the block), the value of the property is undefined.

> Note: this can happen if a property is captured in a lambda expression that is used in some way during other initialization phases

TODO(This needs thorough testing)

Function declaration

***functionDeclaration*:**

[*modifiers*]
`'fun'`
[{*NL*} *typeParameters*]
[{*NL*} *receiverType* {*NL*} `'.'`]
{*NL*}
simpleIdentifier
{*NL*}
functionValueParameters
[{*NL*} `':'` {*NL*} *type*]
[{*NL*} *typeConstraints*]
[{*NL*} *functionBody*]

***functionBody*:**

block
| (`'='` {*NL*} *expression*)

Function declarations assign names to functions — blocks of code which may be called by passing them a number of arguments. Functions have special *function types* which are covered in more detail here.

A simple function declaration consists of four main parts:

- name f
- parameter list $(p_1 : P_1 = v_1, \ldots, p_n : P_n = v_n)$
- return type R
- body b

and creates a function type $f : (P_1, \ldots, P_n) \to R$.

Parameter list $(p_1 : P_1 = v_1, \ldots, p_n : P_n = v_n)$ describes function parameters — inputs needed to execute the declared function. Each parameter $p_i : P_i = v_i$ introduces p_i as a name of value with type P_i available inside function body b; therefore, parameters are final and cannot be changed inside the function. A function may have zero or more parameters.

A parameter may include a default value v_i, which is used if the corresponding argument is not specified in function invocation; v_i should be an expression which evaluates to type $V <: P_i$.

Return type R is optional, if function body b is present and may be inferred to have a valid type $B : B \not\equiv kotlin.Nothing$, in which case $R \equiv B$. In other cases return type R must be specified explicitly.

> As type $kotlin.Nothing$ has a special meaning in Kotlin type system, it must be specified explicitly, to avoid spurious $kotlin.Nothing$ function return types.

Function body b is optional; if it is ommited, a function declaration creates an *abstract* function, which does not have an implementation. This is allowed only inside an abstract classifier declaration. If a function body b is present, it should evaluate to type B which should satisfy $B <: R$.

TODO: `expect` and `external` functions also do not have implementations

A parameterized function declaration consists of five main parts.

- name f
- type parameter list $T_1, \ldots, T_m$
- parameter list $(p_1 : P_1 = v_1, \ldots, p_n : P_n = v_n)$
- return type R
- body b

and extends the rules for a simple function declaration w.r.t. type parameter list. Further details are described here.

Named, positional and default parameters

Kotlin supports *named* parameters out-of-the-box, meaning one can bind an argument to a parameter in function invocation not by its position, but by its name, which is equal to the argument name.

```
fun bar(a: Int, b: Double, s: String): Double = a + b + s.toDouble()

fun main(args: Array<String>) {
    println(bar(b = 42.0, a = 5, s = "13"))
}
```

TODO(Argument names are resolved in compile time)

If one wants to mix named and positional arguments, the argument list must conform to the following form: $P_1, \ldots, P_M, N_1, \ldots, N_Q$, where P_i is a positional argument, N_j is a named argument; i.e., positional arguments must precede all of the named ones.

Kotlin also supports *default* parameters — parameters which have a default value used in function invocation, if the corresponding argument is missing. Note that default parameters cannot be used to provide a value for positional argument *in the middle* of the positional argument list; allowing this would create an ambiguity of which argument for position i is the correct one: explicit one provided by the developer or implicit one from the default value.

```
fun bar(a: Int = 1, b: Double = 42.0, s: String = "Hello"): Double =
    a + b + s.toDouble()

fun main(args: Array<String>) {
    // Valid call, all default parameters used
    println(bar())
    // Valid call, defaults for `b` and `s` used
    println(bar(2))
    // Valid call, default for `b` used
    println(bar(2, s = "Me"))

    // Invalid call, default for `b` cannot be used
    println(bar(2, "Me"))
}
```

In summary, argument list should have the following form:

- Zero or more positional arguments;
- Zero or more named arguments.

Missing arguments are bound to their default values, if they exist.

Variable length parameters

One of the parameters may be designated as being variable length (aka *vararg*). A parameter list $(p_1, \ldots, \text{vararg } p_i : P_i = v_i, \ldots, p_n)$ means a function may be called with any number of arguments in the i-th position. These arguments are represented inside function body b as a value p_i of type, which is the result of *array type specialization* of type `Array<out`P_i`>`.

If a variable length parameter is not last in the parameter list, all subsequent arguments in the function invocation should be specified as named arguments.

If a variable length parameter has a default value, it should be an expression which evaluates to a value of type, which is the result of *array type specialization* of type `Array<out`P_i`>`.

An array of type `Array<Q>` <: `ATS(Array<out`P_i`>)` may be *unpacked* to a variable length parameter in function invocation using [spread operator][Spread operator]; in this case array elements are considered to be separate arguments in the variable length parameter position.

> Note: this means that, for variable length parameters corresponding to specialized array types, unpacking is possible only for these specialized versions; for a variable length parameter of type `Int`, for example, unpacking is valid only for `IntArray`, and not for `Array<Int>`.

A function invocation may include several spread operator expressions corresponding to the vararg parameter.

Extension function declaration

An *extension function declaration* is similar to a standard function declaration, but introduces an additional special function parameter, the *receiver parameter*. This parameter is designated by specifying the receiver type (the type before `.` in function name), which becomes the type of this receiver parameter. This parameter is not named and must always be supplied (either explicitly or implicitly), e.g. it cannot be a variable-argument parameter, have a default value, etc.

Calling such a function is special because the receiver parameter is not supplied as an argument of the call, but as the *receiver* of the call, be it implicit or explicit. This parameter is available inside the scope of the function as the implicit receiver or `this`-expression, while nested scopes may introduce additional receivers that take precedence over this one. See the receiver section for details. This receiver is also available (as usual) in nested scope using labeled `this` syntax using the name of the declared function as the label.

For more information on how a particular receiver for each call is chosen, please refer to the overloading section.

> Note: when declaring extension functions inside classifier declarations, this receiver takes precedence over the classifier object, which is usually the current receiver inside nested functions

For all other purposes, extension functions are not different from non-extension functions.

Examples:

```
fun Int.foo() { println(this + 1) } // this has type Int

fun main(args: Array<String>) {
    2.foo() // prints "3"
}

class Bar {
    fun foo() { println(this) } // this has type Bar
    fun Int.foo() { println(this) } // this has type Int
}
```

Property declaration

***propertyDeclaration*:**
[*modifiers*]
('val' | 'var')
[{*NL*} *typeParameters*]
[{*NL*} *receiverType* {*NL*} '.']
({*NL*} (*multiVariableDeclaration* | *variableDeclaration*))
[{*NL*} *typeConstraints*]
[{*NL*} (('=' {*NL*} *expression*) | *propertyDelegate*)]
[(*NL* {*NL*}) ';']
{*NL*}
(([*getter*] [{*NL*} [*semi*] *setter*]) | ([*setter*] [{*NL*} [*semi*] *getter*]))

Property declarations are used to create read-only (`val`) or mutable (`var`) entities in their respective scope. Properties may also have custom getter or setter — functions which are used to read or write the property value.

Read-only property declaration

A read-only property declaration `val x: T = e` introduces `x` as a name of the result of `e`.

A read-only property declaration may include a custom getter in the form of

```
val x: T = e
    get() { ... }
```

in which case `x` is used as a synonym to the getter invocation. Both the right-hand value `e`, the type `T` and the getter are optional, however, at least one of them must be specified. More so, if both the type of `e` and the return type of the getter cannot be inferred (or, in case of the getter, specified explicitely), the type `T` must be specified explicitly. In case both `e` and `T` are specified, the type of `e` must be a subtype of `T` (see subtyping for more details).

TODO: we never actually say how getters are similar/different to normal functions and, henceworth, how the inference works

The initializer expression `e`, if given, serves as the starting value for the property backing field (see getters and setters section for details) and is evaluated when the property is created. Properties that are not allowed to have backing fields (see getters and setters section for details) are also not allowed to have initializer expressions.

> Note: although a property with an initializer expression looks similar to an assignment, it is different in several key ways: first, a read-only property cannot be assigned, but may have an initializer expression;

second, the initializer expression never invokes the property setter, but assigns the property backing field value directly.

Mutable property declaration

A mutable property declaration `var x: T = e` introduces `x` as a name of a mutable variable with type `T` and initial value equals to the result of `e`. The rules regarding the right-hand value `e` and the type `T` match those of a read-only property declaration.

A mutable property declaration may include a custom getter and/or custom setter in the form of

```
var x: T = e
    get(): TG { ... }
    set(value: TS) { ... }
```

in which case `x` is used as a synonym to the getter invocation when read from and to the setter invocation when written to.

Delegated property declaration

A delegated read-only property declaration `val x: T by e` introduces `x` as a name for the *delegation* result of property `x` to the entity `e`. One may view these properties as regular properties with a special *delegating* getters.

In case of a delegated read-only property, every access to such property (`x` in this case) becomes an [overloadable][Operator overloading] form which is expanded into the following:

```
e.getValue(thisRef, property)
```

where

- `e` is the delegating entity; the compiler needs to make sure that this is accessible in any place `x` is accessible;
- `getValue` is a suitable operator function available on `e`;
- `thisRef` is the receiver object for the property. This argument is `null` for local properties;
- `property` is an object of the type `kotlin.KProperty<*>` that contains information relevant to `x` (for example, its name, see standard library documentation for details).

A delegated mutable property declaration `var x: T by e` introduces `x` as a name of a mutable entity with type `T`, access to which is *delegated* to the entity `e`. As before, one may view these properties as regular properties with special *delegating* getters and setters.

Read access is handled the same way as for a delegated read-only property. Any write access to `x` (using, for example, an assignment operator `x = y`) becomes an overloadable form with the following expansion:

```
e.setValue(thisRef, property, y)
```

where

- `e` is the delegating entity; the compiler needs to make sure that this is accessible in any place `x` is accessible;
- `getValue` is a suitable operator function available on `e`;
- `thisRef` is the receiver object for the property. This argument is `null` for local properties;
- `property` is an object of the type `kotlin.KProperty<*>` that contains information relevant to `x` (for example, its name, see standard library documentation for details);
- `y` is the value `x` is assigned to. In case of complex assignments (see the assignment section), as they are all overloadable forms, first the assignment expansion is performed, and after that, the expansion of the delegated property using normal assignment.

An example on how the delegation expansion may be actually implemented by the compiler is as follows.

```
/*
 * Actual code
 */
class C {
    var prop: Type by DelegateExpression
}

/*
 * Expanded code
 */
class C {
    private val prop$delegate = DelegateExpression
    var prop: Type
        get() = prop$delegate.getValue(this, this::prop)
        set(value: Type) = prop$delegate.setValue(this, this::prop, value)
}
```

The type of a delegated property may be omitted at the declaration site, meaning that it may be inferred from the delegating function itself. If this type is omitted, it is inferred as if it was assigned the value of its expansion. If this inference fails, it is a compile-time error.

TODO(provideDelegate)

Local property declaration

If a property declaration is local, it creates a local entity which follows most of the same rules as the ones for regular property declarations. However, local property declarations cannot have custom getters or setters.

Local property declarations also support *destructive* declaration in the form of

```
val (a: T, b: U, c: V, ...) = e
```

which is a syntactic sugar for the following expansion

```
val a: T = e.component1()
val b: U = e.component2()
val c: V = e.component3()
...
```

where `componentN()` should be a valid operator function available on the result of `e`. Each individual component property follows the rules for regular local property declaration.

Getters and setters

As mentioned before, a property declaration may include a custom getter and/or custom setter (together called *accessors*) in the form of

```
var x: T = e
    get(): TG { ... }
    set(anyValidArgumentName: TS): RT { ... }
```

These functions have the following requirements

- $TG \equiv T$;
- $TS \equiv T$;
- $RT \equiv$ `kotlin.Unit`;
- Types TG, TS and RT are optional and may be omitted from the declaration;
- Read-only properties may have a custom getter, but not a custom setter;
- Mutable properties may have any combination of a custom getter and a custom setter
- Setter argument may have any valid identifier as argument name.

  Note: Regular coding convention recommends `value` as the name for the setter argument

One can also ommit the accessor body, in which case a *default* implementation is used (also known as default accessor).

```
var x: T = e
    get
    set
```

> This notation is usually used if you need to change some aspects of an accessor (i.e., its visibility) without changing the default implementation.

Getters and setters allow one to customize how the property is accessed, and may need access to the property's *backing field*, which is responsible for actually storing the property data. It is accessed via the special `field` property available inside accessor body, which follows these conventions

- For a property declaration of type `T`, `field` has the same type `T`
- `field` is read-only inside getter body
- `field` is mutable inside setter body

However, the backing field is created for a property only in the following cases

- A property has no custom accessors;
- A property has a default accessor;
- A property has a custom accessor, and it uses `field` property;
- A mutable property has a custom getter or setter, but not both/

In all other cases a property has no backing field. Properties without backing fields are not allowed to have initializer expressions.

Read/write access to the property is replaced with getter/setter invocation respectively.

Getters and setters allow for some modifiers available for function declarations (for example, they may be declared `inline`, see grammar for details).

Extension property declaration

An *extension property declaration* is similar to a standard property declaration, but, very much alike an extension function, introduces an additional parameter to the property called *the receiver parameter*. This is different from usual property declarations, that do not have any parameters. There are other differences from standard property declarations:

- Extension properties cannot have initializers;
- Extension properties cannot have backing fields;
- Extension properties cannot have default accessors.

> Note: informally, on can say that extension properties have no state of their own. Only properties that use other objects' storage facilities and/or uses constant data can be extension properties.

Aside from these differences, extension properties are similar to regular properties, but, when accessing such a property one always need to supply a *receiver*, implicit or explicit. Also, unlike regular properties, the type of the receiver must be a subtype of the receiver parameter, and the value that is supplied as the receiver is bound to the receiver parameter. For more information on how a particular receiver for each access is chosen, please refer to the overloading section.

The receiver parameter can be accessed inside getter and setter scopes of the property as the implicit receiver or `this`. It may also be accessed inside nested scopes using labeled `this` syntax using the name of the property declared as the label. For delegated properties, the value passed into the operator functions `getValue` and `setValue` as the receiver is the value of the receiver parameter, rather than the value of the outer classifier. This is also true for local extension properties: while regular local properties are passed `null` as the first argument of these operator functions, local extension properties are passed the value of the receiver argument instead.

> Note: when declaring extension properties inside classifier declarations, this receiver takes precedence over the classifier object, which is usually the current receiver inside nested properties

For all other purposes, extension properties are not different from non-extension properties.

Examples:

```
val Int.foo: Int get() = this + 1

fun main(args: Array<String>) {
    println(2.foo.foo) // prints "4"
}

class Bar {
    val foo get() = this // returns type Bar
    val Int.foo get() = this // returns type Int
}
```

TODO(More examples (delegation, at least))

Property initialization

All non-abstract properties must be definitely initialized before their first use. To guarantee this, Kotlin compiler uses a number of analyses which are described in more detail here.

TODO(maybe it makes more sense to write all the initialization business right here)

Constant properties

A property may be declared **constant**, meaning that its value is known during compilation, by using the special `const` modifier. In order to be declared `const`, a property must meet the following requirements:

- Its type is one of the following:
  - One of the the built-in integral types;
  - `kotlin.Boolean`;
  - `kotlin.Char`;
  - `kotlin.String`;
- It is declared in the top-level scope or inside [an object declaration][Object declarations];
- It has an initializer expression and this initializer expression may be evaluated in the compile-time. Integer literals and string interpolation expressions without evaluated expressions, as well as builtin arithmetic/comparison operations and string concatenation operations on those are such expressions, but it is implementation-defined which other expressions qualify for this;
- It does not have getters, setters or delegation specifiers.

Type alias

***typeAlias*:**
[*modifiers*]
`'typealias'`
{*NL*}
simpleIdentifier
[{*NL*} *typeParameters*]
{*NL*}
`'='`
{*NL*}
type

Type alias introduces an alternative name for the specified type and supports both simple and parameterized types. If type alias is parameterized, its type parameters must be unbounded. Another restriction is that recursive type aliases are forbidden — the type alias name cannot be used in its own right-hand side.

At the moment, Kotlin supports only top-level type aliases. The scope where it is accessible is defined by its [*visibility modifiers*][Visibility].

Declarations with type parameters

TODO()

Declaration modifiers

TODO(declaration scope)

TODO(`open`)

TODO(`abstract`)

TODO(`lateinit`)

TODO(`const`)

TODO(overriding vs overloading vs shadowing)

TODO(visibility)

Statements

TODO()

***statements*:**
[*statement* {*semis statement*} [*semis*]]

***statement*:**
{*label* | *annotation*} (*declaration* | *assignment* | *loopStatement* | *expression*)

Unlike some other languages, Kotlin does not explicitly distinguish between statements, expressions and declarations, i.e., expressions and declarations can be used in statement positions. This section focuses only on those statements that are *not* expressions or declarations. For information on those parts of Kotlin, please refer to the Expressions and Declarations sections of the specification.

> Example: Kotlin supports using conditionals both as expressions and as statements. As their use as expressions is more general, detailed information about conditionals is available in the Expressions section of the specification.

Assignments

***assignment*:**
(*directlyAssignableExpression* `'='` {*NL*} *expression*)
| (*assignableExpression assignmentAndOperator* {*NL*} *expression*)

***assignmentAndOperator*:**
`'+='`
| `'-='`
| `'*='`
| `'/='`
| `'%='`

An *assignment* is a statement that writes a new value to some program entity, denoted by its left-hand side. Both left-hand and right-hand sides of an assignment must be expressions, more so, there are several restrictions for the expression on the left-hand side.

For an expression to be *assignable*, i.e. be allowed to occur on the left-hand side of an assignment, it **must** be one of the following:

- an identifier referring to a mutable property;
- a navigation expression referring to a mutable property;
- an indexing expression.

TODO(switch to navigation paths when we have them?)

Note: Kotlin assignments **are not** expressions and cannot be used as such.

Simple assignments

A *simple assigment* is an assignment which uses the assign operator `=`. If the left-hand side of an assignment refers to a mutable property, a value of that property is changed when an assignment is evaluated, using the following rules (applied in order).

- If a property is delegated, the corresponding operator function `setValue` is called using the right-hand side expression as the `value` argument;
- If a property has a setter, it is called using the right-hand side expression as its argument;
- Otherwise, if a property is a mutable property, its value is changed to the evaluation result of the right-hand side expression.

If the left-hand side of an assignment refers to a mutable property through the usage of safe navigation operator (`?.`), the same rules apply to it, but only if the left-hand side of the navigation operator is not referentially equal to `null` reference, e.g.:

```
a?.b?.z?.x = y
```

is semantically the same as

```
val __tmp = a?.b?.z
if(__tmp !== null) __tmp.x = y
```

TODO(just use setters for everything?)

If the left-hand side of an assignment is an indexing expression, the whole statement is treated as an overloaded operator with the following expansion:

$A[B_1, B_2, B_3, \ldots, B_N] = C$ is the same as calling A.set$(B_1, B_2, B_3, \ldots, B_N, C)$ where `set` is a suitable operator function.

Operator assignments

An *operator assignment* is a combined-form assignment which involves one of the following operators: `+=`, `-=`, `*=`, `/=`, `%=`. All of these operators are overloadable operator functions with the following expansions (applied in order):

- A`+=`B is exactly the same as one of the following:
  - A`.plusAssign(`B`)` if a suitable `plusAssign` operator function exists and is available;
  - A=A`.plus(`B`)` if a suitable `plus` operator function exists and is available.
- A`-=`B is exactly the same as one of the following:
  - A`.minusAssign(`B`)` if a suitable `minusAssign` operator function exists and is available;
  - A=A`.minus(`B`)` if a suitable `minus` operator function exists and is available.
- A`*=`B is exactly the same as one of the following:
  - A`.timesAssign(`B`)` if a suitable `timesAssign` operator function exists and is available;
  - A=A`.times(`B`)` if a suitable `times` operator function exists and is available.
- A`/=`B is exactly the same as one of the following:
  - A`.divAssign(`B`)` if a suitable `divAssign` operator function exists and is available;
  - A=A`.div(`B`)` if a suitable `div` operator function exists and is available;
- A`%=`B is exactly the same as one of the following:
  - A`.remAssign(`B`)` if a suitable `remAssign` operator function exists and is available;
  - A=A`.rem(`B`)` if a suitable `rem` operator function exists and is available.

> Note: as of Kotlin version 1.2.31, there are additional overloadable functions for `%` called `mod`/`modAssign`, which are deprecated.

After the expansion, the resulting [function call expression][Function call expressions] or simple assignment is processed according to their corresponding rules.

> Note: although for most real-world use cases operators `++` and `--` are similar to operator assignments, in Kotlin they are expressions and are described in the corresponding section of this specification.

Loop statements

Loop statements describe an evaluation of a certain number of statements repeatedly until a *loop exit condition* applies.

***loopStatement*:**
forStatement
| *whileStatement*
| *doWhileStatement*

Loops are closely related to the semantics of jump expressions, as these expressions, namely `break` and `continue`, are only allowed in a body of a loop. Please refer to the corresponding sections for details.

While-loop statement

***whileStatement*:**
('`while`' {*NL*} '`(`' *expression* '`)`' {*NL*} *controlStructureBody*)
| ('`while`' {*NL*} '`(`' *expression* '`)`' {*NL*} '`;`')

A *while-loop statement* is similar to an `if` expression in that it also has a condition expression and a body consisting of zero or more statements. While-loop statement evaluating its body repeatedly for as long as its condition expression evaluates to true or a jump expression is evaluated to finish the loop.

> Note: this also means that the condition expression is evaluated before every evaluation of the body, including the first one.

The while-loop condition expression **must be a subtype** of `kotlin.Boolean`.

Do-while-loop statement

***doWhileStatement*:**
'`do`'
{*NL*}
[*controlStructureBody*]

{*NL*}
`'while'`
{*NL*}
`'('`
expression
`')'`

A *do-while-loop statement*, similarly to a while-loop statement, also describes a loop, with the following differences. First, it has a different syntax. Second, it evaluates the loop condition expression **after** evaluating the loop body.

> Note: this also means that the body is always evaluated at least once.

The do-while-loop condition expression **must be a subtype** of `kotlin.Boolean`.

For-loop statement

***forStatement*:**
`'for'`
{*NL*}
`'('`
{*annotation*}
(*variableDeclaration* | *multiVariableDeclaration*)
IN
expression
`')'`
{*NL*}
[*controlStructureBody*]

> Note: unlike most other languages, Kotlin does not have a free-form condition-based for loops. The only form of a for-loop available in Kotlin is the "foreach" loop, which iterates over lists, arrays and other data structures.

A *for-loop statement* is a special kind of loop statement used to iterate over some data structure viewed as an iterable collection of elements. A for-loop statement consists of a loop body, a **container expression** and an **iteration variable declaration**.

The for-loop is actually an overloadable syntax form with the following expansion:

`for(VarDecl in C) Body` is the same as

```
val __iterator = C.iterator()
while (__iterator.hasNext()) {
    val VarDecl = __iterator.next()
    <... all the statements from Body>
}
```

where `iterator`, `hasNext`, `next` are all suitable operator functions available in the current scope. `VarDecl` here may be a variable name or a set of variable name as per [destructuring variable declarations][Destructuring declarations].

> Note: the expansion is hygienic, i.e., the generated iterator variable never clashes with any other variable in the program and cannot be accessed outside the expansion.

TODO(What about iterator value life-time and such?)

Code blocks

***block*:**
'{'
{*NL*}
statements
{*NL*}
'}'

***statements*:**
[*statement* {*semis statement*} [*semis*]]

A *code block* is a sequence of zero or more statements between curly braces separated by newlines or/and semicolons. Evaluating a code block means evaluating all its statements in the order they appear inside of it.

> Note: Kotlin does **not** support code blocks as statements; a curly-braces code block in a statement position is a lambda literal.

A *last expression* of a code block is the last statement in it (if any) if and only if this statement is also an expression. The last expressions are important when defining functions and control structure expressions.

A code block is said to contain no last expression if it does not contain any statements or its last statement is not an expression (e.g., it is an assignment, a loop or a declaration).

> Note: you may consider the case of a missing last expression as if a synthetic last expression with no runtime semantics and type `kotlin.Unit` is introduced in its place.

A *control structure body* is either a single statement or a code block. A *last expression* of a control structure body `CSB` is either the last expression of a code block (if `CSB` is a code block) or the single statement itself (if `CSB` is an expression). If a control structure body is not a code block or an expression, it has no last expression.

> Note: this is equivalent to wrapping the single statement in a new synthetic code block.

In some contexts, a control structure body is expected to have a value and/or a type. The value of a control structure body is:

- the value of its last expression if it exists;
- the singleton `kotlin.Unit` object otherwise.

The type of a control structure body is the type of its value.

TODO

- Labels
- Are declarations statements or not?
  - In the current grammar, they are
- How expansions with new variables actually work

Expressions

Glossary

CSB
: Control structure body

Introduction

TODO()

An expression may be *used as a statement* or *used as an expression* depending on the context. As all expressions are valid statements, free expressions may be used as single statements or inside code blocks.

An expression is used as an expression, if it is encountered in any position where a statement is not allowed, for example, as an operand to an operator or as an immediate argument for a function call. An expression is used as a statement if it is encountered in any position where a statement is allowed.

Some expressions are only allowed to be used as statements, if certain restrictions are met; this may affect the semantics, the compile-time type information or/and the safety of these expressions.

TODO(strong/soft keywords?)

Constant literals

Constant literals are expressions which describe constant values. Every constant literal is defined to have a single standard library type, whichever it is defined to be on current platform. All constant literals are evaluated immediately.

Boolean literals

***BooleanLiteral*:**
`true` | `false`

Keywords `true` and `false` denote boolean literals of the same values. These are strong keywords which cannot be used as identifiers unless [escaped][Escaped identifiers]. Values `true` and `false` always have the type `kotlin.Boolean`.

Integer literals

***IntegerLiteral*:**
DecDigitNoZero {*DecDigitOrSeparator*} *DecDigit* | *DecDigit*
***HexLiteral*:**
`0` (`x`|`X`) *HexDigit* {*HexDigitOrSeparator*} *HexDigit*
| `0` (`x`|`X`) *HexDigit*
***BinLiteral*:**
`0` (`b`|`B`) *BinDigit* {*BinDigitOrSeparator*} *BinDigit*
| `0` (`b`|`B`) *BinDigit*
***DecDigitNoZero*:**
DecDigit - `0`
***DecDigitOrSeparator*:**
DecDigit | *Underscore*
***HexDigitOrSeparator*:**
HexDigit | *Underscore*
***BinDigitOrSeparator*:**
BinDigit | *Underscore*
***DecDigits*:**
DecDigit {*DecDigitOrSeparator*} *DecDigit* | *DecDigit*

Decimal integer literals

A sequence of decimal digit symbols (`0` though `9`) is a decimal integer literal. Digits may be separated by an underscore symbol, but no underscore can be placed before the first digit or after the last one.

> Note: unlike other languages, Kotlin does not support octal literals. Even more so, any decimal literal starting with digit `0` and containing more than 1 digit is not a valid decimal literal.

Hexadecimal integer literals

A sequence of hexadecimal digit symbols (`0` through `9`, `a` through `f`, `A` through `F`) prefixed by `0x` or `0X` is a hexadecimal integer literal. Digits may be separated by an underscore symbol, but no underscore can be placed before the first digit or after the last one.

Binary integer literals

A sequence of binary digit symbols (`0` or `1`) prefixed by `0b` or `0B` is a binary integer literal. Digits may be separated by an underscore symbol, but no underscore can be placed before the first digit or after the last one.

The types for integer literals

Any of the decimal, hexadecimal or binary literals may be suffixed by the long literal mark (symbol `L`). An integer literal with the long literal mark has type `kotlin.Long`. A literal without the mark has a special integer literal type dependent on the value of the literal:

- If the value is greater than maximum `kotlin.Long` value (see built-in integer types), it is an illegal integer literal and a compiler error;
- Otherwise, if the value is greater than maximum `kotlin.Int` value (see built-in integer types), it has type `kotlin.Long`;
- Otherwise, it has an integer literal type containing all the built-in integer types that are guaranteed to be able to represent this value.

  Note: for example, integer literal `0x01` has value 1 and therefore has type `LTS(kotlin.Byte, kotlin.Short, kotlin.Int, kotlin.Long)`. Integer literal `70000` has value 70000, which is not representable using types `kotlin.Byte` and `kotlin.Short` and therefore has type `LTS(kotlin.Int, kotlin.Long)`.

Real literals

***RealLiteral*:**
FloatLiteral | *DoubleLiteral*
***FloatLiteral*:**
DoubleLiteral (`f` | `F`) | *DecDigits* (`f` | `F`)
***DoubleLiteral*:**
[*DecDigits*] `.` *DecDigits* [*DoubleExponent*] | *DecDigits DoubleExponent*

A *real literal* consists of the following parts: the whole-number part, the decimal point (ASCII period character `.`), the fraction part and the exponent. Unlike other languages, Kotlin real literals may only be expressed in decimal numbers. A real literal may also be followed by a type suffix (`f` or `F`).

The exponent is an exponent mark (`e` or `E`) followed by an optionaly signed decimal integer (a sequence of decimal digits).

The whole-number part and the exponent part may be omitted. The fraction part may be omitted only together with the decimal point, if the whole-number part and either the exponent part or the type suffix are present. Unlike other languages, Kotlin does not support omitting the fraction part, but leaving the decimal point in.

The digits of the whole-number part or the fraction part or the exponent may be optionally separated by underscores, but an underscore may not be placed between, before, or after these parts. It also may not be placed before or after the exponent mark symbol.

A real literal without the type suffix has type `kotlin.Double`, a real literal with the type suffix has type `kotlin.Float`.

> Note: this means there is no special suffix associated with type `kotlin.Double`.

Character literals

***CharacterLiteral*:**
`'` (*EscapeSeq* | *<any character except CR, LF, ' and \>*) `'`
***EscapeSeq*:**
UnicodeCharacterLiteral | *EscapedCharacter*
***UnicodeCharacterLiteral*:**
`\ u` *HexDigit HexDigit HexDigit HexDigit*
***EscapedCharacter*:**
`\` (`t` | `b` | `r` | `n` | `'` | `"` | `\` | `$`)

A *character literal* defines a constant holding a unicode character value. A simply-formed character literal is any symbol between two single quotation marks (ASCII single quotation character `'`), excluding newline symbols (*CR* and *LF*), the single quotation mark itself and the escaping mark (ASCII backslash character \).

A character literal may also contain an escaped symbol of two kinds: a simple escaped symbol or a unicode codepoint. Simple escaped symbols include:

(TODO())
(TODO())
(TODO())
(TODO())
(TODO())
(TODO())

- `\t` — the unicode TAB symbol ;
- `\b` — the unicode BACKSPACE symbol ;
- `\r` — *CR*;
- `\n` — *LF*;
- `\'` — the unicode single quotation symbol ;
- `\"` — the unicode double quotation symbol ;
- `\\` — the unicode backslash symbol symbol ;
- `\$` — the unicode DOLLAR symbol .

A unicode codepoint escaped symbol is the symbol `\u` followed by exactly four hexadecimal digits. It represents the unicode symbol with the codepoint equal to the number represented by these four digits.

> Note: this means unicode codepoint escaped symbols support only unicode symbols in range from U+0000 to U+FFFF.

All character literals have type `kotlin.Char`.

String literals

Kotlin supports string interpolation which supersedes traditional string literals. For further details, please refer to the corresponding section.

Null literal

The keyword `null` denotes the **null reference**, which represents an absence of a value and is a valid value only for nullable types. Null reference has type `kotlin.Nothing?` and is, by definition, the only value of this type.

TODO(rearrange these sections)

Try-expression

***tryExpression*:**
`'try'` {*NL*} *block* ((({*NL*} *catchBlock* {{*NL*} *catchBlock*}) [{*NL*} *finallyBlock*]) | ({*NL*} *finallyBlock*))

***catchBlock*:**
`'catch'`
{*NL*}
`'('`
{*annotation*}
simpleIdentifier
`':'`
type
`')'`
{*NL*}
block

***finallyBlock*:**
`'finally'` {*NL*} *block*

A *try-expression* is an expression starting with the keyword `try`. It consists of a code block (*try body*) and one or more of the following kinds of blocks: zero or

more *catch blocks* and an optional *finally block*. A *catch block* starts with the soft keyword `catch` with a single *exception parameter*, which is followed by a code block. A *finally block* starts with the soft keyword `finally`, which is followed by a code block. A valid try-expression must have at least one catch or finally block.

The try-expression evaluation evaluates its body; if any statement in the try body throws an exception (of type E), this exception, rather than being immediately propagated up the call stack, is checked for a matching catch block. If a catch block of this try-expression has an exception parameter of type $T :> E$, this catch block is evaluated immediately after the exception is thrown and the exception itself is passed inside the catch block as the corresponding parameter. If there are several catch blocks which match the exception type, the first one is picked.

TODO(Exception handling?)

If there is a finally block, it is evaluated after the evaluation of all previous try-expression blocks, meaning:

- If no exception is thrown during the evaluation of the try body, no catch blocks are executed, the finally block is evaluated after the try body, and the program execution continues as normal.
- If an exception was thrown, and one of the catch blocks matched its type, the finally block is evaluated after the evaluation of the matching catch block.
- If an exception was thrown, but no catch block matched its type, the finally block is evaluated before propagating the exception up the call stack.

The value of the try-expression is the same as the value of the last expression of the try body (if no exception was thrown) or the value of the last expression of the matching catch block (if an exception was thrown and matched). All other situations mean that an exception is going to be propagated up the call stack, and the value of the try-expression becomes irrelevant.

> Note: as desribed, the finally block (if present) is executed regardless, but it has no effect on the value returned by the try-expression.

(TODO(not that simple)) The type of the try-expression is the least upper bound of the types of the last expressions of the try body and the last expressions of all the catch blocks .

> Note: these rules mean the try-expression always may be used as an expression, as it always has a corresponding result value.

Conditional expression

***ifExpression*:**

('`if`' {*NL*} '`(`' {*NL*} *expression* {*NL*} '`)`' {*NL*} (*controlStructureBody* |

';'))
| ('if' {*NL*} '(' {*NL*} *expression* {*NL*} ')' {*NL*} [*controlStructureBody*]
{*NL*} [';'] {*NL*} 'else' {*NL*} (*controlStructureBody* | ';'))

Conditional expressions use a boolean value of one expression (*condition*) to decide which of the two control structure bodies (*branches*) should be evaluated. If the condition evaluates to `true`, the first branch (the *true branch*) is evaluated if it is present, otherwise the second branch (the *false branch*) is evaluated if it is present.

> Note: this means the following branchless conditional expression, despite being of almost no practical use, is valid in Kotlin
>
> ```
> if (condition) else;
> ```

The value of the resulting expression is the same as the value of the chosen branch.

The type of the resulting expression is the least upper bound of the types of two branches , if both branches are present. If either of the branches are omitted, the resulting conditional expression has type `kotlin.Unit` and may used only as a statement.

(TODO(not that simple))

TODO(Examples?)

The type of the condition expression must be a subtype of `kotlin.Boolean`, otherwise it is an error.

> Note: when used as expressions, conditional expressions are special w.r.t. operator precedence: they have the highest priority (the same as for all primary expressions) when placed on the right side of any binary expression, but when placed on the left side, they have the lowest priority. For details, see Kotlin grammar.

When expression

***whenExpression*:**
'when'
{*NL*}
['(' *expression* ')']
{*NL*}
'{'
{*NL*}
{*whenEntry* {*NL*}}
{*NL*}
'}'

***whenEntry*:**
(*whenCondition* {{*NL*} `','` {*NL*} *whenCondition*} {*NL*} `'->'` {*NL*} *controlStructureBody* [*semi*])
| (`'else'` {*NL*} `'->'` {*NL*} *controlStructureBody* [*semi*])

***whenCondition*:**
expression
| *rangeTest*
| *typeTest*

***rangeTest*:**
inOperator {*NL*} *expression*

***typeTest*:**
isOperator {*NL*} *type*

When expression is similar to a conditional expression in that it allows one of several different control structure bodies (*cases*) to be evaluated, depending on some boolean conditions. The key difference is exactly that a when expressions may include several different conditions with their corresponding control structure bodies. When expression has two different forms: with bound value and without it.

When expression without bound value (the form where the expression enclosed in parentheses after the `when` keyword is absent) evaluates one of the different CSBs based on its condition from the *when entry*. Each when entry consists of a boolean *condition* (or a special `else` condition) and its corresponding CSB. When entries are checked and evaluated in their order of appearance. If the condition evaluates to `true`, the corresponding CSB is evaluated and the value of when expression is the same as the value of the CSB. All remaining conditions and expressions are not evaluated.

The `else` condition is a special condition which evaluates to `true` if none of the branches above it evaluated to `true`. The `else` condition **must** also be in the last when entry of when expression, otherwise it is a compile-time error.

> Note: informally, you can always replace the `else` condition with an always-`true` condition (e.g., boolean literal `true`) with no change to the resulting semantics.

When expression with bound value (the form where the expression enclosed in parentheses after the `when` keyword is present) are similar to the form without bound value, but use a different syntax for conditions. In fact, it supports three different condition forms:

- *Type test condition*: type checking operator followed by a type (`is T`). The resulting condition is a type check expression of the form `boundValue is T`.
- *Contains test condition*: containment operator followed by an expression (`in Expr`). The resulting condition is a containment check expression of

the form `boundValue in Expr`.
- *Any other expression* (`Expr`). The resulting condition is an equality check of the form `boundValue == Expr`.
- The `else` condition, which is a special condition which evaluates to `true` if none of the branches above it evaluated to `true`. The `else` condition **must** also be in the last when entry of when expression, otherwise it is a compile-time error.

  Note: the rule for "any other expression" means that if a when expression with bound value contains a boolean condition, this condition is **checked for equality** with the bound value, instead of being used directly for when entry selection.

TODO(Examples)

The type of the resulting expression is the least upper bound of the types of all its entries . If the when expression is not exhaustive, it has type `kotlin.Unit` and may used only as a statement. (TODO(not that simple))

Exhaustive when expressions

A when expression is called ***exhaustive*** if at least one of the following is true:

- It has an `else` entry;
- It has a bound value and at least one of the following is true:
  - The bound expression is of type `kotlin.Boolean` and the conditions contain both:
    - A [constant expression][Constant expressions] evaluating to `true`;
    - A [constant expression][Constant expressions] evaluating to `false`;
  - The bound expression is of a [`sealed class`][Sealed classes] type and all of its subtypes are covered using type test conditions in this expression. This should include checks for all direct subtypes of this sealed class. If any of the direct subtypes is also a sealed class, there should either be a check for this subtype or all its subtypes should be covered;
  - The bound expression is of an `enum class` type and all its enumerated values are checked for equality using constant expression;
  - The bound expression is of a nullable type T? and one of the cases above is met for its non-nullable counterpart T together with another condition which checks the bound value for equality with `null`.

TODO(Equality check with object behaves kinda like a type check. Or not.)

Note: informally, an exhaustive when expression is guaranteed to evaluate one of its CSBs regardless of the specific when conditions.

Logical disjunction expression

***disjunction*:**
conjunction {{*NL*} `'||'` {*NL*} *conjunction*}

Operator symbol `||` performs logical disjunction over two values of type `kotlin.Boolean`. This operator is **lazy**, meaning that it does not evaluate the right hand side argument unless the left hand side argument evaluated to `false`.

Both operands of a logical disjunction expression must have a type which is a subtype of `kotlin.Boolean`, otherwise it is a type error. The type of logical disjunction expression is `kotlin.Boolean`.

TODO(Types of errors? Compile-time, type, run-time, whatever?)

Logical conjunction expression

***conjunction*:**
equality {{*NL*} `'&&'` {*NL*} *equality*}

Operator symbol `&&` performs logical conjunction over two values of type `kotlin.Boolean`. This operator is **lazy**, meaning that it does not evaluate the right hand side argument unless the left hand side argument evaluated to `true`.

Both operands of a logical conjunction expression must have a type which is a subtype of `kotlin.Boolean`, otherwise it is a type error. The type of logical disjunction expression is `kotlin.Boolean`.

Equality expressions

***equality*:**
comparison {*equalityOperator* {*NL*} *comparison*}

***equalityOperator*:**
`'!='`
| `'!=='`
| `'=='`
| `'==='`

Equality expressions are binary expressions involving equality operators. There are two kinds of equality operators: *reference equality operators* and *value equality operators*.

Reference equality expressions

Reference equality expressions are binary expressions which use reference equality operators: `===` and `!==`. These expressions check if two values are equal (`===`) or non-equal (`!==`) *by reference*: two values are equal by reference if and only if they represent the same runtime value.

For special values created without explicit constructor calls, notably, the constant literals and constant expressions composed of those literals, the following holds:

- If these values are non-equal by value, they are also non-equal by reference;
- Any instance of the null reference `null` is equal by reference to any other instance of the null reference;
- Otherwise, equality by reference is implementation-defined and must not be used as a means of comparing such values.

Reference equality expressions always have type `kotlin.Boolean`.

Value equality expressions

Value equality expressions are binary expressions which use value equality operators: `==` and `!=`. These operators are overloadable with the following expansion:

- `A == B` is exactly the same as `A?.equals(B) ?: (B === null)` where `equals` is a valid operator function available in the current scope;
- `A != B` is exactly the same as `!(A?.equals(B) ?: (B === null))` where `equals` is a valid operator function available in the current scope.

  Note: `kotlin.Any` type has a built-in open operator member function `equals`, meaning there is always at least one available overloading candidate for any value equality expression.

Value equality expressions always have type `kotlin.Boolean`. If the corresponding operator function `equals` has a different return type, it is a compile-time error.

Comparison expressions

***comparison*:**
infixOperation [*comparisonOperator* {*NL*} *infixOperation*]

***comparisonOperator*:**
```
'<'
| '>'
| '<='
| '>='
```

Comparison expressions are binary expressions which use the comparison operators: `<`, `>`, `<=` and `>=`. These operators are overloadable with the following expansion:

- `A < B` is exactly the same as `A.compareTo(B) [<] 0`
- `A > B` is exactly the same as `0 [<] A.compareTo(B)`
- `A <= B` is exactly the same as `!(A.compareTo(B) [<] 0)`
- `A >= B` is exactly the same as `!(0 [<] A.compareTo(B))`

where `compareTo` is a valid operator function available in the current scope and `[<]` (read "boxed less") is a special operator unavailable in user-side Kotlin which performs integer "less-than" comparison of two integer numbers.

The `compareTo` operator function must have a return type `kotlin.Int`, otherwise it is a compile-time error.

All comparison expressions always have type `kotlin.Boolean`.

Type-checking and containment-checking expressions

***infixOperation*:**
elvisExpression {(*inOperator* {*NL*} *elvisExpression*) | (*isOperator* {*NL*} *type*)}

***inOperator*:**
`'in'`
| *NOT_IN*

***isOperator*:**
`'is'`
| *NOT_IS*

Type-checking expression

A type-checking expression uses a type-checking operator `is` or `!is` and has an expression E as a left-hand side operand and a type name T as a right-hand side operand. The type T must be runtime-available, otherwise it is a compiler error. A type-checking expression checks whether the runtime type of E is a subtype of T for `is` operator, or not a subtype of T for `!is` operator.

Type-checking expression always has type `kotlin.Boolean`.

> Note: the expression `null is T?` for any type `T` always evaluates to `true`, as the type of the left-hand side (`null`) is `kotlin.Nothing?`, which is a subtype of any nullable type `T?`.
>
> Note: type-checking expressions may create smart casts, for further details, refer to the corresponding section.

Containment-checking expression

A *containment-checking expression* is a binary expression which uses a containment operator `in` or `!in`. These operators are overloadable with the following expansion:

- `A in B` is exactly the same as `B.contains(A)`;
- `A !in B` is exactly the same as `!(B.contains(A))`.

where `contains` is a valid operator function available in the current scope.

> Note: this means that, contrary to the order of appearance in the code, the right-hand side expression of a containment-checking expression is evaluated before its left-hand side expression

The `contains` function must have a return type `kotlin.Boolean`, otherwise it is a compile-time error. Containment-checking expressions always have type `kotlin.Boolean`.

Elvis operator expression

***elvisExpression*:**
infixFunctionCall {{*NL*} *elvis* {*NL*} *infixFunctionCall*}

An *elvis operator expression* is a binary expression which uses an elvis operator (`?:`). It checks whether the left-hand side expression is reference equal to `null`, and, if it is, evaluates and return the right-hand side expression.

This operator is **lazy**, meaning that if the left-hand side expression is not reference equal to `null`, the right-hand side expression is not evaluated.

The type of elvis operator expression is the least upper bound of the non-nullable variant of the type of the left-hand side expression and the type of the right-hand side expression.

TODO(not that simple either)

Range expression

***rangeExpression*:**
additiveExpression {`'..'` {*NL*} *additiveExpression*}

A *range expression* is a binary expression which uses a range operator `...` It is an overloadable operator with the following expansion:

- `A..B` is exactly the same as `A.rangeTo(B)`

where `rangeTo` is a valid operator function available in the current scope.

The return type of this function is not restricted. A range expression has the same type as the return type of the corresponding `rangeTo` overload variant.

Additive expression

***additiveExpression*:**
multiplicativeExpression {*additiveOperator* {*NL*} *multiplicativeExpression*}

***additiveOperator*:**
`'+'`
| `'-'`

An *additive expression* is a binary expression which uses the addition (`+`) or subtraction (`-`) operators. These are overloadable operators with the following expansions:

- `A + B` is exactly the same as `A.plus(B)`
- `A - B` is exactly the same as `A.minus(B)`

where `plus` or `minus` is a valid operator function available in the current scope.

The return type of these functions is not restricted. An additive expression has the same type as the return type of the corresponding operator function overload variant.

Multiplicative expression

***multiplicativeExpression*:**
asExpression {*multiplicativeOperator* {*NL*} *asExpression*}

***multiplicativeOperator*:**
`'*'`
| `'/'`
| `'%'`

A *multiplicative expression* is a binary expression which uses the multiplication (`*`), division (`/`) or remainder (`%`) operators. These are overloadable operators with the following expansions:

- `A * B` is exactly the same as `A.times(B)`
- `A / B` is exactly the same as `A.div(B)`
- `A % B` is exactly the same as `A.rem(B)`

where `times`, `div`, `rem` is a valid operator function available in the current scope.

> Note: as of Kotlin version 1.2.31, there exists an additional overloadable operator for `%` called `mod`, which is deprecated.

The return type of these functions is not restricted. A multiplicative expression has the same type as the return type of the corresponding operator function overload variant.

Cast expression

***asExpression*:**
prefixUnaryExpression [{*NL*} *asOperator* {*NL*} *type*]

***asOperator*:**
`'as'`
| `'as?'`

A *cast expression* is a binary expression which uses the cast operators `as` or `as?` and has the form `E as/as? T`, where E is an expression and T is a type name.

An **`as` cast expression** `E as T` is called *a unchecked cast* expression. This expression perform a runtime check whether the runtime type of E is a subtype of T and throws an exception otherwise. If type T is a runtime-available type without generic parameters, then this exception is thrown immediately when evaluating the cast expression, otherwise it is platform-dependent whether an exception is thrown at this point.

TODO(We need to sort out undefined/implementation-defined/platform-defined)

> Note: even if the exception is not thrown when evaluating the cast expression, it is guaranteed to be thrown later when its result is used with any runtime-available type.

An unchecked cast expression result always has the same type as the type T specified in the expression.

An **`as?` cast expression** `E as? T` is called *a checked cast* expression. This expression is similar to the unchecked cast expression in that it also does a runtime type check, but does not throw an exception if the types do not match, it returns `null` instead. If type T is not a runtime-available type, then the check is not performed and `null` is never returned, leading to potential runtime errors later in the program execution. This situation should be reported as a compile-time warning.

> Note: if type T is a runtime-available type **with** generic parameters, type parameters are **not** checked w.r.t. subtyping. This is another porentially erroneous situation, which should be reported as a compile-time warning.

The checked cast expression type is the nullable variant of the type T.

Note: cast expressions may create smart casts, for further details, refer to the corresponding section.

Prefix expressions

***prefixUnaryExpression*:**
{*unaryPrefix*} *postfixUnaryExpression*

***unaryPrefix*:**
annotation
| *label*
| (*prefixUnaryOperator* {*NL*})

***prefixUnaryOperator*:**
`'++'`
| `'--'`
| `'-'`
| `'+'`
| *excl*

Annotated and labeled expression

Any expression in Kotlin may be prefixed with any number of annotations and [labels][Labels]. These do not change the value of the expression and can be used by external tools and for implementing platform-dependent features.

Prefix increment expression

A *prefix increment* expression is an expression which uses the prefix form of operator `++`. It is an overloadable operator with the following expansion:

- `++A` is exactly the same as `A = A.inc(); A` where `inc` is a valid operator function available in the current scope.

  Note: informally, `++A` assigns the result of `A.inc()` to `A` and then returns `A` as the result.

For a prefix increment expression `++A` expression `A` must be an assignable expression. Otherwise, it is a compile-time error.

A prefix increment expression has the same type as the return type of the corresponding `inc` overload variant.

Note: as the result of `inc` is assigned to `A`, the return type of `inc` must be a subtype of `A`.

Prefix decrement expression

A *prefix decrement* expression is an expression which uses the prefix form of operator `--`. It is an overloadable operator with the following expansion:

- `--A` is exactly the same as `A = A.dec(); A` where `dec` is a valid operator function available in the current scope.

  Note: informally, `--A` assigns the result of `A.dec()` to `A` and then returns `A` as the result.

For a prefix decrement expression `--A` expression `A` must be an assignable expression. Otherwise, it is a compile-time error.

A prefix decrement expression has the same type as the return type of the corresponding `dec` overload variant.

> Note: as the result of `dec` is assigned to `A`, the return type of `dec` must be a subtype of `A`.

Unary minus expression

An *unary minus* expression is an expression which uses the prefix form of operator `-`. It is an overloadable operator with the following expansion:

- `-A` is exactly the same as `A.unaryMinus()` where `unaryMinus` is a valid operator function available in the current scope.

No additional restrictions apply.

Unary plus expression

An *unary plus* expression is an expression which uses the prefix form of operator `+`. It is an overloadable operator with the following expansion:

- `+A` is exactly the same as `A.unaryPlus()` where `unaryPlus` is a valid operator function available in the current scope.

No additional restrictions apply.

Logical not expression

A *logical not* expression is an expression which uses the prefix operator `!`. It is an overloadable operator with the following expansion:

- `!A` is exactly the same as `A.not()` where `not` is a valid operator function available in the current scope.

No additional restrictions apply.

Postfix operator expressions

***postfixUnaryExpression*:**
primaryExpression
| (*primaryExpression* (*postfixUnarySuffix* {*postfixUnarySuffix*}))

***postfixUnarySuffix*:**
postfixUnaryOperator
| *typeArguments*
| *callSuffix*
| *indexingSuffix*
| *navigationSuffix*

***postfixUnaryOperator*:**
`'++'`
| `'--'`
| (*EXCL_NO_WS excl*)

Postfix increment expression

A *postfix increment* expression is an expression which uses the postfix form of operator `++`. It is an overloadable operator with the following expansion:

- `A++` is exactly the same as `val $freshId = A; A = A.inc(); $freshId` where `inc` is a valid operator function available in the current scope.

  Note: informally, `A++` stores the value of A to a temporary variable, assigns the result of `A.inc()` to `A` and then returns the temporary variable as the result.

For a postfix increment expression `A++` expression `A` must be [assignable expressions][Assignable expressions]. Otherwise, it is a compile-time error.

A postfix increment expression has the same type as the return type of the corresponding `inc` overload variant.

> Note: as the result of `inc` is assigned to `A`, the return type of `inc` must be a subtype of `A`.

Postfix decrement expression

A *postfix decrement* expression is an expression which uses the postfix form of operator `--`. It is an overloadable operator with the following expansion:

- `A--` is exactly the same as `val $freshId = A; A = A.dec(); $freshId` where `dec` is a valid operator function available in the current scope.

> Note: informally, `A--` stores the value of A to a temporary variable, assigns the result of `A.dec()` to `A` and then returns the temporary variable as the result.

For a postfix decrement expression `A--` expression `A` must be [assignable expressions][Assignable expressions]. Otherwise, it is a compile-time error.

A postfix decrement expression has the same type as the return type of the corresponding `dec` overload variant.

> Note: as the result of `dec` is assigned to `A`, the return type of `dec` must be a subtype of `A`.

Not-null assertion expression

TODO(We need to define what "evaluation" is)

A *not-null assertion expression* is a postfix expression which uses an operator `!!`. For an expression `e!!`, if the type of `e` is nullable, a not-null assertion expression checks, whether the evaluation result of `e` is equal to `null` and, if it is, throws a runtime exception. If the evaluation result of `e` is not equal to `null`, the result of `e!!` is the evaluation result of `e`.

If the type of `e` is non-nullable, not-null assertion expression `e!!` has no effect.

The type of non-null assertion expression is the non-nullable variant of the type of `e`.

> Note: this type may be non-denotable in Kotlin and, as such, may be approximated in some situations with the help of type inference.

TODO(Example)

Indexing expressions

***postfixUnaryExpression*:**
primaryExpression
| (*primaryExpression* (*postfixUnarySuffix* {*postfixUnarySuffix*}))

***postfixUnarySuffix*:**
postfixUnaryOperator
| *typeArguments*
| *callSuffix*
| *indexingSuffix*
| *navigationSuffix*

***indexingSuffix*:**
'['
{*NL*}
expression
{{*NL*} ',' {*NL*} *expression*}
{*NL*}
']'

An *indexing expression* is a suffix expression which uses one or more subexpression as *indices* between square brackets (`[` and `]`).

It is an overloadable operator with the following expansion:

- `A[I_0,I_1,...,I_N]` is exactly the same as `A.get(I_0,I_1,...,I_N)`, where `get` is a valid operator function available in the current scope.

An indexing expression has the same type as the corresponding `get` expression.

Indexing expressions are [assignable][Assignable expressions]. For a corresponding assignment form, see [indexing assignment][Indexing assignment].

Call and property access expressions

***postfixUnaryExpression*:**
primaryExpression
| (*primaryExpression* (*postfixUnarySuffix* {*postfixUnarySuffix*}))

***postfixUnarySuffix*:**
postfixUnaryOperator
| *typeArguments*
| *callSuffix*
| *indexingSuffix*
| *navigationSuffix*

***navigationSuffix*:**
{*NL*} *memberAccessOperator* {*NL*} (*simpleIdentifier* | *parenthesizedExpression* | 'class')

***callSuffix*:**
([*typeArguments*] [*valueArguments*] *annotatedLambda*)
| ([*typeArguments*] *valueArguments*)

***annotatedLambda*:**
{*annotation*} [*label*] {*NL*} *lambdaLiteral*

***valueArguments*:**
('(' {*NL*} ')')
| ('(' {*NL*} *valueArgument* {{*NL*} ',' {*NL*} *valueArgument*} {*NL*} ')')

***typeArguments*:**
'<'
{*NL*}
typeProjection
{{*NL*} ',' {*NL*} *typeProjection*}
{*NL*}
'>'

***typeProjection*:**
([*typeProjectionModifiers*] *type*)
| '*'

***typeProjectionModifiers*:**
typeProjectionModifier {*typeProjectionModifier*}

***memberAccessOperator*:**
'.'
| *safeNav*
| '::'

Navigation operators

Expressions which use the navigation binary operators (`.`, `.?` or `::`) are syntactically similar, but, in fact, may have very different semantics.

`a.c` may have one of the following semantics when used as an expression:

- A fully-qualified type, property or object name. The left side of `.` must be a package name, while the right side corresponds to a declaration in that package.

  Note: qualification uses operator `.` only.

- A property access. Here `a` is a value available in the current scope and `c` is a property name.
- A function call if followed by the call suffix (arguments in parentheses). Here `a` is a value available in the current scope and `c` is a function name. These expressions follow the overloading rules.

`a::c` may have one of the following semantics when used as an expression:

- A class literal expression if, instead of an identifier, `c` is the keyword `class`;
- A property reference. Here `a` may be either a value available in the current scope or a type name, and `c` is a property name.
- A function reference. Here `a` may be either a value available in the current scope or a type name, and `c` is a function name.

`a?.c` is a *safe navigation* operator, which has the following expansion:

- `a?.c` is exactly the same as `if (a != null) a.c else null`.

Note: this means the type of `a?.c` is the nullable variant of the type of `a.c`.

Callable references

TODO(this is a stub)

Callable references are a special kind of expressions used to refer to callables (properties and functions) without actually calling/accessing them. It is not to be confused with class literals that use similar syntax, but with the keyword `class` used instead of the identifier.

A callable reference `A::c` where `A` is a type name and `c` is a name of a callable available for type `A` is a *callable reference* for a type. A callable reference `e::c` where `e` is another expression and `c` is a name of a callable available for type `A` is a *callable reference* for expression `e`. The exact callable selected when using this syntax is based on overload resolution much like when accessing the value of a property using the usual navigation syntax.

Depending on the meaning of the left-hand and right-hand sides of the expressions, the value of the expression is different:

- If the left-hand side of the expression is a type, but is not a value (an example of a type which is also used as a value is an object type), while the right-hand side of the expression is resolved to refer to a property of the type on the left-hand side, then the expression is a type-property reference;
- If the left-hand side of the expression is a type, but is not a value (an example of a type which is also used as a value is an object type), while the right-hand side of the expression is resolved to refer to a function available for a receiver of the type on the left-hand side, then the expression is a type-function reference;
- If the left-hand side of the expression is a value, while the right-hand side of the expression is resolved to refer to a property of the value on the left-hand side, then the expression is a value-property reference;
- If the left-hand side of the expression is a value, while the right-hand side of the expression is resolved to refer to a function for the receiver being th value on the left-hand side, then the expression is a value-function reference.

The types of these expressions are implementation-defined, but the following constraints must hold:

- The type of any kind of property reference is a subtype of `kotlin.reflect.KProperty<T>`, where the type parameter `T` is fixed to the type of the property;
- The type of any kind of callable reference is a subtype of function type that allows the corresponding callable to be accessed/called accordingly:

- For a type-callable reference, it is an extension function type `O.(ArgO ... ArgN) -> R`, where `O` is a receiver type same as the left-hand type of the expression, `ArgO, ... , ArgN` are either empty (for a property reference) or are the types of function formal parameters (for a function reference) and `R` is the result type of the callable;
- For a value-callable reference, it is a normal function type `(Args) -> R`, where `ArgO, ... , ArgN` are either empty (for a property reference) or are the types of function formal parameters (for a function reference) and `R` is the result type of the callable. The receiver is bound to the left-hand side expression of the reference expression.

Being of an appropriate function type also means that the values defined by these references are valid callables themselves, with an appropriate `operator invoke` overload, that allows using call syntax to evaluate the value of the callable with the appropriate arguments.

> Note: one may say that any function reference is essentially the same as a lambda literal with the corresponding number of arguments, calling the callable being referenced.

TODO(this is pretty complex, actually. Do we need all the K(Mutable)PropertyN business defined in the specification???) TODO(we need to update overload resolution section with these guys)

Class literals

A class literal is very similar in syntax to a callable reference, with the difference being that it uses the keyword `class` instead of the referenced identifier. Similar to callable references, there are two forms of class literals: with a type used as the left-hand side argument of the expression and with another expression used as such. This is also one of the few cases where a parameterized type may (**and must**) be used without its type parameters.

All class literals have type `kotlin.KClass<T>` and produce a platform-defined object associated with type `T`, which, in turn, is either the type given as the left-hand side of the expression or the runtime type of the value given as the left-hand side of the expression. In both cases, `T` must be a runtime-available type in the current scope. As the runtime type of the expression is not known at compile time, the compile-time type of the expression is `kotlin.KClass<U>` where $T <: U$ and `U` is the compile-time of the expression.

The produced object can be used to allow access to platform-specific capabilities of the runtime type information available on particular platform, either directly or through reflection facilities.

TODO(this is a stub)

TODO(Identifiers, paths, that kinda stuff)

Function literals

Kotlin supports using functions as values. This includes, among other things, being able to use named functions (via function references) as parts of expressions. Sometimes it does not make much sense to provide a separate function declaration, but rather define a function in-place. This is implemented using *function literals*.

There are two types of function literals in Kotlin: *lambda literals* and *anonymous function declarations*. Both of these provide a way of defining a function in-place, but have subtle differences.

> Note: as some may consider function literals to be closely related to function declarations, here is the corresponding section of the specification.

Anonymous function declarations

***anonymousFunction*:**
`'fun'`
[{*NL*} *type* {*NL*} `'.'`]
{*NL*}
functionValueParameters
[{*NL*} `':'` {*NL*} *type*]
[{*NL*} *typeConstraints*]
[{*NL*} *functionBody*]

Anonymous function declarations, despite their name, are not declarations per se, but rather expressions which resemble function declarations. They have a syntax very similar to function declarations, with the following key differences:

- Anonymous functions do not have a name;
- Anonymous functions may not have type parameters;
- Anonymous functions may not have default parameters;
- Anonymous functions may have variable argument parameters, but they are automatically decayed to non-variable argument parameters of the corresponding array type .

(TODO(how does this really work?))

Anonymous function declaration may declare an anonymous extension function.

> Note: as anonymous functions may not have type parameters, you cannot declare an anonymous extension function on a parameterized receiver type.

The type of an anonymous function declaration is the function type constructed similarly to a named function declaration.

Lambda literals

***lambdaLiteral*:**
('{' {*NL*} *statements* {*NL*} '}')
| ('{' {*NL*} [*lambdaParameters*] {*NL*} '->' {*NL*} *statements* {*NL*} '}')

***lambdaParameters*:**
lambdaParameter {{*NL*} ',' {*NL*} *lambdaParameter*}

***lambdaParameter*:**
variableDeclaration
| (*multiVariableDeclaration* [{*NL*} ':' {*NL*} *type*])

Lambda literals are similar to anonymous function declarations in that they define a function with no name. Lambda also use very different syntax, similar to control structure bodies of other expressions.

Every lambda literal consists of an optional lambda parameter list, specified before the arrow (`->`) operator and a body, which is everything after the arrow operator. Lambda body introduces a new statement scope.

Lambda literals has the same restrictions as anonymous function declarations, but also cannot have `vararg` parameters. They can, however, introduce destructuring parameters similar to destructuring property declarations.

TODO(destructuring lambda parameters)

If a lambda expression has no parameter list, it can actually be defining an anonymous function with either zero or one parameter, the exact case dependent on the context of the usage of this expression. The selection of number of parameters in this case is performed during type inference. Any lambda may also define either a normal function or an expansion function, the exact case also dependent on the context of the usage of lambda expression.

If the lambda expression has no parameter list, but has one parameter, this parameter can be accessed inside the lambda body using a special property called `it`. If the lambda expression defines an expansion function, the expansion receiver may be accessed using standard `this` syntax inside the lambda body.

> Note: having no parameter list (and no arrow operator) in a lambda is different from having zero parameters (nothing preceding the arrow operator).

Lambda literals are different from other forms of function definition in that the `return` expressions inside lambda body, unless qualified, refers to the outside non-lambda function the expression is used in rather than the lambda expression

itself. Such returns are only allowed if the function defined by the lambda is guaranteed to be [inlined][Inlining] and are not allowed at all otherwise.

If the lambda expression is labeled, it can also be returned from using the [labeled return expression][Labeled return expression]. In addition to this, if the lambda expression is used as a trailing lambda parameter to a function call, the name of the function used in the call may be used instead of the label. If a particular labeled `return` expression is used inside multiple lambda bodies invoked during the call of the same function, this is an ambiguity and should be reported as a compile-time error.

TODO(Typing)

Any properties used in any way inside the lambda body are **captured** by the lambda expression and, depending on whether it is inlined or not, affect how this properties are processed by other mechanisms, e.g. smart casts.

TODO(Rules of capturing)

Object literals

***objectLiteral*:**
('`object`' {*NL*} '`:`' {*NL*} *delegationSpecifiers* {*NL*} *classBody*)
| ('`object`' {*NL*} *classBody*)

Object literals are used to define anonymous objects in Kotlin. Anonymous objects are similar to regular objects, but they (obviously) have no name and thus can be used only as expressions. Anonymous objects, just like regular object declarations, can have at most one base class and zero or more base interfaces declared in its supertype specifiers.

The main difference between the regular object declaration and an anonymous object is its type. The type of an anonymous object is a special kind of type which is usable (and visible) only in the scope where it is declared. It is similar to a type of a regular object declaration, but, as it cannot be used outside the scope, with some interesting effects.

When a value of an anonymous object type escapes current scope:

- If the type has only one declared supertype, it is implicitly downcasted to this declared supertype;
- If the type has several declared supertypes, there must be an implicit or explicit cast to any suitable type visible outside the scope, otherwise it is a compile-time error.

  Note: an implicit cast may arise, for example, from the results of the type inference.

Note: in this context "escaping" current scope is performed immediately if the corresponding value is declared as a global- or classifier-scope property, as those are a part of package interface.

TODO: This is more complex. From D.Petrov's comment:

This is a bit more complex for anonymous object return types of private functions and properties:

```
class M {
private fun foo() = object {
fun bar() { println("foo.bar") }
}

fun test1() = foo().bar()
fun test2() = foo()
}

fun main() {
M().test1() // OK, prints "foo.bar"
M().test2().bar() // Error: Unresolved reference: bar
}
```

This-expressions

***thisExpression*:**

'this'
| *THIS_AT*

This-expressions are special kind of expressions used to access receivers available in current scope. The basic form of this expression, denoted by `this` keyword, is used to access the current implicit receiver according to the receiver overloading rules. In order to access other receivers, labeled `this` expressions are used. These may be any of the following:

- `this@type`, where `type` is a name of any classifier currently being declared (that is, this-expression is located in the inner scope of the classifier declaration), refers to the implicit object of the type being declared;
- `this@function`, where `function` is a name of any extension function currently being declared (that is, this-expression is located in the function body), refers to the implicit receiver object of the extension function;
- `this@lambda`, where `lambda` is a [label][Labels] provided for a lambda literal currently being declared (that is, this-expression is located in the lambda expression body), refers to the implicit receiver object of the lambda expression.

Any other form of this-expression is illegal and must be a compile-time error.

Super-forms

***superExpression*:**
('super' ['<' {*NL*} *type* {*NL*} '>'] ['@' *simpleIdentifier*])
| *SUPER_AT*

Super-forms are special kind of expression which can only be used as receivers in a function or property access expression. Any use of super-form expression in any other context is a compile-time error.

Super-forms are used in classifier declarations to access method implementations from the supertypes without invoking overriding behaviour.

TODO(The rest...)

Jump expressions

***jumpExpression*:**
('throw' {*NL*} *expression*)
| (('return' | *RETURN_AT*) [*expression*])
| 'continue'
| *CONTINUE_AT*
| 'break'
| *BREAK_AT*

Jump expressions are expressions which redirect the evaluation of the program to a different program point. All these expressions have several things in common:

- They all have type `kotlin.Nothing`, meaning that they never produce any runtime value;
- Any code which follows such expressions is never evaluated.

Throw expressions

TODO(Exceptions go first)

Return expressions

A *return expression*, when used inside a function body, immediately stops evaluating the current function and returns to its caller, effectively making the function call expression evaluate to the value specified in this return expression (if any). A return expression with no value implicitly returns the `kotlin.Unit` object.

There are two forms of return expression: a simple return expression, specified using the `return` keyword, which returns from the innermost function declaration (or Anonymous function declaration) and a labeled return expression of the form `return@Context` where `Context` may be one of the following:

- The name of one of the enclosing function declarations, which refers to this function. If several declarations match one name, it is a compile-time error;
- If `return@Context` is inside a lambda expression body, the name of the function **using** this lambda expression as its argument may be used as `Context` to refer to the lambda literal itself.
- TODO(return from a labeled lambda)

  Note: these rules mean that a simple return expression inside a lambda expression returns **from the innermost function**, in which this lambda expression is defined.

If returning from the referred function is allowed in the current context, the return is performed as usual. If returning from the referred function is not allowed, it is a compile-time error.

TODO(What does it mean for returns to be disallowed?)

Continue expression

A *continue expression* is a jump expression allowed only within loop bodies. When evaluated, this expression passes the control to the start of the next loop iteration (aka "continue-jumps").

There are two forms of continue expressions:

- A simple continue expression, specified using the `continue` keyword, which continue-jumps to the innermost loop statement in the current scope;
- A labeled continue expression, denoted `continue@Loop`, where `Loop` is a label of a labeled loop statement `L`, which continue-jumps to the loop `L`.

  Future use: as of Kotlin 1.2.60, a simple continue expression is not allowed inside `when` expressions.

Break expression

A *break expression* is a jump expression allowed only within loop bodies. When evaluated, this expression passes the control to the next program point immediately after the loop (aka "break-jumps").

There are two forms of break expressions:

- A simple break expression, specified using the `break` keyword, which break-jumps to the innermost loop statement in the current scope;
- A labeled break expression, denoted `break@Loop`, where `Loop` is a label of a labeled loop statement `L`, which break-jumps to the loop `L`.

  Future use: as of Kotlin 1.2.60, a simple break expression is not allowed inside when expressions.

String interpolation expressions

***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 `'}'`

***multiLineStringContent*:**
MultiLineStrText
| *MultiLineStringQuote*
| *MultiLineStrRef*

***multiLineStringExpression*:**
MultiLineStrExprStart
{*NL*}
expression
{*NL*}
`'}'`

String interpolation expressions replace the traditional string literals and supersede them. A string interpolation expression consists of one or more fragments of two different kinds: string content fragments (raw pieces of string content found inside the quoted literal) and *interpolated expressions*, delimited by the special syntax using the `$` symbol. This syntax allows to specify such fragments by directly following the `$` symbol with either a single identifier (if the expression

is a single identifier) or a control structure body. In either case, the interpolated value is evaluated and converted into a `kotlin.String` by a process defined below. The resulting value of a string interpolation expression is the joining of all fragments in the expression.

An interpolated value v is converted to `kotlin.String` according to the following convention:

- If it is equal to the null reference, the result is `"null"`;
- Otherwise, the result is v`.toString()` where `toString` is the `kotlin.Any` member function (no overloading resolution is performed to choose the function in this context).

There are two kinds of string interpolation expressions: line interpolation expressions and multiline (or raw) interpolation expressions. The difference is that some symbols (namely, newline symbols) are not allowed to be used inside line interpolation expressions and they need to be escaped (see grammar for details). On the other hand, multiline interpolation expressions allow such symbols inside them, but do not allow single character escaping of any kind.

> Note: among other things, this means that the escaping of the `$` symbol is impossible in multiline strings. If you need an escaped `$` symbol, use an interpolation fragment instead: `"${'$'}"`

String interpolation expression always has type `kotlin.String`.

TODO(define this using actual `kotlin.StringBuilder` business?)

TODO(list all the allowed escapes here?)

TODOs()

- Class literals
- Smart casts vs compile-time types
- What does `decaying` for vararg actually mean?
- Where to define spread operator?
- Object literal types look just like restricted union types. Are there any traps hidden here?
- The last paragraph in object literals is also pretty shady

Order of evaluation

TODO()

Semantics

TODO()

Control- and data-flow analysis

TODO(Unreachable code w.r.t. Nothing)

Kotlin type constraints

Some complex tasks that need to be solved when compiling Kotlin code are formulated best using *constraint systems* on Kotlin types. These are solved using constraint solvers.

Type constraint definition

A *type constraint* in general is an inequation of the following form: $T <: U$ where T and U are Kotlin types (see type system). It is important, however, that Kotlin has parameterized types and type parameters of T and U (or type parameters of their parameters, or T and U themselves) may be *type variables*, that are unknown types that may be substituted by any other type in Kotlin.

Please note that, in general, type variables of the constraint system are not the same as type parameters of a type or a callable. Some type parameters may be *bound* in the constraint system, meaning that, although they are not known yet in Kotlin code, they are not type variables and are not to be substituted.

When such an ambiguity arises, we will use the notation T_i for a type variable and $\tilde{T}_i$ for a bound type parameter. The main difference between bound parameters and concrete types is that different concrete types may not be equal, but a bound parameter may be equal to another bound parameter or a concrete type.

Several examples of valid type constraints:

- $\texttt{List}\left\langle \tilde{X} \right\rangle <: Y$
- $\texttt{List}\left\langle \tilde{X} \right\rangle <: \texttt{List}\left\langle \texttt{List}\left\langle \texttt{Int} \right\rangle \right\rangle$
- $\tilde{X} <: Y$

Every constraint system has implicit constraints $\texttt{Any} <: T_j$ and $T_j <: \texttt{Nothing}?$ for every type T_j mentioned in constraint, including type variables.

Type constraint solving

There are two tasks that a type constraint solver may perform: checking constraint system for soundness and solving the system, e.g. inferring viable values for all the type variables that have themselves no type variables in them.

Checking a constraint system for soundness can be viewed as a simpler case of solving a constraint, as if there is a solution, then the system is sound. It is, however, a much simpler task with only two possible outcomes. Solving a constraint system, on the other hand, may have several different results as there may be several valid solutions.

> Constraint examples that are sound yet no relevant solutions exist:
>
> - $X <: Y$
> - $\texttt{List}\,\langle X\rangle <: \texttt{Collection}\,\langle X\rangle$

Checking constraint system soundness

TODO()

Finding optimal solution

As any constraint system may have several valid solutions, finding one that is "optimal" in some sense is not possible in general, because the notion of the best solution for a task depends on a particular use-case. To solve this problem, the constraint system allows two additional types of constraints:

- A pull-up constraint for type variable T, denoted $\uparrow T$, signifying that when finding a substitution for this variable, the optimal solution is the least one according to subtyping relation;
- A push-down constraint for type variable T, denoted $\downarrow T$, signifying that when finding a substitution for this variable, the optimal solution is the biggest one according to subtyping relation.

If a variable have no constraints of these two kinds associated with it, it is assumed to have a pull-up constraint, that is, in an ambigious situation, the biggest possible type is chosen.

TODO()

The relations on types as constraints

In the other chapters (see expressions and statements for example) the relations between types may be expressed using the type operations found in the type system section of this document. Not all of these relations are easily converted into their constraint form.

The greatest lower bound of two types is converted directly, as the greatest lower bound is always an intersection type. The least upper bound, however, is a little bit tricky. If type T is defined to be the least upper bound of A and B with all these types being either known, unknown or containing type variables, the following constraints are produced:

- $A <: T$
- $B <: T$
- $\downarrow T$
- $\uparrow A$
- $\uparrow B$

  Example:
  Let's assume we have the following code:

  ```
  val e = if(c) a else b
  ```

  where `a`, `b`, `c` are some expressions with types completely unknown (having no other type constraints besided the implicit ones). Let's assume the type variables generated for them to be A, B and C respectively and the type variable for `e` being E. This, according to the conditional expression chapter, produces the following relations:

  - $C <:$ `kotlin.Boolean`
  - $E = LUB(A, B)$

  These, in turn, produce the following constraints (here we omit the implicit relations of all type variables with `kotlin.Any?` and `kotlin.Nothing`):

  - $C <:$ `kotlin.Boolean`
  - $A <: E$
  - $B <: E$
  - $\downarrow E$
  - $\uparrow A$
  - $\uparrow B$

  Which, according to the semantics of additional constraints (and the default pull-up constraint for C), produce the following solution:

  - $C \rightarrow$ `kotlin.Boolean`
  - $A \rightarrow$ `kotlin.Any`?
  - $B \rightarrow$ `kotlin.Any`?

- $E \to$ `kotlin.Any`?

TODO(prove that these constraints are equivalent to LUB from type system?)

TODO(are they actually?)

TODO(does that matter?)

Type inference

Kotlin has a concept of *type inference* for compile-time type information, meaning some type information in the code may be omitted, to be inferred by the compiler. There are two kinds of type inference supported by Kotlin.

- Local type inference, for inferring types of expressions locally, in statement/expression scope;
- Function signature type inference, for inferring types of function return values and/or parameters.

  Note: type inference is a type constraint problem, and is usually solved by a type constraint solver.

TODO(write about when type inference works and when it does not)

Smart casts

Kotlin introduces a limited form of flow-dependent typing called *smart casting*. Flow-dependent typing means some expressions in the program may introduce changes to the compile-time types of variables. This allows one to avoid unneeded explicit casting of values in cases when their runtime types are guaranteed to conform to the expected compile-time types.

Smart casts are dependent on two main things: *smart cast sources* and *smart cast sink stability*.

Smart cast sources

There are two kinds of smart cast sources: *non-nullability conditions* and *type conditions*. Non-nullability conditions specify that some value is not nullable, i.e., its value is guaranteed to not be `null`. Type conditions specify that some value's runtime type conforms to a constraint $RT <: T$, where T is the assumed

type and *RT* is the runtime type. A smart cast source may be *negated*, meaning it reverses its interpretation.

> Note: non-nullability conditions may be viewed as a special case of type conditions with assumed type `kotlin.Any`.
>
> Note: we may use the terms "negated non-nullability condition" and "nullability condition" interchangeably.

These sources influence the compile-time type of a value in some expression (called *smart cast sink*) only if the sink is *stable* and if the source dominates the sink. The actual compile-time type of a smart casted value for most purposes (including, but not limited to, function overloading and type inference of other values) is as follows.

- If the smart cast source is a non-nullability condition, the type is the [intersection][Type intersection] of the type it had before (including the results of smart casting performed for other conditions) and type `kotlin.Any`;
- If the smart cast source is a negated non-nullability condition, the type is the [intersection][Type intersection] of the type it had before (including the results of smart casting performed for other conditions) and type `kotlin.Nothing?`;
- If the smart cast source is a type condition, the type is the [intersection][Type intersection] of the type it had before (including the results of smart casting performed for other conditions) and the assumed type of the condition.
- If the smart cast source is a negated type condition, the type does not change.

  Note: the most important exception to when smart casts are used in type inference is direct property declaration.

```
fun noSmartCastInInference() {
    var a: Any? = null

    if (a == null) return

    var c = a // Direct property declaration

    c // Declared type of `c` is Any?
      // However, here it's smart casted to Any
}

fun <T> id(a: T): T = a

fun smartCastInInference() {
    var a: Any? = null

    if (a == null) return
```

```
        var c = id(a)

        c // Declared type of `c` is Any
    }
```

Smart cast sources are introduced by:

- Conditional expressions (`if` and `when`);
- Elvis operator (operator `?:`);
- Safe navigation operator (operator `?.`);
- Logical conjunction expressions (operator `&&`);
- Logical disjunction expressions (operator `||`);
- Not-null assertion expressions (operator `!!`);
- Direct casting expression (operator `as`);
- Direct assignments;
- Platform-specific cases: different platforms may add other kinds of expressions which introduce additional smart cast sources.

  Note: property declarations are not listed here, as their types are derived from initializers.

Nullability and type conditions are derived in the following way.

- `x is T` where x is an applicable expression implies a type condition for x with assumed type T;
- `x !is T` where x is an applicable expression implies a negated type condition for x with assumed type T;
- `x != null` or `null != x` where x is an applicable expression implies a non-nullability condition for x;
- `x == null` or `null == x` where x is an applicable expression implies a nullability condition for x;
- `!x` implies all the conditions implied by x, but in negated form;
- `x && y` implies the union of all non-negated conditions implied by x and y and the intersection of all negated conditions implied by x and y;
- `x || y` implies the union of all negated conditions implied by x and y and the intersection of all non-negated conditions implied by x and y;
- `x === y` or `y === x` where x is an applicable expression and y is a known non-nullable value (that is, has a non-nullable compile-time type) implies the non-nullability condition for x;
- `x === y` or `y === x` where x is an applicable expression and y is known to be null (that is, has `Nothing?` type) implies the nullability condition for x;
- `x == y` or `y == x` where x is an applicable expression and y is a known non-nullable value (that is, has a non-nullable compile-time type) implies the non-nullability condition for x, but only if the corresponding `equals` implementation is known to be equivalent to reference equality check.

- `x == y` or `y == x` where x is an applicable expression and y is known to be null (that is, has `Nothing?` type) implies the nullability condition for x, but only if the corresponding `equals` implementation is known to be equivalent to reference equality check.

TODO(x != Nothing? / x !== Nothing?)

  Note: for example, generated `equals` implementation for data classes is considered to be equivalent to reference equality check.

TODO(A complete list of when `equals` is OK?)

Additionally, any type condition with assumed *non-null* type also creates a non-nullability condition for its value. This property is used in bound smart casts.

Smart cast sink stability

A smart cast sink is *stable* for smart casting if its value cannot be changed from the smart cast source to itself; this guarantees the smart cast conditions still hold at the sink. This is one of the necessary conditions for smart cast to be applicable for a given source-sink pair.

Smart cast sink stability breaks in the presence of the following aspects.

- concurrent writes;
- separate module compilation;
- custom getters;
- delegation.

  Note: despite what it may seem at first sight, sink stability is *very* complicated for local variables.

The following smart cast sinks are considered stable.

1. Immutable local or classifier-scope properties without delegation or custom getters;
2. Immutable properties of stable properties without delegation or custom getters;
3. Mutable local properties without delegation or custom getters, if the compiler can prove that they are effectively immutable, i.e., cannot be changed by external means from the smart cast source to the smart cast sink.

Effectively immutable smart cast sinks

We will call redefinition of P ***direct*** redefinition, if it happens in the same declaration scope as the definition of P. If P is redefined in a nested declaration scope (w.r.t. its definition), this is a ***nested*** redefinition.

> Note: informally, a nested redefinition means the property has been captured in another scope and may be changed from that scope in a concurrent fashion.

We define ***direct*** and ***nested*** smart cast sinks in a similar way.

Example:

```
fun example() {
    // definition
    var x: Int? = null

    if (x != null) {
        run {
            // nested smart cast sink
            x.inc()

            // nested redefinition
            x = ...
        }
        // direct smart cast sink
        x.inc()
    }

    // direct redefinition
    x = ...
}
```

A mutable local property P defined at D is considered effectively immutable for a given pair of smart cast source SO and smart cast sink SI, if the following properties hold.

- There are no redefinitions of P on any path between SO and SI
- If SI is a direct sink, there must be no nested redefinitions on any path between D and SI
- If SI is a nested sink, then
  - there must be no nested redefinitions of P
  - all direct redefinitions of P must precede SI

Example:

```
fun directSinkOk() {
    var x: Int? = 42 // definition
    if (x != null)   // smart cast source
        x.inc()      // direct sink
    run {
```

```
        x = null     // nested redefinition
    }
}

fun directSinkBad() {
    var x: Int? = 42 // definition
    run {
        x = null     // nested redefinition
                     //   between a definition
                     //   and a sink
    }
    if (x != null)   // smart cast source
        x.inc()      // direct sink
}

fun nestedSinkOk() {
    var x: Int? = 42     // definition
    x = getNullableInt() // direct redefinition
    run {
        if (x != null)   // smart cast source
            x.inc()      // nested sink
    }
}

fun nestedSinkBad01() {
    var x: Int? = 42     // definition
    run {
        if (x != null)   // smart cast source
            x.inc()      // nested sink
    }
    x = getNullableInt() // direct redefinition
                         //   after the nested sunk
}

fun nestedSinkBad02() {
    var x: Int? = 42     // definition
    run {
        x = null         // nested redefinition
                         //   of a nested sink
    }
    run {
        if (x != null)   // smart cast source
            x.inc()      // nested sink
    }
}
```

Source-sink domination

A smart cast source SO dominates a smart cast sink SI, if SO is a control-flow dominator of SI. This is one of the necessary conditions for smart cast to be applicable for a given source-sink pair.

In the most basic case, smart cast conditions propagate as-is from sources to sinks. However, as a number of expressions have additional semantics, which may influence smart cast conditions, in some cases these conditions are modified along the sink-source chain. This means the following for different smart cast sources.

- Conditional expressions (`if` and `when`):
  - Smart cast conditions derived from expression condition are active inside the true branch scope;
  - Smart cast conditions derived from *negated* expression condition are active inside the false branch scope;
  - If a branch is statically known to be definitely evaluated, that branch's condition is also propagated over to its containing scope after the conditional expression;
- Elvis operator (operator `?:`): if the right-hand side of elvis operator is unreachable, a nullability condition for the left-hand side expression (if applicable) is introduced for the rest of the containing scope;
- Safe navigation operator (operator `?.`) TODO()
- Logical conjunction expressions (operator `&&`): all conditions derived from the left-hand expression are applied to the right-hand expression;
- Logical disjunction expressions (operator `||`): all conditions derived from the left-hand expression are applied *negated* to the right-hand expression;
- Not-null assertion expressions (operator `!!`): a nullability condition for the left-hand side expression (if applicable) is introduced for the rest of the containing scope;
- Unsafe cast expression (operator `as`): a type condition for the left-hand side expression (if applicable) is introduced for the rest of the containing scope; the assumed type is the same as the right-hand side type of the cast expression;
- Direct assignment: if both sides of the assignment are applicable expressions, all the conditions currently applying to the right-hand side are also applied to the left-hand side of the assignment for the rest of the containing scope.

The necessity of source-sink domination also mean that smart cast sources from the loop bodies and conditions are **not** propagated to the containing scope, as the loop body may be evaluated zero or more times, and the corresponding conditions may or may not be true. However, some loop configurations, for which we can have static guarantees about source-sink domination w.r.t. the containing scope, are handled differently.

- do-while loops (as their body is evaluated at least once) propagate the following to the rest of the containing scope:
  - smart cast sources from the loop body, which definitely dominate their sinks
  - smart cast conditions arising from the *negated* loop condition, if the loop body does not contain any `break` expressions
- `while (true)` loops propagate the following to the rest of the containing scope:
  - smart cast sources from the loop body, which definitely dominate their sinks

  Note: in the second case, only the exact `while (true)` form is handled as described; e.g., `while (true == true)` does not work.

  Note: one may extend the number of loop configurations, which are handled by smart casting, if the implementation can statically guarantee the source-sink domination.

  Example:

```
fun breakFromInfiniteLoop() {
    var a: Any? = null

    while (true) {
        if (a == null) continue

        if (randomBoolean()) break
    }

    a // Smart cast to Any
}

fun doWhileAndSmartCasts() {
    var a: Any? = null

    do {
        if (a == null) continue
    } while (randomBoolean())

    a // Smart cast to Any
}

fun doWhileAndSmartCasts2() {
    var a: Any? = null

    do {
        sink(a)
    } while (a == null)
```

```
        a // Smart cast to Any
    }
```

Bound smart casts

Smart casting propagates information forward on the control flow, as by the source-sink domination. However, in some cases it is beneficial to propagate information *backwards*, to reduce boilerplate code. Kotlin supports this feature by bound smart casts.

Bound smart casts apply in the following case. Assume we have two inter-dependent or bound values a and b. Bound smart casts allow to apply smart cast sources for a to b or vice versa, if both values are stable.

Kotlin supports the following bound smart casts (BSC).

- Non-nullability-by-equality BSC. If two values are known to be equal, non-nullability conditions for one are applied to the other.
- Non-nullability-by-safe-call BSC. For a safe-call property `o?.p` of a non-null type T, non-nullability conditions for `o?.p` are applied to `o`.

Two values a and b are considered equals in the following cases.

- there is a known equality or referential-equality condition between a and b
- a is definitely assigned b
  - however, in this case bound smart casts are applied only to b
  - TODO(Why?)

TODO(Do we need additional condition kinds?)

Local type inference

Local type inference in Kotlin is the process of deducing the compile-time types of expressions, lambda expression parameters and properties. As mentioned above, type inference is a type constraint problem, and is usually solved by a type constraint solver.

In addition to the types of intermediate expressions, local type inference also performs deduction and substitution for generic type parameters of functions and types involved in every expression. You can use the Expressions part of this specification as a reference point on how the types for different expressions are constructed.

However, there are some additional clarifications on how these types are constructed. First, the additional effects of smart casting are considered in local type inference, if applicable. Second, there are several special cases.

- If a type T is described as the least upper bound of types A and B, it is represented as a pair of constraints $A <: T$ and $B <: T$;
- TODO(are there other cases?)

Type inference in Kotlin is bidirectional; meaning the types of expressions may be derived not only from their arguments, but from their usage as well. Note that, albeit bidirectional, this process is still local, meaning it processes one statement at a time, strictly in the order of their appearance in a scope; e.g., the type of property in statement S_1 that goes before statement S_2 cannot be inferred based on how S_1 is used in S_2.

As solving a type constraint system is not a definite process (there may be more than one valid solution for a given constraint system), type inference in general may have several valid solutions. In particular, one may always derive a system $A <: T <: B$ for every type variable T, where A and B are both valid solution types. One of these types is always the solution in Kotlin (although from the constraint viewpoint, there are usually more solutions available), but choosing between them is done according to the following rules:

- TODO(what are the rules?)

  Note: this is valid even if T is a variable without any explicit constraints, as every type in Kotlin has an implicit constraint `kotlin.Nothing` $<: T <:$ `kotlin.Any?`.

TODO

- Type approximation for public usage
- Ordering of lambdas (and ordering of overloading vs type inference in general)

Overload resolution

Kotlin supports *function overloading*, that is, the ability for several functions of the same name to coexist in the same scope, with the compiler picking the most suitable one when such a function is called. This section describes this mechanism in detail.

Intro

Unlike other object-oriented languages, Kotlin does not only have object methods, but also top-level functions, local functions, extension functions and function-like values, which complicate the overloading process quite a lot. Kotlin also has infix functions, operator and property overloading which all work in a similar, but subtly different way.

Receivers

Every function or property that is defined as a method or an extension has one or more special parameters called *receiver* parameters. When calling such a callable using navigation operators (`.` or `?.`) the left hand side parameter is called an *explicit receiver* of this particular call. In addition to the explicit receiver, each call may indirectly access zero or more *implicit receivers*.

Implicit receivers are available in some syntactic scope according to the following rules:

- All receivers available in an outer scope are also available in the nested scope;
- In the scope of a classifier declaration, the following receivers are available:
  - The implicit `this` object of the declared type;
  - The companion object (if one exists) of this class;
  - The companion objects (if any exist) of all its superclasses;
- If a function or a property is an extension, `this` parameter of the extension is also available inside the extension declaration;
- The scope of a lambda expression, if it has an extension function type, contains `this` argument of the lambda expression.

TODO(If I'm a companion object, is a companion object of my supertype an implicit receiver for me or not?)

The available receivers are prioritized in the following way:

- The receivers provided in the most inner scope have higher priority;
- In a classifier body, the implicit `this` receiver has higher priority than any companion object receiver;
- Current class companion object receiver has higher priority than any of the base class companion objects.

The implicit receiver having the highest priority is also called the *default implicit receiver*. The default implicit receiver is available in the scope as `this`. Other available receivers may be accessed using labeled this-expressions.

If an implicit receiver is available in a given scope, it may be used to call functions implicitly in that scope without using the navigation operator.

The forms of call-expression

Any function in Kotlin may be called in several different ways:

- A fully-qualified call: `package.foo()`;
- A call with an explicit receiver: `a.foo()`;
- An infix function call: `a foo b`;
- An overloaded operator call: `a + b`;
- A call without an explicit receiver: `foo()`.

For each of these cases, a compiler should first pick a number of *overload candidates*, which form a set of possibly intended callables (*overload candidate set*), and then *choose the most specific function* to call based on the types of the function and the call arguments.

> Important: the overload candidates are picked **before** the most specific function is chosen.

Callables and `invoke` convention

A *callable* X for the purpose of this section is one of the following:

- Function-like callables:
  - A function named X at its declaration site;
  - A function named Y at its declaration site, but imported into the current scope using a renaming import as X;
  - A constructor of the type named X at its declaration site;
- Property-like callables, one of the following with an operator function called `invoke` available as member or extension in the current scope:
  - A property named X at its declaration site;
  - [An object][Object declarations] named X at its declaration site;
  - [A companion object][Companion objects] of a classifier type named X at its declaration site;
  - Any of the above named Y at its declaration site, but imported into the current scope using a renaming import as X.

In the latter case a call $X(Y_0, Y_1, \ldots, Y_N)$ is an overloadable operator which is expanded to X.invoke$(Y_0, Y_1, \ldots, Y_N)$. The call may contain type parameters, named parameters, variable argument parameter expansion and trailing lambda parameters, all of which are forwarded as-is to the corresponding `invoke` function.

The set of explicit receivers itself (denoted by a [`this`][This-expression] expression, labeled or not) may also be used as a property-like callable using `this` as the left-hand side of the call expression. As with normal property-like callables, `this@A`$(Y_0, Y_1, \ldots, Y_N)$ is an overloadable operator which is expanded to `this@A.invoke`$(Y_0, Y_1, \ldots, Y_N)$.

A *member callable* is either a member function-like callable or a member property-like callable with a member operator `invoke`. An *extension callable* is either an extension function-like callable, a member property-like callable with an extension operator `invoke` or an extension property-like callable with an extension operator `invoke`.

When calculating overload candidate sets, member callables produce the following separate sets (ordered by higher priority first):

- Member function-like callables;
- Member property-like callables.

Extension callables produce the following separate sets (ordered by higher priority first):

- Extension functions;
- Member property-like callables with extension invoke;
- Extension property-like callables with member invoke;
- Extension property-like callables with extension invoke.

Let us define this partition as c-level partition (callable-level partition). As this partition is the most fine-grained of all other steps of partitioning resolution candidates into sets, it is always performed last, after all other applicable steps.

Overload resolution for a fully-qualified call

If a callable name is fully-qualified (that is, it contains a full package path), then the overload candidate set S simply contains all the callables with the specified name in the specified package. As a package name can never clash with any other declared entity, after performing c-level partition on S, the resulting sets are the only ones available for further processing.

TODO(Clear up this mess)

Example:

```
package a.b.c

fun foo(a: Int) {}
fun foo(a: Double) {}
fun foo(a: List<Char>) {}
val foo = {}
. . .
a.b.c.foo()
```

Here the resulting overload candidate set contains all the callables named `foo` from the package `a.b.c`.

A call with an explicit receiver

If a function call is done via a navigation operator (`.` or `?.`, not to be confused with a fully-qualified call), then the left hand side operand of the call is the explicit receiver of this call.

A call of callable `f` with an explicit receiver `e` is correct if one (or more) of the following holds:

1. `f` is a member callable of the classifier type of `e` or any of its supertypes;
2. `f` is an extension callable of the classifier type of `e` or any of its supertypes, including local and imported extensions.

   Important: callables for case 2 include not only top-level extension callables, but also extension callables from any of the available implicit receivers. For example, if class P contains a member extension function for another class T and an object of class P is available as an implicit receiver, this extension function may be used for the call if it has a suitable type.

If a call is correct, for a callable named `f` with an explicit receiver `e` of type `T` the following sets are analyzed (in the given order):

TODO(Sync with scopes and stuff when we have them)

1. The sets of non-extension member callables named `f` of type `T`;
2. The sets of local extension callables named `f`, whose receiver type conforms to type `T`, in all declaration scopes containing the current declaration scope, ordered by the size of the scope (smallest first), excluding the package scope;
3. The sets of explicitly imported extension callables named `f`, whose receiver type conforms to type `T`;
4. The sets of extension callables named `f`, whose receiver type conforms to type `T`, declared in the package scope;
5. The sets of star-imported extension callables named `f`, whose receiver type conforms to type `T`;
6. The sets of implicitly imported extension callables named `f`, whose receiver type conforms to type `T`.

   Note: here type `U` conforms to type `T`, if $T <: U$.

When analyzing these sets, the **first** set that contains **any** callable with the corresponding name and conforming types is picked. This means, among other things, that if the set constructed on step 2 contains the overall most suitable candidate function, but the set constructed on step 1 is not empty, the functions from set 1 will be picked despite them being less suitable overload candidates.

Infix function calls

Infix function calls are a special case of function calls with an explicit receiver in the left hand side position, i.e., `a foo b` may be an infix form of `a.foo(b)`.

However, there is an important difference: during the overload candidate set construction the only functions considered for inclusion are the ones with the `infix` modifier. All other functions (and any properties) are not even considered for inclusion. Aside from this difference, candidates are selected using the same rules as for normal calls with explicit receiver.

> Note: this also means that all properties available through the `invoke` convention are non-eligible for infix calls, as there is no way of specifying the `infix` modifier for them.

Different platform implementations may extend the set of functions considered as infix functions for the overload candidate set.

Operator calls

According to the overloadable operators section, some operator expressions in Kotlin can be overloaded using specially-named functions. This makes operator expressions semantically equivalent to function calls with explicit receiver, where the receiver expression is selected based on the operator used. The selection of an exact function called in each particular case is based on the same rules as for function calls with explicit receivers, the only difference being that only functions with `operator` modifier are considered for inclusion when building overload candidate sets. Any properties are never considered for the overload candidate sets of operator calls.

> Note: this also means that all the properties available through the `invoke`convention are non-eligible for operator calls, as there is no way of specifying the `operator` modifier for them, even though the `invoke` callable is required to always have such modifier. As `invoke` convention itself is an operator call, it is impossible to use more than one `invoke` conventions in a single call.

Different platform implementations may extend the set of functions considered as operator functions for the overload candidate set.

> Note: these rules are valid not only for dedicated operator expressions, but also for any calls arising from expanding `for`-loop iteration conventions, assignments or property delegates.

A call without an explicit receiver

A call which is performed with unqualified function name and without using a navigation operator is a call without an explicit receiver. It may have one or more implicit receivers or reference a top-level function.

> Note: this does not include calls using the `invoke` operator function where the left side of the call operator is not an identifier, but some other kind of expression. These cases are handled the same way as covered in the previous section and need no special treatment

As with function calls with explicit receiver, we should first pick a valid overload candidate set and then search this set for the *most specific function* to match the call.

For an idenitifer named `f` the following sets are analyzed (in the given order):

1. The sets of local non-extension functions named `f` available in the current scope, in order of the scope they are declared in, smallest scope first;
2. The overload candidate sets for each implicit receiver `r` and `f`, calculated as if `r` is the explicit receiver, in order of the receiver priority (see the corresponding section);
3. Top-level non-extension functions named `f`, in the order of:
    a. Functions explicitely imported into current file;
    b. Functions declared in the same package;
    c. Functions star-imported into current file;
    d. Implicitly imported functions (either Kotlin standard library or platform-specific ones).

When analyzing these sets, the **first** set which contains **any** function with the corresponding name and conforming types is picked.

Calls with named parameters

Most of the calls in Kotlin may use named parameters in call expressions, e.g., `f(a = 2)`, where `a` is a parameter specified in the declaration of `f`. Such calls are treated the same way as normal calls, but the overload resolution sets are filtered to only contain callables which have matching formal parameters for all named parameters from the call.

> Note: for properties called via `invoke` convention, the named parameters must be present in the declaration of the `invoke` operator function.

Unlike positional arguments, named arguments are matched by name directly to their respective formal parameters; this matching is performed separately for each function candidate. While the number of defaults (see the MSC selection

process) does affect resolution process, the fact that some argument was or was not mapped as a named argument does not affect this process in any way.

Calls with trailing lambda expressions

A call expression may have a single lambda expression placed outside of the argument list parentheses or even completely replacing them (see [this section][Call expression] for further details). This has no effect on the overload resolution process, aside from the argument reordering which may happen because of variable argument parameters or parameters with defaults.

> Example: this means that calls `f(1,2) { g() }` and `f(1,2, body = { g() })` are completely equivalent w.r.t. the overload resolution, assuming `body` is the name of the last formal parameter of `f`.

Calls with specified type parameters

A call expression may have a type argument list explicitly specified before the argument list (see [this section][Call expression] for further details).. In this case all the potential overload sets only include callables which contain exactly the same number of formal type parameters at declaration site. In case of a property callable via `invoke` convention, type parameters must be present at the `invoke` operator function declaration.

Determining function applicability for a specific call

Rationale

A function is *applicable* for a specific call if and only if the function parameters may be assigned the values of the arguments specified at call site and all type constraints of the function hold.

Description

Determining function applicability for a specific call is a [type constraint][Type constraints] problem. First, for every non-lambda argument of the function supplied in the call, type inference is performed. Lambda arguments are excluded, as their type inference needs the results of overload resolution to finish.

Second, the following constraint system is built:

- For every non-lambda parameter inferred to have type T_i, corresponding to the function argument of type U_j, a constraint $T_i <: U_j$ is constructed;

- All declaration-site type constraints for the function are also added to the constraint system;
- For every lambda parameter with the number of lambda arguments known to be K, corresponding to the function argument of type U_m, a special constraint of the form $R(L_1, \ldots, L_K) <: U_m$ is added to the constraint system, where $R, L_1, \ldots, L_K$ are fresh variables;
- For each lambda parameter with an unknown number of lambda arguments (that is, being equal to 0 or 1), a special constraint of the form $kotlin.Function <: U_m$ is added to the constraint system, where $kotlin.Function$ is the common base of all functional types .

(TODO(what's the spec name?))

If this constraint system is sound, the function is applicable for the call. Only applicable functions are considered for the next step: finding the most specific overload candidate from the candidate set.

Choosing the most specific function from the overload candidate set

Rationale

The main rationale in choosing the most specific function from the overload candidate set is the following:

> The most specific function can forward itself to any other function from the overload candidate set, while the opposite is not true.

If there are several functions with this property, none of them are the most specific and an ambiguity error should be reported by the compiler.

Consider the following example with two functions:

```
fun f(arg: Int, arg2: String) {}        // (1)
fun f(arg: Any?, arg2: CharSequence) {} // (2)
...
f(2, "Hello")
```

Both functions (1) and (2) are applicable for the call, but function (1) could easily call function (2) by forwarding both arguments into it, and the reverse is impossible. As a result, function (1) is more specific of the two.

The following snippet should explain this in more detail.

```
fun f1(arg: Int, arg2: String) {
    f2(arg, arg2) // valid: can forward both arguments
}
fun f2(arg: Any?, arg2: CharSequence) {
    f1(arg, arg2) // invalid: function f1 is not applicable
}
```

The rest of this section will try to clarify this mechanism in more detail.

Description

When an overload resolution set S is selected and it contains more than one callable, the next step is to choose the most appropriate candidate from these callables.

The selection process uses the [type constraint][Type constraints] system of Kotlin, in a way similar to the process of determining function applicability. For every two distinct members of the candidate set F_1 and F_2, the following constraint system is constructed and solved:

- For every non-default argument of the call, the corresponding value parameter types $X_1, X_2, X_3, \ldots, X_N$ of F_1 and $Y_1, Y_2, Y_3, \ldots, Y_N$ of F_2, a type constraint $X_K <: Y_K$ is built **unless both X_K and Y_K are built-in integer types.** During construction of these constraints, all type parameters $T_1, T_2, \ldots, T_M$ of F_1 are considered bound to fresh type variables $T_1^{\sim}, T_2^{\sim}, \ldots, T_M^{\sim}$, and all type parameters of F_2 are considered free;
- All declaration-site type constraints of $X_1, X_2, X_3, \ldots, X_N$ and $Y_1, Y_2, Y_3, \ldots, Y_N$ are also added to the constraint system.

If the resulting constraint system is sound, it means that F_1 is equally or more applicable than F_2 as an overload candidate (aka applicability criteria). The check is then repeated with F_1 and F_2 swapped. If F_1 is equally or more applicable than F_2 and F_2 is equally or more applicable than F_1, this means that the two callables are equally applicable and additional decision steps are needed to choose the most specific overload candidate. If neither F_1 nor F_2 is equally or more applicable than its counterpart, it also means that F_1 and F_2 are equally applicable and additional decision steps are needed.

All members of the overload candidate set are ranked according to the applicability criteria. If there are several callables which are more applicable than all other candidates and equally applicable to each other, an additional step is performed.

- Any non-generic (meaning that it does not have type parameters in its declaration) callable is a more specific candidate than any generic (containing type parameters in its declaration) callable. If there are several non-generic candidates, further steps are limited to those candidates;
- For every non-default argument of the call consider the corresponding value parameter types $X_1, X_2, X_3, \ldots, X_N$ of F_1 and $Y_1, Y_2, Y_3, \ldots, Y_N$ of F_2. If, for any K, both X_K and Y_K are different built-in integer types and one of them is `kotlin.Int`, then this parameter is preferred over the other parameter of the call. If all such parameters of F_1 are preferred based on this criteria over the parameters of F_2, then F_1 is a more specific candidate than F_2, and vice versa.

- For each candidate, we count the number of default parameters *not* specified in the call (i.e., the number of parameters for which we use the default value);
- The candidate with the least number of non-specified default parameters is a more specific candidate;
- If the number of non-specified default parameters is equal for several candidates, the candidate having any variable-argument parameters is less specific than any candidate without them.

  Note: it may seem strange to process built-in integer types in a way different from other types, but it is important in cases where the actual call argument is an integer literal having an integer literal type. In this particular case, several functions with different built-in integer types for the corresponding parameter may be applicable, and it is preferred to have the `kotlin.Int` overload as the most specific.

If after this additional step there are still several candidates that are equally applicable for the call, this is an **overload ambiguity** which must be reported as a compiler error.

> Note: unlike the applicability test, the candidate comparison constraint system is **not** based on the actuall call, meaning that, when comparing two candidates, only constraints visible at *declaration site* apply.

About type inference

Type inference in Kotlin is a pretty complicated process, which is performed after resolving all the overload candidates. Due to the complexity of the process, type inference may not affect the way overload resolution candidate is picked up.

TODOs

- Property business
- Function types (type system section?)
- Definition of "type parameter level"
- Calls with trailing lambda without parameter type
  - Lambdas with parameter types seem to be covered (**nope, they are not**)
- Calls with specified type parameters `f<Double>(3)`
- Widen the notion of "function" and "property" during overloading
  - Constructors and companion object `invoke` (clash with functions)
  - Singleton objects (clash with properties)
  - Enum constants (clash with properties)

- Explicit `this` cannot clash with properties, but can clash with other explicit `this`, meaning it effectively overloads over all the available receivers in the scope
- Can `super` be overloaded? I suppose

Concurrency

TODO()

Coroutines

TODO(everything: state machine, context, etc.)

Note: as of Kotlin 1.2, the support for coroutines is experimental

Suspending functions

Any function declaration or a getter/setter declaration in Kotlin may be marked *suspending* using the special `suspend` modifier. A function type for a particular value may also be marked suspending using the `suspend` modifier. Both normal functions and extension functions, top-level and member, anonymous and named, may be marked as suspending.

TODO(`suspend val`?)

A suspending function is different from normal functions by potentially having zero or more *suspension points* — statements in its body that may pause the function execution to be resumed at a later moment in time. Each call to another suspending function inside the body of a suspending function is a suspension point. Suspension points are important because at such a point another function may start being executed in the same flow of execution, leading to potential changes in shared state in the middle of a function execution flow.

Normal functions may not call suspending functions directly, meaning they do not have suspension points. Suspending functions may call normal functions without any limitation, such calls are not suspension points. The exception for this rule are inlined lambda parameters that, if the inlined higher-order function invoking them is called from a suspending function, may also have suspension points and call other suspending functions.

> Note: suspending functions interleaving each other in this manner is not dissimilar to how functions from different threads interact on platforms that support multithreading. There are, however, several key differences. First, suspending functions may pause only at suspension points, this process cannot be paused at arbitrary execution point. Second, this may happen in one platform thread. In multithreaded environment, suspension functions may also be interleaved by the usual rules of concurrent execution on this platform, independent of the interleaving of coroutines.

The implementation of suspending functions on a particular platform is platform-dependent. Please refer to the platform documentation for details.

Coroutine intrinsics

TODO(Do we need them?)

Annotations

Annotations are a form of syntactically-defined metadata that may be associated with different entities in a Kotlin program. Annotations are specified in the source code of the program and may be accessed on a particular platform using platform-specific mechanisms both by the compiler (and source-processing tools) and during runtime (using [reflection][Reflection] facilities). Values of annotation types cannot be created directly, but can be operated when accessed using platform-specific facilities.

Annotation values

An annotation value is a value of a special [Annotation type][Annotation types]. An annotation type is a special kind of class type that is allowed to include properties of the following types:

- Integer types;
- String type;
- Other annotation types;
- Arrays of any type listed above.

Annotation classes are not allowed to have any member functions, constructors or mutable properties. They are also not allowed to have base classes besides `kotlin.Annotation`.

Annotation retention

The retention level of an annotation declares which compilation artifacts a particular compiler on a particular platform do retain this kind of annotation. There are the following types of retention available:

- Source retention (accessible by source-processing tools);
- Binary retention (retained in compilation artifacts);
- Runtime retention (accessible during runtime).

For availability and particular ways of accessing the metadata specified by these annotations please refer to the corresponding platforms' documentation.

Annotation targets

The *targets* of a particular type of annotations is the kind of entity which this annotations may be placed on. There are the following targets available:

- A class declaration (including annotation classes);
- An annotation class declaration;
- A type parameter;
- A property declaration;
- A property backing field;
- A property getter;
- A property setter;
- A local property declaration;
- A value parameter (function or constructor declaration);
- A constructor;
- A function declaration;
- A type usage;
- An [expression][Expression];
- A [Kotlin file][Kotlin file scope];
- A [type alias declaration][Type alias declaration].

Annotation declarations

Annotations are declared using annotation class declarations. See the corresponding section for details.

TODO()

Builtin annotations

- Deprecated / ReplaceWith

- Suppress
- SinceKotlin
- UnsafeVariance
- DslMarker
- PublishedApi
- Contract-related stuff???

Documentation comments

TODO()

Exceptions

TODO(This is a stub)

An *exception* type declaration is any type declaration that meets the following criteria:

- It is a class or object declaration;
- It has `kotlin.Throwable` as supertype;
- It has no type parameters.

Any object of an exception type may be *thrown* or *catched*.

Catching exceptions

A `try`-expression becomes *active* once the execution of the program enters it and stops being active once the execution of the program leaves it. If there are several active `try`-expressions, the one that became active last is *currently active*.

If an exception is thrown while a try-expression is currently active and this try-expression has any `catch`-blocks, those `catch`-blocks are checked for applicability for this exception. A `catch`-block is applicable for an exception object if the runtime type of this expression object is a subtype of the bound exception parameter of this `catch`-block. Note that this is subject to Kotlin runtime type information limitations and may be dependent on the platform implementation of runtime type information, as well as the implementation of exception classes.

If a `catch`-block is applicable for the exception thrown, the code inside the block is evaluated and the value of the block is returned as the value of a `try`-expression. If this `try`-expression contains a `finally`-block, the body of

this block is evaluated after the body of the selected `catch` block. The `try`-expression itself is not considered active inside its own `catch` and `finally` blocks. If this results in throwing other exceptions (including the one caught by the `catch`-block), they are propagated as normal.

If none of the `catch`-blocks of the currently active `try`-expression are applicable for the exception, the `finally` block (if any) is still evaluated and the exception is propagated, meaning that the next active `try`-expression becomes currently active and is checked for applicability.

If there is not a single active `try`-block, the execution of the program finishes, signaling that the exception has reached top level.

Throwing exceptions

Throwing an exception object is performed using `throw`-expression. A valid throw expression `throw e` requires that:

- `e` is a value of a runtime-available type;
- `e` is a value of an exception type (see above).

Throwing an exception results in checking active `try`-blocks as described above.

> Note: Kotlin does not specify whether throwing exceptions involves construction of a program stack trace and how the actual exception handling is performed internally. This is a platform-dependent mechanism.

TODO: control flow?

TODO: concurrency?

TODO: write it better