Chapter 13
Syntax Summary

The following descriptions of Scala tokens uses literal characters `‘c’` when referring to the ASCII fragment `\u0000` – `\u007F`.

Unicode escapes are used to represent the Unicode character with the given hexadecimal code:

```
UnicodeEscape ::= ‘\’ ‘u’ {‘u’} hexDigit hexDigit hexDigit hexDigit
hexDigit      ::= ‘0’ | … | ‘9’ | ‘A’ | … | ‘F’ | ‘a’ | … | ‘f’
```

13.1 Lexical Syntax

The lexical syntax of Scala is given by the following grammar in EBNF form:

```
whiteSpace       ::=  ‘\u0020’ | ‘\u0009’ | ‘\u000D’ | ‘\u000A’
upper            ::=  ‘A’ | … | ‘Z’ | ‘$’ | ‘_’  // and Unicode category Lu
lower            ::=  ‘a’ | … | ‘z’ // and Unicode category Ll
letter           ::=  upper | lower // and Unicode categories Lo, Lt, Nl
digit            ::=  ‘0’ | … | ‘9’
paren            ::=  ‘(’ | ‘)’ | ‘[’ | ‘]’ | ‘{’ | ‘}’
delim            ::=  ‘`’ | ‘'’ | ‘"’ | ‘.’ | ‘;’ | ‘,’
opchar           ::= // printableChar not matched by (whiteSpace | upper | lower |
                     // letter | digit | paren | delim | opchar | Unicode_Sm | Unicode_So)
printableChar    ::= // all characters in [\u0020, \u007F] inclusive
charEscapeSeq    ::= ‘\’ (‘b’ | ‘t’ | ‘n’ | ‘f’ | ‘r’ | ‘"’ | ‘'’ | ‘\’)

op               ::=  opchar {opchar}
varid            ::=  lower idrest
plainid          ::=  upper idrest
                 |  varid
                 |  op
id               ::=  plainid
                 |  ‘`’ { charNoBackQuoteOrNewline | UnicodeEscape | charEscapeSeq } ‘`’
idrest           ::=  {letter | digit} [‘_’ op]

integerLiteral   ::=  (decimalNumeral | hexNumeral) [‘L’ | ‘l’]
decimalNumeral   ::=  ‘0’ | nonZeroDigit {digit}
hexNumeral       ::=  ‘0’ (‘x’ | ‘X’) hexDigit {hexDigit}
digit            ::=  ‘0’ | nonZeroDigit
nonZeroDigit     ::=  ‘1’ | … | ‘9’

floatingPointLiteral
                 ::=  digit {digit} ‘.’ digit {digit} [exponentPart] [floatType]
                 |  ‘.’ digit {digit} [exponentPart] [floatType]
                 |  digit {digit} exponentPart [floatType]
                 |  digit {digit} [exponentPart] floatType
exponentPart     ::=  (‘E’ | ‘e’) [‘+’ | ‘-’] digit {digit}
floatType        ::=  ‘F’ | ‘f’ | ‘D’ | ‘d’

booleanLiteral   ::=  ‘true’ | ‘false’

characterLiteral ::=  ‘'’ (charNoQuoteOrNewline | UnicodeEscape | charEscapeSeq) ‘'’

stringLiteral    ::=  ‘"’ {stringElement} ‘"’
                 |  ‘"""’ multiLineChars ‘"""’
stringElement    ::=  charNoDoubleQuoteOrNewline
                 |  UnicodeEscape
                 |  charEscapeSeq
multiLineChars   ::=  {[‘"’] [‘"’] charNoDoubleQuote} {‘"’}

symbolLiteral    ::=  ‘'’ plainid
```

```
comment          ::=  ‘/*’ “any sequence of characters; nested comments are allowed” ‘*/’
                 |  ‘//’ “any sequence of characters up to end of line”

nl               ::=  “newlinecharacter”
semi             ::=  ‘;’ |  nl {nl}
```

13.2 Context-free Syntax

The context-free syntax of Scala is given by the following EBNF grammar:

```
Literal           ::=  [‘-’] integerLiteral
                    |  [‘-’] floatingPointLiteral
                    |  booleanLiteral
                    |  characterLiteral
                    |  stringLiteral
                    |  symbolLiteral
                    |  ‘null’

QualId            ::=  id {‘.’ id}
ids               ::=  id {‘,’ id}

Path              ::=  StableId
                    |  [id ‘.’] ‘this’
StableId          ::=  id
                    |  Path ‘.’ id
                    |  [id ‘.’] ‘super’ [ClassQualifier] ‘.’ id
ClassQualifier    ::=  ‘[’ id ‘]’

Type              ::=  FunctionArgTypes ‘=>’ Type
                    |  InfixType [ExistentialClause]
FunctionArgTypes  ::= InfixType
                    | ‘(’ [ ParamType {‘,’ ParamType } ] ‘)’
ExistentialClause ::=  ‘forSome’ ‘{’ ExistentialDcl {semi ExistentialDcl} ‘}’
ExistentialDcl    ::=  ‘type’ TypeDcl
                    |  ‘val’ ValDcl
InfixType         ::=  CompoundType {id [nl] CompoundType}
CompoundType      ::=  AnnotType {‘with’ AnnotType} [Refinement]
                    |  Refinement
AnnotType         ::=  SimpleType {Annotation}
SimpleType        ::=  SimpleType TypeArgs
                    |  SimpleType ‘#’ id
                    |  StableId
                    |  Path ‘.’ ‘type’
                    |  ‘(’ Types ‘)’
TypeArgs          ::=  ‘[’ Types ‘]’
Types             ::=  Type {‘,’ Type}
Refinement        ::=  [nl] ‘{’ RefineStat {semi RefineStat} ‘}’
RefineStat        ::=  Dcl
                    |  ‘type’ TypeDef
                    |
TypePat           ::=  Type

Ascription        ::=  ‘:’ InfixType
                    |  ‘:’ Annotation {Annotation}
                    |  ‘:’ ‘_’ ‘*’

Expr              ::=  (Bindings | [‘implicit’] id | ‘_’) ‘=>’ Expr
                    |  Expr1
Expr1             ::=  ‘if’ ‘(’ Expr ‘)’ {nl} Expr [[semi] ‘else’ Expr]
                    |  ‘while’ ‘(’ Expr ‘)’ {nl} Expr
                    |  ‘try’ (‘{’ Block ‘}’ | Expr) [‘catch’ ‘{’ CaseClauses ‘}’] [‘finally’ Expr]
                    |  ‘do’ Expr [semi] ‘while’ ‘(’ Expr ‘)’
                    |  ‘for’ (‘(’ Enumerators ‘)’ | ‘{’ Enumerators ‘}’) {nl} [‘yield’] Expr
                    |  ‘throw’ Expr
                    |  ‘return’ [Expr]
                    |  [SimpleExpr ‘.’] id ‘=’ Expr
                    |  SimpleExpr1 ArgumentExprs ‘=’ Expr
                    |  PostfixExpr
                    |  PostfixExpr Ascription
                    |  PostfixExpr ‘match’ ‘{’ CaseClauses ‘}’
PostfixExpr       ::=  InfixExpr [id [nl]]
InfixExpr         ::=  PrefixExpr
                    |  InfixExpr id [nl] InfixExpr
PrefixExpr        ::=  [‘-’ | ‘+’ | ‘~’ | ‘!’] SimpleExpr
```

```
SimpleExpr        ::=  ‘new’ (ClassTemplate | TemplateBody)
                    |  BlockExpr
                    |  SimpleExpr1 [‘_’]
SimpleExpr1       ::=  Literal
                    |  Path
                    |  ‘_’
                    |  ‘(’ [Exprs] ‘)’
                    |  SimpleExpr ‘.’ id
                    |  SimpleExpr TypeArgs
                    |  SimpleExpr1 ArgumentExprs
                    |  XmlExpr
Exprs             ::=  Expr {‘,’ Expr}
ArgumentExprs     ::=  ‘(’ [Exprs] ‘)’
                    |  ‘(’ [Exprs ‘,’] PostfixExpr ‘:’ ‘_’ ‘*’ ‘)’
                    |  [nl] BlockExpr
BlockExpr         ::=  ‘{’ CaseClauses ‘}’
                    |  ‘{’ Block ‘}’
Block             ::=  BlockStat {semi BlockStat} [ResultExpr]
BlockStat         ::=  Import
                    |  {Annotation} [‘implicit’ | ‘lazy’] Def
                    |  {Annotation} {LocalModifier} TmplDef
                    |  Expr1
                    |
ResultExpr        ::=  Expr1
                    |  (Bindings | ([‘implicit’] id | ‘_’) ‘:’ CompoundType) ‘=>’ Block

Enumerators       ::=  Generator {semi Generator}
Generator         ::=  Pattern1 ‘<-’ Expr {[semi] Guard | semi Pattern1 ‘=’ Expr}

CaseClauses       ::=  CaseClause { CaseClause }
CaseClause        ::=  ‘case’ Pattern [Guard] ‘=>’ Block
Guard             ::=  ‘if’ PostfixExpr

Pattern           ::=  Pattern1 { ‘|’ Pattern1 }
Pattern1          ::=  varid ‘:’ TypePat
                    |  ‘_’ ‘:’ TypePat
                    |  Pattern2
Pattern2          ::=  varid [‘@’ Pattern3]
                    |  Pattern3
Pattern3          ::=  SimplePattern
                    |  SimplePattern { id [nl] SimplePattern }
SimplePattern     ::=  ‘_’
                    |  varid
                    |  Literal
                    |  StableId
                    |  StableId ‘(’ [Patterns] ‘)’
                    |  StableId ‘(’ [Patterns ‘,’] [varid ‘@’] ‘_’ ‘*’ ‘)’
                    |  ‘(’ [Patterns] ‘)’
                    |  XmlPattern
Patterns          ::=  Pattern [‘,’ Patterns]
                    |  ‘_’ ‘*’

TypeParamClause   ::=  ‘[’ VariantTypeParam {‘,’ VariantTypeParam} ‘]’
FunTypeParamClause::=  ‘[’ TypeParam {‘,’ TypeParam} ‘]’
VariantTypeParam  ::=  {Annotation} [‘+’ | ‘-’] TypeParam
TypeParam         ::=  (id | ‘_’) [TypeParamClause] [‘>:’ Type] [‘<:’ Type]
                       {‘<%’ Type} {‘:’ Type}
ParamClauses      ::=  {ParamClause} [[nl] ‘(’ ‘implicit’ Params ‘)’]
ParamClause       ::=  [nl] ‘(’ [Params] ‘)’
Params            ::=  Param {‘,’ Param}
Param             ::=  {Annotation} id [‘:’ ParamType] [‘=’ Expr]
ParamType         ::=  Type
                    |  ‘=>’ Type
                    |  Type ‘*’
ClassParamClauses ::=  {ClassParamClause}
                       [[nl] ‘(’ ‘implicit’ ClassParams ‘)’]
ClassParamClause  ::=  [nl] ‘(’ [ClassParams] ‘)’
ClassParams       ::=  ClassParam {‘,’ ClassParam}
ClassParam        ::=  {Annotation} {Modifier} [(‘val’ | ‘var’)]
                       id ‘:’ ParamType [‘=’ Expr]
Bindings          ::=  ‘(’ Binding {‘,’ Binding} ‘)’
Binding           ::=  (id | ‘_’) [‘:’ Type]

Modifier          ::=  LocalModifier
```

```
                    |  AccessModifier

                    |  ‘override’
LocalModifier     ::=  ‘abstract’
                    |  ‘final’
                    |  ‘sealed’
                    |  ‘implicit’
                    |  ‘lazy’
AccessModifier    ::=  (‘private’ | ‘protected’) [AccessQualifier]
AccessQualifier   ::=  ‘[’ (id | ‘this’) ‘]’

Annotation        ::=  ‘@’ SimpleType {ArgumentExprs}
ConstrAnnotation  ::=  ‘@’ SimpleType ArgumentExprs

TemplateBody      ::=  [nl] ‘{’ [SelfType] TemplateStat {semi TemplateStat} ‘}’
TemplateStat      ::=  Import
                    |  {Annotation [nl]} {Modifier} Def
                    |  {Annotation [nl]} {Modifier} Dcl
                    |  Expr
                    |
SelfType          ::=  id [‘:’ Type] ‘=>’
                    |  ‘this’ ‘:’ Type ‘=>’

Import            ::=  ‘import’ ImportExpr {‘,’ ImportExpr}
ImportExpr        ::=  StableId ‘.’ (id | ‘_’ | ImportSelectors)
ImportSelectors   ::=  ‘{’ {ImportSelector ‘,’} (ImportSelector | ‘_’) ‘}’
ImportSelector    ::=  id [‘=>’ id | ‘=>’ ‘_’]

Dcl               ::=  ‘val’ ValDcl
                    |  ‘var’ VarDcl
                    |  ‘def’ FunDcl
                    |  ‘type’ {nl} TypeDcl

ValDcl            ::=  ids ‘:’ Type
VarDcl            ::=  ids ‘:’ Type
FunDcl            ::=  FunSig [‘:’ Type]
FunSig            ::=  id [FunTypeParamClause] ParamClauses
TypeDcl           ::=  id [TypeParamClause] [‘>:’ Type] [‘<:’ Type]

PatVarDef         ::=  ‘val’ PatDef
                    |  ‘var’ VarDef
Def               ::=  PatVarDef
                    |  ‘def’ FunDef
                    |  ‘type’ {nl} TypeDef
                    |  TmplDef
PatDef            ::=  Pattern2 {‘,’ Pattern2} [‘:’ Type] ‘=’ Expr
VarDef            ::=  PatDef
                    |  ids ‘:’ Type ‘=’ ‘_’
FunDef            ::=  FunSig [‘:’ Type] ‘=’ Expr
                    |  FunSig [nl] ‘{’ Block ‘}’
                    |  ‘this’ ParamClause ParamClauses
                       (‘=’ ConstrExpr | [nl] ConstrBlock)
TypeDef           ::=  id [TypeParamClause] ‘=’ Type

TmplDef           ::=  [‘case’] ‘class’ ClassDef
                    |  [‘case’] ‘object’ ObjectDef
                    |  ‘trait’ TraitDef
ClassDef          ::=  id [TypeParamClause] {ConstrAnnotation} [AccessModifier]
                       ClassParamClauses ClassTemplateOpt
TraitDef          ::=  id [TypeParamClause] TraitTemplateOpt
ObjectDef         ::=  id ClassTemplateOpt
ClassTemplateOpt  ::=  ‘extends’ ClassTemplate | [[‘extends’] TemplateBody]
TraitTemplateOpt  ::=  ‘extends’ TraitTemplate | [[‘extends’] TemplateBody]
ClassTemplate     ::=  [EarlyDefs] ClassParents [TemplateBody]
TraitTemplate     ::=  [EarlyDefs] TraitParents [TemplateBody]
ClassParents      ::=  Constr {‘with’ AnnotType}
TraitParents      ::=  AnnotType {‘with’ AnnotType}
Constr            ::=  AnnotType {ArgumentExprs}
EarlyDefs         ::= ‘{’ [EarlyDef {semi EarlyDef}] ‘}’ ‘with’
EarlyDef          ::=  {Annotation [nl]} {Modifier} PatVarDef

ConstrExpr        ::=  SelfInvocation
                    |  ConstrBlock
ConstrBlock       ::=  ‘{’ SelfInvocation {semi BlockStat} ‘}’
SelfInvocation    ::=  ‘this’ ArgumentExprs {ArgumentExprs}
```

```
TopStatSeq        ::=  TopStat {semi TopStat}
TopStat           ::=  {Annotation [nl]} {Modifier} TmplDef
                    |  Import
                    |  Packaging
                    |  PackageObject
                    |
Packaging         ::=  ‘package’ QualId [nl] ‘{’ TopStatSeq ‘}’
PackageObject     ::=  ‘package’ ‘object’ ObjectDef

CompilationUnit   ::=  {‘package’ QualId semi} TopStatSeq
```