Scala 言語仕様
Version 2.8

DRAFT
July 13,2010

Martin Odersky
水尾学文(和訳)

PROGRAMMING METHODS LABORATORY
EPFL
SWITZERLAND

目次

序文

Scala はオブジェクト指向と関数型プログラミングを融合した Java に似たプログラミング言語です。 すべての値がオブジェクトであるという点で、純粋なオブジェクト指向言語です。 クラスによってオブジェクトの型と振る舞いを記述します。 クラスはミックスイン合成を使って構築できます。 Scala は、 2つのそれほど純粋ではないが主流のオブジェクト指向言語 - Java、C# とシームレスに動作するよう設計されています。

すべての関数が値であるという意味で、Scala は関数型言語です。 関数定義のネストと高階関数は当然サポートされています。 Scala はパターン・マッチングの汎用的な概念もサポートし、 多くの関数型言語で使われる代数型をモデル化できます。

Scala は Java（.NET でも Scala の代替実装が動作します）とシームレスに相互運用できるよう設計されてきました。 Scala のクラスは、Java メソッドの 呼び出し、Java オブジェクトの生成、Java クラスの継承、そして Java インタフェースを実装できます。 それはインターフェース定義あるいは接着コードを全く必要としません。

Scala は EPFL で 2001年からプログラミング方式研究所で開発されてきました。 バージョン1.0 が 2003年 11月にリリースされました。 このドキュメントは 2006年 3月にリリースされた、言語の2番目のバージョンを記述しています。 これは言語定義といくつかのコアライブラリモジュールのリファレンスの役を果たします。 このドキュメントは Scala やその概念を伝えることを意図していません; そのためのドキュメントとしては [Oa04、Ode06、OZ05b、 OCRZ03、OZ05a] 等があります。

Scala は多くの人達の共同の努力のたまものです。バージョン1.0 の設計と実装は Philippe Altherr、Vincent Cremet、Gilles Dubochet、Burak Emir、 Stephane Micheloud、Nikolay Mihaylov、Michel Schinz、Erik Stenman、 Matthias Zenger と著者によって完成されました。Iulian Dragos、 Gilles Dubochet、Philipp Haller、Sean McDirmid、Lex Spoon と Geoffrey Washburn は、 言語およびツールの 2 番目のバージョン開発の取り組みに参加しました。 Gilad Bracha、Craig Chambers、Erik Ernst、Matthias Felleisen、Shriram Krishnamurti、Gary Leavens、Sebastian Maneth、 Erik Meijer、Klaus Ostermann、Didier Remy、Mads Torgersen、 and Philip Wadler は、この文書の活発な、 そして人を元気づけるディスカッションと前バージョンについてのコメントを通して、言語設計を具体化しました。 Scala メーリングリストへの貢献者からも大変有用な意見をもらい、 言語とそのツールの改善を助けられました。

訳者注

用語・訳

● element type -> 要素型 、result type -> 結果型 、expected type -> 要請型、 parameter type -> パラメータ型 と訳しましたが、 実際にそういった特定の型があるわけではなく、それぞれ、要素の型、返される結果の型、要求される(期待される)型、パラメータの型、と読み替えると理解できる場合がよくあります。 (Chapter A の BNF 記法での用語にも「パラメータ型」が出てきますが、 本文中で使う場合とは意味合いが違うことがあります) しかし、subtype -> サブ型、supertype -> スーパー型 その他、「の」を入れて解釈することができないものも多数あります。

● parameter ->パラメータ、argument -> 引数と訳しました。 parameterは仮引数、argumentは実引数を表すのかもしれません。

● value -> 「値」、value types -> 「値型」と単純に訳しました。 一般に値型とは Int,shortなどの数値型のことみたいですが、この資料ではそうとは限らないようです。 「値定義」とは val x:T ...のことであり、クラスの値メンバとは val 定義されたメンバのこと、メソッドの「値パラメータ」とは型パラメータではない方の 従来のパラメータ部を意味するようです。 「関数値(function values)」とは関数が返す値のことではなく、「値としての関数」というような意味合いだそうです。 たんに「値」といっても、val変数を指すこともたまにはあるようなので、すみませんが、適宜読み替えてみてください。

● 書籍にならい view bound ->可視境界 と訳しましたが、ビュー境界 と読み替えたほうがよいと思います。 単なる view を 「可視」ではなく「ビュー」と訳したのは、英文中では "view" を機能名として使うことがよくあり、それを「可視」とは言い難いからです。

●「/」を 「あるいは」の代わりに用いていることがあります。 用い方は恣意的です。 例:クラス/オブジェクト <---> class or object

・ ps = パラメータ部(parameter section) 、 tps = 型パラメータ部(type parameter section) 、 targs = 型引数部(type arguments section) の略と思われます。

・ 第 1 章では parentheses を ()[]{} と定義していますが、他の場所では、 ()の意味で使っているようなので、第 1 章と Chapter A を除き、基本的には parenthesesを丸括弧と訳しました。

・ stable memberを「安定メンバー」と訳しましたが「不変メンバー」とでもしたほうがいいのかもしれません。

字体・表示

Wiki上の制約、日文PDFで使う restructuredText との兼ね合い・制約、労力、日文中での 見栄え等を考慮して、字体(太字・斜体字・他)は英文PDFとは異なっています。

・ 5章 英文PDFでは、見た目が大きく異なるフォントを記号的に使っていることがありますが、日文では代わりに字を重ねて表現しています。 例 : 線形化のシンボル L(C) --> LL(C)

・ Scala予約語は英文では基本的に太字ですが、本文中では通常の字体です。 英文中で斜体字になっているもののうち、T や Anといった記号的なものは日文では通常の字体に、用語的なものは斜体字ではなく太字です。 添え字は、いくつかの例外を除き、基本的には小さくなっていません。

● wikiの PDF表示では記号の類が正しく表示されないことがあります。 たとえば、プライム記号(´)や±です。

その他

・構文(BNF記法)については、本文中では 英文のまま記載していますが、 Chapter A では、日文と英文両方を記載しています。

・PDF中では、少しだけですが英語原文を表示しました。

2011年2月24日

1 字句構文

Scala プログラムはユニコード基本多言語面（BMP）文字セットを使って書かれます; ユニコード補助文字は現在サポートされていません。 この章では Scala の字句構文の 2 つのモード、Scala モードと XML モードを定義します。 以下では断りがなければ、 Scala トークンの記述は Scala モードを意味し、リテラル文字 'c' は ASCIIフラグメント \u0000-\u007F を意味します。

Scala モードでは、**ユニコードエスケープ**は与えられた 16進コードに対応するユニコード文字で置き換えられます。

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

トークンの組み立てにあたり、文字は次の区分に従って区別されます（括弧内はユニコード一般カテゴリを表します)。

1. 空白文字(Whitespace characters) \u0020 | \u0009 | \u000D | \u000A
2. 文字（Letters）小文字（Ll)、大文字（Lu)、タイトルケース文字（Lt)、 他の文字（Lo)、数字（Nl)、大文字扱いの 2 つの文字 \u0024 '$' と \u005F '_' などを含む。
3. 10進数字（Digits）'0' | … | '9'
4. 括弧（Parentheses）'(' | ')' | '[' | ']' | '{' | '}'
5. 区切り文字(Delimiter characters) '`' | ''' | '"' | '.' | ';' | ','
6. 演算子文字（Operator characters）印字可能な ASCIIキャラクタ \u0020-\u007F のうち上記を除くものすべてと、数学記号(Sm)、他の記号(So)とから成ります。

1.1 識別子（Identifiers)

構文:

```
op         ::=   opchar {opchar}
varid      ::=   lower idrest
plainid    ::=   upper idrest
               |   varid
               |   op
id         ::=   plainid
               |   '\`' stringLit '\`'
idrest     ::=   {letter | digit} ['_' op]
```

There are three ways to form an identifier. First, an identifier can start with a letter which can be followed by an arbitrary sequence of letters and digits. This may be followed by underscore '_' characters and another string composed of either letters and digits or of operator characters. Second, an identifier can start with an operator ch

aracter followed by an arbitrary sequence of operator characters. The preceding two fo rms are called plain identifiers. Finally, an identifier may also be formed by an arbi trary string between back-quotes (host systems may impose some restrictions on which s trings are legal for identifiers). The identifier then is composed of all characters e xcluding the backquotes themselves.

識別子には 3 つの形があります。 1 つめは、文字で始まり、文字または 10進数字の任意の並びが後に続く識別子です。 これには、下線 '_'を後に続けることができ、文字あるいは 10進数字から構成される他の文字列、または演算子文字から構成される文字列を続けることができます。 2 つめは、演算子文字で始まり、演算子文字の任意の並びが続く識別子です。 これら前の 2 つの形は**プレーン**な識別子（訳注:訳は、一般識別子)と呼ばれます。 最後の形は、任意の文字列をバッククォートで囲んだものです（ホストシステムは、文字列が識別子として正しいということに制限を課すかもしれません)。 識別子はバッククォート自身を除くすべての文字で構成されることになります。

通常は、最長一致規則が適用されます。 例えば、文字列

```
big_bob++=`def`
```

は、3 つの識別子 big_bob、++=、def に分解されます。 パターンマッチングの規則では、さらに、小文字から始まる**変数識別子(variable identifiers)**と、そうでない**定数識別子(constant identifiers)**を区別します。

'$' 文字はコンパイラが生成する識別子用に予約されています。 ユーザープログラムは '$' 文字を含む識別子を定義すべきではありません。

次の名前は予約語であり、字句上の識別子の構文上の識別種には属していません。

abstract	**case**	**catch**	**class**	def
do	**else**	**extends**	**false**	**final**
finally	**for**	**forSome**	**if**	**implicit**
import	lazy	**match**	**new**	**null**
object	override	**package**	private	**protected**
return	**sealed**	**super**	**this**	**throw**
trait	try	**true**	**type**	val
var	**while**	**with**	**yield**	
_	**:**	**=**	**=>**	**<-**
<:	**<%**	**>:**	**#**	**@**

ASCII の '=>'と '<-' に等価な、ユニコード演算子 \u21D2 '⇒' と \u2190 '←' も予約されています。

Example 1.1.1 次は識別子の例です:

```
x           Object        maxIndex        p2p         empty_?
+           `yield`       αρετη      _y          dot_product_*
__sysstem   _MAX_LEN_
```

Example 1.1.2文字列をバッククォートで囲むことは、Scala の予約語である Java 識別子にアクセスする必要があるときの 1 つの解決策です。 例えば、文 Thread.yield() は、yield が Scala の予約語ですから、文法違反です。 しかし、次の回避策があります:

```
Thread.`yield`()
```

1.2 改行文字

構文:

```
semi ::= ';' |  nl {nl}
```

Scala は、文をセミコロンあるいは改行によって終えてもよい行指向(line-oriented)言語です。 Scala ソーステキスト中の改行は、もし次の 3 つの条件が満たされるなら、特別なトークン "nl" として扱われます:

1. 改行の直前のトークンで文を終結できる。
2. 改行の直後のトークンで文を始めることができる。
3. 改行が可能な領域でトークンが現れる。

文を終結させることができるトークンは、リテラル、識別子と次の区切り文字と予約語です。

```
this      null      true      false     return      type      <xml-start>
_         )         ]         }
```

文を始めることができるトークンは、次のデリミタと予約語**以外の**、すべての Scala トークンです。

```
catch    else       extends     finally      forSome      match
with     yield      ,    .      ;    :      =    =>     <-    <:    <%
>:    #    [      )    ]      }
```

case トークンは、class あるいは object トークンが後に続くときのみ、文を始めることができます。

改行は次の場所で可能です。

1. Scala ソースファイルのすべて。ただし改行できないネストした領域を除く。
2. 対応する波括弧トークン { と } の間。ただし改行できないネストした領域を除く。

改行できない場所は次です

1. 対応する丸括弧トークン (と) の間。 ただし改行が可能なネストした領域を除く。
2. 対応する角括弧トークン [と] の間。 ただし改行が可能なネストした領域を除く。
3. case トークンと対応する => トークンの間。 ただし改行が可能なネストした領域を除く。
4. XML モード(§1.5)で解析される全ての領域。

XML における {...}エスケープの波括弧文字と、文字列リテラルはトークンではないことに注意してください。 ですから、それらは改行可能な領域を囲みません。

Normally, only a single nl token is inserted between two consecutive non-newline tokens which are on different lines, even if there are multiple lines between the two tokens. However, if two tokens are separated by at least one complete blank line (i.e. a line which contains no printable characters), then two nl tokens are inserted.

通常、ただ 1 つの nl トークンが、異なる行にある継続する 2 つの非改行トークンの間に、たとえその 2 つのトークン間に多数の行があるとしても、挿入されます。 しかし、もし 2 つのトークンが、少なくとも 1 つの完全な空白行(すなわち、印字可能な文字を含んでいない行)で

分割されているなら、 2 つの nl トークンが挿入されます。

Scala 文法(詳細は Chapter A)には、セミコロンではなく、オプションの nl トークンが受け入れられる場所の規則があります。 それには、そういった場所の 1 つで改行しても式あるいは文は終結しない、という効果があります。 それらの場所は次のように要約できます:

複数の改行トークンは次の場所で受け入れられます（改行の代わりにセミコロンにすると、 すべての場合において不正となることに注意してください)。

- 条件式（§6.16）あるいは whileループ(§6.17)の条件と、次に続く式との間。
- for内包表記の列挙子リストと、次に続く式との間。
- 型定義あるいは型宣言(§4.3)中の 最初の type キーワードの後。

次の場合、ただ 1 つの改行が受け入れられます。

- 左波括弧 "{"の前、ただしその左波括弧が現在の文あるいは式の 正しい続きである場合。
- 中置演算子の後、ただし次行の最初のトークンが式（§6.12）を始めることができる場合。
- パラメータ節（§4.6）の前。
- アノテーション（§11）の後。

Example 1.2.1次のコードは、それぞれ 2 行からなる 4 つの正しい形の文です。 2 行の間の改行トークンは、文の区切りとして扱われません。

```
if (x > 0)
  x = x - 1

while (x > 0)
  x = x / 2

for (x <- 1 to 10)
  println(x)

type
  IntList = List[Int]
```

Example 1.2.2 次のコードは無名クラスを定めます。

```
new Iterator[Int]
{
  private var x = 0
  def hasNext = true
  def next = { x += 1; x }
}
```

改行を入れると、同じコードはローカルなブロックが後に続く、オブジェクト生成として解釈されます。

```
new Iterator[Int]

{
  private var x = 0
```

```
  def hasNext = true
  def next = { x += 1; x }
}
```

Example 1.2.3 次のコードはただ 1 つの式を定めます。

```
x < 0 ||
x > 10
```

改行を入れると、同じコードは 2 つの式として解釈されます。

```
x < 0 ||

x > 10
```

Example 1.2.4 次のコードはただ 1 つのカリー化された関数を定めます。

```
def func(xa: Int)
        (y: Int) = x + y
```

改行を入れると、同じコードは抽象関数定義と文法違反の文として解釈されます。

```
def func(x: Int)

        (y: Int) = x + y
```

Example 1.2.5

次のコードは属性を持った定義を定めます。

```
@serializable
protected class Data { ... }
```

改行を入れると、同じコードは 1 つの属性と 1 つの別の文(文法違反の文)として解釈されます。

```
@serializable

protected class Data { ... }
```

1.3 リテラル（Literals）

整数、浮動小数点数、文字、ブール値、シンボル、文字列用のリテラルがあります。 これらのリテラル構文は、それぞれ Java と同じです。

構文:

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

1.3.1 整数リテラル（Integer Literals）

構文:

```
integerLiteral ::=  (decimalNumeral | hexNumeral | octalNumeral) ['L' | 'l']
decimalNumeral ::=  '0' | nonZeroDigit {digit}
hexNumeral     ::=  '0' 'x' hexDigit {hexDigit}
octalNumeral   ::=  '0' octalDigit {octalDigit}
digit          ::=  '0' | nonZeroDigit
nonZeroDigit   ::=  '1' | … | '9'
octalDigit     ::=  '0' | … | '7'
```

整数リテラルは通常 Int 型であり、L あるいは l 接尾辞が後に続くなら Long 型です。 Int 型の値は、-2^{31} 以上 $2^{31} - 1$ 以下のすべての整数です。Long 型の値は、-2^{63} 以上 $2^{63} - 1$ 以下のすべての整数です。 もし整数リテラルがこれらの範囲外の数を表すなら、コンパイルエラーが発生します。

しかし、もし式中のリテラルの要請型(expected type) pt (§6.1)が Byte、Short、 あるいは Char のいずれかであり、その整数値が型によって定義された数値の範囲内なら、その数は型 pt に変換されます。 それらの型の数値範囲は次です:

```
Byte          -2^7 以上 2^7 - 1 以下
Short         -2^15 以上 2^15 - 1 以下
Char          0 以上 2^16 - 1 以下
```

Example 1.3.1 次は整数リテラルの例です:

```
0             21           0xFFFFFFFF          0777L
```

1.3.2 浮動小数点リテラル（Floating Point Literals）

構文:

```
floatingPointLiteral ::=   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'
```

浮動小数点リテラルは、接尾辞 F あるいは f が後に続くなら Float 型であり、そうでなければ Double 型です。 Float 型はすべての IEEE 754 32ビット単精度バイナリ浮動小数点値からなるのに対して、 Double 型はすべての IEEE 754 64ビット倍精度バイナリ浮動小数点値からなります。

もしプログラム中の浮動小数点リテラルの後に文字で始まるトークンが続くなら、2 つのトークン間に少なくとも 1 つの空白文字がなければなりません。

Example 1.3.2 次は浮動小数点リテラルの例です。

```
0.0       1e30f          3.14159f      1.0e-100          .1
```

Example 1.3.3

語句 '1.toString' は 3 つの異なるトークン '1'、'.'、'toString' に分解されます。 他方、もしピリオドの後に空白を入れると、'1 .toString' は、浮動小数点リテラル '1.' の後に識別子 'toString'が続いていると解析されます。

1.3.3 ブーリアンリテラル（Boolean Literals）

構文:

```
booleanLiteral   ::=    'true' | 'false'
```

ブーリアンリテラル true と false は、Boolean型のメンバーです。

1.3.4 文字リテラル（Character Literals）

構文:

```
characterLiteral   ::=    '\'' printableChar '\''
                        |    '\'' charEscapeSeq '\''
```

文字リテラルは引用符で囲まれた単一文字です。 文字は、印字可能なユニコード文字であるか、エスケープシーケンス(§1.3.6)で記述されるかのいずれかです。

Example 1.3.4 次は文字リテラルの例です。

```
'a'   '\u0041'        '\n'   '\t'
```

'\u000A' は有効な文字リテラル**ではない**ことに注意してください。 なぜなら、リテラル解析の前にユニコード変換され、ユニコード文字 \u000A（ラインフィード）は印字可能な文字ではないからです。 その代わりに、エスケープシーケンス 'n' あるいは 8進法のエスケープ '\12' (§1.3.6)を使えます。

1.3.5 文字列リテラル (String Literals)

構文:

```
stringLiteral    ::=   '\"' {stringElement} '\"'
stringElement    ::=   printableCharNoDoubleQuote  |  charEscapeSeq
```

文字列リテラルはダブルクォートの中の文字の並びです。 文字は、印字可能なユニコード文字であるか、エスケープシーケンス(§1.3.6) で記述されるかのいずれかです。 もし文字列リテラルがダブルクォート文字を含むなら、エスケープして \" としなくてはなりません。 文字列リテラルの値は、クラス String のインスタンスです。

Example 1.3.5 次は文字列リテラルの例です

```
"Hello,\nWorld!"
"This string contains a \" character."
```

複数行文字列リテラル (Multi-Line String Literals)

構文:

```
stringLiteral      ::=   '"""' multiLineChars '"""'
multiLineChars     ::=   {['"'] ['"'] charNoDoubleQuote} {'"'}
```

複数行文字列リテラルは 3 つの引用符 """ ...""" で囲まれた文字の並びです。 文字の並びは、その最後にだけ 3 つ以上の連続した引用符を含むことができる以外、任意です。 文字は必ずしも印字可能である必要はありません。 改行あるいは他の制御文字も同様に許されます。 ユニコードエスケープは他の場所と同様に機能しますが、しかし(§1.3.6) 中のエスケープシーケンスはどれも解釈されません。

Example 1.3.6 次は複数行文字列リテラルの例です。

```
"""the present string
   spans three
   lines."""
```

これは次の文字列になります。

```
the present string
     spans three
     lines.
```

Scala ライブラリには実用的なメソッド stripMargin があり、複数行文字列から先頭の空白文字を取り去るのに使えます。 式

```
"""the present string
  |spans three
  |lines.""".stripMargin
```

は、次のように評価されます。

```
the present string
spans three
lines.
```

メソッド stripMargin はクラス scala.collection.immutable.StringLike の中で定義されています。 String から StringLike への事前定義された暗黙変換(§6.26)があるので、 このメソッドはすべての文字列に適用可能です。

1.3.6 エスケープシーケンス（Escape Sequences）

次のエスケープシーケンスは、文字あるいは文字列リテラル中で認識されます。

```
\b          \u0008: バックスペース BS
\t          \u0009: 水平タブ HT
\n          \u000a: ラインフィード LF
\f          \u000c: フォームフィード FF
\r          \u000d: キャリッジリターン CR
\"          \u0022: ダブルクォート "
\'          \u0027: シングルクォート '
\\          \u005c: バックスラッシュ \
```

ユニコードで 0 から 255 の文字は、8進法のエスケープ、すなわちバックスラッシュ '\' と、それに続く最大 3 つの 8進文字の並びで表せます。

文字あるいは文字列リテラル中のバックスラッシュ文字が、有効なエスケープシーケンスを開始しないなら、コンパイルエラーが発生します。

1.3.7 シンボルリテラル（Symbol literals）

構文:

```
symbolLiteral   ::=   ''' plainid
```

シンボルリテラル 'x は、式 scala.Symbol("x") の略記表現です。 Symbol はケースクラス(§5.3.2)であり、次のように定義されています。

```
package scala
final case class Symbol private (name: String) {
  override def toString: String = "'" + name
}
```

Symbol のコンパニオンオブジェクトの apply メソッドは、 Symbols への弱い参照をキャッシュしており、同じ識別子のシンボルリテラルが参照等価性の観点で等しいことを保証しています。

1.4 空白文字とコメント (Whitespace and Comments)

トークンは空白文字あるいはコメントで分割できます。 コメントには 2 つの形があります:

一行コメントは // で始まる文字の並びであり、行末に及びます。

複数行コメントは /* と */ の間の文字の並びです。 複数行コメントはネストできます。 しかし適切にネストされるよう要求されます。 ですから、/* /* */ のようなコメントは終わらないコメントとして却下されます。

1.5 XML モード (XML mode)

XML 小部分のリテラル記述を可能とするために、字句解析は、次の環境中で 始め山括弧 '<' に出会うと Scala モードから XML モードへと変わります: '<' の前には空白文字、左丸括弧あるいは左波括弧がなけれならず、そしてすぐ後に XML 名を始める文字が続かなくてはなりません。

構文:

```
( whitespace | '(' | '{' ) '<' (XNameStart | '!' | '?')

XNameStart ::= '_' | BaseChar | Ideographic  (W3C XMLと同様、但し ':' なし
```

次のいずれかの場合、スキャナは XML モードから Scala モードへと変わります。

- the XML expression or the XML pattern started by the initial '<' has been successfully parsed, or if
- the parser encounters an embedded Scala expression or pattern and forces the Scanner back to normal mode, until the Scala expression or pattern is successfully parsed. In this case, since code and XML fragments can be nested, the parser has to maintain a stack that reflects the nesting of XML and Scala expressions adequately.

- 最初の '<' で始まる XML 式あるいは XML パターンが成功裏に解析されるか、あるいは、
- パーサーが、埋め込まれた Scala 式/パターンに遭遇して、 Scala 式/パターンが成功裏に解析されるまで、 スキャナを通常モードに戻すことを強制するとき。 この場合、コードと XML 小部分はネストできるので、パーサーは XML と Scala 式のネストを反映するスタックを適切に維持する必要があります。

Scala トークンは XML モードでは構築されず、コメントはテキストとして解釈されることに注意してください。

Example 1.5.1

次の値定義は、2つの埋め込まれた Scala 式をもつ XML リテラルを使っています。

```
val b = <book>
          <title>The Scala Language Specification</title>
          <version>{scalaBook.version}</version>
          <authors>{scalaBook.authors.mkList("", ", ", "")}</authors>
        </book>
```

2 識別子・名前・スコープ（Identifiers, Names and Scopes）

Scala における名前は、ひとまとめに**エンティティ**と呼ばれる、型、値、メソッド、そしてクラスを識別します。 名前はローカルな定義と宣言(§4)、継承(§5.1.3)、インポート節(§4.7)、あるいはパッケージ節(§9.2) によって導入され、それらはひとまとめに、**束縛(bindings)**と呼ばれます。

異なる種類の束縛は、それぞれ優先順位を持っています:

1. Definitions and declarations that are local, inherited, or made available by a pa ckage clause in the same compilation unit where the definition occurs have highes t precedence.
2. Explicit imports have next highest precedence.
3. Wildcard imports have next highest precedence.
4. Definitions made available by a package clause not in the compilation unit where the definition occurs have lowest precedence.

1. 定義と宣言は、ローカルなもの、継承されたもの、 あるいは定義が現れる同じコンパイル単位中のパッケージ節を通して 利用可能となったものが、最も高い優先順位を持っています。
2. 明示的なインポートが次に最も高い優先順位を持っています。
3. ワイルドカードによるインポートが次に最も高い優先順位を持っています。
4. 定義が現れるコンパイル単位中にないパッケージ節を通して利用可能となった定義が、 最も低い優先順位を持っています。

2 つの異なる名前空間があり、一つは型(types §3)のためのもの、もう一つは項(terms §6)のためのものです。 その名前が使われるコンテキストによっては、同じ名前で 1 つの型と 1 つの項を指定できます。

束縛は**スコープ**を持ち、そこでは、ただ 1 つの名前によって定義されたエンティティは、単純名を使ってアクセスできます。 スコープはネストされます。 内側のスコープ中の束縛は、同じスコープ中のより低い優先順位の束縛を**隠します(shadow)**。 また、外側のスコープ中の、同じかあるいはより低い優先順位の束縛も隠します。

隠すことは、半順序関係にすぎないことに注意してください。 次の状況では

```
val x = 1;
{ import p.x;
  x }
```

x のいずれの束縛も、他を隠しません。 したがって、上記 3 行目の x への参照は曖昧です。

限定修飾されていない（型あるいは項の）識別子 x への参照は、ただ 1 つの束縛に結びつけられます。 それは、

- 識別子と同じ名前空間中で名前 x のエンティティを定義し、
- その名前空間中で名前 x のエンティティを定義する、他のすべての束縛を隠します。

It is an error if no such binding exists. If x is bound by an import clause, then the simple name x is taken to be equivalent to the qualified name to which x is mapped by the import clause. If x is bound by a definition or declaration, then x refers to the entity introduced by that binding. In that case, the type of x is the type of the refe renced entity.

もしそのような束縛が存在しないなら、エラーです。 もし x がインポート節によって束縛されるなら、単純名 x は、インポート節によって x がマップされる、限定修飾された名前に等しいと解釈されます。 もし x が定義あるいは宣言によって束縛されるなら、x はその束縛によって導入されたエンティティを参照します。 この場合、x の型は参照されるエンティティの型です。

Example 2.0.2パッケージ P と Q 中に X という名前の 2 つのオブジェクト定義があるとします。

```
package P {
  object X { val x = 1; val y = 2 }
}

package Q {
  object X { val x = true; val y = "" }
}
```

次のプログラムは、それらの間の異なる種類の束縛と優先順位を示します。

```
package P {                    // 'X' パッケージ節による束縛
  import Console._             // 'println' ワイルドカード節による束縛
  object A {
    println("L4: "+X)          // ここでは、'X' は 'P.X' を参照
    object B {
      import Q._               // 'X' ワイルドカード節による束縛
      println("L7: "+X)        // ここでは 'X' は 'Q.X' を参照
      import X._               // 'x' と 'y' ワイルドカード節による束縛
      println("L8: "+x)        // ここでは 'x' は 'Q.X.x' を参照
      object C {
        val x = 3              // 'x' ローカル定義よって束縛
        println("L12: "+x)     // ここでは、'x' は定数 '3' を参照
        { import Q.X._         // 'x' と 'y' は、ワイルドカード節による束縛
//        println("L14: "+x)    // ここでは 'x' への参照は曖昧
          import X.y           // 'y' 明示的インポートによる束縛
          println("L16: "+y)   // ここでは 'y' は 'Q.X.y' を参照
          { val x = "abc"      // 'x' ローカル定義よって束縛
            import P.X._       // 'x' と 'y' ワイルドカード節による束縛
//          println("L19: "+y) // ここでは 'y' への参照は曖昧
            println("L20: "+x) // ここでは 'x' は文字列 ''abc''あ を参照
}}}}}}
```

限定修飾された(型あるいは項の)識別子 e.x への参照は、識別子と同じ名前空間中の名前 x を持つ、e の型 T のメンバーを参照します。 もし T が値型(§3.2)でないなら、エラーです。 e.x の型は、T 中の参照されたエンティティのメンバー型です。

3 型（Types）

構文:

```
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}
```

We distinguish between first-order types and type constructors, which take type parameters and yield types. A subset of first-order types called value types represents sets of (first-class) values. Value types are either concrete or abstract.

一階型と、パラメータを指定して型を生み出す型コンストラクタとを区別します。 **値型(value types)**と呼ばれる一階型の部分集合は、(first class:第一級の)値の集合を表します。 値型は**具象あるいは抽象**のいずれかです。

すべての具象値型は、**クラス型**として表せます。 すなわち、クラスやトレイト(*1)(§5.3)を参照する型指定子（§3.2.3)、あるいは、型の論理積(intersection)を表す**複合型**(§3.2.7)としてです。 複合型には、そのメンバーの型にさらなる制約を与える細別(refinement §3.2.7)を加えることができます。 抽象値型は、型パラメータ(§4.4)と抽象型束縛(§4.3)によって導入されます。 型の中では、丸括弧をグルーピングのために使えます。

(*1）我々はオブジェクトとパッケージも暗黙のうちに（ユーザープログラムはアクセスできない、オブジェクトあるいはパッケージと同じ名前の）クラスを定義すると考えます。

非値型は、値でない識別子のプロパティを取り込みます(§3.3)。 例えば、型コンストラクタ(§3.3.3）は直接には値の型を指定しません。 しかし、型コンストラクタが正しい型引数に適用されれば、それは値型であろう一階型をもたらします。

非値型は Scalaでは間接的に表現されます。 例えば、メソッド型はメソッドのシグニチャを書き出して記述できますが、それ自身は実際の型でないけれども、対応するメソッド型(§3.3.1) を引き起こします。型コンストラクタは他の例です。 型を Swap[m[_, _], a,b] = m[b, a] と書けますが、しかし、対応する無名の型関数を直接的に書くための構文はありません。

3.1 パス（Paths）

構文:

```
Path              ::= StableId
                    | [id '.'] this
StableId          ::= id
                    | Path '.' id
                    | [id '.'] 'super' [ClassQualifier] '.' id
ClassQualifier    ::= '[' id ']'
```

パスそれ自身は型ではありません。 しかしそれは、名前付きの型(named type)の一部であり、その機能は Scala 型システムの中で中心的役割を果たします。

パスは次のうちの１つです。

- 空のパス（ユーザープログラム中では、明示的には書けません)。
- C.this。ここで C はクラスを参照します。 パス this は C.this の略記表現として解釈されます。 ここで C は、参照を直接囲むクラスの名前です。
- p.x where p is a path and x is a stable member of p. Stable members are packages or members introduced by object definitions or by value definitions of non-volatile types (§3.6).
- p.x。ここで p はパス、x は p の安定メンバー。 **安定メンバー(stable members)**とは、オブジェクト定義によってあるいは非 volatile 型の値定義によって導入されたメンバーあるいはパッケージのことです(§3.6)。
- C.super.x or C.super[M].x where C references a class and x references a stable member of the super class or designated parent class M of C. The prefix super is taken as a shorthand for C.super where C is the name of the class directly enclosing the reference.
- C.super.x あるいは C.super[M].x。 ここで C はクラスを参照し、x は、スーパークラスあるいは C の指定された親クラス M の、安定メンバーを参照します。 前置子 super は、C.super の略記表現として解釈されます。 ここで C は、参照を直接囲むクラスの名前です。

安定識別子(stable identifier)とは、最後が識別子で終わるパスです。

3.2 値型（Value Types）

Scala のすべての値は、次のどれか 1 つの形をとる型を持っています。

3.2.1 シングルトン型（Singleton Types）

構文:

```
SimpleType  ::=  Path '.' type
```

シングルトン型は p.type の形の型であり、ここで p はパスで、scala.AnyRefに適合する(§6.1)ことが要請される値を指し示します。 この型は、null と p で示される値からなる集合を表します。

安定型(stable type)とは、シングルトン型あるいは、トレイト scala.Singleton のサブ型（subtype）であると宣言された型、のいずれかです。

3.2.2 型射影（Type Projection）

構文:

```
SimpleType  ::=  SimpleType '#' id
```

型射影 T#x は、型 T の x という名前の型メンバーを参照します。

3.2.3 型指定子（Type Designators）

構文:

```
SimpleType  ::=  StableId
```

型指定子は、名前付きの値型を参照します。 それは単純名であるか、あるいは限定修飾されます。 そのような型指定子はすべて、型射影の略記表現です。

特に、限定修飾されていない型名 t は、C.this.type#t の略記表現として解釈されます。 ここで t はあるクラス、オブジェクト、あるいはパッケージ C 内で束縛されているものとします。 もし t がクラス、オブジェクト、あるいはパッケージ内で束縛されていないなら、t は ε.type#t の略記表現と解釈されます。

限定修飾された型指定子は、p.t の形をしています。 ここで p はパス(§3.1)、t は型名です。 そのような型指定子は型射影 p.type#t に等価です。

Example 3.2.1

次は、いくつかの型指定子とそれらの展開をリストしています。 ここで、ローカルな型パラメータ t、型メンバー Nodeを持つ値 maintable、標準的なクラス scala.Int があると仮定します。

```
t                       ε.type#t
Int                     scala.type#Int
scala.Int               scala.type#Int
```

```
data.maintable.Node       data.maintable.type#Node
```

3.2.4 パラメータ化された型 (Parameterized Types)

構文:

```
SimpleType  ::=   SimpleType TypeArgs
TypeArgs    ::=   '[' Types ']'
```

パラメータ化された型 T[U1,...,Un] は、型指定子 T と型パラメータ U1,...,Un (n >= 1) から成ります。 T は、n 個の型パラメータ a1,...,an をとる型コンストラクタを参照していなければなりません。

例えば、型パラメータが下限境界 L1,...,Ln と上限境界 U1,...,Un を持っているとします。 もし、それぞれ実際の型パラメータがその**両境界に適合する**なら、つまり σLi <: Ti <: σUi なら (ここで σは置換[a1 := T1,...,an := Tn])、このパラメータ化された型は正しい形(well formed)です。

Example 3.2.2 次のような部分的な型定義が与えられているとします。

```
class TreeMap[A <: Comparable[A], B] { ... }
class List[A] { ... }
class I extends Comparable[I] { ... }

class F[M[_], X] { ... }
class S[K <: String] { ... }
class G[M[ Z <: I ], I] { ... }
```

このとき、次のパラメータ化された型は正しい形です。

```
TreeMap[I, String]
List[I]
List[List[Boolean]]

F[List, Int]
G[S, String]
```

Example 3.2.3例3.2.2の型定義が与えられているとき、次の型は不正な形(ill-formed)です。

```
TreeMap[I]              // 不正: パラメータ数誤り。
TreeMap[List[I], Int]   // 不正: 型パラメータが境界外。

F[Int, Boolean]         // 不正: Int は型コンストラクタではない。
F[TreeMap, Int]         // 不正: TreeMap は 2 つのパラメータをとるが、
                        //       F は 1 パラメータのコンストラクタである。
G[S, Int]               // 不正: S は、Stringに適合するパラメータを含み、
                        //       G は、Intに適合するパラメータをもつ
```

```
                    //          型コンストラクタが要請される。
```

3.2.5 タプル型

構文:

```
SimpleType  ::=     '(' Types ')'
```

タプル型 (T1,...,Tn) は、クラス scala.Tuplen[T1,...,Tn] のエイリアスです。ここで n >= 2。

タプルクラスはケースクラスであり、セレクター _1, ..., _n を使ってそのフィールドにアクセスできます。

それらの関数は、対応する Product トレイト内にまとめられています。 n 項タプルクラスと product トレイトは、標準的な Scala ライブラリにおいて、少なくとも次のように定義されています (他のメソッドも加え、また、他のトレイトも実装しているかもしれません)。

```
case class Tuplen[+T1, ..., +Tn](_1: T1, ..., _n : Tn)
extends Productn[T1, ..., Tn] {}

trait Productn[+T1, +T2, +Tn] {
  override def arity = n
  def _1: T1
  ...
  def _n :Tn
}
```

3.2.6 アノテーション型 (Annotated Types)

構文:

```
AnnotType   ::=   SimpleType {Annotation}
```

アノテーション型 T a1...an は、型 T にアノテーション a1,...,an を付加します(§11)。

Example 3.2.4次の型は、文字列型に @suspendable@ アノテーションを付加します:

```
String @suspendable
```

3.2.7 複合型 (Compound Types)

(訳注:複合型については [[A Tour of Scala:Compound Types>http://www.scala-lang.org/node/110]] に簡単な説明があります)

構文:

```
CompoundType    ::=  AnnotType {'with' AnnotType} [Refinement]
                   |  Refinement
Refinement      ::=  [nl] '{' RefineStat {semi RefineStat} '}'
RefineStat      ::=  Dcl
                   |  'type' TypeDef
                   |
```

複合型 T1 with ... with Tn {R} は、構成要素型 T1,...,Tn と細別 {R} で与えられたメンバーをもつオブジェクトを表わします。 細別 {R} は、宣言と型定義を含みます。 もし宣言あるいは定義が、構成要素型 T1,...Tn の 1 つの中の宣言あるいは定義をオーバライドするなら、オーバライドの通常の規則が適用されます(§5.1.4); そうでないなら、宣言あるいは定義は「構造的」と言われます(*2)。

(*2) 構造的に定義されたメンバーへの参照 (メソッド呼び出しあるいは、値/変数へのアクセス) は、非構造的メンバーを参照する等価なコードより、際立って遅いバイナリコードを生成することがあります。

構造的な細別内のメソッド宣言では、すべての値パラメータの型は、その細別内に含まれる型パラメータあるいは抽象型だけを参照できます。 すなわち、メソッド自身の型パラメータへの参照であるか、あるいは、細別内の型定義への参照のいずれかです。 この制限は関数の結果型には適用されません。

もし細別が与えられていなければ、空の細別が暗黙のうちに加えられます。 つまり、T1 with ... with Tn は、T1 with ... with Tn {} の略記表現です。

複合型は、先行する構成要素型のない、細別 {R} だけから成ることもあります。 そのような型は AnyRef {R} に等価です。

Example 3.2.5次の例は、関数のパラメータ型が構造的宣言をもつ細別を含む場合の、宣言方法と使い方を示しています。

```
case class Bird (val name: String) extends Object {
  def fly(height: Int) = ...
  ...
}
case class Plane (val callsign: String) extends Object {
  def fly(height: Int) = ...
  ...
}
def takeoff(
      runway: Int,
      r: { val callsign: String; def fly(height: Int) }) = {
  tower.print(r.callsign + " requests take-off on runway " + runway)
  tower.read(r.callsign + " is clear for take-off")
  r.fly(1000)
  }
val bird = new Bird("Polly the parrot"){ val callsign = name }
val a380 = new Plane("TZ-987")
takeoff(42, bird)
```

```
takeoff(89, a380)
```

Bird と Plane は Object 以外の親クラスを共有していませんが、関数 takeoff のパラメータ r は、構造的宣言の細別を使って定義されていて、値 callsign と fly 関数を宣言する任意のオブジェクトを受けつけます。

3.2.8 中置型 (Infix Types)

構文:

```
InfixType   ::=    CompoundType {id [nl] CompoundType}
```

中置型 T1 op T2 は、2 つのオペランド型 T1 と T2 へ適用される中置演算子 op からなります。 この型は型適用 op[T1,T2] に等価です。 中置演算子 op には、* 以外の任意の識別子が使えます。 * は反復パラメータ型(§4.6.2)を表す後置修飾子として予約されています。

すべての中置型演算子は同じ優先順位を持っています; グルーピングには丸括弧を使われなければなりません。 型演算子の結合性(§6.12)は項演算子(term operator)については決まっています。 ':'で終わる型演算子は右結合です。他のすべての演算子は左結合です。

連続する中置型演算のシーケンス t0 op1 t1 op2 ... opn tn では、すべての演算子 op1,...,opn が同じ結合性を持っていなくてはなりません。 もしそれらがすべて左結合なら、シーケンスは (... (t0 op1 t1) op2 ...) opn tn と解釈され、そうでなければ t0 op1 (t1 op2 (... opn tn) ...) と解釈されます。

3.2.9 関数型 (Function Types)

構文:

```
Type            ::=   FunctionArgs '=>' Type
FunctionArgs    ::=   InfixType
                  |   '(' [ ParamType {',' ParamType } ] ')'
```

型 (T1,...,Tn) => U は、型 T1,...,Tn を引数にとり、型 U の結果をもたらす関数値の集合を表します。 引数が正確に 1 つだけの場合は、型 T => U は (T) => U の略記表現です。 形が => T の引数型は、型 T の名前呼出しパラメータ (§4.6.1)を表します。

関数型は右へ結合します。 例えば、S => T => U は S => (T => U) と同じです。

関数型は、apply 関数を定義するクラス型の略記表現です。 特に、n 項関数型 (T1,...,Tn) => U は、クラス型 Functionn[T1,...,Tn,U] の略記表現です。 このようなクラス型は、0 以上 9 以下の n に対して Scala ライブラリ中で、次のように定義されています。

```
package scala
trait Functionn[-T1,...,-Tn , +R] {
  def apply(x1 : T1,...,xn : Tn ): R
  override def toString = "<function>"
}
```

したがって、関数型は、その結果型については共変(§4.5)、その引数型については反変です。

3.2.10 存在型 (Existential Types)

構文:

```
Type                ::= InfixType ExistentialClauses
ExistentialClauses  ::= 'forSome' '{' ExistentialDcl
                        {semi ExistentialDcl} '}'
ExistentialDcl      ::= 'type' TypeDcl
                      | 'val' ValDcl
```

存在型は、T forSome { Q } の形をしています。 ここで Q は型宣言(§4.3)の並びです。 t1[tps1] >: L1 <: U1,...,tn[tpsn] >: Ln <: Un を Q 内で宣言された型であるとします (型パラメータ部 [tpsi] のどれかが欠けていても構いません)。 各型 ti のスコープは、型 T と存在節 Q を含みます。 型変数 ti は、型 T forSome { Q } 内で**束縛**されていると言われます。 型 T の中に現れるけれど T 内で束縛されない型変数は、T 内で**自由(free)**であると言われます。

T forSome { Q }の**型インスタンス**は、型 σT です。 ここで σは、各 i に対し σLi <: σti <: σUi であるような t1,...,tn の置換とします。 存在型 T forSome { Q } によって示される値の集合は、そのすべての型インスタンスの値の集合の和集合です。

T forSome { Q } の skolemization は、型インスタンス σT です。ここで σは、[t´1/t1,...,t´n/tn] の置換で、各 t´i は、下限境界 σLi と上限境界 σUi をもつ新規の抽象型です。

簡略化規則

存在型は、次の 4 つの等価規則に従います。

1. 存在型中の複数の for 節をマージできます。 例えば、T forSome { Q } forSome { Q´} は T forSome { Q ; Q´} に等価です。
2. 使われていない存在量化(quantification)を外すことができます。 例えば、T forSome { Q ; Q´} は、T forSome { Q } に等価です。 ただしここで Q´中で 定義されたどの型も、T あるいは Q によって参照されないものとします。
3. 空の存在量化を外すことができます。 例えば、T forSome { } は T に等価です。
4. 存在型 T forSome { Q } (ここで Q は節 type t[tps] >: L <: U を含む) は、 型 T´ forSome { Q } に等価です。 ここで T´は、T 中で共変(§4.5)として出現するすべての t を U で置き換え、 T 中で反変として出現するすべての t を L で置き換えて得られるものです。

値の存在量化 (Existential Quantification over Values)

文法上の便宜として、存在型中の束縛節に値宣言 val x : T を含めることができます。 存在型 T forSome { Q; val x : S ; Q´ } は、型 T´ forSome { Q ; type t <: S with Singleton; Q´ } の略記表現として扱われます。 ここで t は新たな型名で、T´ は T 中に現れる x.type を t で置き換えて得られるものです。

存在型のプレースホルダ構文

構文:

```
WildcardType      ::=   '_' TypeBounds
```

Scala は存在型のプレースホルダ構文をサポートしています。 **ワイルドカード型**は _ >: L <: U の形です。 それぞれの境界節はともに省略されるかもしれません。 もし下限境界節 >: L がないなら、>: scala.Nothing が想定されます。 もし上限境界節 <: U がないなら、<: scala.Any が想定されます。 ワイルドカード型は存在量化された型変数の略記表現です。 ここで、存在量化は暗黙的です。

ワイルドカード型は、パラメータ化された型の型引数として現われなくてはなりません。 いま、T = p.c[targs,T,targs´] をパラメータ化された型とします。 ここで targs,targs´ は空きでもよく、T はワイルドカード型 _ >: L <: U です。 このとき T は、次の存在型

```
p.c[targs,t,targs´] forSome { type t >: L <: U }
```

と等価になります。 ここで t は、ある新しい型変数とします。 ワイルドカード型は、中置型(§3.2.8)、関数型(§3.2.9)、あるいはタプル型(§3.2.5)等の一部にも現れることがあります。 それらの展開は、等価なパラメータ化された型内での展開となります。

Example 3.2.6 次のクラス定義を仮定します。

```
class Ref[T]
abstract class Outer { type T }
```

次は、存在型のいくつかの例です。

```
Ref[T] forSome { type T <: java.lang.Number }
Ref[x.T] forSome { val x: Outer }
Ref[x_type# T] forSome { type x_type <: Outer with Singleton }
```

上記リストの最後の 2 つの型は等価です。 最初の型をワイルドカード構文を使って書き換えると次のようになります。

```
Ref[_ <: java.lang.Number]
```

Example 3.2.7型 List[List[_]] は、次の存在型に等価です。

```
List[List[t] forSome { type t }]
```

Example 3.2.8 次の共変型があるとします。

```
class List[+T]
```

型

```
List[T] forSome { type T <: java.lang.Number }
```

は、上記の簡略化規則の 4 番目により、次と等価です。

```
List[java.lang.Number] forSome { type T <: java.lang.Number }
```

これは今度は(上記の簡略化規則 2 と 3 により)、List[java.lang.Number] と等価になります。

3.3 非値型 (Non-Value Types)

次に説明する型は、値の集合を表しているわけではなく、また、プログラム中に明示的に現れるわけではありません。 これらは、定義された識別子の内部型としてこのレポートで紹介しています。

3.3.1 メソッド型

メソッド型は、内部的に (Ps)U として表現されます。 ここで (Ps) はパラメータ名と型の並び (p1 : T1,...,pn : Tn)、U は (値またはメソッド) 型です(n >= 0)。 この型は、型 T1,...,Tn の p1,...,pn という名前の引数をとり、型 U の結果を返す、名前付きメソッドとして表現されます。

メソッド型は右結合です。: すなわち、(Ps1)(Ps2)U は (Ps1)((Ps2)U) として扱われます。

特別な場合として、いかなるパラメータもとらないメソッドの型があります。 それらはここでは「 => T」と書きます。 パラメータなしのメソッドは、パラメータなしのメソッド名が参照される度に再評価される式に対して名前を与えます。

メソッド型は、値の型としては存在しません。 もしメソッド名が値として使われるなら、その型は暗黙のうちに対応する関数型に変換されます(§6.26)。

Example 3.3.1 : 宣言

```
def a: Int
def b (x: Int): Boolean
def c (x: Int) (y: String, z: String): String
```

は、次の型付けを生みます。

```
a: => Int
b: (Int) Boolean
c: (Int) (String, String) String
```

3.3.2 多相的メソッド型 (Polymorphic Method Types)

多相的メソッド型は、内部的に [tps]T として表現されます。 ここで n >= 0、[tps] は型パラメータ部 [a1 >: L1 <: U1,...,an >: Ln <: Un]であり、T は(値あるいはメソッドの)型です。この型は、下限境界 L1,...,Ln と上限境界 U1,...,Un に適合する(§3.2.4)型引数 S1,...,Sn をとり、型 T の結果をもたらす、名前付きメソッドとして表現されます。

Example 3.3.2 : 宣言

```
def empty[A]: List[A]
def union[A <: Comparable[A]] (x: Set[A], xs: Set[A]): Set[A]
```

は、次の型付けを生みます。

```
empty : [A >: Nothing <: Any] List[A]
union : [A >: Nothing <: Comparable[A]] (x: Set[A], xs: Set[A]) Set[A]
```

3.3.3 型コンストラクタ (Type Constructors)

型コンストラクタは内部的に、多相的メソッド型とほとんど同じように表現されます。 [± a1 >: L1 <: U1,..., ± an >: Ln <: Un]T は、型コンストラクタパラメータ(§4.4)あるいは、対応する型パラメータ節をもつ抽象型コンストラクタ束縛(§4.3)によって要請される型を表します。

Example 3.3.3 : 次の 断片的な Iterable[+X]クラスについて考えます。

```
trait Iterable[+X] {
  def flatMap[newType[+X] <: Iterable[X], S](f: X => newType[S]): newType[S]
}
```

Conceptually, the type constructor Iterable is a name for the anonymous type [+X] Iterable[X], which may be passed to the newType type constructor parameter in flatMap.

概念的には、型コンストラクタ Iterable は、flatMap 中の型コンストラクタパラメータ newType に渡される、無名の型 [+X] Iterable[X] の名前です。

3.4 基本型とメンバー定義 (Base Types and Member Definitions)

クラスメンバの型は、メンバーが参照される方法に依存します。3 つの主要な概念があります。

1. 型 T の基本型の集合としての概念。

2. ある前置型 S からみた、あるクラス C 中の、型 T としての概念。
3. ある型 T のメンバー束縛の集合としての概念。

これらの概念は、次のように相互に再帰的に定義されます。

1. ある型の基本型(base types)の集合は、次で与えられるクラス型の集合です。

   - The base types of a class type C with parents T1,...,Tn are C itself, as well as the base types of the compound type T1 with ... with Tn {R}.
   - The base types of an aliased type are the base types of its alias.
   - The base types of an abstract type are the base types of its upper bound.
   - The base types of a parameterized type C[T1,...,Tn] are the base types of type C , where every occurrence of a type parameter ai of C has been replaced by the corresponding parameter type Ti .
   - The base types of a singleton type p.type are the base types of the type of p.
   - 親 T1,...,Tn をもつクラス型 C の基本型は、複合型 T1 with ... with Tn {R} の基本型と同様、C 自身。
   - エイリアスされた型の基本型は、そのエイリアスの基本型。
   - 抽象型の基本型は、その上限境界の基本型。
   - パラメータ化された型 C[T1,...,Tn] の基本型は、型 C の基本型。 ここで C の型パラメータ ai の出現を対応するパラメータ型 Ti で置き換えたとします。
   - シングルトン型 p.type の基本型は、p の型の基本型。
   - The base types of a compound type T1 with ... with Tn {R} are the reduced union of the base classes of all Ti's. This means: Let the multi-set SS be the multi-set-union of the base types of all Ti's. If SS contains several type instances of the same class, say Si#C[T1,...,Tin] (i ∈ I), then all those instances are replaced by one of them which conforms to all others. It is an error if no such instance exists. It follows that the reduced union, if it exists, produces a set of class types, where different types are instances of different classes.
   - 複合型 T1 with ... with Tn {R} の基本型は、 すべての Ti の基底クラスの**縮小和集合(reduced union)**。 これは次を意味します:マルチ集合 SS をすべての Ti の基本型のマルチ集合の和とします。 もし SS が同じクラスの型インスタンスをいくつか含むなら、 例えば S^i#C[T^i1,...,T^in] (i ∈ I)、 それらすべてのインスタンスを、 その他のすべてに適合するものの１つで置き換えます。 もしそのようなインスタンスが存在しないなら、エラーです。 したがって、縮小和集合は、もしそれが存在するなら、 異なる型は異なるクラスのインスタンスであるような、 クラス型の集合を作り出します。

   - 型選択 S#T の基本型は、次のように決定されます。 もし T がエイリアスあるいは抽象型なら、上述が適用されます。 そうでなければ T は、あるクラス B で定義された（パラメータ化されていてもよい）クラス型でなければなりません。 このとき S#T の基本型は、前置型 S から見た B 中の T の基本型です。
   - 存在型 T forSome { Q } の基本型は、すべての S forSome { Q } 型。 ここで S は T の基本型とします。

2. ある**前置型 S からみたクラス C 中の型 T**という考えは、前置型 S がクラス C の型インスタンスを基本型として持つ場合に限り意味を持ちます。 例えば S´#C[T1,...,Tn]。このとき、次のように定義します。

   - もし S = ε.type なら、S から見た C 中の T は、T 自身です。

- そうでなければ、もし S が存在型 S′ forSome { Q }で、 S′から見た C 中の T が T′なら、S から見た C 中の T は、 T′ forSome { Q } です。

- そうでなければ、もし T があるクラス D の i 番目の型パラメータなら、

  - もし S がある型パラメータ [U1,...,Un] に対して基本型 D[U1,...,Un] を持つなら、S から見た C 中の T は Ui です。
  - そうでなければ、もし C がクラス C′ 中で定義されていれば、 S から見た C 中の T は、S′から見た C′中の T と同じです。
  - そうでなければ、もし C が他のクラス中で定義されていないなら、 S から見た C 中の T は T 自身です。

- そうでなければ、もし T があるクラス D に対するシングルトン型 D.this.type なら、

  - もし D が C のサブクラスで、S がその基本型の中にクラス D の型インスタンスをもっているなら、S から見た C 中の T は S です。
  - そうでなければ、もし C がクラス C′中で定義されているなら、S から見た C 中の T は、S′から見た C′中の T と同じです。
  - そうでなければ、もし C が他のクラス中で定義されていないなら、 S から見た C 中の T は T 自身です。

- もし T が他のある型なら、 ここで記述したマッピングをそのすべての型構成に対して実行します。

もし T がパラメータ化されたクラス型である可能性があるとき、ただしここで T のクラスは他のあるクラス D 中で定義されていて、S が何らかの前置型であるとして、「S から見た T 」を「S から見た D 中の T」の略記表現として使います。

3. The member bindings of a type T are (1) all bindings d such that there exists a type instance of some class C among the base types of T and there exists a definition or declaration d′ in C such that d results from d′ by replacing every type T′ in d′ by T′ in C seen from T , and (2) all bindings of the type's refinement (§3.2.7), if it has one.

3. 型 T の**メンバー束縛**とは次です。 (1) T の基本型の中に何らかのクラス C の型インスタンスが存在し、C 中の定義/宣言 d′ で、d′ 中のすべての型 T′を T から見た C 中の T′で置き換えると束縛 d が得られるような d′があるとき、それらすべての束縛 d 。 (2) 型の細別(§3.2.7)の、もしあれば、すべての束縛。

型射影**S#tの定義**は、S 中の型 t の メンバー束縛 d_t です。 この場合、S#t は **dt によって定義される**、ともいいます。 (訳注:share a to ?)

3.5 型の関係

2 つの型の関係を定義します。

型の等価性	T≡U	T と U はあらゆるコンテキストにおいて交換可能です。
適合性	T **<:** U	型 T は型 U に適合します。

3.5.1 型の等価性

型の等価(≡)は、次を満たす最も小さい一致(congruence *3)です:

(*3) 一致(congruence)は、コンテキスト形成の下で閉じた等価関係です。

- もし t が型エイリアス type t = T によって定義されているなら、t は T に等価。
- もしパス p がシングルトン型 q.type をもつなら、p.type ≡ q.type 。
- もし O がオブジェクト定義によって定義されていて、 p がパッケージあるいはオブジェクトセレクタだけからなり O の中で終わるパスなら、O.this.type ≡ p.type。
- 2 つの複合型(§3.2.7)は、もしそれらの構成要素の並びが対で等価で、 同じ順序で現れ、それらの細別が等価なら、等価です。 2 つの細別は、もしそれらが同じ名前を束縛し、 すべての宣言されたエンティティの修飾子、型、境界が両方の細別で等価なら、 等価です、
- 2 つのメソッド型(§3.3.1)は、もしそれらが等価な結果型を持ち、 両方とも同じ数のパラメータを持ち、 対応するパラメータが等価な型を持っているなら、等価です。 パラメータの名前は、メソッド型の等価性では重要でないことに注意してください。
- 2 つの多相的メソッド型(§3.3.2)は、 もしそれらが同じ数の型パラメータを持ち、 型パラメータの 1 つのセットを他の名前でリネームした後でも、 その結果型が等価で、対応する型パラメータの上下限境界も等価なら、 等価です。
- 2 つの存在型(§3.2.10)は、もしそれらが同じ数の存在量化子をもち、 型存在量化子の 1 つのリストを他でリネームした後でも、 存在量化された型が等価で、対応する存在量化子の上下限境界も等価なら、 等価です。
- 2 つの型コンストラクタ(§3.3.3)は、 もしそれらが同じ数の型パラメータをもち、 型パラメータの 1 つのリストを他でリネームした後でも、 対応する型パラメータの上下限境界や変位指定と同様にその結果型も等価なら、 等価です。

3.5.2 適合性 (Conformance)

適合性の関係(<:)は、次の条件を満たす最小の推移律です。

- 適合性は等価性を包含します。もし T≡U なら T <: U。
- すべての値型 T について、scala.Nothing <: T <: scala.Any。

- すべての型コンストラクタ T (型パラメータがいくつあっても)について、 scala.Nothing <: T <: scala.Any。
- T <: scala.AnyRef であり、T <: scala.NotNull1 ではないような、すべてのクラス型 T について、scala.Null <: T 。

- 型変数あるいは抽象型 t はその上限境界に適合し、 その下限境界は t に適合します。
- クラス型あるいはパラメータ化された型は、その基本型のいずれにも適合します。
- シングルトン型 p.type は、パス p の型に適合します。
- シングルトン型 p.type は、型 scala.Singleton に適合します。
- もし T が U に適合するなら、型射影 T#t は U#t に適合します。

- パラメータ化された型 T[T1,...,Tn] は、もし次の 3 つの条件が i = 1,...,n に対して満たされるなら、T[U1,...,Un] に適合します 。

  - もし T の i 番目の型パラメータが共変と宣言されているなら、Ti <: Ui 。
  - もし T の i 番目の型パラメータが反変と宣言されているなら、Ui <: Ti 。
  - もし T の i 番目の型パラメータが共変でもなく 反変でもないと宣言されているなら、Ui ≡ Ti 。

- 複合型 T1 with ... with Tn {R} は、その構成要素型 Ti のそれぞれに適合します。
- もし T <: Ui (i = 1,...,n) で、R 中の型/値 x のあらゆる束縛 d について、 d を包含する T 中の メンバー束縛 x が存在するなら、 T は複合型 U1 with ... with Un {R} に適合します。
- 存在型 T forSome { Q } は、もしその skolemization(§3.2.10) が U に適合するなら、U に適合します。
- 型 T は、T が U forSome { Q } の型インスタンス(§3.2.10) の 1 つに適合するなら、存在型 U forSome { Q } に適合します。

- もし Ti ≡ T′ (i = 1,...,n) で、U が U′に適合するなら、 メソッド型 (p1 : T1,...,pn : Tn)U は (p′1 : T′1,...,p′n : T′n)U′ に適合します。
- 多相的な型 [a1 >: L1 <: U1,...,an >: Ln <: Un]T は、 L′1<: a1 <: U1′,...,L′n <: an <: U′n と仮定して、 もし i = 1,...,n に対して T <: T′かつ Li <: L′i かつ U′i <: Ui なら、 多相的な型 [a1 >: L′1 <: U′1,...,an >: L′n <: U′n]T′に適合します。
- Type constructors T and T′ follow a similar discipline. We characterize T and T′ by their type parameter clauses [a1,...,an] and [a′1,...,a′n], where an ai or a′i may include a variance annotation, a higher-order type parameter clause, and bounds. Then, T conforms to T′ if any list [t1,...,tn] - with declared variances, bounds and higher-order type parameter clauses - of valid type arguments for T′ is also a valid list of type arguments for T and T[t1,...,tn] <: T′[t1,...,tn]. Note that this entails that:
- 型コンストラクタ T と T′ は、似たような規則に従います。 我々は T と T′を、それらの型パラメータ節 [a1,...,an]、[a′1,...,a′n] によって特徴付けます。 ここで ai あるいは a′i は変位指定アノテーション、高階の型パラメータ節、 境界をもっていてもかまいません。 このとき、もし T′の有効な型引数の任意のリスト [t1,...,tn] --- 宣言された変位指定、境界、高階の型パラメータ節をもっていてもよい --- もまた T の型引数の有効なリストで T[t1,...,tn] <: T′[t1,...,tn] なら、 T は T′ に適合します。この

ことは次を必要とすることに注意してください:

- ai の境界は、a′i に対して宣言された対応する境界よりも弱くなければならない。
- ai の変位指定は、a′i の変位指定と一致しなければならない。 ここで、共変は共変に、反変は反変に、 他の任意の変位指定は不変(invariance)と一致すること。
- これらの制限は、ai と a′i の対応する高階の型パラメータ節に 再帰的に適用されます。

もし次のうちの１つが満たされるなら、クラス型 C の何らかの複合型中の宣言/定義は、クラス型 C′あるいは何らかの複合型中の同じ名前の他の宣言を**包含します(subsume)**。

- 型 T をもつ 名前 x を定義する値宣言/定義が、T <: T′ のときに、 型 T′ をもつ x を定義する値/メソッド宣言を包含する。
- 型 T をもつ名前 x を定義するメソッド宣言/定義が、 T <: T′ のときに、型 T′ をもつ x を定義するメソッド宣言を包含する。
- もし T≡T′なら、型エイリアス type t[T1,...,Tn] = T は型エイリアス type t[T1,...,Tn] = T′を包含します。
- 型宣言 type t[T1,...,Tn] >: L <: U が、もし L′ <: L かつ U <: U′ ならば、型宣言 type t[T1,...,Tn] >: L′ <: U′を包含する。
- 型名 t を束縛する型/クラス定義が、もし L <: t <: U ならば、 抽象型宣言 type t[T1,...,Tn] >: L <: U を包含する。

(<:)関係は、型の間に前秩序(pre-order)を形成します。 すなわち、推移律と反射律を満たします。 型の集合の**最小の上限境界**と**最大の下限境界**は、この順序に関して相対的であると考えられます。

注意:型の集合の最小の上限境界と最大の下限境界は、常に存在するわけではありません。 例えば、次のクラス定義を考えてみてください。

```
class A[+T] {}
class B extends A[B]
class C extends A[C]
```

このとき、型 A[Any]、A[A[Any]]、A[A[A[Any]]], ... は、B と C に対する上限境界を降順に構成します。 最小の上限境界はこのシーケンスの無限の彼方にあり、Scala の型としては存在しません。 このような場合は一般的に検出できないので、また、そういった境界はコンパイラセットの制限(*4)よりも複雑であるかもしれず、Scala コンパイラは最小の上限境界あるいは最大の下限境界として指定された型をもつ項(term)を拒絶する自由を持ちます。

(*4) 現在の Scala コンパイラは、そのような境界中のパラメータ化のネストレベルを、 オペランド型の最大ネストレベルよりせいぜい 2 つまで深いと制限しています。

最小の上限境界あるいは最大の下限境界は、同様にただ一つとは限りません。 例えば、A with B と B with A は、両方とも A および B の最大の下限境界です。 もし最小の上限境界あるいは最大の下限境界が複数あるときは、Scala コンパイラはそれらの 1 つを勝手に選びます。

3.5.3 弱い適合性（Weak Conformance）

Scalaはいくつかの状況では、より一般的な適合性関係を用います。 もし S <: T であるか、あるいは、S と T 両方がプリミティブな数値型で、次の順序中で S が T の前にあるなら、型 S は型 T に**弱く適合する**といい、S <:w T と書きます。

```
Byte <:w Short
Short <:w Int
Char <:w Int
Int <:w Long
Long <:w Float
Float <:w Double
```

弱い最小の上限境界(weak least upper bound)とは、弱い適合性に関する最小の上限境界です。

3.6 volatile 型（Volatile Types）

Type volatility approximates the possibility that a type parameter or abstract type instance of a type does not have any non-null values. As explained in (§3.1), a value member of a volatile type cannot appear in a path.

型の揮発性(volatility)は、型の抽象型インスタンスあるいは型パラメータがいかなる非 null 値も持たない可能性を近似します。 (§3.1)で説明したように、volatile 型の値メンバーはパス中に現れることはできません。

もし型が 4 つのカテゴリの 1 つにあてはまるなら、型は **volatile** です:

複合型 T1 with ... with Tn {R} は、もし次の 2 つの(訳注: 3 つの誤記?) 条件のうちの 1 つを満たすなら、volatile です。

1. T2,...,Tn の１つが型パラメータあるいは抽象型である。あるいは、
2. T1 が抽象型で、細別 R あるいは型 Tj (j > 1)が、 複合型に抽象メンバを提供している。あるいは
3. T1,...,Tn の 1 つがシングルトン型である。

ここで、もし型 S が型 T のメンバーでもあるような抽象メンバを含むなら、型 S は型 T に**抽象メンバを提供する**といいます。 もし細別 R が型 T のメンバーでもあるような抽象宣言を含むなら、細別 R は型 T に抽象メンバを提供するといいます。

もし型指定子が volatile 型のエイリアスであるか、あるいは、もし型指定子がその上限境界として volatile 型を持つ型パラメータあるいは抽象型を指定するなら、その型指定子は volatile です。

もしパス p の基礎をなしている型が volatile なら、シングルトン型 p.type は volatile です。

存在型 T forSome { Q } は、もし T が volatile なら、volatile です。

3.7 型消去（Type Erasure）

型は、もしそれが型引数あるいは型変数を含むなら、**ジェネリック**である言われます。 **型消去(Type erasure)**は、(ジェネリックでもよい)型から非ジェネリックな型へのマッピングです。 T の型消去を |T| と書きます。 型消去のマッピングは次のように定義されます。

- エイリアス型の型消去は、その右側の型消去。
- 抽象型の型消去は、その上限境界の型消去。
- パラメータ化された型 scala.Array[T1] の型消去は、scala.Array[|T1|]。
- 他のすべてのパラメータ化された型 T[T1,...,Tn] の型消去は、|T|。
- シングルトン型 p.type の型消去は、p の型消去。
- 型射影 T#x の型消去は、|T|#x。
- 複合型 T1 with ... with Tn {R} の型消去は、T1,...,Tn の論理積ドミネータ（intersection dominator)の型消去。
- 存在型 T forSome { Q } の型消去は、|T|。

型のリスト T1,...,Tn の**論理積ドミネータ(intersection dominator)**は、次のように計算されます。 まず、Ti1,...,Tim を型 Ti の部分列、ただし Ti は他のある型 Tj のスーパー型ではないようなものとします。 もしこの部分列がトレイトではないクラスを参照する型指定子 Tc を含むなら、論理積ドミネータは Tc です。 そうでなければ、論理積ドミネータは、部分列の最初の要素 Ti1 です。

4 基本的な宣言と定義

構文:

```
Dcl         ::=   'val' ValDcl
                | 'var' VarDcl
                | 'def' FunDcl
                | 'type' {nl} TypeDcl
PatVarDef   ::=   'val' PatDef
                | 'var' VarDef
Def         ::=  PatVarDef
                | 'def' FunDef
                | 'type' {nl} TypeDef
                | TmplDef
```

宣言は名前を導入し、それらに型を割り当てます。 それはクラス定義(§5.1)の一部あるいは、複合型(§3.2.7)における細別(refinement) の一部を形成します。

定義は項(term)あるいは型を表す名前を導入します。 それはオブジェクトあるいはクラス定義の一部を形成したり、ブロックにローカルにできます。 宣言と定義の両方とも、型名に型定義あるいは境界を結びつける**束縛**を生み出し、それは項名に型を結びつけます。

宣言/定義によって導入された名前のスコープは、束縛を含む文並び全体です。 しかし、ブロック中での前方参照には制限があります。: ブロックを構成する文並び s1 ... sn 中で、もし si 中の単純名が sj によって定義されたエンティティを参照するなら (ただしここで j >= i)、そのときは si と sj の間およびそれらを含めたいかなる定義も、値あるいは変数定義であってはなりません。

4.1 値宣言と定義 (Value Declarations and Definitions)

構文:

```
Dcl         ::=   'val' ValDcl
ValDcl      ::=   ids ':' Type
Def         ::=   'val' PatDef
PatDef      ::=   Pattern2 {',' Pattern2} [':' Type] '=' Expr
ids         ::=   id {',' id}
```

値宣言 val x : T は、型 T の値の名前として x を導入します。

値定義 val x : T = e は、e の評価から得られる値の名前として x を定義します。 もし値定義が再帰的でないなら、型 T は省略されることがあり、その場合は、式 e のパックされた型(§6.1)が想定されます。 もし型 T が与えられていれば、e はそれに適合することが要請されます。

値定義の評価は、修飾子 lazy がなければ、その右辺 e の評価を意味します。 値定義の効果は、型 T に変換された e の値に x を束縛することです。 遅延評価値(lazy value definition)

定義は、値が最初にアクセスされたときに、その右辺 e を評価します。

定数値定義(constant value definition)は、次の形です。

```
final val x = e
```

ここで e は定数式(§6.24)です。 final 修飾子は必須ですが、型アノテーションは全くないかもしれません。 定数値 x への参照は、それ自身定数式として扱われます。 つまり、それらは生成コード中で定義の右辺 e によって置き換えられます。

値定義は、左辺にパターン(§8.1)をとることができます。 もし p が、単純名あるいは名前の後にコロンと型が続くパターン以外なら、値定義 val p = e は次のように展開されます:

1. もしパターン p が束縛変数 x1,...,xn（n > 1）を持っているなら

```
val $x = e match {case p => {x1,...,xn}}
val x1 = $x._1
  ...
val xn = $x._n
```

ここで $x は新規の名前です。

2. もし p がただ 1 つの束縛変数 x を持つなら:

```
val x = e match { case p => x }
```

3. もし p が束縛変数を持たないなら:

```
e match { case p => ()}
```

Example 4.1.1 : 次は値定義の例です

```
val   pi = 3.1415
val   pi: Double = 3.1415        // 最初の定義と等価
val   Some(x) = f()              // パターン定義
val   x :: xs = mylist           // 中置パターン定義
```

最後の 2 つの定義は、次のように展開されます。

```
val x = f() match { case Some(x) => x }

val x$ = mylist match { case x :: xs => {x, xs} }
val x = x$._1
val xs = x$._2
```

宣言されたあるいは定義された値のいかなる名前も _= で終わってはなりません。

値宣言 val x1,...,xn : T は、値宣言の並び val x1 : T ; ...; val xn : T の略記表現です。 値定義 val p1,...,pn = e は、値定義の並び val p1 = e ; ...; val pn = e の略記表現です。 値定義 val p1,...,pn : T = e は、値定義の並び val p1 : T = e ; ...; val pn : T = e の略記表現です。

4.2 変数宣言と定義 (Variable Declarations and Definitions)

構文:

```
Dcl         ::= 'var' VarDcl
Def         ::= 'var' VarDef
VarDcl      ::= ids ':' Type
VarDef      ::= PatDef
              | ids ':' Type '=' '_'
```

変数宣言 `var x : T` は、次のように定義された**ゲッター関数** `x` と**セッター関数** `x_=` 宣言に等価です:

```
def x : T
def x_= (y : T): Unit
```

変数宣言を含むクラスの実装では、変数定義を使ってそれら変数を定義することがあり、あるいはまた、直接的にゲッター、セッター関数を定義することがあります。

変数定義 `var x : T = e` は、型 `T` をもつミュータブル変数と、式 `e` で与えられた初期値を導入します。 型 `T` は省略されることがあり、その場合は、`e` の型が想定されます。 もし `T` が与えられていれば、`e` はそれに適合することが要請されます(§6.1)。

変数定義は、左辺にパターン(§8.1)をとることができます。 変数定義 `var p = e` (ただしここで `p` は単純名あるいは名前の後にコロンと型が続く以外のパターン) は、 `p` 中の自由な名前が値ではなくミュータブル変数として導入されること以外、値定義 `val p = e` と同じ方法(§4.1) で展開されます。

いかなる宣言されたあるいは定義された名前も、`_=` で終わってはなりません。

変数定義 `var x : T = _` は、テンプレートのメンバーとしてだけ現われることができます。 これは型 `T` のミュータブルなフィールドとデフォルトの初期値を導入します。 デフォルト値は、次のように型 `T` に依存します:

```
0       T が Int あるいはその部分領域型の 1 つのとき
0L      T が Long のとき
0.0f    T が Float のとき
0.0d    T が Double のとき
false   T が Boolean のとき
{}      T が Unit のとき
null    他のすべての型 T について
```

それらがテンプレートのメンバーとして現れるときは、変数定義の両形式とも、変数に現在割り当てられている値を返すゲッター関数 `x` が導入され、変数に現在割り当てられている値を変えるセッター関数 `x_=` も導入されます。 関数は、変数宣言については同じシグニチャを持ちます。 このときテンプレートはそれらゲッターとセッター関数をメンバーとして持ちますが、一方、オリジナルの変数には、テンプレートメンバーとして直接アクセスすることはできません。

Example 4.2.1 : 次の例は、Scala において**プロパティ**が どのようにシミュレートされるかを示します。 これは、hours、minutes、seconds で表される更新可能な整数フィールドを持つ、時間値からなるクラス TimeOfDayVar を定義しています。 この実装は、それらフィールドに正しい値のみ代入できるようにするテストを含んでいます。 一方、ユーザーコードは、それらのフィールドに普通の変数とまったく同じようにアクセスします。

```
class TimeOfDayVar {
  private var h: Int = 0
  private var m: Int = 0
  private var s: Int = 0

  def hours             =  h
  def hours_= (h: Int)  =  if (0 <= h && h < 24) this.h = h
                           else throw new DateError()

  def minutes           =  m
  def minutes_= (m: Int) =  if (0 <= m && m < 60) this.m = m
                           else throw new DateError()

  def seconds           =  s
  def seconds_= (s: Int) =  if (0 <= s && s < 60) this.s = s
                           else throw new DateError()
}
val d = new TimeOfDayVar
d.hours = 8; d.minutes = 30; d.seconds = 0
d.hours = 25                  // DateError 例外送出
```

変数宣言 var x1,...,xn : T は、変数宣言の並び var x1 : T ; ...; var xn : T の略記表現です。 変数定義 var x1,...,xn = e は、変数定義の並び var x1 = e ; ...; var xn = e の略記表現です。 変数定義 var x1,...,xn : T = e は、変数定義の並び var x1 : T = e ; ...; var xn : T = e の略記表現です。

4.3 型宣言と型エイリアス (Type Declarations and Type Aliases)

構文:

```
Dcl         ::= 'type' {nl} TypeDcl
TypeDcl     ::= id [TypeParamClause] ['>:' Type] ['<:' Type]
Def         ::= type {nl} TypeDef
TypeDef     ::= id [TypeParamClause] '=' Type
```

A type declaration type t[tps] >: L <: U declares t to be an abstract type with lower bound type L and upper bound type U . If the type parameter clause [tps] is omitted, t abstracts over a first-order type, otherwise t stands for a type constructor that accepts type arguments as described by the type parameter clause .

型宣言 type t[tps] >: L <: U は、 t が下限境界型 L と上限境界型 U をもつ抽象型であることを宣言します。 型パラメータ節 [tps] が省略された場合、t は一階型の抽象化であるか、そうでなければ t は型コンストラクタを表し、型パラメータ節によって記述されたとして型引数を受け入れます。

もし型宣言が型のメンバー宣言として現われるなら、その型の実装は、t を L <: T <: U である任意の型 T をもつとして実装するかもしれません。 もし L が U に適合しないなら、コンパイルエラーとなります。 いずれかあるいは両方の境界とも省略されるかもしれません。 もし下限

境界 L がないなら、最下位の型 scala.Nothing が想定されます。 もし上限境界 U がないなら、最上位の型 scala.Any が想定されます。

型コンストラクタ宣言は、t が表すであろう具象型に制限を付け加えます。 L と U の境界ほかに、型パラメータ節は、型コンストラクタ(§3.5.2) の適合性によって左右される高階の境界や変位指定を課すかもしれません。

型パラメータのスコープは、境界 >: L <: U と型パラメータ節 tps 自身に及びます。 (抽象型コンストラクタ tc の)高階の型パラメータ節は同種のスコープを持ち、型パラメータ tc の宣言へ制限されます。

To illustrate nested scoping, these declarations are all equivalent: type t[m[x] <: Bound[x], Bound[x]], type t[m[x] <: Bound[x], Bound[y]] and type t[m[x] <: Bound[x], Bound[_]], as the scope of, e.g., the type parameter of m is limited to the declaration of m. In all of them, t is an abstract type member that abstracts over two type constructors: m stands for a type constructor that takes one type parameter and that must be a subtype of Bound , t's second type constructor parameter. t[MutableList, Iterable] is a valid use of t .

ネストしたスコープを説明すると、次は全て同じスコープです。: type t[m[x] <: Bound[x], Bound[x]]、type t[m[x] <: Bound[x], Bound[y]] と type t[m[x] <: Bound[x], Bound[_]]。例えば、m の型パラメータのスコープは m の宣言に制限されるからです。 これらのすべてにおいて t は、2 つの型コンストラクタ上の抽象である抽象型メンバーです。:

m は、1 つの型パラメータをとり、**Bound** のサブ型でなければならないような型コンストラクタを表します。 ここで Bound は t の 2 番目の型コンストラクタパラメータとなっています。 t[MutableList、Iterable]は、t の有効な使い方です。

型エイリアス type t = T は、t が型 T の別名であることを定義します。 型エイリアスの左辺は型パラメータ節を持っていてもかまいません。 たとえば type t[tps] = T です。 型パラメータのスコープは、右辺 T と型パラメータ節 tps 自身に及びます。

定義(§4)と型パラメータのスコープ規則(§4.6)では、型名がそれ自身の境界あるいはその右辺中に現れることを可能としています。 しかし、もし型エイリアスが、定義された型コンストラクタ自身を再帰的に参照するなら、それは静的エラーです。 すなわち、型エイリアス type t[tps] = T 中の型 T は、直接的にも間接的にも名前 t を参照できません。 もし抽象型が、直接あるいは間接的にそれ自身の上限/下限境界なら、それも同じくエラーです。

Example 4.3.1 : 次は正しい型宣言と定義です:

```
type   IntList = List[Integer]
type   T <: Comparable[T]
type   Two[A] = Tuple2[A, A]
type   MyCollection[+X] <: Iterable[X]
```

次は不正です:

```
type Abs = Comparable[Abs]     // 再帰的な型エイリアス

type S <: T                    // S, T はそれら自身によって
type T <: S                    //      境界付けられている

type T >: Comparable[T.That]   // T から選択できない
                               // T は型であり、値ではない
type MyCollection <: Iterable  // 型コンストラクタメンバーは明示的に
                               //      ??? を記述しなければならない。
```

もし型エイリアス type t[tps] = S がクラス型 S を参照するなら、名前 t は、型 S のオブジェクトのコンストラクタとしても使えます。

Example 4.3.2 : 事前定義済みオブジェクト(Predef object)は、パラメータ化されたクラス Tuple2 のエイリアスとして、Pair を定める定義を含んでいます。

```
type Pair[+A, +B] = Tuple2[A, B]
object Pair {
  def apply[A, B](x: A, y: B) = Tuple2(x, y)
  def unapply[A, B](x: Tuple2[A, B]): Option[Tuple2[A, B]] = Some(x)
}
```

その結果、任意の 2 つの型 S と T に対して、型 Pair[S, T]は、型 Tuple2[S, T] に等価です。 次のように、Pair は Tuple2 の代わりにコンストラクタとしても使えます。

```
val x: Pair[Int, String] = new Pair(1, "abc")
```

4.4 型パラメータ（Type Parameters）

構文:

```
TypeParamClause  ::= '[' VariantTypeParam {',' VariantTypeParam} ']'
VariantTypeParam ::= {Annotation} ['+' | '-'] TypeParam
TypeParam ::= (id | '_') [TypeParamClause] ['>:' Type] ['<:' Type] [':' Type]
```

型パラメータは、型定義、クラス定義と関数定義の中に現れます。 この節では、下限境界 >:L と上限境界 <: U をもつ型パラメータ定義だけについて考え、コンテキスト境界 : U と 可視境界 <% U については、あとで §7.4 で議論します。

一階の型パラメータの最も一般的な形は、@a1 ... @an ± t >: L <: U です。 ここで、L と U は、そのパラメータの可能な型引数を制約する下限と上限境界です。 もし L が U に適合しなければ、それは実行時エラーとなります。 ±は**変位指定(variance)**、すなわち、+ も - もオプションの前置子です。 1 つ以上のアノテーションが型パラメータに先行しても構いません。

すべての型パラメータの名前は、型パラメータ節を囲む中で対で(pairwise)異なっていなければなりません。 型パラメータのスコープは、それぞれ、型パラメータ節全体を含みます。 ですから型パラメータは、それ自身の境界の一部あるいは、同じ節中の他の型パラメータの境界の一部に現れることができます。 しかし型パラメータは、それ自身によっては直接あるいは間接に境界づけられません。

型コンストラクタパラメータは、ネストした型パラメータ節をその型パラメータに加えます。 型コンストラクタパラメータの最も一般的な形は、@a1 ... @an ± t[tps] >: L <: U です。

上記のスコープ制約は、高階の型パラメータを宣言するネストした型パラメータ節の場合へ一般化されます。 高階の型パラメータ(型パラメータ t の型パラメータ)は、それらを直に囲むパラメータ節(より深いネストレベルにおける節も含む)中と t の境界中においてだけ、可視です。 そのため、それらの名前は、他の可視のパラメータ名と対で異なっていなければなりません。 高階の型パラメータの名前は、しばしば重要でないことがあり、その場合、'_' で表すことができ、名前はどこにも見えません。

Example 4.4.1 : 次は、型パラメータ節の正しい形の例です:

```
[S, T]
[@specialized T, U]
[Ex <: Throwable]
[A <: Comparable[B], B <: A]
[A, B >: A, C >: A <: B]
[M[X], N[X]]
[M[_], N[_]]                // 上の節に等価
[M[X <: Bound[X]], Bound[_]]
[M[+X] <: Iterable[X]]
```

次の型パラメータ節は不正です:

```
[A >: A]                  // 不正、'A'はそれ自身を境界として持つ
[A <: B, B <: C, C <: A]  // 不正、'A'はそれ自身を境界として持つ
[A, B, C >: A <: B]       // 不正、'C'の不正な下限境界 'A'は、
                          // 上限境界 'B' に適合しない。
```

4.5 変位指定アノテーション (Variance Annotations)

変位指定アノテーションは、パラメータ化された型のインスタンスがサブ型(§3.5.2)に関してどのように異なるかを示します。 '+' 変位指定は共変の従属性を表し、'-' 変位指定は反変の従属性を表し、変位指定がなければ不変の従属性を表します。

A variance annotation constrains the way the annotated type variable may appear in the type or class which binds the type parameter. In a type definition type T[tps] = S, or a type declaration type T[tps] >: L <: U type parameters labeled '+' must only appear in covariant position whereas type parameters labeled '-' must only appear in contravariant position. Analogously, for a class definition class C[tps](ps) extends T { x : S => ...}, type parameters labeled '+' must only appear in covariant position in the self type S and the template T , whereas type parameters labeled '-' must only appear in contravariant position .

変位指定アノテーションは、型パラメータを束縛する型あるいはクラス中において、アノテーションされた型変数の現れ方に制約を課します。 型定義 type T[tps] = S あるいは型宣言 type T[tps] >: L <: U において、'+' と印された型パラメータは共変のポジションにのみ現れることができ、他方、'-' と印された型パラメータは反変のポジションにのみ現れることができます。 同様に、クラス定義 class C[tps](ps) extends T { x : S => ...} においては、'+' と印された型パラメータは、自己型 S とテンプレート T 中の共変のポジションにのみ現れることができ、他方、'-' と印された型パラメータは反変のポジションにのみ現れることができます。

型あるいはテンプレートにおける型パラメータの変位指定のポジションは、次のように定義されます。 共変の反対は反変、非変の反対はそれ自身とします。 型あるいはテンプレートのトップレベルは、常に共変のポジションです。 変位指定のポジションは、次の言語要素において変化します。

- メソッドパラメータの変位指定のポジションは、 取り囲むパラメータ節の変位指定のポジションの反対です。

- 型パラメータの変位指定のポジションは、 取り囲む型パラメータ節の変位指定のポジションの反対です。
- 型宣言あるいは型パラメータの下限境界の変位指定のポジションは、 型宣言あるいは型パラメータの変位指定のポジションの反対です。
- ミュータブルな変数の型は、常に非変のポジション中にあります。
- 型選択 `S#T` の前置子 `S` は、常に非変のポジション中にあります。
- 型 `S[... T ...]` の型引数 `T` について、もし対応する型パラメータが非変なら、 `T` は非変のポジション中にあります。 もし対応する型パラメータが反変なら、`T` の変位指定のポジションは `S[...T...]` を囲む変位指定のポジションの反対です。

オブジェクト非公開 な値、変数、あるいはクラスのメソッド(§5.2)中の型パラメータへの参照は、 それらの変位指定のポジションはチェックされません。 これらのメンバーでは、型パラメータは、正しい変位指定アノテーションと限定されずに、どこにでも現われることができます。

Example 4.5.1 : 次の変位指定アノテーションは正しい。

```
abstract class P[+A, +B] {
  def fst: A; def snd: B   }
```

この変位指定アノテーションでは、P の型インスタンスは、それら引数を共変にサブ型付けします。 例えば、

```
P[IOException, String] <: P[Throwable, AnyRef]
```

もし P のメンバーがミュータブルな変数なら、同じ変位指定アノテーションは不正になります。

```
abstract class Q[+A, +B](x: A, y: B) {
  var fst: A = x           // **** error: illegal variance:
  var snd: B = y           // 'A', 'B' が不変のポジションに現れている
}
```

もしミュータブルな変数がオブジェクト非公開 なら、クラス定義は再び正しくなります:

```
abstract class R[+A, +B](x: A, y: B) {
  private[this] var fst: A = x        // OK
  private[this] var snd: B = y        // OK
}
```

Example 4.5.2 : 次の変位指定アノテーションは不正です。 なぜなら、append パラメータ中の反変のポジションに A が現れるからです。

```
abstract class Sequence[+A] {
  def append(x: Sequence[A]): Sequence[A]
                // **** error: illegal variance:
                // 'A' が反変のポジションに現れている
}
```

問題を、下限境界を使って append の型を一般化することで回避できます。

```
abstract class Sequence[+A] {
  def append[B >: A](x: Sequence[B]): Sequence[B]
}
```

Example 4.5.3 : 次は反変の型パラメータが役に立つ場合です。

```
abstract class OutputChannel[-A] {
  def write(x: A): Unit
}
```

このアノテーションにより、OutputChannel[AnyRef] は OutputChannel[String] に適合します。 つまり、任意のオブジェクトを書き込めるチャネルは、文字列だけを書き込めるチャネルを代理できます。

4.6 関数宣言と定義

構文:

```
Dcl                 ::=  'def' FunDcl
FunDcl              ::=  FunSig ':' Type
Def                 ::=  'def' FunDef
FunDef              ::=  FunSig [':' Type] '=' Expr
FunSig              ::=  id [FunTypeParamClause] ParamClauses
FunTypeParamClause  ::=  '[' TypeParam {',' TypeParam} ']'
ParamClauses        ::=  {ParamClause} [[nl] '(' 'implicit' Params ')']
ParamClause         ::=  [nl] '(' [Params] ')'}
Params              ::=  Param {',' Param}
Param               ::=  {Annotation} id [':' ParamType] ['=' Expr]
ParamType           ::=  Type
                       |  '=>' Type
                       |  Type '*'
```

関数宣言は def f psig: T の形をとります。 ここで、f は関数名、psig はそのパラメータシグニチャ、T はその結果型です。 関数定義 def f psig: T = e は、**関数本体** e も含んでいます。 すなわち、関数の結果を定義する式です。 パラメータシグニチャは、オプションの型パラメータ節 [tps] と、その後に続く 0 個以上の値パラメータ節(ps1)...(psn) からなります。 そのような宣言または定義は、パラメータ型と結果型が与えられた(多相的でもよい)メソッド型をもつ値を導入します。

関数本体の型は、もしそれが与えられているなら、関数の宣言された結果型に適合する(§6.1)ことが要請されます。 もし関数定義が再帰的でないなら、結果型は省略されることがあります。 その場合、結果型は関数本体のパックされた型から決定されます。

型パラメータ節 tps は、1 つ以上の型宣言 (§4.3)からなり、 場合によっては境界のついた型パラメータを導入します。 型パラメータのスコープはシグニチャ全体を含み、もしあれば、すべての型パラメータ境界も関数本体と同様に含みます。

値パラメータ節 ps は、値パラメータを束縛しそれらと型を結びつける x : T あるいは x : T = e のような、0 個以上の形式上のパラメータ束縛からなります。 値パラメータ宣言はそれぞれ、オプションでデフォルト引数を定義できます。 デフォルト引数式 e は、T 中の関数の型パラメータのすべての出現を未定義の型で置き換えて得られる要請型 T で型チェックされます。

デフォルト引数をもつすべてのパラメータ $p_{i,j}$ に対して、デフォルト引数式を計算する f$default$n という名前のメソッドが生成されます。 ここで n は、メソッド宣言中のパラメータ位置を表します。 これらのメソッドは、型パラメータ節 [tps] と $p_{i,j}$ に先行するすべての値パラメータ節 $(ps_1)...(ps_{i-1})$ によってパラメータ化されます。 ユーザープログラムは f$default$n メソッドにはアクセスできません。

形式上の値パラメータ名 x のスコープは、後に続くすべてのパラメータ節およびメソッドの戻り値型と関数本体などから、もしそれらが与えられているなら(*1)、構成されます。 型パラメータ名と値パラメータ名の両方とも、対として異なっていなければなりません。

(*1) しかし、現在のメソッドパラメータのシングルトン型は、メソッド本体の中にだけ現れることができます。 ですから、**依存メソッド型(dependent method types)**はサポートされていません。

Example 4.6.1 : メソッド中

```
def compare[T](a: T = 0)(b: T = a) = (a == b)
```

デフォルト式 0 は、未定義の要請型で型チェックされます。 compare()を適用するとき、デフォルト値 0 が挿入され、 T は Int にインスタンス化されます。 デフォルト引数を計算するメソッドは、次のような形をとります。

```
def compare$default$1[T]: Int = 0
def compare$default$2[T](a: T): T = a
```

4.6.1 名前呼び出しパラメータ（By-Name Parameters）

構文:

```
ParamType              ::= '=>' Type
```

値パラメータの型には => が手前につくかもしれません。たとえば、x : => T。 そのようなパラメータの型は、パラメータなしのメソッド型 => T です。 これは、対応する引数が関数適用の時点で評価されないことを示します。 しかしその代わりに、関数内での使用の度に評価されます。 すなわち、引数は**名前呼出し**を使って評価されます。

名前呼び出し修飾子は、val あるいは var 前置子のつくクラスのパラメータに対しては、val 前置子が暗黙の内に生成されるケースクラスのパラメータも含めて、許されていません。 名前呼び出し修飾子は、暗黙のパラメータ(§7.2)に対しても許されていません。

Example 4.6.2 : 宣言

```
def whileLoop (cond: => Boolean) (stat: => Unit): Unit
```

は、whileLoop の両方のパラメータが名前呼出しを使って評価されることを示しています。

4.6.2 反復パラメータ（Repeated Parameters）

構文:

```
ParamType               ::= Type '*'
```

パラメータ部の最後の値パラメータには、あとに "*" がつくかもしれません。 たとえば（..., x :T *）。 メソッド内部のそのような**反復**パラメータの型は、シーケンス型 scala.Seq[T] です。 反復パラメータ T * をもつ メソッドは、型 T の引数を可変個とります。 すなわち、もし型 (p1:T1,...,pn:Tn,ps:S*)U をもつメソッド m が引数（e1,...,ek）に適用されるなら(k >= n)、 m はその適用中に、型 S が k - n 個出現する型(p1:T1,...,pn:Tn,ps:S,..., ps´S)U を持っていると解釈されます。 ここで ps より後のパラメータ名はすべて新規とします。（訳注:最後のメソッド型は(p1:T1,...,pn:Tn,ps:S,..., ps´:S)U の誤記と思われる）この規則の唯一の例外は、最後の引数が _* 型アノテーションをもつ**シーケンス引数**と印されている場合です。 もし上の m が引数（e1,...,en , e´ : _*）に適用されるなら、適用中の m の型は（p1 : T1,..., pn: Tn , ps :scala.Seq[S]）と解釈されます。

反復パラメータをもつパラメータ部中では、いかなるデフォルト引数も定義することは許されていません。

Example 4.6.3 : 次のメソッド定義は、可変個の整数引数の二乗の和を計算します。

```
def sum(args: Int*) = {
  var result = 0
  for (arg <- args) result += arg * arg
  result
}
```

このメソッドの次の適用は、順に 0、1、6 をもたらします(訳注:2 乗なので 14 と思われる)。

```
sum()
sum(1)
sum(1, 2, 3)
```

さらに、次の定義を仮定します:

```
val xs = List(1, 2, 3)
```

メソッド sum の次の適用は不正です:

```
sum(xs)          // ***** error: expected: Int, found: List[Int]
```

それと対照して、次の適用は正しい形であり、再び結果 6 をもたらします:

```
sum(xs: _*)
```

4.6.3 手続き（Procedures）

構文:

```
FunDcl      ::=   FunSig
FunDef      ::=   FunSig [nl] '{' Block '}'
```

手続き(procedures)のための特別な構文があります。 たとえば、Unit 値 {} (訳注:() の誤記と思われる)を返す手続きです。 手続き宣言は、結果型が省略された関数宣言です。 この場合、結果型は暗黙のうちに Unit 型となります。 すなわち、def f(ps) は def f(ps): Unit と同じです。

手続き定義は、結果型と等号が省略された関数定義です; その定義式は 1 つのブロックでなければなりません。 すなわち、def f(ps) {stats} は def f(ps): Unit = {stats} と同じです。

Example 4.6.4 : 次は write という名前の手続きの宣言と定義です。

```
trait Writer {
  def write(str: String)
}
object Terminal extends Writer {
  def write(str: String) { System.out.println(str) }
}
```

上記のコードは、暗黙のうちに次のコードになります。

```
trait Writer {
  def write(str: String): Unit
}
object Terminal extends Writer {
  def write(str: String): Unit = { System.out.println(str) }
}
```

4.6.4 メソッドの戻り値型の推論 (Method Return Type Inference)

A class member definition m that overrides some other function m′ in a base class of C may leave out the return type, even if it is recursive. In this case, the return type R′ of the overridden function m′ , seen as a member of C , is taken as the return type of m for each recursive invocation of m. That way, a type R for the right-hand side of m can be determined, which is then taken as the return type of m. Note that R may be different from R′ , as long as R conforms to R′ .

C の基底クラス中の他のある関数 m′をオーバライドするクラスメンバ定義 m について、たとえそれが再帰的であっても、戻り値型を書き落とすかもしれません。 この場合、オーバライドされた関数 m′の戻り値型 R′は、C のメンバーとして見られ、m の各再帰呼出しに対して m の戻り値型とみなされます。 このように m の右辺の型 R を決定でき、それは m の戻り値型とみなされます。 R が R′ に適合する限り、R は R′ と異なっていてもよいことに注意してください。

Example 4.6.5 : 次の定義を仮定します。

```
trait   I {
  def   factorial(x: Int): Int
```

```
}
class   C extends I {
  def   factorial(x: Int) = if (x == 0) 1 else x * factorial(x - 1)
}
```

ここで、C 中の factorial の結果型を書き落としても、メソッドは再帰的ですが、問題ありません。

4.7 インポート節（Import Clauses）

構文:

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

インポート節は、import p.I の形をしています。 ここで、p は安定識別子(§3.1)、I はインポート式です。インポート式は、限定修飾なしで使える、 p のインポート可能なメンバー名の集合を決定します。 p のメンバー m は、もしそれがオブジェクト非公開（§5.2）でなければ、**インポート可能**です。インポート式の最も一般的な形は、次のインポートセレクタのリストです。

```
{ x1 => y1,...,xn => yn , _ }
```

ここで n >= 0、最後のワイルドカード '_' はないかもしれません。 これにより、限定修飾されていない名前 yi を使って、各インポート可能なメンバー p.xi を利用できます。 すなわち、すべてのインポートセレクタ xi => yi は、p.xi を yi にリネームします。 もし 最後にワイルドカードがあれば、x1,...,xn 以外の p のすべてインポート可能なメンバー z を、限定修飾されていないそれら自身の名前を使って利用できます。

インポートセレクタは、型や項メンバーに対しても同じようにうまく働きます。 例えば、インポート節 import p.{ x => y } は、項名 p.x を項名 y へ、型名 p.x を型名 y へリネームします。 これらの 2 つの名前の少なくとも 1 つは、p のインポート可能なメンバーを参照しなくてはなりません。

もしインポートセレクタ中のターゲットがワイルドカードなら、インポートセレクタはそのソースメンバーへのアクセスを隠蔽します。 例えばインポートセレクタ x => _ は、x をワイルドカード記号へ「リネーム」し(ユーザープログラム中では名前としてはアクセスできない)、 これによって限定修飾されていない名前 x へのアクセスを効率的に妨ぎます。 もし同じインポートセレクタリストの最後にワイルドカードがあれば、それより前のインポートセレクタで言及されていないすべてのメンバーをインポートするのに役立ちます。

The scope of a binding introduced by an import-clause starts immediately after the import clause and extends to the end of the enclosing block, template, package clause, or compilation unit, whichever comes first .

インポート節によって導入された束縛のスコープは、インポート節のすぐ後に始まり、取り囲むブロック、テンプレート、パッケージ節、あるいはコンパイル単位等の、いずれか最初にくるものの最後にまで及びます。

略記表現がいくつかあります。 インポートセレクタは、たんに単純名 x であるかもしれません。 この場合、x はリネームなしでインポートされます。 ですからインポートセレクタは x => x に等価です。 さらに、ただ 1 つの識別子あるいはワイルドカードを使って、インポートセレクタリスト全体を置き換えできます。 インポート節 import p.x は、import p.{x} に等価です。 すなわち p のメンバー x を限定修飾なしで利用可能にします。 インポート節 import p._ は、import p{_} に等価です。 すなわち p のすべてのメンバーを限定修飾なしで利用可能にします (これは Java の import p* と似ています)。

複数のインポート式をもつインポート節 p1.I1,...,pn.In は、インポート節の並び import p1.I1 ; ...; import pn.In と解釈されます。

Example 4.7.1 : 次のオブジェクト定義を考えます。

```
object M {
  def z = 0, one = 1
  def add(x: Int, y: Int): Int = x + y
}
```

すると、ブロック

```
{ import M.{one, z => zero, _}; add(zero, one) }
```

は、次のブロックに等価です。

```
{ M.add(M.z, M.one) }
```

5 クラスとオブジェクト

構文:

```
TmplDef     ::= ['case'] 'class' ClassDef
              | ['case'] 'object' ObjectDef
              | 'trait' TraitDef
```

クラス(§5.3)とオブジェクト(§5.4)は共にテンプレートの言葉で定義されています。

5.1 テンプレート (Templates)

構文:

```
ClassTemplate ::=   [EarlyDefs] ClassParents [TemplateBody]
TraitTemplate ::=   [EarlyDefs] TraitParents [TemplateBody]
ClassParents  ::=   Constr {'with' AnnotType}
TraitParents  ::=   AnnotType {'with' AnnotType}
TemplateBody  ::=   [nl] '{' [SelfType] TemplateStat {semi TemplateStat} '}'
SelfType      ::=   id [':' Type] '=>'
                  |   this ':' Type '=>'
```

A template defines the type signature, behavior and initial state of a trait or class of objects or of a single object. Templates form part of instance creation expressions, class definitions, and object definitions. A template sc with mt1 with ... with mtn {stats} consists of a constructor invocation sc which defines the template's superclass, trait references mt1,...,mtn (n >= 0), which define the template's traits, and a statement sequence stats which contains initialization code and additional member definitions for the template.

テンプレートは、クラス/トレイトあるいはオブジェクト群あるいは単一のオブジェクトの、型シグニチャ、振る舞い、初期状態を定義します。 テンプレートは、インスタンス生成式、クラス定義とオブジェクト定義等の一部を形成します。 テンプレート `sc with mt1 with ... with mtn {stats}` は、テンプレートの**スーパークラス**を定義するコンストラクタ呼び出し `sc`、テンプレートの**トレイト**を定義するトレイト参照 `mt1,...,mtn (n >= 0)`、テンプレートの初期化コードと追加のメンバー定義を含む文並び `stats` からなります。

各トレイト参照 `mti` は、トレイト(§5.3.3)を表わします。 これと対照して、スーパークラスコンストラクタ `sc` は通常、トレイトではないクラスを参照します。 トレイト参照で始まる親のリストを書くことが可能です。 例えば、`mt1 with ... with mtn`。この場合、親のリストは、最初の親型として `mt1` のスーパー型を含むように暗黙のうちに展開されます。 新しいスーパー型は、パラメータをとらない少なくとも 1 つのコンストラクタを持っていなくてはなりません。 以降では、常にこの暗黙の展開が実行されると想定します。 ですから、テンプレートの最初の親クラスは、通常のスーパークラスコンストラクタであり、トレイト参照ではありません。

あらゆるクラスの親のリストは同様に、最後のミックスインである `scala.ScalaObject` トレイトへの参照により、常に暗黙のうちに拡張されます。 つまり、

```
sc with mt1 with ... with mtn {stats }
```

は、次になります。

```
mt1 with ... with mtn {stats } with ScalaObject {stats}
```

テンプレートの親のリストは、正しい形でなければなりません。 これはスーパークラスコンストラクタ sc によって表されるクラスが、すべてのトレイト mt1,...,mtn のスーパークラスのサブクラスでなければならないことを意味します。 言い換えれば、テンプレートによって継承される非トレイトクラスは、テンプレートのスーパークラスで始まる継承階層の中で、1 つのチェインを形成します。

テンプレートの**最小固有のスーパー型(least proper supertype)**とは、そのすべての親クラス型から構成される複合型(§3.2.7)あるいはクラス型です。

文並び stats は、新しいメンバーを定義するメンバー定義あるいは、親クラス中のメンバーを上書きするメンバー定義を含みます。 もしテンプレートが抽象クラスあるいはトレイト定義の一部を形づくるなら、文部分 stats も抽象メンバの宣言を含んでいて構いません。 もしテンプレートが具象クラス定義の一部を形づくるなら、stats はそれでも、抽象型メンバの宣言を含んでいて構いません。 ただし、抽象項メンバの宣言は含みません。 さらに、stats は、いずれの場合も式を含み得ます。; それらはテンプレートの初期化の一部として、与えられた順に実行されます。

テンプレートの文並びには、形式上のパラメータ定義と矢印、たとえば x => あるいは x :T => が前置されることがあります。 もし形式上のパラメータが与えられていれば、それをテンプレート本体全体にわたって this 参照のエイリアスとして使えます。

もし形式上のパラメータが型 T が一緒に書いてあるなら、その定義は、下地となっているクラス/オブジェクトの**自己型** S に次のような影響を与えます。: まず、C をテンプレートを定義するクラス/トレイト/オブジェクトの型であると仮定します。 もし型 T が形式上の自己パラメータに対して与えられているなら、S は T と C の最大の下限境界です。もし型 T が与えられていないなら、 S はたんに C です。 テンプレート内では、this の型は S であると想定されます。

クラスあるいはオブジェクトの自己型は、テンプレート t によって継承されるすべてのクラスの自己型に適合しなくてはなりません。

自己型アノテーションの 2 つめの形は、たんに this: S => と書きます。 これにより、エイリアス名を導入せずに this の型 S を記述できます。

Example 5.1.1 : 次のクラス定義について考えます:

```
class Base extends Object {}
trait Mixin extends Base {}
object O extends Mixin {}
```

この場合、O の定義は次のように展開されます。

```
object O extends Base with Mixin {}
```

Java 型からの継承 :テンプレートは、そのスーパークラスとして Java クラスを、そのミックスインとして Java インターフェースを持っていても構いません。

テンプレートの評価 :

テンプレート sc with mt1 with mtn {stats } について考えます。

もしこれがトレイト(§5.3.3)のテンプレートなら、その**ミックスイン評価**は、文並び stats の**評価**から構成されます。

もしこれがトレイトのテンプレートでないなら、その評価は、次のステップによって構成されます。

- 最初に、スーパークラスコンストラクタ sc が評価されます(§5.1.1)。
- 次に、sc によって表わされるテンプレートのスーパークラスに至る、 テンプレート線形化(§5.1.2)中のすべての基底クラスは、 ミックスイン評価されます。 ミックスイン評価が、線形化中での出現の逆順でされます。
- 最後に、文並び stats が評価されます。

5.1.1 コンストラクタ呼び出し (Constructor Invocations)

構文:

```
Constr      ::=   AnnotType {'(' [Exprs] ')'}
```

Constructor invocations define the type, members, and initial state of objects created by an instance creation expression, or of parts of an object's definition which are inherited by a class or object definition. A constructor invocation is a function application x.c[targs](args1)...(argsn) , where x is a stable identifier (§3.1), c is a type name which either designates a class or defines an alias type for one, targs is a type argument list, args1,...,argsn are argument lists, and there is a constructor of that class which is applicable (§6.6) to the given arguments. If the constructor invocation uses named or default arguments, it is transformed into a block expression using the same transformation as described in (§6.6.1) .

コンストラクタ呼び出しは、型、メンバー、そしてインスタンス生成式によって生成されるオブジェクトの初期状態、あるいはクラス/オブジェクト定義によって継承されたオブジェクト定義部分の初期状態等を定義します。 コンストラクタ呼び出しは、関数適用 x.c[targs](args1)...(argsn)です。 ここで、x は安定識別子(§3.1)であり、c は型名で、クラスを指定するかあるいはクラスのエイリアス型を定義するかのいずれかであり、targs は型引数リスト、 args1,...,argsn は引数リスト、そして与えられた引数に適用可能(§6.6)なそのクラスのコンストラクタがあります。 もしコンストラクタ呼び出しが、名前付きあるいはデフォルト引数を用いるときは、(§6.6.1)中で記述したのと同じ変形を使ったブロック式に変換されます。

前置子 'x.' は省略できます。 クラス c には、それが型パラメータをとる場合に限り、型引数リストを与えることができます。 このときにも、それを省略できます。 その場合、型引数リストはローカルな型推論(§6.26.4)を使って合成されます。 もし明示的な引数が与えられていないなら、空リスト()が暗黙のうちに補充されます。

コンストラクタ呼び出し x.c[targs](args1)...(argsn)の評価は、次のステップからなります。

- 最初に、前置子 x が評価されます。
- 次に、引数 args1,...,argsn が左から右へ評価されます。
- 最後に、構成されつつあるクラスが、 c によって参照されるクラスのテンプレートを評価することで初期化されます。

5.1.2 クラス線形化 (Class Linearization)

クラス C からの直接の継承関係の推移的なクロージャを通して到達可能なクラスは、C の**基底クラス(base classes)**と呼ばれます。 ミックスインにより、基底クラス上の継承関係は一般に有向非巡回グラフになります。 このグラフの線形化は、次のように定義されています。

定義 5.1.2 : C をテンプレート C1 with ... with Cn { stats } をもつクラスとします。 C の線形化 LL(C)は、次のように定義されます。:

```
LL(C) = C , LL(Cn) ⊕ ... ⊕ LL(C1)
```

ここで ⊕ は連結を表し、右オペランドを左オペランドの同一の要素で置き換えます。:

```
{a, A} ⊕ B   = a,(A ⊕ B)  if a ∉ B
             = A ⊕ B      if a ∈ B
```

Example 5.1.3 : 次のクラス定義について考えます。

```
abstract class AbsIterator extends AnyRef { ... }
trait RichIterator extends AbsIterator { ... }
class StringIterator extends AbsIterator { ... }
class Iter extends StringIterator with RichIterator { ... }
```

クラス Iter の線形化は、次のようになります。

```
{Iter,RichIterator,StringIterator,AbsIterator,ScalaObject,AnyRef,Any}
```

このリストの中ではトレイト ScalaObject が現れます。 なぜなら、それはすべての Scala クラスに最後のミックスインとして加えられるからです(§5.1)。

クラス線形化は、継承関係を精緻化することに注意してください。: もし C が D のサブクラスなら、C と D 両方が現れるどのような線形化中でも、C は D より先に現れます。定義 5.1.2 は、 クラス線形化が常にその直上のスーパークラスの線形化を接尾部として含む、という特性も満たします。

例えば、StringIterator の線形化は、

```
{ StringIterator, AbsIterator, ScalaObject, AnyRef, Any }
```

です。 これは、そのサブクラス Iter の線形化の接尾部となっています。 同じことはミックスインの線形化では、正しくありません。 例えば、RichIterator の線形化は次です。

```
{ RichIterator, AbsIterator, ScalaObject, AnyRef, Any }
```

これは Iter の線形化の接尾部ではありません。

5.1.3 クラスメンバ (Class Members)

テンプレート C1 with ... with Cn { stats } によって定義されたクラス C は、その文並び stats の中でメンバーを定義でき、また、すべての親クラスからメンバーを継承できます。 Scala はメソッドの静的なオーバーロードに関して Java と C# の規則を採用しています。 ですからクラスは同じ名前の複数のメソッドを定義あるいは継承できます。

クラス C の定義されたメンバーがどの親クラスのメンバーをオーバライドするか、あるいは、C 中のオーバーロードされた変位指定として 2 つが共存するかどうかを決めるために、Scala は次のメンバー**マッチング**の定義を用います。

定義 5.1.4もし M と M′ が同じ名前を束縛し、次の 1 つが満たされるなら、メンバー定義 M はメンバー定義 M′と**マッチします**。

1. M と M′のいずれもメソッド定義ではない。
2. M と M′は両方とも、等価な引数型をもつ単相的なメソッドを定義する。
3. M はパラメータなしのメソッドを定義し、M′ は空きのパラメータリスト () のメソッドを定義するかあるいは、**その逆**である。
4. M と M′は両方とも、同数の引数型 T、T′ と、 同数の型パラメータ t、t′をもつ (たとえば T′ = [t′/t]T) 多相的メソッドを定義する (訳注:英語原文では、Tや t には上線がついていました)。

メンバー定義は 2 つのカテゴリ、具象または抽象、に分けられます。 クラス C のメンバーは、**直接定義される**(すなわち、それらは C の文並び stats 中に現れる)かあるいは、**継承された**かのいずれかです。 クラスのメンバーの集合を決定する 2 つの規則があり、それぞれ 1 つのカテゴリーに対応します。

定義 5.1.5 : クラス C の**具象メンバー**とは、あるクラス Ci∈LL(C) 中のすべての具象定義 M である。 ただしここで、M にマッチする具象メンバーM′を直接定義するような、先行するクラス Cj∈LL(C) (j < i)がある場合を除く。

クラス C の**抽象メンバー**とは、あるクラス Ci∈LL(C) 中のすべての抽象定義 M である。 ただしここで、C がすでに M にマッチする具象メンバーM′ を含むか、あるいは、M にマッチする抽象メンバーM′を直接定義するような、先行するクラス Cj∈LL(C) (j < i)がある場合を除く。

この定義は、クラス C のマッチするメンバーとその親との間のオーバーライド関係も決定します (§5.1.4)。 第一に、具象定義は常に抽象定義をオーバライドします。 第二に、共に具象であるかあるいは共に抽象的な定義 M と M'について、もし M が、M′ が定義されているクラスよりも (C の線形化で)先行するクラス中に現れるなら、M は M′をオーバライドします。

もしテンプレートが 2 つのマッチするメンバーを直接定義しているなら、エラーです。 もしテンプレートが同じ名前と同じ消去型(erased type)(§3.7)をもつ 2 つの(直接定義されたか、あるいは継承された) メンバーを含むなら、同様にエラーです。 最後に、テンプレートは、同じ名前で共にデフォルト引数を定義する(直接定義されたか、あるいは継承された) 2 つのメソッドを含むことは許されていません。

Example 5.1.6 : 次のトレイト定義について考えます。

```
trait  A  { def f: Int }
trait  B  extends A { def f: Int = 1 ; def g: Int = 2 ; def h: Int = 3 }
trait  C  extends A { override def f: Int = 4 ; def g: Int }
trait  D  extends B with C { def h: Int }
```

このときトレイト D は直接定義された抽象メンバ h を持っています。 それは、トレイト B からメンバー g を、トレイト C からメンバー f を継承します。

5.1.4 オーバーライド (Overriding)

C の基底クラスの非 private なメンバー M′ にマッチ(§5.1.3)する、クラス C のメンバー M は、そのメンバーを**オーバライド**すると言われます。 この場合、オーバーライドするメンバー M の束縛は、オーバライドされるメンバー M′ の束縛を包含しなくてはなりません(§3.5.2)。 さらに、修飾子に関する次の制限が M および M′へ適用されます。

- M′ は final と印されていないこと。
- M は private ではないこと(§5.2)。
- もし M が、ある取り囲むクラス/パッケージ C について private[C] と印されているなら、M′はある取り囲むクラス/パッケージ C について private[C′] と印されていること。ただしここで、C′は C に等しいか、あるいは C′は C に含まれるものとします。
- もし M が protected と印されているなら、M′もまた protected と印されていること。
- もし M′ が抽象メンバでないなら、M は override と印されていること。 さらに、次の 2 つの可能性の 1 つが満たされなくてはなりません。
    - M は、M′ が定義されているクラスのサブクラス内で定義されているか、
    - あるいは M と M′ の両方とも、M や M′ を含む両クラスの (1 つの)基底クラス内で定義されている第 3 のメンバー M′′ をオーバライドする。
- もし M′が C 中で不完全(incomplete §5.2)なら、 M は abstract override と印されていること。
- もし M と M′が両方とも具象値定義なら、そのどちらも lazy と印されないか、あるいは両方とも lazy と印されるかのどちらかであること。

パラメータなしのメソッドに関係する特別の規則があります。 もし def f : T = ... あるいは def f = ... として定義されたパラメータなしのメソッドが、空きのパラメータリストをもつ型 ()T′ のメソッドをオーバライドするなら、 f も同じく空きのパラメータリストを持つと想定されます。

もう 1 つの制限が抽象型メンバーに適用されます。: その上限境界として volatile 型(§3.6)をもつ抽象型メンバーは、volatile 型の上限境界をもたない抽象型メンバーをオーバーライドできません。

オーバーライドするメソッドは、スーパークラス中の定義からすべてのデフォルト引数を継承します。 オーバーライドするメソッド中でデフォルト引数を指定して、(もしスーパークラス中の対応するパラメータがデフォルトを持たないなら) 新しいデフォルトを追加したり、あるいは(そうでなければ)、スーパークラスのデフォルトをオーバライドできます。

Example 5.1.7 : 次の定義を考えます。

```
trait   Root { type T <: Root }
trait   A extends Root { type T <: A }
trait   B extends Root { type T <: B }
trait   C extends A with B
```

このとき、クラス定義 C は正しくありません。 なぜなら、C 中の T の束縛は type T <: B であり、型 A 中の T の束縛 type T <: A の包含に失敗するからです。 この問題は、クラス C 中に型 T のオーバーライド定義を加えることで解決できます:

```
class C extends A with B { type T <: C }
```

5.1.5 継承クロージャ（Inheritance Closure）

C はクラス型であるとします。C の**継承クロージャ(inheritance closure)**とは、次のような型の最小の集合 SS です。

- もし T が SS 中にあれば、T の一部を構文的に形成するすべての型 T′ もまた SS 中にある。
- もし T が SS 中のクラス型なら、T のすべての親(§5.1)は同じく SS 中にある。

もしクラス型の継承クロージャが無数の型からなるなら、静的エラーです（この制限は、サブ型付けを決定可能とするために必要です[KP07])。

5.1.6 事前定義（Early Definitions）

構文:

```
EarlyDefs             ::= '{' [EarlyDef {semi EarlyDef}] '}' 'with'
EarlyDef              ::= {Annotation} {Modifier} PatVarDef
```

テンプレートを**事前フィールド定義(early field definition)**節で始めることができ、それによりスーパー型のコンストラクタがコールされる前に、ある特定のフィールド値を定義できます。

次のテンプレート中で

```
{ val p1 : T1 = e1
  ...
  val pn : Tn = en
} with sc with mt1 with mtn {stats}
```

p1,...,pn 定義の最初のパターンは**事前定義(early difinition)**と呼ばれます。 それらはテンプレートの一部をなすフィールドを定義します。 すべての事前定義は、少なくとも 1 つの変数を定義していなくてはなりません。

事前定義は型チェックされ、そのテンプレートが定義される直前に有効なスコープ中で評価されます。 また、取り囲むクラスのすべての型パラメータと、定義しているものに先行するすべて

の事前定義によって拡張されます。 特に、事前定義の右辺における this への参照はすべて、テンプレートのすぐ外の this 識別子への参照です。 したがって、事前定義が、テンプレートによって構築中のオブジェクトを参照したり、あるいは、同じセクション中の先行する他の事前定義以外のフィールドやメソッドの 1 つを参照することはできません。 さらに、先行する事前定義への参照は、そこで定義されている値を常に参照し、オーバーライド定義は考慮されません。 言い換えると、事前定義ブロックはまさに、複数の値定義を含むローカルなブロックであるかのように評価されます。

事前定義は、テンプレートのスーパークラスコンストラクタが呼ばれる前に、それらが定義された順番で評価されます。

Example 5.1.8 : 事前定義は、通常のコンストラクタパラメータをもたないトレイトに対して特に役立ちます。 例:

```
trait Greeting {
  val name: String
  val msg = "How are you, "+name
}
class C extends {
  val name = "Bob"
} with Greeting {
  println(msg)
}
```

上記のコードで、フィールド name は Greeting のコンストラクタが呼び出される前に初期化されます。 ですから、クラス Greeting 中のフィールド msg は "How are you, Bob" に適切に初期化されます。

代わりに、もし name が C の通常のクラス本体中で初期化されると、それは Greeing のコンストラクタの後に初期化されます。 その場合、msg は "How are you, <null>" と初期化されます。

5.2 修飾子 (Modifiers)

構文:

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

メンバー定義には修飾子が先行することがあり、それらは結びつく識別子のアクセス性と使用法に影響を与えます。 複数の修飾子が与えられていても、その順番は重要ではありません。 しか

し同じ修飾子は一度以上は現れはなりません。 定義の繰り返しに先行する修飾子は、各定義すべてに適用されます。 修飾子の有効性と意味を決定する規則は次の通りです。

・ private 修飾子はテンプレート中の任意の定義または宣言と共に使えます。 そのようなメンバーは、直接に取り囲むテンプレートとそのコンパニオンモジュールあるいはコンパニオンクラス(§5.4)中からのみアクセスできます。それらはサブクラスによって継承されません。 また、それらは親クラス中の定義をオーバライドできません。

修飾子は識別子 C で**限定修飾(qualified)**でき（たとえば private[C])、 この C は定義を囲むパッケージあるいはクラスを表していなければなりません。 そのような修飾子が印されたメンバーは、パッケージ C のコード中から、あるいはクラス C とそのコンパニオンモジュール(§5.4)のコード中からのみ、それぞれアクセス可能です。 そのようなメンバーはまた、C 中のテンプレートからのみ継承されます。

private[this] は限定修飾の異なる形です。 この修飾子を使ってマークされたメンバー M は、それが定義されたオブジェクト内からのみアクセスできます。 すなわち、選択 p.M は、その参照を囲むあるクラス O があって、前置子が this あるいは O.this である場合のみ正しいとされます。 加えて、限定修飾なしの private の制約が適用されます。

限定修飾子がない private と印されたメンバーは、**クラス非公開(class-private)**と呼ばれるのに対して、 private[this] と印されたメンバーは、 **オブジェクト非公開(object-private)**と呼ばれます。 メンバーは、もしそれがクラス非公開あるいはオブジェクト非公開で、private[C] (ここで C は識別子)とマークされていないなら、**非公開(private)** です。; 後者の場合、そのメンバーは**限定非公開 (qulified private)**と呼ばれます。

クラス非公開あるいはオブジェクト非公開のメンバーは、抽象であってはならず、また、protected あるいは override 修飾子を持ってはいけません。

・ protected 修飾子はクラスメンバ定義に適用されます。クラスの protected メンバーは、次の中からアクセスできます。

- 定義しているクラスのテンプレート、
- 定義しているクラスを基底クラスとしてもつ、すべてのテンプレート
- それらクラスの任意のコンパニオンモジュール

protected 修飾子は、識別子 C で限定修飾できます(たとえば、protected[C])。 ここで C は、その定義を囲むクラスあるいはパッケージを表さなくてはなりません。 そのような修飾子を印されたメンバーもまた、パッケージ C 中のすべてのコードから、 あるいはクラス C とそのコンパニオンモジュール(§5.4)中のすべてのコードから、 それぞれアクセス可能です。

protected 識別子 x は、次の 1 つが適用される場合に限り、選択 r.x 中でメンバー名として使えます。

- そのアクセスがメンバーを定義しているテンプレート中にあるか、 あるいは、限定修飾 C が与えられている場合は、 そのアクセスがパッケージ C 中あるいはクラス C 中、 あるいはそのコンパニオンモジュール中にある。あるいは、
- r は、予約語 this あるいは super の 1 つ。あるいは
- r の型が、そのアクセスを含むクラスの型インスタンスに適合する。

protected[this] は、限定修飾の異なる形です。 この修飾子を印されたメンバー M は、それが定義されたオブジェクト内からのみアクセスできます。 すなわち、選択 p.M は、その参照を囲むあるクラス O に対して、 前置子が this あるいは O.this である場合のみ正しいとされます。 加えて、限定修飾なしの protected の制約が適用されます。

・ override 修飾子は、クラスメンバ定義または宣言に適用されます。 これは親クラス中の他のある具象メンバー定義をオーバライドする、メンバー定義/宣言について必須です。もし override 修飾子が与えられていれば、 少なくとも 1 つのオーバライドされる(具象あるいは抽象の)

メンバー定義/宣言がなければなりません。

・ override 修飾子は abstract 修飾子と組み合わされるとき、意味が追加されます。 この修飾子の組合せは、トレイトの値メンバーに対してのみ許されます。

テンプレートのメンバー M は、もしそれが抽象(すなわち、宣言によって定義されている)であるか、 あるいは、abstract かつ override と印されているなら、**不完全(incomplete)**と呼ばれます。 そして、M によってオーバライドされたすべてのメンバーは、再び不完全です。

abstract override 修飾子の組合せは、メンバーが具象か抽象かということに影響を与えないことに注意してください。 メンバーは、もしそれに関して宣言だけが与えられていれば**抽象(abstract)**であり、 もし完全な定義が与えられていれば**具象(concrete)**です。

・ abstract 修飾子はクラス定義の中で使います。 これはトレイトには不必要であり、不完全なメンバーを持つ他のすべてのクラスでは必須です。 抽象クラスは、クラスのすべての不完全なメンバーをオーバライドするミックスインや細別(refinement) が後に続かないなら、コンストラクタ呼び出しでインスタンス化(§6.10)することはできません。 抽象クラスとトレイトだけが抽象項(term)メンバーを持てます。

abstract 修飾子は、クラスメンバ定義についても override と一緒に使えます。 その場合、前の議論が適用されます。

・ final 修飾子は、クラスメンバ定義とクラス定義に適用されます。 サブクラス中で final クラスメンバー定義をオーバライドしてはなりません。 テンプレートは final クラスを継承できません。 final はオブジェクト定義には不必要です。 final クラス/オブジェクトのメンバーは、暗黙のうちに同様に final です。 ですから、それらについても final 修飾子は不必要です。 final は不完全なメンバーに適用できず、また、1 つの修飾子リスト中で sealed とは一緒には使えません。

・ The sealed modifier applies to class definitions. A sealed class may not be directly inherited, except if the inheriting template is defined in the same source file as the inherited class. However, subclasses of a sealed class can be inherited anywhere .

・ sealed 修飾子は、クラス定義に適用されます。 sealed クラスは、継承するテンプレートを継承されるクラスと同じソースファイル中で定義する場合を除き、直接には継承できません。 しかし、sealed クラスのサブクラスはどこででも継承できます。

・ lazy 修飾子は値定義に適用されます。 遅延評価 Val は、それが最初にアクセスされる(決して起きないかもしれない)ときに初期化されます。遅延評価 Val をその初期化中にアクセスしようとすると、動作がループするかもしれません。 初期化中に例外が送出された場合は、その値は初期化されていないとみなされ、後のアクセスでその右辺の評価が再び試みられるでしょう。

Example 5.2.1 : 次のコードは限定修飾された private の使用を示します。:

```
package outerpkg.innerpkg
class Outer {
  class Inner {
    private[Outer] def f()
    private[innerpkg] def g()
    private[outerpkg] def h()
  }
}
```

ここで、メソッド f へのアクセスは、OuterClass 内ならどこにでも現れることができますが、その外では現われることはできません。 メソッド g へのアクセスは、Java の package-private の場合と同じように、パッケージ outerpkg.innerpkg 内のどこにでも現われることができます。 最後に、メソッド h へのアクセスは、パッケージ outerpkg 内の、そのパッケージが含むパッケージを含めて、どこにでも現われることができます。

Example 5.2.2 : クライアントにクラスの新しいインスタンスを構築させないための 1 つの方法は、そのクラスを abstart かつ sealed と宣言することです。:

```
object m {
  abstract sealed class C (x: Int) {
    def nextC = new C(x + 1) {}
  }
  val empty = new C(0) {}
}
```

たとえば、上記のコード中でクライアントは、既存の m.C オブジェクトの nextC メソッドを呼び出すことによってのみ、クラス m.C のインスタンスを生成できます。; クライアントはクラス m.C のオブジェクトを直接生成できません。 実際、次の 2 つの行はどちらもエラーです。

```
new m.C(0)    // **** error: C は abstract なのでインスタンス化できない
new m.C(0) {} // **** error: sealed クラスからの不正継承
```

基本コンストラクタを private とすることで、同じようなアクセス制限を課すことができます(Example 5.3.2 参照)。

5.3 クラス定義 (Class Definitions)

構文:

```
TmplDef            ::= 'class' ClassDef
ClassDef           ::= id [TypeParamClause] {Annotation}
                       [AccessModifier] ClassParamClauses ClassTemplateOpt
ClassParamClauses  ::= {ClassParamClause}
                       [[nl] '(' implicit ClassParams ')']
ClassParamClause   ::= [nl] '(' [ClassParams] ')'
ClassParams        ::= ClassParam {',' ClassParam}
ClassParam         ::= {Annotation} [{Modifier} ('val' | 'var')]
                       id [':' ParamType] ['=' Expr]
ClassTemplateOpt   ::= 'extends' ClassTemplate | [['extends'] TemplateBody]
```

クラス定義の最も一般的な形は次です。

```
class c[tps] as m(ps1)...(psn) extends t    (n >= 0)
```

ここで、

c は定義されるクラスの名前です。

tps は定義されるクラスの型パラメータの非空リストです。 型パラメータのスコープは、それ自身の型パラメータ部を含むクラス定義全体です。 同じ名前の 2 つの型パラメータを定義することは不正です。 型パラメータ部 [tps] は省略されるかもしれません。 型パラメータ部をもつクラスは**多相的(polymorphic)**と呼ばれ、 そうでなければ、**単相的(monomorphic)**と呼ばれます。

as はアノテーション(§11)の並びで、空きでも構いません。 もしアノテーションが与えられていれば、 それらはクラスの基本コンストラクタへ適用されます。

m は private/protected のようなアクセス修飾子(§5.2)で、 限定修飾子をともなうかもしれません。 もしそのようなアクセス修飾子が与えられていれば、 クラスの基本コンストラクタへ適用されます。

(ps1) ... (psn) are formal value parameter clauses for the primary constructor of the class. The scope of a formal value parameter includes all subsequent parameter sections and the template t . However, a formal value parameter may not form part of the types of any of the parent classes or members of the class template t . It is illegal to define two formal value parameters with the same name. If no formal parameter sections are given, an empty parameter section () is assumed.

(ps1)...(psn) は、クラスの**基本コンストラクタ(primary constructor)**の形式上の値パラメータ節です。 形式上の値パラメータのスコープは、 後に続くすべてのパラメータ部とテンプレート t を含みます。 しかし、形式上の値パラメータは、いかなる親クラスの、 あるいはクラステンプレート t のメンバーの型部分も形成しません。 同じ名前をもつ 2 つの形式上の値パラメータを定義することは不正です。 もし形式上のパラメータ部が与えられていないなら、 空きのパラメータ部 () が想定されます。

もし形式上のパラメータ宣言 x : T に val または var キーワードが先行するなら、 このパラメータ用のアクセス子(ゲッター)定義(§4.2) が暗黙のうちにクラスに加えられます。 ゲッターは、パラメータのエイリアスとして定義される、 クラス c の値メンバー x を導入します。 もし導入するキーワードが var なら、セッターアクセス子 x _= (§4.2)も暗黙のうちにクラスに加えられます。 セッターの呼び出しにおいて、x _=(e) は、パラメータの値を e の評価結果に変えます。 形式上のパラメータ宣言は修飾子を含むことができ、それはアクセス子定義へ持ち越されます。 val または var が前置された形式上のパラメータは、同時に名前呼び出しパラメータ(§4.6.1)にはできません。

t は次の形のテンプレート(§5.1)です。

```
sc with mt1 with ... with mtm { stats } (m >= 0)
```

これは、基底クラス、そのクラスのオブジェクトの初期状態と振る舞いを定義します。 継承節 extends sc with mt1 with ... with mtm は、省略されるかもしれません。 その場合、extends scala.AnyRef が想定されます。 クラス本体 {stats} も同じく省略されるかもしれません。その場合は、空の本体 {} が想定されます。

このクラス定義は、型 c[tps] とコンストラクタを定義し、 そのコンストラクタは、 型 ps に適合するパラメータへ適用されるときにテンプレート t を評価して、 型 c[tps] のインスタンスを初期化します。

Example 5.3.1 : 次の例は、クラス C の val と var パラメータを示します:

```
class C(x: Int, val y: String, var z: List[String])
val c = new C(1, "abc", List())
c.z = c.y :: c.z
```

Example 5.3.2 : 次のクラスは、そのコンパニオンモジュールからのみ生成できます。

```
object Sensitive {
  def makeSensitive(credentials: Certificate): Sensitive =
    if (credentials == Admin) new Sensitive()
```

```
    else throw new SecurityViolationException
}
class Sensitive private () {
  ...
}
```

5.3.1 コンストラクタ定義（Constructor Definitions）

構文:

```
FunDef         ::= 'this' ParamClause ParamClauses
                   ('=' ConstrExpr | [nl] ConstrBlock)
ConstrExpr     ::= SelfInvocation
                 | ConstrBlock
ConstrBlock    ::= '{' SelfInvocation {semi BlockStat} '}'
SelfInvocation ::= 'this' ArgumentExprs {ArgumentExprs}
```

クラスは基本コンストラクタのほかに、追加のコンストラクタを持てます。 それらは `def this(ps1)...(psn) = e` の形のコンストラクタ定義で定義できます。 このような定義は、取り囲むクラスに対し、形式上のパラメータリスト `ps1,...,psn` として与えられたパラメータと、その評価がコンストラクタ式 `e` で定義された追加のコンストラクタを導入します。 各形式上のパラメータのスコープは、後に続くパラメータ部とコンストラクタ式 `e` です。コンストラクタ式は、自己コンストラクタ呼び出し `this(args1)...(argsn)` 、あるいは自己コンストラクタ呼び出しで始まるブロックです。 自己コンストラクタ呼び出しは、クラスのジェネリックなインスタンスを構築しなくてはなりません。 すなわち、もし問題のクラスが名前 `C` と型パラメータ `[tps]` をもつなら、自己コンストラクタ呼び出しは `C[tps]` のインスタンスを生成しなくてはなりません。; 形式上の型パラメータをインスタンス化することは許されていません。

コンストラクタ定義のシグニチャと自己コンストラクタ呼び出しは、取り囲むクラス定義の実際の場所のスコープ内で、型チェックおよび評価されます。 また、取り囲むクラスのすべての型パラメータと取り囲むテンプレートのすべての事前定義(§5.1.6)によって拡張されます。 コンストラクタ式の残りは、現在のクラス内の関数本体として型チェックされ、評価されます。

もしクラス `C` の補助コンストラクタがあれば、それらは `C` の基本コンストラクタ(§5.3)と共に、オーバーロードされたコンストラクタ定義を形成します。 オーバーロード解決(§6.26.3)の通常の規則が、クラス `C` のコンストラクタ呼び出しに、それ自身のコンストラクタ式中の自己コンストラクタ呼び出しの場合も含めて、適用されます。 しかし他のメソッドと異なり、コンストラクタは決して継承されません。 コンストラクタ呼び出しの無限ループを防ぐために、すべての自己コンストラクタ呼び出しは、それに先行するコンストラクタ定義を参照しなければならないという制約があります（すなわち、先行する補助コンストラクタあるいはクラスの基本コンストラクタのいずれかを参照しなくてはなりません)。

Example 5.3.3: 次のクラス定義について考えます。

```
class   LinkedList[A]() {
  var   head = _
  var   tail = null
  def   isEmpty = tail != null
  def   this(head: A) = { this(); this.head = head }
  def   this(head: A, tail: List[A]) = { this(head); this.tail = tail }
}
```

これは 3 つのコンストラクタをもつクラス LinkedList を定義しています。 2 番目のコンストラクタはシングルトンリストを構築し、他方、3 番目は与えられた head と tail をもつリストを構築します。

5.3.2 ケースクラス (Case Classes)

構文:

```
TmplDef ::=     'case' 'class' ClassDef
```

クラス定義の前に case が置かれていると、クラスはケースクラスと言われます。

ケースクラスの最初のパラメータ部中の形式上のパラメータは、**要素(elements)**と呼ばれます;それらは特別に扱われます。 第一に、そのようなパラメータの値は、コンストラクタパターンのフィールドとして抽出できます。 第二に、パラメータに既に val または var 修飾子がついていなければ、val 前置子がそのようなパラメータに暗黙の内に加えられます。 ですから、パラメータのアクセス子定義が生成されます(§5.3)。

型パラメータ tps と値パラメータ ps をもつケースクラス定義 c[tps](ps1)...(psn) は、次のような、抽出子オブジェクト(§8.1.8)の定義を暗黙のうちに生成します。

```
object c {
  def apply[tps](ps1)...(psn): c[tps] = new c[Ts](xs1)...(xsn)
  def unapply[tps](x : c[tps]) =
    if (x eq null) scala.None
    else scala.Some(x.xs11,...,x.xs1k)
}
```

ここで、Ts は型パラメータ部 tps で定義された型のベクトル(一次元配列)を表し、各 xsi はパラメータ部 psi のパラメータ名を示し、xs11,...,xs1k は最初のパラメータ部 xs1 中のすべてのパラメータの名前を示します。 もし型パラメータ部がクラスにないなら、それは apply および unapply メソッドにもありません。 もしクラス c が abstract なら、apply の定義は省略されます。

もしケースクラス定義が空の値パラメータリストをもつなら、unapply メソッドはオプション型の代わりに Boolean を返し、次のように定義されます。:

```
def unapply[tps](x : c[tps]) = x ne null
```

もし c の最初のパラメータ部 ps1 が反復パラメータ(§4.6.2)で終わるなら、unapply メソッドの名前は unapplySeq に変わります。 もしコンパニオンオブジェクト c が既に存在するなら、新しいオブジェクトは生成されず、代わりに apply および unapply メソッドが既存のオブジェクトに加えられます。

copy と名付けられたメソッドが、クラスがそういう名前の(直接定義されたか、あるいは継承した)メンバーを持たない限り、暗黙のうちにすべてのケースクラスに加えられます。 メソッドは次のように定義されます:

```
def copy[tps](ps′1)...(ps′n): c[tps] = new c[Ts](xs1)...(xsn)
```

ここで再び、Ts は型パラメータ部 tps で定義された型のベクトルを表し、各 xsi はパラメータ部 ps′i のパラメータ名を示します。 copy メソッドのすべての値パラメータ $ps'_{i,j}$ は、 $x_{i,j}$:$T_{i,j}$ =this.$x_{i,j}$ の形をしています。 ここで、$x_{i,j}$ と $T_{i,j}$ は、対応するクラスパラメータ

$ps_{i,j}$ の名前と型を参照します。

すべてのケースクラスは、クラス scala.AnyRef(§12.1)のいくつかのメソッド定義を暗黙のうちにオーバライドします。 ただしそれは、同じ名前のメソッド定義がケースクラス自身で既に与えられていないか、あるいは同じ名前のメソッドの具象定義がケースクラスの AnyRef 以外の基底クラス中で与えられていない場合です。特に:

メソッド equals:(Any)Boolean は構造的な等価(structural equality)を表します。ここで 2 つのインスタンスは、もしそれらが両方とも問題のケースクラスに属し、そしてそれらが (equals に関して) 等価なコンストラクタ引数を持つなら、等価です。

Method hashCode: Int computes a hash-code. If the hashCode methods of the data structure members map equal (with respect to equals) values to equal hash-codes, then the case class hashCode method does too .

メソッド hashCode:Int は、ハッシュ・コードを計算します。もしデータ構造メンバーの hashCode メソッドが、(equals に関して)等しい値を等しいハッシュ・コードにマップするなら、ケースクラスの hashCode メソッドも同様にマップします。

メソッド toString:String は、クラスとその要素名を含む文字列表現を返します。

Example 5.3.4 :

次は、λ計算のための抽象構文定義です。:

```
class Expr
case class Var   (x: String)          extends Expr
case class Apply (f: Expr, e: Expr)   extends Expr
case class Lambda(x: String, e: Expr) extends Expr
```

これはケースクラス Var、Apply と Lambdaをもつクラス Expr を定義します(訳注:関係が逆?)。 ラムダ式に対する値呼出し評価子は、このとき次のように書けます。

```
type Env = String => Value
case class Value(e: Expr, env: Env)

def eval(e: Expr, env: Env): Value = e match {
  case Var (x) =>
    env(x)
  case Apply(f, g) =>
    val Value(Lambda (x, e1), env1) = eval(f, env)
    val v = eval(g, env)
    eval (e1, (y => if (y == x) v else env1(y)))
  case Lambda(_, _) =>
    Value(e, env)
}
```

プログラムの他の場所で、型 Expr を拡張するケースクラスをさらに定義できます。 例えば、

```
case class Number(x: Int) extends Expr
```

この形の拡張性は、基底クラス Expr を sealed と宣言することで排除できます。; その場合、Expr を直接拡張するすべてのクラスは、Expr と同じソースファイル中になければなりません。

5.3.3 トレイト (Traits)

構文:

```
TmplDef          ::= 'trait' TraitDef
TraitDef         ::= id [TypeParamClause] TraitTemplateOpt
TraitTemplateOpt ::= 'extends' TraitTemplate | [['extends'] TemplateBody]
```

トレイトは、他のあるクラスにミックスインとして加えることを意図したクラスです。 通常のクラスと異なり、トレイトはコンストラクタパラメータを持つことはできません。 さらに、トレイトのスーパークラスにコンストラクタ引数を渡せません。 これは必要ではありません。 なぜなら、スーパークラスが初期化された後でトレイトは初期化されるからです。

トレイト D が、型 C のインスタンス x のある特徴を定義しているとします(つまり、D は C の基底クラス)。 このとき、x 中の D の**実際のスーパー型(actual supetype)**は、D を継承する LL(C)中のすべての基底クラスからなる複合型です。 実際のスーパー型は、トレイトにおける super の参照(§6.5)を解決するためのコンテキストを与えます。 実際のスーパー型は、ミックスイン合成中にトレイトが付加される型に依存することに注意してください; トレイトが定義された時点でそれを静的に知ることはできません。

もし D がトレイトでないなら、その実際のスーパー型は単に、その最小固有のスーパー型です(静的に知ることができます)。

Example 5.3.5 : 次のトレイトは、 ある型のオブジェクトと比較可能にするプロパティを定義します。 これは抽象メソッド < と、他の比較演算子 <=、>、>= のデフォルト実装を含みます。

```
trait Comparable[T <: Comparable[T]] { self: T =>
  def < (that: T): Boolean
  def <=(that: T): Boolean = this < that || this == that
  def > (that: T): Boolean = that < this
  def >=(that: T): Boolean = that <= this
}
```

Example 5.3.6 : キー A の型から値 B の型へのマップを実装する、抽象クラス Table について考えます。 このクラスは、新しい キー/値 ペアをテーブルに入れるためのメソッド set と、与えられたキーにマッチするオプション値を返すメソッド get を持ちます。 最後に、get に似たメソッド apply があり、それは、もし与えられたキーに対してテーブルが未定義なら、与えられたデフォルト値を返す点が get と異なります。

このクラスは次のように実装されます。

```
abstract class Table[A, B](defaultValue: B) {
  def get(key: A): Option[B]
  def set(key: A, value: B)
  def apply(key: A) = get(key) match {
    case Some(value) => value
    case None => defaultValue
  }
}
```

次は Table クラスの具象実装です。

```
class ListTable[A, B](defaultValue: B) extends Table[A, B](defaultValue) {
  private var elems: List[(A, B)]
  def get(key: A) = elems.find(._1.==(key)).map(._2)
  def set(key: A, value: B) = { elems = (key, value) :: elems }
}
```

次は、その親クラスの get および set 操作への並行アクセスを防ぐトレイトです。

```
trait SynchronizedTable[A, B] extends Table[A, B] {
  abstract override def get(key: A): B =
    synchronized { super.get(key) }
  abstract override def set((key: A, value: B) =
    synchronized { super.set(key, value) }
}
```

Table が形式上のパラメータを用いて定義されていても、 SynchronizedTable はそのスーパークラス Table に引数を渡さないことに注意してください。 SynchronizedTable の get と set メソッド内の super 呼び出しは、クラス Table 中の抽象メソッドを静的に参照していることにも注意してください。 呼び出すメソッドが abstract override (§5.2) と印されている限り、これは正しいです。

最終的に、次のミックスイン合成は、文字列をキーとして整数を値とする(デフォルト値 0)、同期リストテーブルを生成します。:

```
object MyTable extends ListTable[String, Int](0) with SynchronizedTable
```

オブジェクト MyTable は、SynchronizedTable からその get と set メソッドを継承します。これらメソッド内の super 呼び出しは、ListTable 中の対応する実装への参照へ再束縛され、そしてそれは MyTable 中の SynchronizedTable の実際のスーパー型です。

5.4 オブジェクト定義 (Object Definitions)

構文:

```
ObjectDef      ::=   id ClassTemplate
```

オブジェクト定義は、新しいクラスのただ 1 つのオブジェクトを定義します。 その最も一般的な形は object m extends t です。 ここで、m は定義されるオブジェクトの名前、t は次の形のテンプレート(§5.1) です。

```
sc with mt1 with ... with mtn {stats}
```

これは m の基底クラス、振る舞いと初期状態を定義します。 継承節 extends sc with mt1 with ... with mtn は省略されることがあり、その場合、scala.AnyRef が想定されます。 クラス本体 {stats} も省略されることがあり、その場合、空の本体 {} が想定されます。

オブジェクト定義はテンプレート t に適合するただ 1 つのオブジェクト(あるいは:**モジュール**)を定義します。 それは次の遅延評価値定義と、大まかに言って同じです。

```
lazy val m = new sc with mt1 with ... with mtn { this: m.type => stats }
```

オブジェクト定義によって定義された値は、遅れてインスタンス化されることに注意してください。 new m$cls コンストラクタは、オブジェクト定義の時点では評価されません。 しかし代わりに、m がプログラム実行中に最初に逆参照されるとき(そういうことは全くないかもしれない)、評価されます。 コンストラクタの評価途中で m の逆参照を再び試みると、無限ループあるいは実行時エラーを招きます。 コンストラクタ評価中に m の逆参照を試みる他のスレッドは、評価が完了するまでブロックします。

上記で与えられた拡張は、トップレベルのオブジェクトについては正確ではありません。 それはありません。 なぜなら、変数とメソッド定義はパッケージオブジェクト(§9.3)の外側のトップレベルに現われることができないからです。 その代わり、トップレベルのオブジェクトは静的なフィールドに翻訳されます。

Example 5.4.1 : Scala のクラスは静的メンバを持ちません。; しかし、オブジェクト定義を随伴させることで、同様の効果を達成できます。

```
abstract class Point {
  val x: Double
  val y: Double
  def isOrigin = (x == 0.0 && y == 0.0)
}
object Point {
  val origin = new Point() { val x = 0.0; val y = 0.0 }
}
```

これはクラス Pointと、メンバーとして origin を含むオブジェクト Point を定義します。 名前 Point の２重使用が正しいことに注意してください。 クラス定義は名前 Point を型-名前空間内で定義し、他方、オブジェクト定義は名前を項-名前空間内で定義するからです。

この方法は、静的メンバをもつ Java クラスを解釈するときに Scala コンパイラによって用いられます。 そのようなクラス C は、C のすべてのインスタンスメンバーを含む Scala クラスと、 C のすべての静的メンバーを含む Scala オブジェクトのペアとして概念的にとらえることができます。

一般に、クラスの**コンパニオンモジュール**はクラスと同じ名前をもつオブジェクトであり、同じスコープとコンパイル単位で定義されます。 逆に、そのクラスはモジュールの**コンパニオンクラス**と呼ばれます。

6 式 (Expressions)

構文:

```
Expr          ::=  (Bindings | id | '_') '=>' Expr
                 |  Expr1
Expr1         ::=  'if' '(' Expr ')' {nl} Expr [[semi] else Expr]
                 |  'while' '(' Expr ')' {nl} Expr
                 |  'try' '{' Block '}' ['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 s
                 |  SimpleExpr TypeArgs
                 |  SimpleExpr1 ArgumentExprs
                 |  XmlExpr

Exprs         ::=  Expr {',' Expr}
BlockExpr     ::=  '{' CaseClauses '}'
                 |  '{' Block '}'
Block         ::=  {BlockStat semi} [ResultExpr]
ResultExpr    ::=  Expr1
                 |  (Bindings | (id | '_') ':' CompoundType) '=>' Block
Ascription    ::=  ':' InfixType
                 |  ':' Annotation {Annotation}
                 |  ':' '_' '*'
```

式は演算子とオペランドから構成されます。 式の形については、後で、優先順位の低い順に論じます。

6.1 式の型付け (Expression Typing)

式の型付けは、何らかの(未定義かもしれない)**要請型(expected type)**にしばしば関係します。 「式 e は型 T に適合することが要請される」と書くとき、次を意味します: (1) e の要請型は T であり、(2) 式 e の型は T に適合しなくてはならない。

The following skolemization rule is applied universally for every expression: If the type of an expression would be an existential type T , then the type of the expression is assumed instead to be a skolemization (§3.2.10) of T .

Skolemization is reversed by type packing. Assume an expression e of type T and let t1[tps1] >: L1 <: U1,...,tn[tpsn] >: Ln <: Un be all the type variables created by skolemization of some part of e which are free in T . Then the packed type of e is

次の skolemization 規則は、すべての式に広く適用されます。: すなわち、もし式の型が存在型 T なら、式の型は代わりに T の skolemization(§3.2.10)であると仮定されます。

Skolemization は型パッキングによって逆にされます。 型 T の式 e があって、t1[tps1] >: L1 <: U1,...,tn[tpsn] >: Ln <: Un は、T 中で自由な e のある部分の skolemization によって生成されたすべての型変数であると仮定します。 このとき e の**パックされた型**は次です。

```
T forSome { type t1[tps1] >: L1 <: U1 ; ...; type tn[tpsn] >: Ln <: Un }
```

6.2 リテラル (Literals)

構文:

```
SimpleExpr  ::=  Literal
```

リテラル(§1.3)の型付けで記述されています。 それらは直ちに評価されます。

6.3 null 値 (The Null Value)

null値は scala.Null 型であり、従って、全ての参照型と互換です。 これは特別な "null" オブジェクトを参照する参照値を表します。 このオブジェクトは、クラス scala.AnyRef 中で次のメソッドを実装します。:

- eq(x) と == (x) は、もし引数 x も同じく "null" オブジェクトである時、 かつそのときに限り true を返します。
- ne(x) and !=(x) return true iff the argument x is not also the "null" object.
- ne(x) と !=(x) は、もし引数 x が "null" オブジェクトでない時、 かつそのときに限り true を返します。

- isInstanceOf[T] は、常に false を返します。
- asInstanceOf[T] は、もし T が scala.AnyRef に適合するなら "null" オブジェクトそれ自身を返し、そうでなければ NullPointerException を送出します。

"null" オブジェクトの他のメンバーへの参照は、すべて NullPointerException 送出を引き起こします。

6.4 指定子 (Designators)

構文:

```
SimpleExpr   ::=   Path
                 |   SimpleExpr '.' id
```

指定子は、名前付きの項(named term)を参照します。

それは**単純名**または**選択(selection)**です。 単純名 x は、 §2 中で定めた値を参照します。 もし x が取り囲むクラス/オブジェクト C 中の定義/宣言によって束縛されるなら、それは、選択 C.this.x と等価であるとみなされます。 ここで C は、たとえ型名 C が x の出現時に隠されていても(§2)、x を含むクラスを参照するとみなされます。

もし r が型 T の安定識別子(§3.1)なら、選択 r.x は、T において名前 x と同一視される、r の項メンバー m を静的に参照します。

他の式 e については、 e.x はそれが { val y = e ; y.x } (y は、ある新規の名前)であるかのように、型付けされます。

指定子の前置子の要請型は、常に未定義です。 指定子の型は、それが参照するエンティティの型 T ですが、次の例外があります: 安定型(§3.2.1)が要求されるコンテキスト中に現れるパス(§3.1) p の型は、シングルトン型 p.type です。

安定型が要求されるコンテキストは、次の条件の 1 つを満たすものです。:

1. パス p は選択の前置子として出現し、定数を指定しない。あるいは
2. 要請型 pt は安定型である。あるいは
3. 要請型 pt は下限境界に安定型をもつ抽象型で、 p によって参照されるエンティティの型 T が pt に適合しない。あるいは
4. パス p はモジュールを指定する。

選択 e.x の評価では、はじめに限定修飾している式 e を評価します。 それは例えば、オブジェクト r をもたらします。 選択の結果は、m によって定義されているか、あるいは m をオーバーライドする定義によって定義されている、r のメンバーです。 もしそのようなメンバーが scala.NotNull に適合する型を持っていれば、メンバーの値は null と異なる値に初期化されなくてはなりません。 そうでなければ scala.UnitializedError が送出されます。

6.5 this と super (This and Super)

構文:

```
SimpleExpr   ::=   [id '.'] 'this'
                 |   [id '.'] 'super' [ClassQualifier] '.' id
```

式 this は、テンプレートあるいは複合型の文部分中に現われます。 それは、参照を囲む最も内側のテンプレートあるいは複合型によって定義されているオブジェクトを表します。 もしそれが複合型なら、this の型はその複合型です。 もしそれが単純名 C をもつクラス/オブジェクト定義のテンプレートなら、this の型は C.this の型と同じです。

式 C.this は、単純名 C をもつ取り囲むクラス/オブジェクト定義の文部分内では、正しいです。 これは、最も内側のそのような定義によって定義されつつあるオブジェクトを表します。 もし、式の要請型が安定型であるか、あるいは C.this が選択の前置子として現れるなら、その型は C.this.type であり、そうでなければクラス C の自己型です。

参照 super.m は、参照を含む最も内側のテンプレートの最小固有のスーパー型中の、メソッドあるいは型 m を静的に参照します。 それは、m に等しいかあるいは m をオーバライドする、そのテンプレートの実際のスーパー型中のメンバー m′へ評価されます。 静的に参照されるメンバー m は、型あるいはメソッドでなければなりません。 もしそれがメソッドなら、それは具象であるか、あるいは、その参照を含むテンプレートが m をオーバライドする abstract override と印されたメンバー m′を持たなくてはなりません。

参照 C.super.m は、その参照を囲む、最も内側の取り囲む C と名付けられたクラス/オブジェクト定義の最小固有のスーパー型中の、メソッドあるいは型 m を静的に参照します。 それは、m に等しいかあるいは m をオーバライドする、そのクラス/オブジェクトの実際のスーパー型中のメンバー m′へ評価されます。 静的に参照されるメンバー m は、型あるいはメソッドでなければなりません。 もし静的に参照されるメンバー m がメソッドなら、それは具象であるか、あるいは、最も内側の取り囲む C と名付けられたクラス/オブジェクト定義が、m をオーバライドする abstract override と印されたメンバー m′を持たなくてはなりません。

super 前置子の後には、C.super[T].x のように、トレイト限定修飾子 [T] が続くことがあります。 これは、**静的 super 参照(static super reference)**と呼ばれます。 この場合、参照は、その単純名が T である C の、親トレイト中の型あるいはメソッド x への参照です。 そのようなメンバーはユニークに定義されていなければなりません。 もしそれがメソッドなら、具象でなければなりません。

Example 6.5.1 : 次のクラス定義について考えます。

```
class Root { def x = "Root" }
class A extends Root { override def x = "A" ; def superA = super.x }
trait B extends Root { override def x = "B" ; def superB = super.x }
class C extends Root with B {
  override def x = "C" ; def superC = super.x
}
class D extends A with B {
  override def x = "D" ; def superD = super.x
}
```

クラス C の線形化は {C, B, Root} であり、クラス D の線形化は {D, B, A, Root} です。このとき次を得ます。

```
(new A).superA == "Root",
                          (new C).superB = "Root", (new C).superC = "B",
```

```
(new D).superA == "Root", (new D).superB = "A",    (new D).superD = "B",
```

B がクラス Root あるいは A のどちらとミックスインされるかによって、 superB 関数が異なる結果を返すことに注意してください。

(訳注:この例について(他の例も含め)、訳者は確認していません)

6.6 関数適用 (Function Applications)

構文:

```
SimpleExpr    ::=  SimpleExpr1 ArgumentExprs
ArgumentExprs ::=  '(' [Exprs] ')'
                 |  '(' [Exprs ','] PostfixExpr ':' '_' '*' ')'
                 |  [nl] BlockExpr
Exprs         ::=  Expr {',' Expr}
```

適用 f(e1,...,em)は、関数 f を引数式 e1,...,em に適用します。 もし f がメソッド型 (p1 : T1,...,pn :Tn)U をもつなら、各引数式 ei の型は、対応するパラメータ型 Ti を要請型として型付けされます。 Si を引数 ei (i = 1,...,m)の型であるとします。 もし f が多相的メソッドなら、f の型引数の決定にローカルな型推論(§6.26.4) が使われます。 もし f が何らかの値型を持っていれば、その適用は f.apply(e1,...,em)、すなわち、f で定義された apply メソッドの適用とみなされます。

関数 f は、型 S1,...,Sn の引数 e1,...,en に**適用可能**でなければなりません。

もし f がメソッド型 (p1 : T1,...,pn : Tn)U を持ち、引数式 ei が xi = e′i の形 (xi はパラメータ名 p1,...,pn の 1 つ)なら、引数式 ei を**名前付き**引数と言います。 もし次の条件がすべて満たされるなら、関数 f は適用可能です:

- すべての名前付き引数 xi = e′i について、型 Si は、 xi にマッチする名前 pj のパラメータ型 Tjと互換である。
- すべての位置的な(positional)引数 ei について、型 Si は Ti と互換である。
- もし要請型が定義されているなら、結果型 U はそれに互換である。

もし f が多相的メソッドで、もしインスタンス化されたメソッドが適用可能であるようにローカルな型推論(§6.26.4)が型引数を決定できるなら、f は適用可能です。もし f が何らかの値型を持ち、 apply と名付けられた適用可能なメソッドメンバーをもつなら、 f は適用可能です。

f(e1,...,en)の評価は通常、f と e1,...,en のこの順番での評価を必要とします。 各引数式は、その対応する形式上のパラメータの型に変換されます。 そのあと、その適用は、形式上のパラメータの代わりに実際の引数を用いて、関数の右側へ書き直されます。 書き直された右側の評価結果は、最終的に関数の宣言された結果型へ、もし与えられていれば、変換されます。

パラメータなしのメソッド型 =>T をもつ形式上のパラメータの場合は、特別に扱われます。 この場合、対応する実際の引数式 e は適用の前には評価されません。 代わりに、形式上のパラメータを使用する度に、書き直し規則の右側で e の再評価が必要となります。 言い替えれば、=>-パラメータ の評価規則は**名前呼出し**であり、他方、標準的なパラメータの評価規則は**値呼出し**です。 さらに、e のパックされた型(§6.1)は、パラメータ型 T に適合することが要求されます。
名前呼び出しパラメータの動作は、もし適用が名前付き/デフォルト引数に起因してブロックに

変換されるなら、維持されます。 この場合、パラメータのローカルな値は val yi = () => e の形であり、関数に渡される引数は yi() です。

適用における最後の引数は、シーケンス引数と印されているかもしれません。 たとえば e : _* です。 そのような引数は、型 S* の反復パラメータ(§4.6.2) に対応していなければならず、そしてこのパラメータと一致する唯一の引数でなければなりません (すなわち、形式上のパラメータ数と実際の引数の数が同じでなければなりません)。 さらに、e の型は、S に適合する何らかの型 T があって、scala.Seq[T] に適合しなくてはなりません。 この場合、引数リストは、シーケンス e をその要素で置き換えて変換されます。 適用が名前付き引数を使うときは、可変引数 (vararg)パラメータは正確にただ 1 度だけ指定されなければなりません。

関数適用は通常、プログラムのランタイムスタック上に新しいフレームを割り当てます。 しかし、もしローカル関数あるいは final メソッドがそれ自身を最後の動作として呼び出すなら、その呼び出しは呼び出す側のスタックフレームを使って実行されます。

Example 6.6.1 : 引数の変数の合計を数える次の関数を仮定します:

```
def sum(xs: Int*) = (0 /: xs) ((x, y) => x + y)
```

このとき

```
sum(1, 2, 3, 4)
sum(List(1, 2, 3, 4): _*)
```

の両方とも、10 という結果をもたらします。

他方、

```
sum(List(1, 2, 3, 4))
```

は、型チェックを通らないでしょう。

6.6.1 名前付き引数とデフォルト引数 (Named and Default Arguments)

もし適用が、名前付き引数 p = e あるいはデフォルト引数を使うなら、次の条件を満たさなくてはなりません。

- 名前付き引数が引数リスト e1,...,em の接尾部を形成する、 つまり、いかなる位置的な引数も名前付き引数の後には続かない。
- すべての名前付き引数の名前 xi は対として異なり、いかなる名前付き引数も、 位置的引数によって既に指定されたパラメータを定義しない。
- 位置的あるいは名前付き引数のいずれにも指定されていない、 すべての形式上のパラメータ pj : Tj は、デフォルト引数を持つ。

もし適用が名前付き/デフォルト引数を使うなら、次の変形が、それを名前付き/デフォルト引数のない適用へ変えます。 もし関数 f が p.m[targs] の形なら、それは次のようなブロックに変換されます。

```
{ val q = p
  q.m[targs]
}
```

もし関数 f がそれ自身適用式なら、変形は f に再帰的に適用されます。

f の変形結果は、次の形のブロックです

```
{ val q = p
  val x1 = expr1
    ...
    val xk = exprk
    q.m[targs](args1),..., (argsl)
}
```

ここで、(args1),..., (argsl)中の各引数は、値 x1,...,xk の 1 つへの参照です。 現在の適用をブロックへまとめるために、 はじめに、e1,...,em 中の各引数に対して新規の名前 yi を使った値定義が生成されます。 位置的引数は ei へ初期化され、xi = e´i の形の名前付き引数は e´i へ初期化されます。 つぎに、引数リストで指定されてないすべてのパラメータについては、新規の名前 zi を使った値定義が生成されます。 それは、このパラメータのデフォルト引数を計算するメソッドを使って初期化されます(§4.6)。

args を生成された名前 yi と zi の置換(permutation)で、各名前の位置がメソッド型 (p1 : T1,...,pn : Tn)U 中の対応するパラメータ位置にマッチするものとします。 変形の最終結果は、次の形のブロックです。

```
{ val q = p
  val x1 = expr1
  ...
  val xl = exprk
  val y1 = e1
  ...
  val ym = em
  val z1 = q.m$default$i[targs](args1),..., (argsl)
  ...
  val zd = q.m$default$j[targs](args1),..., (argsl)
  q.m[targs](args1),..., (argsl)(args)
}
```

6.7 メソッド値 (Method Values)

構文:

```
SimpleExpr        ::=   SimpleExpr1 '_'
```

もし e がメソッド型であるかあるいは、もし e が名前呼び出しパラメータなら、式 e _ は正しい形です。 もし e がパラメータをもつメソッドなら、e _ は、イータ展開 (§6.26.5)によって関数型に変換された e を表します。 もし e がパラメータなしのメソッドあるいは、型= > T

の名前呼び出しパラメータなら、e _ は型() => T の関数を表し、それが空きのパラメータリスト () に適用されたときに e を評価します。

Example 6.7.1 : 次の左カラム中のメソッド値は、それぞれ右の無名関数(§6.23)に等価です。

```
Math.sin _                x =>Math.sin(x)
Array.range _             (x1,x2) => Array.range(x1, x2)
List.map2 _               (x1,x2) => (x3) => List.map2(x1, x2)(x3)
List.map2(xs, ys)_        x =>List.map2(xs, ys)(x)
```

メソッド名とお終いの下線の間に空白が必要であることに注意してくださいなぜなら、そうでなければ下線は名前の一部とみなされるからです。

6.8 型適用 (Type Applications)

構文:

```
SimpleExpr        ::=    SimpleExpr TypeArgs
```

型適用 e[T1,...,Tn] は、引数型 T1,...,Tn をもつ型 [a1 >: L1 <: U1,...,an >: Ln <: Un]S の多相的な値 e をインスタンス化します。 すべての引数型 Ti は、対応する上下限境界 Li と Ui に従わなくてはなりません。 すなわち、各 i = 1,...,n に対し σLi <: Ti <: σUi でなければなりません。 ここで σは置換 [a1: = T1,...,an:= Tn] です。 適用の型は σS です。

もし関数部 e が何らかの値型なら、型適用は e.apply[T1,...,Tn] と同じとされます。 すなわち、e によって定義された apply メソッドの適用です。

多相的な関数に対し、もしローカルな型推論(§6.26.4)が実際の関数引数の型と要請される結果型から最も良い型パラメータを推論できる場合は、型適用は省略できます。

6.9 タプル (Tuples)

構文:

```
SimpleExpr    ::=       '(' [Exprs] ')'
```

タプル式 (e1,...,en) は、クラスインスタンス生成 scala.Tuplen(e1,...,en)のエイリアスです。 ここで、n >= 2 。 空タプル () は、型 scala.Unit のただ 1 つの値です。

6.10 インスタンス生成式 (Instance Creation Expressions)

構文:

```
SimpleExpr        ::=     'new' (ClassTemplate | TemplateBody)
```

単純なインスタンス生成式は new c の形をしています。 ここで、c はコンストラクタ呼び出し(constructor invocation §5.1.1)です。 c の型を T とします。 このとき T は、scala.AnyRef の非抽象サブクラス(の型インスタンス)を表さなければなりません。 さらに、式の**具象自己型**は、T によって表されるクラスの自己型に適合しなくてはなりません(§5.1)。 具象自己型は通常 T です。 ただし、式 new c が値定義の右辺として現われる場合を除きます。

```
val x : S = new c
```

(ここで、型アノテーション :S はないかもしれません) 後者の場合、式の具象自己型は複合型 T with x.type です。

式は、型 T の新しいオブジェクトを生成することで評価され、そのオブジェクトは c を評価することで初期化されます。 式の型は T です。

一般的なインスタンス生成式は、あるクラステンプレート t (§5.1)については、new t の形をしています。 そのような式はブロック

```
{ class a extends t ; new a }
```

に等価です。 ここで a は、ユーザープログラムはアクセスできない**無名クラス**の新規の名前です。

構造型(structural types)の値を生成するための略記表現もあります。 もし {D} がクラス本体なら、new {D} は一般的なインスタンス生成式 new AnyRef {D} と同じです。

Example 6.10.1 : 次の構造的なインスタンス生成式について考えます。:

```
new { def getName() = "aaron" }
```

これは次の一般的なインスタンス生成式の略記表現です。

```
new AnyRef{ def getName() = "aaron" }
```

後者は、今度は次のブロックの略記表現です。

```
{ class anon$X extends AnyRef{ def getName() = "aaron" }; new anon$X }
```

ここで anon$X は、ある新規に生成された名前です。

6.11 ブロック（Blocks）

構文:

```
BlockExpr   ::=   '{' Block '}'
Block       ::=   {BlockStat semi} [ResultExpr]
```

ブロック式 {s1 ; ...; sn ; e } は、ブロック文 s1,...,sn の並びと最後の式 e から構成されます。 文並びは同じ名前空間中で同じ名前を束縛する 2 つの定義/宣言を含んではなりません。 最後の式は省略されることがあり、その場合、unit 値()が想定されます。

最後の式 e の要請型は、ブロックの要請型です。 先行するすべての文の要請型は未定義です。

The type of a block s1 ; ...; sn ; e is T forSome { Q }, where T is the type of e and Q contains existential clauses (§3.2.10) for every value or type name which is free in T and which is defined locally in one of the statements s1,...,sn . We say the existential clause binds the occurrence of the value or type name. Specifically,

ブロック s1 ; ...; sn ; e の型は T forSome { Q } です。 ここで T は e の型であり、Q は、T 中で自由でかつ文 s1,...,sn の 1 つでローカルに定義された、すべての値あるいは型名の存在節(§3.2.10) を含みます。我々は、 存在節が値あるいは型名の存在(occurrence)を**束縛**すると言います。 特に、

- ローカルに定義された型定義 type t = T は、 存在節 type t >: T <: T によって束縛されます。 もし t が型パラメータをもつなら、エラーです。
- ローカルに定義された値定義 val x : T = e は、 存在節 val x : T によって束縛されます。
- ローカルに定義されたクラス定義 class c extends t は、 存在節 type c <: T によって束縛されます。 ここで T は、型 c の固有のスーパー型である、 最小のクラス型あるいは細別型です。 もし c が型パラメータをもつなら、エラーです。
- ローカルに定義されたオブジェクト定義 object x extends t は、 存在節 val x : T によって束縛されます。 ここで T は、型 x.type の固有のスーパー型である、 最小のクラス型あるいは細別型です。

ブロックの評価は、その文並びの評価と、後に、最後の式 e の評価を必要とし、式 e の評価はブロックの結果を定義します。

Example 6.11.1 : クラス Ref[T](x: T)を仮定します。ブロック

```
{ class C extends B {...} ; new Ref(new C) }
```

は、型 Ref[_1] forSome { type _1 <: B } を持ちます。 ブロック

```
{ class C extends B {...} ; new C }
```

は、単なる 型 B をもちます。 なぜなら、(§3.2.10) 中の規則を用いて、存在量化された型 _1 forSome { type _1 <: B} は B に簡略化できるからです。

6.12 前置、中置、後置演算 (Prefix, Infix, and Postfix Operations)

構文:

```
PostfixExpr    ::=    InfixExpr [id [nl]]
InfixExpr      ::=    PrefixExpr
                  |   InfixExpr id [nl] InfixExpr
PrefixExpr     ::=    ['-' | '+' | '!' | '~'] SimpleExpr
```

式は、オペランドと演算子から構築できます。

6.12.1 前置演算 (Prefix Operations)

前置演算 `op e` は前置子オペレータ `op` からなり、それは識別子 '+'、'-'、'!'、'~' の 1 つでなければなりません。 式 `op e` は、後置メソッド適用 `e.unary_op` に等価です。

前置演算子は、それらのオペランド式がアトミックである必要がない点で、通常の関数適用と異なります。 例えば、入力シーケンス `-sin(x)` は、`-(sin(x))` として読まれ、一方、関数適用 `negate sin(x)` は、中置演算子 `sin` の、オペランド `negate`(反転) と (x) への適用として構文解析されます。

6.12.2 後置演算 (Postfix Operations)

後置演算子は任意の識別子です。 後置演算 `e op` は `e.op` と解釈されます。

6.12.3 中置演算 (Infix Operations)

中置演算子は任意の識別子です。 中置演算子は次のように定義された優先順位と結合性をもちます。:

中置演算子の**優先順位**は演算子の最初の文字で決まります。 次にリストした文字は、優先順位の増加順で、同じ行の文字は同じ優先順位です。

```
(すべての文字(letters))
|
^
&
< >
= !
:
```

```
+ -
* / %
(他のすべての特殊文字(special characters))
```

すなわち、文字で始まる演算子が最も低い優先順位を持ち、次に '|'、その他が続きます。

この規則には 1 つの例外があり、**代入演算子**(§6.12.4)に関係しています。 代入演算子の優先順位は、単純な代入(=)のそれと同じです。 すなわち、それは他のすべての演算子の優先順位よりも低いです。

演算子の**結合性**は演算子の最後の文字によって決まります。 コロン ':' で終わるオペレータは右結合です。 他のすべての演算子は左結合です。

演算子の優先順位と結合性は、次のように式の一部をグループ化します。

- もし１つの式の中に複数の中置演算があるなら、より高い優先順位の演算子が、 より低い優先順位の演算子よりもより強く結びつきます。
- もし、同じ優先順位の演算子 op1,...,opn をもつ連続した中置演算 e0 op1 e1 op2 ... opn en があるなら、 すべての演算子は同じ結合性を持たなくてはなりません。 もしすべての演算子が左結合なら、その連続は、 (...(e0 op1 e1)op2 ...) opn en と解釈されます。 そうでなく、もしすべての演算子が右結合なら、その連続は、 e0 op1(e1 op2(... opn en)...)と解釈されます。
- 後置演算子は常に、中置演算子より低い優先順位を持ちます。 たとえば、e1 op1 e2 op2 は常に、(e1 op1 e2)op2 に等価です。

左結合演算子の右オペランドは、丸括弧で囲まれる複数の引数からできていても構いません。 たとえば、e op (e1,...,en)。 この式は e.op(e1,...,en)と解釈されます。

左結合バイナリ演算 e1 op e2 は、e1.op(e2) と解釈されます。 もし op が 右結合なら、同じオペレーションは { val x =e1 ; e2.op(x) } と解釈されます。 ここで x は新規の名前です。

6.12.4 代入演算子 (Assignment Operators)

代入演算子は、末尾が等号文字「=」になっている演算子シンボル((§1.1)中の構文カテゴリ op)であり、次の条件の 1 つを満たす、演算子の例外があります。:

(1) 演算子の始まりも等号文字である。あるいは、

(2) 演算子が (<=)、(>=)、(!=)のうちの１つ。

代入演算子は、もし他のどのような解釈も有効でないなら、それらは代入に拡張されるという点で、特別に扱われます。

中置演算 l += r 中の += のような代入演算子について考えてみましょう。 ここで l、r は式です。 このオペレーションは、代入に対応するオペレーション

```
l = l + r
```

として、オペレーションの左辺 l がただ 1 度だけ評価されることを除き、再解釈されます。

もし次の 2 つの条件がフルに満たされるなら、再解釈が起きます。

1. 左辺 l が += という名前のメンバーを持たず、暗黙の変換 (§6.26) によって += という名前のメンバーを用いて値へ変換できない。

2. 代入 l = l + r が型的に正しい。 特に、このことは l が、それへの代入が可能な変数/オブジェクトを参照し、+ という名前のメンバーを用いて値に変換できることを意味します。

6.13 型付けされた式（Typed Expressions）

構文:

```
Expr1          ::=    PostfixExpr ':' CompoundType
```

型付けされた式 e : T は型 T を持ちます。 式 e の型は T に適合することが要請されます。式の結果は、型 T に変換された e の値です。

Example 6.13.1 : 次は、正しく型付けされた式と、不正に型付けされた式の例です。

```
1: Int                    // 正しい、Int 型
1: Long                   // 正しい、Long 型
// 1: string              // ***** 不正
```

6.14 アノテーション(注釈)付きの式（Annotated Expressions）

構文:

```
Expr1            ::=    PostfixExpr ':' Annotation {Annotation}
```

アノテーション付きの式 e : @a1 ...@an は、アノテーション a1,...,an を式 e へ加えます(§11)。

6.15 代入（Assignments）

構文:

```
Expr1         ::=   [SimpleExpr '.'] id '=' Expr
                |   SimpleExpr1 ArgumentExprs '=' Expr
```

単純変数への代入 x = e の解釈は、x の定義に依存します。 もし x がミュータブル変数を表すなら、代入は x の現在の値を式 e の評価結果に変えます。 e の型は、x の型に適合することが要請されます。 もし x があるテンプレート中で定義されたパラメータなしの関数で、同じテンプレートがメンバーとしてセッター関数 x_= を含むなら、代入 x = e はそのセッター関数の呼び出し x_=(e)と解釈されます。 同様に、パラメータなしの関数 x への代入 f.x = e は、呼び出し f.x_=(e) と解釈されます。

'=' 演算子の左辺への関数適用をもつ代入 f(args) = e は、f.update(args,e)、すなわち、f で定義された update 関数の呼び出しと解釈されます。

Example 6.15.1 : 次は、行列乗算の通常の命令型コードです。

```
def matmul(xss: Array[Array[Double]], yss: Array[Array[Double]]) = {
  val zss: Array[Array[Double]] = new Array(xss.length, yss(0).length)
  var i = 0
  while (i < xss.length) {
    var j = 0
    while (j < yss(0).length) {
      var acc = 0.0
      var k = 0
      while (k < yss.length) {
        acc = acc + xss(i)(k) * yss(k)(j)
        k += 1
      }
      zss(i)(j) = acc
      j += 1
    }
    i += 1
  }
  zss
}
```

配列のアクセスと代入の糖衣を取り除くと、次の展開バージョンを得ます。:

```
def matmul(xss: Array[Array[Double]], yss: Array[Array[Double]]) = {
  val zss: Array[Array[Double]] = new Array(xss.length, yss.apply(0).length)
  var i = 0
  while (i < xss.length) {
    var j = 0
    while (j < yss.apply(0).length) {
      var acc = 0.0
      var k = 0
      while (k < yss.length) {
        acc = acc + xss.apply(i).apply(k) * yss.apply(k).apply(j)
        k += 1
      }
      zss.apply(i).update(j, acc)
      j += 1
    }
    i += 1
  }
```

```
  zss
}
```

6.16 条件式 (Conditional Expressions)

構文:

```
Expr1          ::=   'if' '(' Expr ')' {nl} Expr [[semi] 'else' Expr]
```

条件式 if (e1) e2 else e3 は、e1 の値に応じて、e2 と e3 の値の 1 つを選択します。 条件 e1 は Boolean 型に適合することが要請されます。 then 部分 e2 と else 部分 e3 は共に、条件式の要請型に適合することが要請されます。 条件式の型は、e2 と e3 の型の、弱い最少の上限境界(§3.5.3)です。 条件式の else シンボルの手前のセミコロンは無視されます。

条件式は、はじめの e1 の評価に基づいて評価されます。 もしこの評価が true なら、e2 の評価が返されます。 そうでなければ、e3 評価結果が返されます。

条件式の短縮形として else 部分がない場合があります。 条件式 if (e1) e2 は、それが if (e1) e2 else () であるかのように評価されます。

6.17 while ループ式 (While Loop Expressions)

構文:

```
Expr1          ::=   'while' '(' Expr ')' {nl} Expr
```

while ループ式 while (e1) e2 は、whileLoop (e1) (e2)の適用であるかのように型付けされ評価されます。 ただしここで、仮想関数 whileLoop は次のように定義されます。

```
def whileLoop(cond: => Boolean)(body: => Unit): Unit =
  if (cond) { body ; whileLoop(cond)(body) } else {}
```

6.18 do ループ式 (Do Loop Expressions)

構文:

```
Expr1          ::=   'do' Expr [semi] 'while' '(' Expr ')'
```

do ループ式 do e1 while (e2)は、式 (e1 ; while (e2) e1)であるかのように型付けされ評価されます。 do ループ式の while シンボルの前のセミコロンは無視されます。

6.19 for 内包表記と for ループ (For Comprehensions and For Loops)

構文:

```
Expr1          ::=   'for' ('(' Enumerators ')' | '{' Enumerators '}')
                       {nl} ['yield'] Expr
Enumerators    ::=   Generator {semi Enumerator}
Enumerator     ::=   Generator
                   |   Guard
                   |   'val' Pattern1 '=' Expr
Generator      ::=   Pattern1 '<-' Expr [Guard]
Guard          ::=   'if' PostfixExpr
```

for ループ for (enums) e は、列挙子 enums によって生成された各束縛に対して式 e を実行します。 for 内包表記 for (enums) yield e は、列挙子 enums によって生成された各束縛に対して式 e を評価し、結果を集めます。 列挙子シーケンスは常に生成子から始まります。; その後に、さらなる生成子、値定義、あるいはガードを続けることができます。 **生成子** p <- e は、パターン p に対して何らかの方法でマッチした式 e から、束縛を生み出します。 **値定義** p = e は、値名 p(あるいは、パターン p 中のいくつかの名前)を式 e の評価結果へ束縛します。 **ガード** if e は、列挙された束縛を限定する boolean 式を含みます。 生成子とガードの正確な意味は、4 つのメソッド map、withFilter、flatMap、foreach 呼び出しへの変換によって定義されます: これらのメソッドは、異なるタイプに応じて異なる方法で実装されます。

変換方式は次の通りです。 最初のステップでは、全ての生成子 p <- e は次で置き換えられます (ここで p は e の型について明白(irrefutable)ではないとします(§8.1))。

```
p <- e.withFilter { case p => true; case _ => false }
```

次に、すべての内包表記がなくなるまで、以下の規則が繰り返し適用されます。

- for 内包表記 for (p <- e) yield e′は e.map { case p => e′ } へ変換されます。
- for ループ for (p <- e) e′は e.foreach { case p => e′ } へ変換されます。
- for 内包表記

  ```
  for (p <- e ; p′ <- e′ ...) yield e′′
  ```

  (ここで ... は、生成子、定義、あるいはガードの(空きでもよい)並び)
  は、次へ変換されます。

  ```
  e.flatMap { case p => for (p′ <- e′ ...) yield e′′ }
  ```

```
for (p <- e ; p′ <- e′ ...) e′′
```

(ここで ... は、生成子、定義、あるいはガードの(空きでもよい)並び)
は、次へ変換されます。

```
e.foreach { case p => for (p′ <- e′ ...) e′′ }
```

- ガード if g が後に続く生成子 p <- e は、ただ1つの生成子 p <- e.withFilter((x1,..., xn) => g)へ変換されます。 ここで x1,...,xn は p の自由変数です。
- 値定義 p′ = e′が後に続く生成子 p <- e は、次の、値の対の生成子へ変換されます。 ここで x と x′ は新規の名前です:

```
(p , p′) <- for (x@p <- e) yield { val x′@p′ = e′ ; (x , x′) }
```

Example 6.19.1 : 次のコードは、その和が素数である、 1 以上 n - 1 以下の数の全ての対を作り出します。

```
for  { i <- 1 until n
       j <- 1 until i
       if isPrime(i+j)
} yield (i, j)
```

for 内包表記は次へ変換されます。

```
(1 until n)
  .flatMap {
     case i => (1 until i)
       .withFilter { j => isPrime(i+j) }
       .map { case j => (i, j) } }
```

Example 6.19.2 : for 内包表記は、ベクトルと行列アルゴリズムを簡潔に表現するのに役立ちます。 例えば、次は与えられた行列の転置を計算する関数です。:

```
def transpose[A](xss: Array[Array[A]]) = {
  for (i <- Array.range(0, xss(0).length)) yield
    for (xs <- xss) yield xs(i)
}
```

次は、2 つのベクトルのスカラ積を計算する関数です。:

```
def scalprod(xs: Array[Double], ys: Array[Double]) = {
  var acc = 0.0
  for ((x, y) <- xs zip ys) acc = acc + x * y
  acc
}
```

最後に、次は 2 つの行列の積を計算する関数です。 Example 6.15.1 の命令型バージョンと比較してみてください。

```
def matmul(xss: Array[Array[Double]], yss: Array[Array[Double]]) = {
  val ysst = transpose(yss)
  for (xs <- xss) yield
    for (yst <- ysst) yield
      scalprod(xs, yst)
}
```

上記のコードは、map、flatMap、withFilter、foreach がクラス scala.Array のインスタンスに対して定義されている、という事実を利用しています。

6.20 return 式 (Return Expressions)

構文:

```
Expr1       ::=   'return' [Expr]
```

return 式 return e は、取り囲む名前付きメソッド/関数本体の内側に現れなくてはなりません。 ソースプログラム中の、取り囲む最内の名前付きメソッド/関数 f は、明示的に宣言された結果型を持たなければならず、e の型はそれに適合しなくてはなりません。 return 式は式 e を評価し、その値を f の結果として返します。 return 式の後に続く、どのような文あるいは式の評価も除かれます。return 式の型は scala.Nothing です。

式 e は省略されるかもしれません。 return 式 return は、型チェックされ、return ()であるかのように評価されます。

無名関数の展開としてコンパイラが生成する apply メソッドは、ソースプログラム中の名前付き関数としては扱われず、従って、return 式のターゲットには決してなりません。

Returning from a nested anonymous function is implemented by throwing and catching a scala.runtime.NonLocalReturnException . Any exception catches between the point of return and the enclosing methods might see the exception . A key comparison makes sure that these exceptions are only caught by the method instance which is terminated by the return .

ネストした無名関数から戻ることは、scala.runtime.NonLocalReturnException の送出と捕捉として実装されます。 return の場所と取り囲むメソッド間のどのような例外捕捉も、例外となるでしょう。 キーを比較すれば、これらの例外が return によって終了するメソッドインスタンスによってのみ捕捉できる、ということを確かめることができます。

If the return expression is itself part of an anonymous function, it is possible that the enclosing instance of f has already returned before the return expression is executed . In that case, the thrown scala.runtime.NonLocalReturnException will not be caught, and will propagate up the call stack .

もし return 式がそれ自身無名関数の一部なら、f の取り囲むインスタンスは return 式が実行される前にさっさと戻ることが可能です。 そのような場合、送出された scala.runtime.NonLocalReturnException は捕えられません。 そして、呼び出しスタックは増殖し続けます。

6.21 throw 式 (Throw Expressions)

構文:

```
Expr1       ::=   'throw' Expr
```

throw 式 throw e は、式 e を評価します。 この式の型は Throwable に適合しなくてはなりません。 もし e が例外参照へ評価されるなら、評価は送出される例外でアボートされます。 もし e が null へ評価されるなら、評価は代わりに、NullPointerException でアボートされます。 もし送出された例外を処理するアクティブな try 式(§6.22)があるなら、評価はハンドラでレジュームします。; そうでなければ、throw を実行するスレッドはアボートされます。 throw 式の型は scala.Nothing です。

6.22 try 式 (Try Expressions)

構文:

```
Expr1     ::=  'try' '{' Block '}' ['catch' '{' CaseClauses '}']
               ['finally' Expr]
```

try 式は try { b } catch h の形です。 ここで、ハンドラ h は次のパターンマッチング無名関数です(§8.5)。

```
{ case p1 => b1 ... case pn => bn } .
```

この式は、ブロック b の評価に基づいて評価されます。 もし b の評価が例外送出を引き起こさなければ、b の結果が返されます。 そうでなければ、ハンドラ h が送出された例外に適用されます。 もしハンドラが送出された例外にマッチするケースを含むなら、そのような最初のケースが呼び出されます。 もしハンドラが、送出された例外に対応するケースを含まないなら、その例外は再送出されます。

pt を try 式の要請型であるとします。 ブロック b は pt に適合することが要請されます。 ハンドラ h は、型 scala.PartialFunction[scala.Throwable,pt] に適合することが要請されます。 try 式の型は、b の型と h の結果型の弱い最少の上限境界(§3.5.3)です。

try 式 try { b } finally e は、ブロック b を評価します。 もし b の評価が例外送出を引き起こさないなら、式 e が評価されます。 もし e の評価中に例外が送出されるなら、try 式の評価は送出される例外でアボートされます。 もし e の評価中に例外が送出されないなら、b の結果が try 式の結果として返されます。

もし b の評価中に例外が送出されるなら、finally ブロック e も評価されます。 もし e の評価中に他の例外 e が送出されたなら、try 式の評価は送出される例外でアボートされます。 もし e の評価中に例外が送出されないなら、b 中で送出されたオリジナルの例外が、e の評価完了後すぐに再送出されます。 ブロック b は、try 式の要請型に適合することが要請されます。 finally 式 e は、型 Unit に適合することが要請されます。

try 式 try { b } catch e1 finally e2 は、 try { try { b } catch e1 } finally e2 の略記表現です。

6.23 無名関数 (Anonymous Functions)

構文:

```
Expr         ::=  (Bindings | ['implicit'] id | '_') '=>' Expr
ResultExpr   ::=  (Bindings | (id | '_') ':' CompoundType) '=>' Block
Bindings     ::=  '(' Binding {',' Binding} ')'
Binding      ::=  (id | '_') [':' Type]
```

無名関数 (x1 : T1,...,xn:Tn) => e は、型 Ti のパラメータ xi を式 e によって与えられる結果へマップします。 各形式上のパラメータ xi のスコープは e です。 形式上のパラメータは、対として異なる名前でなくてはなりません。

もし無名関数の要請型が scala.Functionn[S1,...,Sn,R] の形なら、e の要請型は R であり、そしていずれのパラメータ xi の型 Ti も省略でき、 その場合、Ti = Si が想定されます。 もし無名関数の要請型が他のある型なら、すべての形式上のパラメータ型は明示的に与えられなければならず、e の要請型は未定義となります。 無名関数の型は scala.Functionn[S1,...,Sn,T] であり、ここで T は e のパックされた型(§6.1)です。 T は形式上のパラメータ xi のいずれをも参照しない型に等価でなければなりません。

無名関数は、次のインスタンス生成式として評価されます。

```
new scala.Functionn[T1,...,Tn , T] {
  def apply(x1 : T1,...,xn : Tn): T = e
}
```

ただ 1 つの 型付けされていない形式上のパラメータについては、(x) => e を x => e と短く書けます。 もしただ 1 つの型付けされたパラメータをもつ無名関数 (x : T) => e が、ブロックの結果式として現われるなら、これを x : T => e と短く書けます。

形式上のパラメータは、下線 _ で表されるワイルドカードでも構いません。 そのような場合、パラメータのために新規の名前が勝手に選択されます。

無名関数の名前付きパラメータに、オプションとして implicit 修飾子が先行するかもしれません。 そのような場合、パラメータは implicit と印されます(§7)。; しかし、パラメータ部それ自身は、(§7.2)の意味での暗黙のパラメータ部としては扱われません。 ですから、無名関数への引数は常に明示的に与えられなければなりません。

Example 6.23.1 無名関数の例:

```
x => x                          // 同じ値の関数 (The identity function)

f => g => x => f(g(x))          // カリー化された関数合成
                                // Curried function composition

(x: Int,y: Int) => x + y        // 和の関数

() => { count += 1; count }     //  空きのパラメータリスト ()をとり
                                //  非ローカルな変数 'count'を
                                //  インクリメントし、新しい値を返す。

_ => 5                          // 引数を無視する関数。常に 5 を返す。
                                //
```

無名関数のためのプレースホルダー構文(placeholder syntax for anonymous functions)

構文:

```
SimpleExpr1     ::=   '_'
```

式(構文カテゴリ Expr)には、識別子があってよい場所に、下線シンボル _ が埋め込まれているかもしれません。 そのような式は、それに続く下線の出現が一連のパラメータを表す、無名関数を表します。

Define an underscore section to be an expression of the form _:T , where T is a type, or else of the form _, provided the underscore does not appear as the expression part of a type ascription _:T .

下線部を _:T の形（ここで T は型)、あるいはそうでなければ _ の形の式で定義してください。 ただしここで、下線は型帰属(type ascription) _:T の式部分として現われるのではないとします。

もし次の 2 つの条件が満たされるなら、構文カテゴリ Expr の式 e は下線部 u を**束縛します**。: (1) e が適切に u を含む。 そして (2) e に適切に含まれ、それ自身適切に u を含む構文カテゴリ Expr の式が他にない。

もし式 e が、下線部 u1,...,un をこの順で束縛するなら、それは無名関数(u´1、...u´n) => e´ に等価です。 ここで、各 u´i は ui の下線部を新規の識別子で置き換えたもの、e´ は e の各下線部 ui を u´i で置き換えたものとします。

Example 6.23.2左カラム中の無名関数は、プレースホルダー構文を使っています。 それぞれ、右側の無名関数に等価です。

```
_ + 1                  x => x + 1
_ * _                  (x1, x2) => x1 * x2
(_: Int) * 2           (x: Int) => (x: Int) * 2
if (_) x else y        z => if (z) x else y
_.map(f)               x => x.map(f)
_.map(_ + 1)           x => x.map(y => y + 1)
```

6.24 定数式 (Constant Expressions)

定数式は Scala コンパイラが定数へ評価できる式です。 「定数式」の定義はプラットフォームに依存します。 しかし、少なくとも次の形の式があります。:

- 値クラスのリテラル。 例としては 1 つの整数。
- 文字列リテラル。
- Predef.classOf (§12.5) で構築されるクラス。
- 基盤となっているプラットフォームからの列挙要素。
- Array(c1,...,cn) の形のリテラル配列。 ここで ci はすべてそれ自身が定数式。
- 定数値定義(§4.1) によって定義された識別子。

6.25 文 (Statements)

構文:

```
BlockStat    ::=        Import
                |       {Annotation} ['implicit'] Def
                |       {Annotation} {LocalModifier} TmplDef
                |       Expr1
                |
TemplateStat ::=        Import
                |       {Annotation} {Modifier} Def
                |       {Annotation} {Modifier} Dcl
                |       Expr
                |
```

文はブロックおよびテンプレートの一部として現れます。 文はインポート、定義あるいは式、あるいは空でも構いません。 クラス定義のテンプレート中で使われる文は、宣言であることもあります。 文として使われる式は、任意の値型を持てます。 式の文 e は、e を評価し評価結果を捨てることで、評価されます。

ブロックの文は、ブロック内のローカルな名前を束縛する定義でも構いません。 ブロックにローカルなあらゆる定義中で許される唯一の修飾子は、implict です。 クラス/オブジェクト定義に前置するものとしては、修飾子 abstract、final と sealed も許されます。

文並びの評価は、それらが書かれた順で評価されます。

6.26 暗黙の変換 (Implicit Conversions)

Implicit conversions can be applied to expressions whose type does not match their exp ected type, as well as to unapplied methods . The available implicit conversions are g iven in the next two sub-sections .

暗黙の変換は、その型が要請型と一致しない式に、(引数リストに)適用されていないメソッドにも同様に、適用されます。 利用可能な暗黙の変換については、以降の 2 つの副節で述べます。

もし 型 T がイータ展開(§6.26.5)とビュー適用(§7.3)の後で型 U に適合するなら、型 T は型 U に**互換(compatible)**であると言います。

6.26.1 値変換 (Value Conversions)

ある値型 T をもち、ある要請型 pt で型チェックされる式 e に対して、次の 5 つの暗黙の変換が適用されます。

オーバーロード解決 (overloading resolution)

もし式がクラスの複数の可能なメンバーを表すなら、ただ 1 つのメンバーを選ぶために、オーバーロード解決(§6.26.3) が適用されます。

型インスタンス化（Type Instantiation）

式 e は次の多相型で

```
[a1 >: L1 <: U1,...,an >: Ln <: Un]T
```

which does not appear as the function part of a type application is converted to a type instance of T by determining with local type inference (§6.26.4) instance types T1,...,Tn for the type variables a1,...,an and implicitly embedding e in the type application e[T1,...,Tn] (§6.8) .

型適用の関数部分としては現れないとします。 式 e は T のインスタンスの型へ変換されますが、それは、ローカルな型推論(§6.26.4)を用いて型変数 a1,...,an のインスタンス型 T1,...,Tn を決定し、e を型適用 e[T1,...,Tn] 中に 暗黙のうちに埋め込む(§6.8)ことでなされます。

数の拡張（Numeric Widening）

もし e が、要請型に弱く適合(§3.5.3)するプリミティブな数値型なら、`§12.2.1`_ で定義された数値変換メソッド toShort、toChar、toInt、toLong、toFloat、toDouble の 1 つを使って要請型に拡張されます。

数値リテラルの縮小（Numeric Literal Narrowing）

もし要請型が Byte、Short あるいは Char であり、式 e がそれらの型の範囲の整数リテラルなら、その型の同じリテラルに変換されます。

値の廃棄（Value Discarding）

もし e がある値型で要請型が Unitなら、e は、それを項 { e ; () } の中へ埋め込むことで要請型に変換されます。

ビュー適用（View Application）

もし前記変換のいずれも適用されず、e の型が要請型 pt に適合しないなら、ビュー(§7.3)を使って e を要請型に変換することが試みられます。

6.26.2 メソッド変換（Method Conversions）

つぎの 4 つの暗黙の変換が、ある引数リストに適用されないメソッドに適用されます。

評価（Evaluation）

パラメータなしの、型 => T のメソッド m は常に、m が束縛される式へ評価することで、型 T へ変換されます。

暗黙の適用（Implicit Application）

もしメソッドが暗黙のパラメータのみをとるなら、暗黙の引数が §7.2 の規則に従って渡されます。

イータ展開 (eta expansion)

そうでなければ、もしメソッドがコンストラクタでなく、要請型 pt が関数型 (Ts´) ⇨ T´なら、式 e のイータ展開(§6.26.5)が実行されます。

空の適用 (Empty Application)

そうでなければ、もし e がメソッド型 ()T なら、それは暗黙のうちに空の引数リストに適用され、e() をもたらします。

6.26.3 オーバーロード解決 (Overloading Resolution)

もし識別子あるいは選択 e がクラスの複数のメンバーを参照するなら、ただ 1 つのメンバーを決定するために、参照のコンテキストが使われます。 この方法は e が関数として使用されるかどうかによります。 AA を、e によって参照されるメンバーの集合とします。

最初に、e は適用中に関数として、たとえば e(e1,...,em) のように現れると仮定します。

まず、引数の **形状(shape)**に基づいて潜在的に適用可能な関数の集合を決定します。

引数式 e の形状は、shape(e) と書き、次のように定義される型です。:

- 関数式 (p1 : T1,...,pn : Tn) => b に対し : (Any,...,Any) => shape(b)。 ここで Any は引数型中に n 回現れます。
- 名前付き引数 n = e に対し : shape(e)
- 他のすべての式に対し : Nothing

BB を、型 (shape(e1),...,shape(en)) の式 (e1,...,en) へ**適用可能**(§6.6)な、AA 中の代替物の集合とします。 もし BB 中に 正確に 1 つの代替物があるなら、その代替物が選ばれます。

Otherwise, let S1,...,Sm be the vector of types obtained by typing each argument with an undefined expected type . For every member m in BB one determines whether it is applicable to expressions (e1,...,em) of types S1,...,Sm . It is an error if none of the members in BB is applicable . If there is one single applicable alternative , that alternative is chosen . Otherwise, let CC be the set of applicable alternatives which don't employ any default argument in the application to e1,...,em . It is again an error if CC is empty . Otherwise, one chooses the most specific alternative among the alternatives in CC , according to the following definition of being "as specific as", and "more specific than":

そうでなければ、S1,...,Sm を、各引数を未定義の要請型で型付けして得られる、型のベクトルとします。 BB 中のすべてのメンバー m に対して、それが型 S1,...,Sm の式 (e1,...,em) に適用可能かどうかを決定します。 もし BB 中のどのメンバーも適用可能でないなら、エラーです。 もし ただ 1 つの適用可能な代替物があるなら、その代替物が選ばれます。 そうでなければ、CC を、e1,...,em への適用中でデフォルト引数を使用しない適用可能な代替物の集合とします。 もし CC が空なら、それは再びエラーです。 そうでなければ、CC 中の代替物の中で**最も特化した**代替物を選びます。 ここで「同じくらい特化した」、「より特化した」という定義は次です。

- A parameterized method m of type (p1 : T1,...,pn : Tn)U is as specific as some other member m′ of type S if m′ is applicable to arguments (p1,...,pn) of types T1,...,Tn .
- A polymorphic method of type [a1 >: L1 <: U1,...,an >: Ln <: Un]T is as specific as some other member of type S if T is as specific as S under the assumption that for i = 1,...,n each ai is an abstract type name bounded from below by Li and from above by Ui .
- A member of any other type is always as specific as a parameterized method or a polymorphic method .
- Given two members of types T and U which are neither parameterized nor polymorphic method types, the member of type T is as specific as the member of type U if the existential dual of T conforms to the existential dual of U . Here, the existential dual of a polymorphic type [a1 >: L1 <: U1,...,an >: Ln <: Un]T is T forSome { type a1 >: L1 <: U1,...,type an >: Ln <: Un }. The existential dual of every other type is the type itself .
- 型(p1 : T1,...,pn:Tn)U のパラメータ化されたメソッド m が、 型 S の他のあるメンバー m′と同じくらい特化しているとは、 m′が型 T1,...,Tn の引数 (p1,...,pn)に適用可能な場合をいいます。
- 型 [a1 >: L1 <: U1,...,an >: Ln <: Un]T の多相的メソッドが、 型 S の 他のあるメンバーと同じくらい特化しているとは、 i = 1,...,n について各 ai が 下から Li によって、 上から Ui によって境界付けられている抽象型名であるという仮定の下、 T が S と同じぐらい特化している場合をいいます。
- 他のすべての型のメンバーは常に、パラメータ化された メソッドあるいは多相的メソッドと同じぐらい特化しています。
- パラメータ化されておらず多相的メソッド型でもない型 T と U の 2 つの与えられたメンバーについて、もし T の存在的双対が U の存在的双対に適合するなら、 型 T のメンバーは型 U のメンバーと同じぐらい特化しています。 ここで、多相型 [a1 >: L1 <: U1,...,an >: Ln <: Un]T の存在的双対とは、 T forSome { type a1 >: L1 <: U1,...,type an >: Ln <: Un } です。 他のすべての型の存在的双対はその型自身です。

代替物 Bに対する代替物 A の**相対的重み**は、0 から 2 までの数値で、次の 2 項の和として定義されます。

- もし A が B と同じぐらい特化しているなら 1、そうでなければ 0。
- もし A が、B を定義しているクラス/オブジェクトから派生する クラス/オブジェクト内で定義されているなら 1、そうでなければ 0。

もし次の 1 つが満たされるなら、クラス/オブジェクト C は、クラス/オブジェクト D から**派生**されます:

- C は D のサブクラス。あるいは、
- C は D から派生したクラスのコンパニオンオブジェクト。あるいは、
- D は C の派生元クラスのコンパニオンオブジェクト。

もし B に対する A の相対的重みが、A に対する B の相対的重みより大きいなら、代替物 A は代替物 B **より特化**しています。

もし CC 中の他のすべての代替物より特化した代替物が CC 中に一つもないなら、エラーです。

次に、e が型適用の中で関数として、たとえば e[targs] のように現れると仮定します。 このとき、targs 中の型引数と同じ個数の型パラメータをとる、AA 中のすべての代替物を選びます。

もしそのような代替物がないなら、エラーです。 もしそのような代替物が複数あるなら、オーバーロード解決を再び式全体 e[targs] に適用します。

最後に、e は適用あるいは型適用のいずれのうちにも関数として現われないと仮定します。 もし要請型が与えられていれば、それに互換(§6.26)な AA 中の代替物の集合を BB とします。 そうでなければ、BB を AA と同じとします。 この場合、BB 中のすべての代替物の中から最も特化した代替物を選びます。 もし BB 中の他のすべての代替物より特化した代替物が BB 中に一つもないなら、エラーです。

Example 6.26.1 次の定義を考えます。

```
class A extends B {}
def f(x: B, y: B) = ...
def f(x: A, y: B) = ...
val a: A
val b: B
```

このとき、適用 f(b、b)は f の最初の定義を参照するのに対して、適用 f(a、a) は 2 番目を参照します。 ここで次の、3 番目のオーバーロードされた定義を加えるとします。

```
def f(x: B, y: A) = ...
```

そうすると、適用 f(a、a) は、最も特化した適用可能なシグニチャが存在しないため、曖昧であるとして却下されます。

6.26.4 ローカルな型推論 (Local Type Inference)

ローカルな型推論は、多相型の式に渡す型引数を推論します。 たとえば、e の型を型 [a1 >: L1 <: U1,...,an >: Ln <: Un]T とし、明示的な型パラメータが与えられていないとします。

ローカルな型推論は、この式を型適用 e[T1,...,Tn] へ変換します。 型引数 T1,...,Tn の取捨は、式が現われるコンテキストと要請型 pt に依存します。 3 つの場合があります。

Case 1: 選択 (Selections)

もし式が、名前 x の選択の前置子として現われるなら、型推論は式 e.x 全体へ**持ち越され**(deferred)ます。 すなわち、もし e.x が型 S を持つなら、それは型 [a1 >: L1 <: U1,...,an >: Ln <: Un]S を持っているとして扱われ、ローカルな型推論が今度は、e.x が現われるコンテキストを用いて a1,...,an の型引数の推論に使われます。

Case 2: 値 (Values)

もし式 e が、値引数へ適用されずに値として現われるなら、その型引数の推論は、要請型 pt をもつ式の型 T にかかわる制約システムを解決することでなされます。 一般性を失うことなく、T は値型であると仮定できます。; つまり、もしそれがメソッド型なら、イータ展開(§6.26.5)を適用して関数型に変換します。 ここで解決とは次のような、型パラメータ ai に対する型 Ti の置換 σ を見つけることです。

- すべての型パラメータ境界を順守する。 すなわち、i = 1,...,n に対して σLi <: σai かつ σai <: σUi 。

- 式の型は要請型に適合する。すなわち、σT <: σpt

もしそのような置換が存在しないなら、実行時エラーとなります。 もし複数の置換が存在するなら、ローカルな型推論は、各型変数 ai に対して解空間の最小/最大の型 Ti を選びます。 もし型パラメータ ai が、式の型 T 中に反変的に(§4.5)現れるなら、**最大の型** Ti が選ばれます。 他のすべての状況、つまり、もし変数が共変的、不変的、あるいはまったく型 T 中に現れなければ、**最小の型** Ti が選ばれます。 そのような置換を、型 T に対する与えられた制約システムの**最適解**と呼びます。

Case 3: メソッド (Methods)

もし式 e が適用 e(d1,...,dm) 中に現れるなら、最後のケースが適用されます。 この場合、T はメソッド型 (p1 : R1,...,pm:Rm)T′です。 一般性を失うことなく、結果型 T′ が値型であると仮定できます。; もしこれがメソッド型なら、我々はこれを関数型に変換するためにイータ展開(§6.26.5)を適用します。 まず、2 つの選択スキームを使って、引数式 dj の型 Sj を計算します。 各引数式 dj は、最初に要請型 Rj で型付けされます。 そのとき、型パラメータ a1,..., an は型定数とみなされます。 もしこれが失敗するなら、引数 dj は、その代わりに要請型 R′j で型付けされます。 ここで R′j は、Rj を a1,...,an 中のすべての型パラメータを未定義で置き換えて得られるものです。

2 つめのステップでは、型引数の推論は、要請型 pt および引数型 S1,...,Sm をもつメソッド型にかかわる制約システムを解決することでなされます。 制約システムの解決とは、次のような、型パラメータ ai に対する型 Ti の置換を見つけることです。

- すべての型パラメータ境界を順守する。 すなわち、i = 1,...,n に対して σLi <: σai かつ σai <: σUi 。
- メソッドの結果型 T′ は要請型に適合する。 すなわち、σT′ <: σpt 。
- 各引数型は、対応する形式上のパラメータ型に弱く適合(§3.5.3)する。 すなわち j = 1,...,m に対して σSj <:w σRj 。

もしそのような置換が存在しないなら、実行時エラーとなります。 もし複数の解決が存在するなら、型 T′に対して最適なものが選ばれます。

要請型 pt のすべてあるいは一部は未定義かもしれません。 この場合、適合性の規則(§3.5.2)は、任意の型 T に対して次の 2 つの文が常に真である、という規則を加えて拡張されます。:

```
undefined <: T    と    T <: undefined
```

型変数に対して最小/最大の解が存在しないことはあり得ます。 その場合、実行時エラーが生じます。 なぜなら、<: は前順序(pre-order)であり、解集合が 1 つの型に対して複数の最適解を持つこともあるからです。 そのような場合 Scala コンパイラは、それらの 1 つを勝手に選びます。

Example 6.26.2次の 2 つのメソッドと定義について考えます。:

```
def cons[A](x: A, xs: List[A]): List[A] = x :: xs
def nil[B]: List[B] = Nil

val xs = cons(1, nil)
```

cons の適用は、未定義の要請型で型付けされます。 この適用は cons[Int](1、nil)へのローカルな型推論によって完成されます。 ここで次の論法を使って、型パラメータ a に対して型引数 Int を推論します。:

最初に、引数式が型付けされます。 最初の引数 1 は Int 型を持ち、他方、2 番目の引数 nil はそれ自身が多相的です。 要請型 List[a] で nil の型チェックを試みます。 これは次の制約システムを導きます。

```
List[b?] <: List[a]
```

ここで、制約システム中の変数であることを示すクエスチョンマークのついた b? という印を使っています。 クラス List は共変なので、この制約の最適解は、

```
b = scala.Nothing .
```

です。

2 つめのステップで、cons の型パラメータ a に対する次の制約システムを解きます。:

```
Int <: a?
List[scala.Nothing] <: List[a?]
List[a?] <: undefined
```

この制約システムの最適解は次です。

```
a = Int ,
```

ですから、Int が a の推論される型です。

Example 6.26.3 今度は、次の定義を考えます。

```
val ys = cons("abc", xs)
```

ここで、xs は前に型 List[Int] と定義されています。 この場合、ローカルな型推論は次のように進みます。

最初に、引数式が型付けされます。最初の引数 "abc" は文字列型です。 2 番目の引数 xs は最初に、要請型 List[a] での型付けを試みられます。 List[Int] が List[a] のサブ型ではないので、これは失敗します。 このため、2 番目の戦略が試みられます。; xs は今度は、要請型 List[undefined] で型付けされます。これは成功し、引数型 List[Int] をもたらします。

2 つめのステップで、cons の型パラメータに対する次の制約システムを解きます。:

```
String <: a?
List[Int] <: List[a?]
List[a?] <: undefined
```

この制約システムの最適解は次です。

```
a = scala.Any
```

ですから、scala.Any が推論される a の型です。

6.26.5 イータ展開（Eta Expansion）

イータ展開は、メソッド型の式を等価な関数型の式に変換します。 それは、2 つのステップで進みます。

最初に、e の最大の部分式を識別します。; たとえば、それらが e1,...,em であるとします。 それらの各々に対し、新規の名前 xi を生成します。 e´を、e 中のすべての最大の部分式 ei を対応する新規の名前 xi で置き換えて得られる式とします。 次に、メソッドの各引数型 Ti(i = 1,...,n）に対して新規の名前 yi を生成します。 イータ変換の結果は次のようになります。

```
{ val x1 = e1 ;
    ...
  val xm = em ;
  ( y1 : T1,...,yn : Tn) => e´( y1,...,yn )
}
```

7 暗黙のパラメータとビュー（Implicit Parameters and Views）

7.1 implicit 修飾子（The Implicit Modifier）

構文:

```
LocalModifier    ::= 'implicit'
ParamClauses     ::= {ParamClause} [nl] '(' 'implicit' Params ')'
```

implicit 修飾子が印されたテンプレートのメンバーとパラメータを暗黙のパラメータ（§7.2）へ渡すことができ、ビュー(§7.3)と呼ばれる暗黙の変換を使えます。 implicit 修飾子は、トップレベル(§9.2)のオブジェクトに対してと同様に、すべての型メンバーに対しても不正です。

Example 7.1.1

次のコードは monoids の抽象クラスと 2 つの具象実装、StringMonoid と IntMonoid を定義しています。 2 つの実装は implicit とマークされています。

```
abstract class Monoid[A] extends SemiGroup[A] {
  def unit: A
  def add(x: A, y: A): A
}
object Monoids {
  implicit object stringMonoid extends Monoid[String] {
    def add(x: String, y: String): String = x.concat(y)
    def unit: String = ""
  }
  implicit object intMonoid extends Monoid[Int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }
}
```

7.2 暗黙のパラメータ (Implicit Parameters)

メソッドの暗黙のパラメータリスト(implicit p1,...,pn) は、パラメータ p1,...,pn を implicit とマークします。 メソッドあるいはコンストラクタは、ただ 1 つの暗黙のパラメータリストを持つことができ、それは与えられた最後のパラメータリストでなければなりません。

暗黙のパラメータをもつメソッドは、通常のメソッドとまったく同じように引数に適用されます。 この場合、implicit ラベルは効果を持ちません。 しかし、もしメソッドがその暗黙のパラメータ引数を失っているなら、そのような引数は自動的に供給されます。

The actual arguments that are eligible to be passed to an implicit parameter of type T fall into two categories . First, eligible are all identifiers x that can be accessed at the point of the method call without a prefix and that denote an implicit definition (§7.1) or an implicit parameter . An eligible identifier may thus be a local name, or a member of an enclosing template, or it may be have been made accessible without a prefix through an import clause (§4.7). If there are no eligible identifiers under this rule, then, second, eligible are also all implicit members of some object that belongs to the implicit scope of the implicit parameter's type, T .

型 T の暗黙のパラメータに渡すに適した実際の引数は、2 つのカテゴリに分けられます。 1つめは、メソッド呼び出しの時点でアクセスできる、前置子のつかない、implicit 定義(§7.1)あるいは暗黙のパラメータを表す、すべての識別子 x が適しています。 適した識別子は、このようにローカル名、あるいは、取り囲むテンプレートのメンバー、あるいは、インポート節(§4.7)を通して前置子なしでアクセス可能になっているものです。 もしこの規則の下で適した識別子がないなら、2つめは、暗黙のパラメータの型 T の暗黙のスコープに入っている、あるオブジェクトのすべての暗黙のメンバーも適しています。

型 T の**暗黙のスコープ**は、暗黙のパラメータの型に随伴するクラスのすべてのコンパニオンモジュール(§5.4)から成ります。 ここで、クラス C が型 T に**随伴する**とは、それが T のある部分の基底クラス (§5.1.2) である場合を言います。 型 T の部分とは、次です。

- if T is a compound type T1 with ... with Tn , the union of the parts of T1,...,Tn , as well as T itself,
- if T is a parameterized type S[T1,...,Tn], the union of the parts of S and T1,...,Tn ,
- if T is a singleton type p.type, the parts of the type of p,
- if T is a type projection S#U , the parts of S as well as T itself,
- in all other cases, just T itself .
- もし T が複合型 T1 with ... with Tn なら、T 自身と同様、T1,...,Tn の部分の和集合。
- もし T がパラメータ化された型 S[T1,...,Tn] なら、S および T1,...,Tn の部分の和集合。
- もし T がシングルトン型 p.type なら、p の型の部分。
- もし T が型投影 S#U なら、T 自身と同様、S の部分。
- 他のすべての場合、T それ自身。

もし暗黙のパラメータの型と一致する適した引数が複数あるなら、最も特化したものが静的なオーバーロード解決の規則(§6.26.3)を使って選ばれます。 もしパラメータがデフォルト引数を持っていて、暗黙の引数がみつからないなら、デフォルト引数が使われます。

Example 7.2.1

Example 7.1.1 のクラスを仮定します。次は、monoid の add と unit 操作を使って、リストの要素の合計を計算するメソッドです。

```
def sum[A](xs: List[A])(implicit m: Monoid[A]): A =
    if (xs.isEmpty) m.unit else m.add(xs.head, sum(xs.tail))
```

問題の monoid は暗黙のパラメータとしてマークされており、したがってリストの型に基づいて推論されます。 たとえば次のインスタンス呼び出しを考えます。

```
sum(List(1, 2, 3))
```

これは stringMonoid と intMonoid が見えているコンテキスト中にあるものとします。 我々は、sum の形式上の型パラメータ a が Int にインスタンス化される必要があることを知っています。 暗黙の形式上のパラメータ型 Monoid[Int]とマッチする唯一の適したオブジェクトは intMonoid です。 ですからこのオブジェクトは暗黙のパラメータとして渡されます。

この議論は、すべての型引数が推論された(§6.26.4)後で、暗黙のパラメータが推論されることも示しています。

暗黙のメソッドは、それ自身暗黙のパラメータを持てます。 1 つの例は次のメソッドで、これは scala.Ordered クラスにリストを注入(inject)するモジュール scala.List にあります。 リストの要素型もこの型に互換であるとします。

```
implicit def list2ordered[A](x: List[A])
  (implicit elem2ordered: A => Ordered[A]): Ordered[List[A]] =
...
```

さらに加えて、次のメソッドが Ordered クラスに整数を注入すると仮定します。

```
implicit def int2ordered(x: Int): Ordered[Int]
```

そうすると、順序づけられたリスト(ordered list)上に ソートメソッドを定義できます:

```
def sort[A](xs: List[A])(implicit a2ordered: A => Ordered[A]) = ...
```

次のように、整数のリストのリスト yss: List[List[Int]] にソートを適用できます:

```
sort(yss)
```

上記の呼び出しは、2 重にネストした暗黙の引数を渡すことで完成します。:

```
sort(yss)(xs: List[Int] => list2ordered[Int](xs)(int2ordered))
```

暗黙の引数に暗黙の引数を渡せることは、無限再帰を引き起こす可能性があります。 例えば、**すべての**型を Ordered クラスに注入する、次のメソッドを定義しようとするかもしれません。:

```
implicit def magic[A](x: A)
  (implicit a2ordered: A => Ordered[A]): Ordered[A] =
  a2ordered(x)
```

ここで Ordered クラスへの他の注入を持っていない型の引数 arg に、ソートを適用しようすると無限展開になるでしょう:

```
sort(arg)(x => magic(x)(x => magic(x)(x => ... )))
```

To prevent such infinite expansions, the compiler keeps track of a stack of "open impl icit types" for which implicit arguments are currently being searched . Whenever an im plicit argument for type T is searched, the "core type" of T is added to the stack . H ere, the core type of T is T with aliases expanded, top-level type annotations (§11) a nd refinements (§3.2.7) removed, and occurrences of top-level existentially bound vari ables replaced by their upper bounds . The core type is removed from the stack once th e search for the implicit argument either definitely fails or succeeds . Everytime a c ore type is added to the stack, it is checked that this type does not dominate any of the other types in the set .

そのような無限の展開を防ぐために、コンパイラは、現在検索されている暗黙の引数に関する「オープンな暗黙型」のスタックを追跡します。 型 T に対する暗黙の引数を検索するときはいつでも、T の「コア型」がスタックに加えられます。 ここで T の**コア型**とは、拡張されたエイリアス、削除されたトップレベルの型アノテーション(§11)と細別 (§3.2.7)、それらの上限境界で置き換えられたトップレベルの存在的に束縛された変数の出現などを備えた T です。 コア型は暗黙の引数探索が確実に失敗あるいは成功するとすぐにスタックから取り除かれます。 コア型がスタックに加えられるたびに、その型が集合中の他のいずれの型もドミネートしないことがチェックされます。

ここで、コア型 T が型 U を**ドミネート**するとは、T が U に等価 (§3.5.1) であるか、あるいは、T と U のトップレベルの型コンストラクタが共通の要素を持っていて T が U よりも複雑な場合です。

型 T の**トップレベルの型コンストラクタ**の集合 ttcs(T)は、型の形に依存します。:

```
型指定子に対して、
ttcs(p.c) = {c};
パラメータ化された型に対して、
ttcs(p.c[targs]) = {c};
シングルトン型に対して、
ttcs(p.type) = ttcs(T) 。ただし p が型 T をもつとして;
複合型に対して、
ttcs(T1 with ... with Tn) = ttcs(T1)∪ ... ∪ttcs(Tn)
```

コア型の**複雑さ**complexity(T) も、型の形に依存する整数です:

```
型指定子に対して、
complexity(p.c) = 1 + complexity(p)
パラメータ化された型に対して、
complexity(p.c[targs]) = 1 + Σcomplexity(targs)
パッケージ p を表すシングルトン型に対して、
complexity(p.type) = 0
他のすべてのシングルトン型に対して、
complexity(p.type) = 1 + complexity(T) 。ただし p が型 T をもつとして;
複合型に対して
complexity(T1 with ... with Tn) = Σcomplexity(Ti)
```

Example 7.2.2

型 List[List[List[Int]]] の、あるリスト xs に対して sort(xs) を型付けするとき、暗黙の引数が検索される、型のシーケンスは次です。

```
List[List[Int]] => Ordered[List[List[Int]]],
List[Int] => Ordered[List[Int]]
Int => Ordered[Int]
```

すべての型は共通の型コンストラクタ scala.Function1 を共有しますが、しかし、各新しい型の複雑さは前の型の複雑さより低いです。 ですから、コードは型チェックを通ります。

Example 7.2.3

ys を Ordered へ変換できないある型のリストとします。たとえば、

```
val ys = List(new IllegalArgumentException,new ClassCastException,new Error)
```

前に書いた magic の定義がスコープ中にあると仮定します。 このとき、暗黙の引数が検索される、型のシーケンスは次です。

```
Throwable => Ordered[Throwable],
Throwable => Ordered[Throwable],
...
```

シーケンスの 2 番目の型は最初の型と等価ですから、コンパイラは暗黙展開の不一致を知らせるエラーを発行するでしょう。

7.3 ビュー (Views)

暗黙のパラメータとメソッドも、**ビュー(views)**と呼ばれる暗黙の変換を定義できます。型 S から型 T へのビューは、関数型 S=>T あるいは (=>S)=>T をもつ暗黙の値によって、あるいはその型の値に変換可能なメソッドによって、定義されます。

ビューは、2 つの状況で適用されます。

1. If an expression e is of type T , and T does not conform to the expression's expected type pt . In this case an implicit v is searched which is applicable to e and whose result type conforms to pt . The search proceeds as in the case of implicit parameters, where the implicit scope is the one of T => pt . If such a view is found, the expression e is converted to v(e) .

1. もし式 e の型が T で、T が式の要請型 pt に適合しない場合。 この場合、e に適用可能でその結果型が pt に適合する、暗黙の v が検索されます。 検索は暗黙のパラメータの場合と同じように進み、 そこでは暗黙のスコープは T => pt のそれです。 もしそのようなビューが見つかれば、式 e は v(e) に変換されます。

2. e の型が T で、e の選択 e.m 中で、セレクタ m が T のメンバーを表さない場合。 この場合、e に適用可能でその結果が m という名前のメンバーを含む、 ビュー v が検索されます。検索は暗黙のパラメータの場合と同じように進み、 そこでは暗黙のスコープは T のそれです。 もしそのようなビューが見つかれば、選択 e.m は v(e).m に変換されます。

As for implicit parameters, overloading resolution is applied if there are several possible candidates (of either the call-by-value or the call-by-name category) .

暗黙のビューが受け入れできるのは、もしそれが見つかれば、値呼出しあるいは名前呼び出しパラメータとしての引数 e です。 しかし、暗黙の値呼出しは暗黙の名前呼出しよりも優先順位が

高くなります。

暗黙のパラメータについては、もし複数の可能な候補(値呼出しあるいは名前呼出しカテゴリのいずれでも)があれば、オーバーロード解決が適用されます。

Example 7.3.1 クラス scala.Ordered[A] は、次のメソッドを含みます。

```
def <= [B >: A](that: B)(implicit b2ordered: B => Ordered[B]): Boolean
```

型 List[Int] の 2 つのリスト xs と ys があるとし、`§7.2`_ で定義された list2ordered と int2ordered メソッドがスコープ中にあると仮定します。

このとき、操作

```
xs <= ys
```

は正しく、そして次へ展開されます。

```
list2ordered(xs)(int2ordered).<=
  (ys)
  (xs => list2ordered(xs)(int2ordered))
```

list2ordered の最初の適用は、リスト xs をクラス Ordered のインスタンスへ変換するのに対して、2 番目の出現は、<= メソッドに渡される暗黙のパラメータの一部です。

7.4 コンテキスト境界と可視境界(ビュー境界) (Context Bounds and View Bounds)

構文:

```
TypeParam ::= (id | '_') [TypeParamClause] ['>:' Type] ['<:'Type]
              {'<%' Type} {':' Type}
```

メソッド/非トレイトクラスの型パラメータ A は、1 つ以上の可視境界(view bound) A <% T を持てます。 この場合、型パラメータは、境界 T へのビュー適用により変換可能な、任意の型 S へインスタンス化されます。

A type parameter A of a method or non-trait class may also have one or more context bounds A : T . In this case the type parameter may be instantiated to any type S for which evidence exists at the instantiation point that S satisfies the bound T . Such evidence consists of an implicit value with type T[S] .

メソッド/非トレイトクラスの型パラメータ A は、1 つ以上のコンテキスト境界 A : T も持てます。 この場合、型パラメータは、型 S が 境界 T をみたすインスタンス化地点でその**証拠(evidence)**が存在するような、任意の型 S へインスタンス化されるでしょう。 そのような証拠は、型 T[S] をもつ暗黙の値からなります。

可視/コンテキスト境界をもつ型パラメータを含むメソッド/クラスは、暗黙のパラメータをもつメソッドと同じように扱われます。 はじめに、次のような可視あるいはコンテキストあるいはその両方の境界をもつ、ただ 1 つのパラメータの場合を考えます。

```
def f[A <% T1 ... <% Tm : U1 : Un](ps): R = ...
```

このとき、上のメソッド定義は次へ展開されます。

```
def f[A](ps)(implicit v1 : A => T1 ,..., vm : A => Tm ,
                      w1 : U1[ A ],..., wn : Un[ A ]): R = ...
```

ここで vi と wj は、新たに導入された暗黙のパラメータのための新規の名前です。 これらのパラメータは**証拠パラメータ(evidence parameters)**と呼ばれます。

もしクラス/メソッドが、複数の可視-/コンテキスト-境界付けられた型パラメータを持つなら、そのような各型パラメータは出現順に証拠パラメータへ展開され、得られるすべての証拠パラメータは 1 つの暗黙のパラメータ部にまとめられます。 トレイトはコンストラクタパラメータをとらないので、この変換処理はそれらには機能しません。 したがって、トレイト中の型パラメータは、可視-/コンテキスト-境界付きではありません。 同様に、可視-/コンテキスト-境界をもつメソッド/クラスは、いかなる追加の暗黙のパラメータも定義できません。

Example 7.4.1 Example 7.3.1 中で言及された <= メソッドは、次のように、より簡潔に宣言できます。

```
def <= [B >: A <% Ordered[B]](that: B): Boolean
```

7.5 マニフェスト (Manifests)

マニフェスト(目録)は、暗黙のパラメータへの引数として、Scala コンパイラによって自動的に生成される、型ディスクリプタです。Scala 標準ライブラリは 4 つのマニフェストクラスの階層構造を含み、トップは OptManifest です。次はそれらシグニチャの概観です。

```
trait OptManifest[+T]
object NoManifest extends OptManifest[Nothing]
trait ClassManifest[T] extends OptManifest[T]
trait Manifest[T] extends ClassManifest[T]
```

もしメソッド/コンストラクタの暗黙のパラメータの型がクラス OptManifest[T] のサブ型 M[T] なら、次の規則に従って、**M[S] に対するマニフェスト**が決まります。

最初に、もしすでに M[T] にマッチする暗黙の引数があるなら、その引数が選ばれます。

そうでない場合、ここで、もし M がトレイト Manifest なら、Mobj を scala.reflect.Manifest のコンパニオンオブジェクトとし、そうでなければ、 Mobj を scala.reflect.ClassManifest のコンパニオンオブジェクトとします。 もし M がトレイト Manifest なら、M′を トレイト Manifest とし、そうでなければ M′ をトレイト OptManifest とします。 このとき次の規則が適用されます。

1. もし T が値クラスあるいは、Any、AnyVal、Object、Null、Nothing の 1 つなら、 それに対するマニフェストは、Manifest モジュール中に存在する、 対応するマニフェスト値 Manifest.T を選ぶことで作成されます。

2. もし T が Array[S] のインスタンスなら、マニフェストは、Mobj.arrayType[S](m) の呼び出しで作成されます。 ここで m は M[S] に対して決定されるマニフェストです。
3. もし T が他のあるクラス型 S#C[U1,...,Un] なら、 ここで前置型 S はクラス C から静的に決定できないとして、マニフェストは Mobj.classType[T](m0 , classOf[T], ms) の呼び出しで作成されます。 ここで m0 は M′[S] に対して決定されるマニフェストであり、 ms は M′[U1],...,M′[Un] に対して決定されるマニフェストです。
4. もし T が型引数 U1,...,Un をもつ他のあるクラス型なら、 マニフェストは Mobj.classType[T](classOf[T], ms) の呼び出しで作成されます。 ただしここで ms は M′[U1],...,M′[Un] に対して決定されるマニフェストです。
5. もし T がシングルトン型 p.type なら、マニフェストは Mobj.singleType[T](p) の呼び出しで作成されます。
6. もし T が細別型 T′{R} なら、マニフェストは T′に対して作成されます。 (すなわち、細別はマニフェストに決して反映されません)。
7. もし T が論理積型 T1 with,...,with Tn (n > 1)なら、その結果は、 フルのマニフェストが決定されるか否かによります。 もし M がトレイト Manifest なら、マニフェストは Manifest.intersectionType[T](ms) の呼び出しで作成されます。 ただしここで ms は M[T1],...,M[Tn] に対して決定されるマニフェストです。 そうでなければ、もし M がトレイト ClassManifest なら、 マニフェストは型 T1,...,Tn の論理積ドミネータ(§3.7) に対して作成されます。
8. もし T が他のある型なら、そのときもし M がトレイト OptManifest なら、 マニフェストは指定子 scala.reflect.NoManifest から作成されます。 もし M が OptManifest と異なる型なら、静的エラーとなります。

8 パターンマッチング（Pattern Matching）

8.1 パターン（Patterns）

構文:

```
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}
```

パターンは定数、コンストラクタ、変数と型テストから構成されます。 パターンマッチングは、与えられた値(あるいは値のシーケンス)がパターンによって定義された形をしているかテストし、もしそうなら、パターン中の変数を対応する値(あるいは値のシーケンス)の構成要素に束縛します。 同じ変数名はパターン中で 1 度より多くは束縛できません。

Example 8.1.1 次はパターンのいくつかの例です:

1. パターン ex:IOException は、クラス IOException のすべてのインスタンスとマッチし、変数 ex をインスタンスに束縛します。
2. パターン Some(x)は、Some(v) の形の値と一致し、x を Some コンストラクタの引数値 v に束縛します。
3. パターン(x,_)は、値のペアに一致し、 x をそのペアの最初の構成要素に束縛します。 2 番目の構成要素はワイルドカードパターンとマッチします。
4. パターン x :: y :: xs は、長さ 2 以上のリストとマッチし、 x をリストの最初の要素へ、y をリストの 2 番目の要素へ、 xs を残りへ束縛します。
5. パターン 1 | 2 | 3 は、1 以上 3 以下の整数とマッチします。

パターンマッチングは常に、パターンの要請型を提供するコンテキスト中で実行されます。 パターンには次の種類の区別があります。

8.1.1 変数パターン (Variable Patterns)

構文:

```
SimplePattern    ::=   '_'
                     |   varid
```

変数パターン x は小文字で始まる単純な識別子です。 それは任意の値とマッチし、変数名をその値に束縛します。 x の型は、外側で与えられた、パターンの要請型です。 ワイルドカードパターン _ は特別な場合で、出現の度に新しい変数であるかのように扱われます。

8.1.2 型付きパターン (Typed Patterns)

構文:

```
Pattern1         ::=   varid ':' TypePat
                     |   '_' ':' TypePat
```

型付きパターン x : T は、パターン変数 x と型パターン T から成ります。 x の型は型パターン T です。ここで各型変数とワイルドカードは、新規の、未知の型によって置き換えられます。 このパターンは型パターン T によってマッチされる任意の値にマッチします(§8.2); これは変数名をその値に束縛します。

8.1.3 パターンバインダー (Pattern Binders)

構文:

```
Pattern2         ::=   varid '@' Pattern3
```

パターンバインダー(訳注:変数の束縛、あるいは、変数識別子束縛) x@p は、パターン変数 x とパターン p からなります。 変数 x の型はパターン p の静的な型 T です。 このパターンは、v の実行時型も同じく T のインスタンスなら、パターン p によってマッチされる任意の値 v にマッチします。 そして変数名をその値に束縛します。

8.1.4 リテラルパターン (Literal Patterns)

構文:

```
SimplePattern       ::=   Literal
```

リテラルパターン `L` は、リテラル `L` に(==の意味で)等価な任意の値にマッチします。 `L` の型はパターンの要請型に適合しなくてはなりません。

8.1.5 安定識別子パターン (Stable Identifier Patterns)

構文:

```
SimplePattern       ::=   StableId
```

安定識別子パターンは安定識別子 `r` (§3.1)です。 `r` の型はパターンの要請型に適合しなくてはなりません。 パターンは、`r == v` である任意の値にマッチします(§12.1)。

変数パターンとの構文的な重なりを解決するため、安定識別子パターンは小文字で始まる単純名ではいけません。 しかし、これはそのような変数名をバッククォートで囲めば可能で、そうすれば安定識別子パターンとして扱われます。

Example 8.1.2 次の関数定義を考えます。:

```
def f(x: Int, y: Int) = x match {
  case y => ...
}
```

ここで `y` は、任意の値にマッチする変数パターンです。 もしこのパターンを安定識別子パターンに変えたければ、次のようにます。:

```
def f(x: Int, y: Int) = x match {
  case `y` => ...
}
```

これで、パターンは取り囲む関数 `f` の `y` パラメータにマッチします。 すなわち、`f` の `x` 引数と `y` 引数が等しい場合に限り、マッチングは成功します。

8.1.6 コンストラクタパターン (Constructor Patterns)

構文:

```
SimplePattern       ::=   StableId '(' [Patterns] ')
```

コンストラクタパターンは `c(p1,...,pn)` の形です。ここで `n >= 0`。 これは、安定識別子 `c` とそれに続く要素パターン `p1,...,pn` からなります。

コンストラクタ c はケースクラス(§5.3.2)を表す単純名あるいは修飾された名前です。 もしケースクラスが単相的なら、それはパターンの要請型に適合しなくてはならず、また、x の基本コンストラクタ(§5.3)の形式上のパラメータ型は、要素パターン p1,...,pn の要請型とみなされます。（訳注: x とあるのは c の誤記?）

もしケースクラスが多相的なら、c のインスタンス化がパターンの要請型に適合するように、型パラメータがインスタンス化されます。 c の基本コンストラクタのインスタンス化された形式上のパラメータ型は、構成要素パターン p1,...,pn の要請型とみなされます。 パターンは、コンストラクタ呼び出し c(v1,...,vn) から生成されるすべてのオブジェクトにマッチします。ただしここで、各要素パターン pi は対応する値 vi にマッチします。

c の形式上のパラメータ型が反復パラメータで終わるとき、特別な場合が生じます。 これについては、さらに(§8.1.9)で論じます。

8.1.7 タプルパターン（Tuple Patterns）

構文:

```
SimplePattern   ::=   '(' [Patterns] ')'
```

タプルパターン (p1,...,pn) は、コンストラクタパターン scala.Tuplen(p1,...,pn) のエイリアスです。 ここで n >= 2。空タプル()は、型 scala.Unit のただ 1 つの値です。

8.1.8 抽出子パターン（Extractor Patterns）

構文:

```
SimplePattern        ::=   StableId '(' [Patterns] ')'
```

抽出子パターン x(p1,...,pn) は、ここで n >= 0、コンストラクタパターンと同じ構文上の形をしています。 しかし、ケースクラスの代わりに、安定識別子 x は、パターンにマッチする unapply あるいは unapplySeq という名前のメンバーメソッドを持つオブジェクトを表します。

オブジェクト x 中の unapply メソッドは、もしそれが正確にただ 1 つの引数をとり、次の 1 つが当てはまるなら、パターン x(p1,...,pn) と**マッチ**します:

- n = 0 かつ unapply の結果型は Boolean 。 この場合、抽出子パターンは x.unapply(v) が true となる、 すべての値 v とマッチします。
- n = 1 かつ unapply の結果型が、ある型 T から構成される Option[T] 。 この場合、その(ただ 1 つの)引数パターン p1 は、今度は要請型 T で型付けられます。このとき抽出子パターンは、x.unapply(v) が Some(v1) の形の値をもたらしそして p1 が v1 にマッチする、 すべての値 v とマッチします。
- n > 1 かつ unapply の結果型が、ある型 T1,...,Tn から構成される Option[(T1,...,Tn)] 。この場合、引数パターン p1,...,pn は、今度は要請型 T1,...,Tn で型付けされます。このとき抽出子パターンは、x.unapply(v) が Some((v1,...,vn)) の形の値をもたらし、そして各パターン pi が対応する vi にマッチする、すべての値 v とマッチします。

オブジェクト x 中の unapplySeq メソッドは、もしそれが正確にただ 1 つの引数をとり、その結果型が Option[S] の形なら、パターン x(p1,...,pn)とマッチします。 ここで S はある要素型 T から構成される Seq[T] のサブ型です。 これについては(§8.1.9)でさらに議論されます。

Example 8.1.3 Predef (事前定義済み)オブジェクトは抽出子オブジェクト Pair の定義を含んでいます:

```
object Pair {
  def apply[A, B](x: A, y: B) = Tuple2(x, y)
  def unapply[A, B](x: Tuple2[A, B]): Option[Tuple2[A, B]] = Some(x)
}
```

このことは、名前 Pair が Tuple2 の代わりに、パターンにおけるタプルの逆構築(deconstruction)と同様に、タプル形成にも使えることを意味します。 ですから、次が可能です:

```
val x = (1, 2)
val y = x match {
  case Pair(i, s) => Pair(s + i, i * i)
}
```

8.1.9 シーケンスパターン (Pattern sequences)

構文:

```
SimplePattern ::= StableId '(' [Patterns ','] [varid '@'] '_' '*' ')'
```

シーケンスパターン p1,...,pn は、2 つのコンテキスト中に現れます。 1 つは、コンストラクタパターン c(q1,...,qm , p1,...,pn) 中です。 ここで c は、m + 1 個の基本コンストラクタパラメータをもつケースクラスで、パラメータの最後が型 S の反復パラメータ(§4.6.2)のものです。 2 つめは、抽出子パターン x(p1,...,pn) 中です。 ただし、抽出子オブジェクト x は S eq[S] に適合する結果型を返す unapplySeq メソッドを持つが、しかし p1,...,pn にマッチする unapply メソッドを持たない場合です。 シーケンスパターンの要請型は、それぞれ、型 S です。

シーケンスパターン中の最後のパターンは、**ワイルドカードシーケンス** _* でも構いません。 各要素パターン pi は、それがワイルドカードシーケンスでない限り、S を要請型として型チェックされます。 もし最後にワイルドカードシーケンスがあれば、そのパターンは、パターン p1,...,pn-1 にマッチする要素で始まるシーケンスである、すべての値 v にマッチします。 もし最後にワイルドカードシーケンスが与えられていないなら、そのパターンは、パターン p1,...,pn にマッチする要素からなる長さ n のシーケンスである、すべての値 v にマッチします。

8.1.10 中置演算パターン (Infix Operation Patterns)

構文:

```
Pattern3   ::= SimplePattern {id [nl] SimplePattern}
```

中置演算パターン p op q は、コンストラクタあるいは抽出子パターン op(p,q) の略記表現です。

パターン中の演算子の優先順位と結合性は、式(§6.12)中の場合と同じです。 中置演算パターン p op (q1,...,qn) は、コンストラクタあるいは抽出子パターン op(p, q1,...,qn) の略記表現です。

8.1.11 パターン選択 (Pattern Alternatives)

構文:

```
Pattern ::= Pattern1 { '|' Pattern1 }
```

パターン選択 p1 | ... | pn は、多数の選択肢パターン pi からなります。 すべての選択肢パターンは、パターンの要請型で型チェックされます。 それらはワイルドカード以外の変数を束縛しません。 選択パターンは、もし少なくとも 1 つの選択肢が 値 v にマッチするなら、v にマッチします。

8.1.12 XML patterns (XML Patterns)

XML パターンは §10.2 で扱います。

8.1.13 正規表現パターン (Regular Expression Patterns)

Later version of Scala provide a much simplified version of regular expression patterns that cover most scenarios of non-text sequence processing .

正規表現パターンは、Scala バージョン 2.0 から廃止になりました。

Scala の後のバージョンでは、非テキストシーケンス処理のほとんどのシナリオをカバーする、正規表現パターンのより単純化されたバージョンを提供します。

シーケンスパターンとは、次のいずれかの場所にあるパターンです。(1) 要請される、ある A から構成される Seq[A] に適合する型 T のパターン、あるいは、(2) 反復する形式上のパラメータ A* を持つケースクラスコンストラクタ。 最も右に位置するワイルドカードの星印パターン _* は、任意長さのシーケンスを表します。 通常、@を使って変数へ束縛できます。 その場合、変数は型 Seq[A] を持ちます。

8.1.14 明白パターン（Irrefutable Patterns）

パターン p は、もし次の 1 つが当てはまるなら、型 T について**明白(irrefutable)**です。:

1. p は変数パターン。
2. p は型付きパターン x : T′で、T <: T′。
3. p はコンストラクタパターン c(p1,...,pn) で、 型 T はクラス c のインスタンス、型 T の基本コンストラクタ(§5.3) が引数型 T1,...,Tn をもち、各 pi が Ti について明白(irrefutable)。

8.2 型パターン（Type Patterns）

構文:

```
TypePat     ::= Type
```

型パターンは、型、型変数、ワイルドカードからなります。 型パターン T は、次の形の 1 つをとります。:

- クラス C、p.C、あるいは T#C への参照。 この型パターンは、与えられたクラスの任意の非 null インスタンスとマッチします。 クラスの前置子は、もしそれが与えられていれば、 クラスインスタンスの決定に関係があるあることに注意してください。 例えばパターン p.C は、パス p を前置子として生成されたクラス C のインスタンスとのみマッチします。

  最下位の型 scala.Nothing と scala.Null は型パターンとして使用できません。 なぜなら、それらは如何なる場合も何ともマッチしないからです。

- A singleton type p.type . This type pattern matches only the value denoted by the path p (that is, a pattern match involved a comparison of the matched value with p using method eq in class AnyRef) .
- A compound type pattern T1 with ... with Tn where each Ti is a type pattern . This type pattern matches all values that are matched by each of the type patterns Ti .
- シングルトン型 p.type。 この型パターンは、パス p で示される値とだけマッチします（すなわち、クラス AnyRef 中の eq メソッドを使っての、 p とマッチされた値との比較を含むパターンマッチ）。
- 複合型パターン T1 with ... with Tn。ここで、各 Ti は型パターン。 この型パターンは、各型パターン Ti がマッチするすべての値とマッチします。
- A parameterized type pattern T[a1,...,an], where the ai are type variable patterns or wildcards _. This type pattern matches all values which match T for some arbitrary instantiation of the type variables and wildcards . The bounds or alias type of these type variable are determined as described in (§8.3) .
- パラメータ化された型パターン T[a1,...,an]。 ここで ai は型変数パターンあるいはワイルドカード _ 。 この型パターンは、型変数とワイルドカードの 何らかの任意のインス

タンス化について T にマッチする、 すべての値とマッチします。 これら型変数の境界 あるいはエイリアス型は、(§8.3) の記述に従って決定されます。

・パラメータ化された型パターン scala.Array[T1]。 ただしここで T1 は 1 つの型パターン。 この型パターンは、型が scala.Array[U1] である任意の非 null インスタンスにマッチします。ここで U1 は T1 とマッチする型です。

上述した形の 1 つではないものも、型パターンとして受け入れられます。 しかし、そのような型パターンは、それらの型消去(§3.7)に変換されます。 Scala コンパイラは、これらのパターンが型安全でない可能性を合図する「unchecked」という警告を発します。

型変数パターン(type variable pattern)は小文字で始まる単純な識別子です。 しかし、事前定義されたプリミティブな型エイリアス unit、boolean、byte、short、char、int、float、double は、型変数パターン に分類されません。

8.3 パターン中の型パラメータ推論 (Type Parameter Inference in Patterns)

型パラメータ推論は、型付きパターンあるいはコンストラクタパターン中の束縛された型変数の境界を見つけるプロセスです。 推論は、パターンの要請型を計算に入れます。

型付きパターンの型パラメータ推論(Type parameter inference for typed patterns..)

型付きパターン p : T′を仮定します。 T を、 T′中のすべてのワイルドカードを新規の変数名へリネームして得られたものとします。 a1,...,an を T 中の型変数とします。 これらの型変数は、パターン中で束縛されていると考えられます。 パターンの要請型を pt とします。

Type parameter inference constructs first a set of subtype constraints over the type variables ai . The initial constraints set CC0 reflects just the bounds of these type variables . That is, assuming T has bound type variables a1,...,an which correspond to class type parameters a′1,...,a′n with lower bounds L1,...,Ln and upper bounds U1,..., Un , CC0 contains the constraints

型パラメータ推論は最初に、型変数 ai 上のサブ型制約の集合を構築します。 最初の制約集合 CC0 は、それら型変数の境界そのものの反映です。 すなわち、T が束縛された型変数 a1,...,an をもつと仮定します。 ただしここで a1,...,an は、下限境界 L1,...,Ln と上限境界 U1,..., Un をもつクラス型パラメータ a′1,...,a′n に対応するものであり、CC0 は次の制約を含みます。

```
ai      <: σUi     (i = 1,...,n)
σLi    <: ai       (i = 1,...,n)
```

ここで σ は置換 [a′1: = a1,...,a′n := an] です。

このとき集合 CC0 は、さらなるサブ型制約によって大きくなります。 次の 2 つの場合があります。

Case 1:

もし σT が pt に適合するような型変数 ai,...,an 上の置換が存在するなら、 CC0∧CC1 が T の pt への適合を意味するような、型変数 a1,...,an 上の最も弱いサブ型制約 CC1 を決定します。

Case 2:

Otherwise, if T can not be made to conform to pt by instantiating its type variables, one determines all type variables in pt which are defined as type parameters of a method enclosing the pattern . Let the set of such type parameters be b1,...,bm . Let CC′0 be the subtype constraints reflecting the bounds of the type variables bi . If T denotes an instance type of a final class, let CC2 be the weakest set of subtype constraints over the type variables a1,...,an and b1,...,bm such that CC0∧CC′0∧CC2 implies that T conforms to pt .

そうでなければ、もし T の型変数のインスタンス化で T を pt に適合させることができないなら、パターンを囲むメソッドの型パラメータとして定義された、pt 中のすべての型変数を決定します。そのような型パラメータのセットを b1,...,bm とします。 CC′0 を型変数 bi の境界を反映するサブ型制約とします。 もし T が final クラスのインスタンス型を表すなら、CC2 を、CC0∧CC′0∧CC2 が T の pt への適合を意味するような、型変数 a1,...,an と b1,...,bm 上のサブ型制約の最も弱い集合とします。

If T does not denote an instance type of a final class, let CC2 be the weakest set of subtype constraints over the type variables a1,...,an and b1,...,bm such that CC0∧CC′0∧CC2 implies that it is possible to construct a type T′ which conforms to both T and pt . It is a static error if there is no satisfiable set of constraints CC2 with this property . The final step consists in choosing type bounds for the type variables which imply the established constraint system . The process is different for the two cases above.

もし T が final クラスのインスタンス型を表さないなら、CC2 を、CC0∧CC′0∧CC2 が T と pt の両方に適合する型 T′の構成可能性を意味するような、型変数 a1,...,an と b1,...,bm 上のサブ型制約の最も弱い集合とします。 もしこの属性をもつ制約 CC2 の集合がないなら、静的エラーです。

最後のステップは型変数の型境界を選ぶことであり、制約システムの確定を意味します。 そのプロセスは、上記 2 つの場合で異なります。

Case 1:

我々は ai >: Li <: Ui を得ます。 ここで i = 1,...,n に対して ai >: Li <: Ui が CC0∧CC1 を意味するような、<: に関して各 Li は最小、各 Ui は最大のものです。

Case 2:

我々は ai >: Li <: Ui と bi >: L′i <: U′i を得ます。 ここで i = 1,...,n、j = 1,...m に対して ai >: Li <: Ui と bj >: L′j <: U′j が CC0∧CC′0∧CC2 を意味するような、各 Li と L′j は最小、各 Ui と U′j は最大のものです。

どちらの場合も、ローカルな型推論は、推論される境界の複雑さを制限することが許されます。 型の最小性と最大性は、型集合の受け入れ可能な複雑さと比較して解釈されなければなりません。

コンストラクタパターンの型パラメータ推論(Type parameter inference for constructor patterns..)

コンストラクタパターン C(p1,...,pn) を仮定します。 ここで、クラス C は、型パラメータ a1,...,an を持つとします。 これらの型パラメータは、型付きパターン (_: C[a1,...,an]) と同じ方法で推論されます。

Example 8.3.1 次のプログラム断片について考えます。:

```
val x: Any
x match {
  case y: List[a] => ...
}
```

ここで、型パターン List[a] は、要請型 Any に対してマッチされます。 パターンは型変数 a を束縛します。 List[a] はすべての型引数に対して Any に適合するので、a 上に何も制約がありません。 ですから、a は境界をもたない抽象型として導入されます。 a のスコープは、そのケース節の右側です。

他方、もし x が次のように宣言されれば

```
val x: List[List[String]],
```

これは 制約 List[a] <: List[List[String]] を生み出し、List が共変なので a <: List[String] へ単純化されます。 ですから、a は上限境界 List[String] をもって導入されます。

Example 8.3.2 次のプログラム断片について考えます。:

```
val x: Any
x match {
  case y: List[String] => ...
}
```

Scala は型引数の情報を実行時に保持しません。 ですから x が文字列のリストであることをチェックする方法がありません。 その代わりに、Scala コンパイラはパターンを List[_] に型消去(§3.7)します。; つまり、値 x のトップレベルの実行時クラスが List に適合するかテストするだけです。 そしてもしそうなら、パターンマッチングは成功します。 これは、リスト x が文字列以外の要素を含む場合に、後でクラスキャスト例外を発生させるかもしれません。 Scala コンパイラは、「unchecked」というメッセージで、型安全でない可能性を合図します。

Example 8.3.3 次のプログラム断片について考えます。

```
class Term[A]
class Number(val n: Int) extends Term[Int]
def f[B](t: Term[B]): B = t match {
  case y: Number => y.n
}
```

パターン y: Number の要請型は Term[B] です。 型 Number は Term[B] に適合しません。;ですから、上記規則の Case 2 が適用されます。

This means that b is treated as another type variable for which subtype constraints are inferred . In our case the applicable constraint is Number <: Term[B], which entails B = Int . Hence, B is treated in the case clause as an abstract type with lower and upper bound Int . Therefore, the right hand side of the case clause, y.n, of type Int, is found to conform to the function's declared result type, Number .

このことは b が、そのサブ型制約が推論された他の型変数として扱われることを意味します。 この場合、適用可能な制約は Number <: Term[B] です。 ここで B = Int が引き起こされます。 ですから、B は、ケース節で上・下限境界 Int をもつ抽象型として扱われます。 したがって、ケース節の右側 y.n は型 Int をもち、関数の宣言された結果型 Number に適合することが判明します。 (訳注: Numberは B の誤記?)

8.4 パターンマッチ式 (Pattern Matching Expressions)

構文:

```
Expr            ::=   PostfixExpr 'match' '{' CaseClauses '}'
CaseClauses     ::=   CaseClause {CaseClause}
CaseClause      ::=   'case' Pattern [Guard] '=>' Block
```

パターンマッチ式

```
e match { case p1 => b1 ... case pn => bn }
```

は、セレクター式 e と n 個(>0)の ケースからなります。 各ケースは、(ガードがあってもよい) パターン pi とブロック bi からなります。 各 pi には、ガード if e が補完されていても構いません。 ここで e は boolean 式です。 pi 中のパターン変数のスコープは、パターンのガードと対応するブロック bi から構成されます。

セレクター式 e の型を T 、パターンマッチ式を囲むすべてのメソッドの型パラメータを a1,...,am とします。 各 ai に対し、Li をその下限境界、Ui をそのより高い境界(higher bound)とします。 各パターン p ∈ {p1 ,,...,pn} は、次の 2 つの方法で型付けされます。

最初に、その要請型である T で p の型付けを試みます。 もしそれが失敗するなら、T の代わりに、T 中の型パラメータ ai のすべての出現を**未定義**で置き換えて得られる、修正された要請型 T′ で p を型付けします。 もしこの 2 つめのステップも失敗するなら、実行時エラーとなります。 もし 2 つめのステップが成功したなら、式として見たパターン p の型を Tp とします。 このとき、最小の境界 L′1,...,L′m と最大の境界 U′1,...,U′m を、各 i に対し Li <: L′i かつ U′i <: Ui、そして次の制約システムが満たされるようなものとして決定します。

```
L1 <: a1 <: U1∧ ...∧ Lm <: am <: Um => Tp <: T
```

もしそのような境界が見つからないなら、実行時エラーとなります。 もしそのような境界が見つかれば、p で始まるパターンマッチ節はこのとき、各 ai が Li の代わりに下限境界 L′i を持ち、Ui の代わりに上限境界 U′i を持つという仮定の下で型付けされます。

どのブロック bi の要請型も、パターンマッチ式全体の要請型です。 パターンマッチ式の型はこのとき、全てのブロック bi の型の、弱い最少の上限境界(§3.5.3)です。

パターンマッチ式をセレクター値へ適用するとき、セレクター値(§8.1)にマッチするものが 1 つ見つかるまで次々にパターンが試みられます。 たとえば、この場合は case pi ⇨ bi です。 式全体の結果はこのとき、bi の評価結果です。 ここで pi のすべてのパターン変数は、セレクター値の対応する部分に束縛されます。 もしマッチするパターンが見つからなければ、scala.MatchError 例外が送出されます。

ケース中のパターンには、boolean 式 e を伴うガード接尾部 if e が続いても構いません。 ガード式は、もしケース中の先行するパターンがマッチするなら、評価されます。 もしガード式

が true に評価されるなら、パターンマッチは通常通り成功します。 もしガード式が false に評価されるなら、ケース中のパターンはマッチしないと考えられ、マッチするパターンの検索が継続されます。

効率化のため、パターンマッチ式の評価は原文の並びといくぶん異なる順序でパターンを試みることがあります。 これはガード中の副作用を通して、評価に影響を与えるかもしれません。 しかし、そのガードするパターンがマッチする場合に限り、ガード式が評価されることは保証されています。

もしパターンマッチのセレクタが sealed クラス(§5.2)のインスタンスなら、パターンマッチングのコンパイル時に、与えられたパターンセットが網羅的ではないと診断する警告が出ます。 すなわち、実行時に MatchError が発生する可能性があるということです。

Example 8.4.1

つぎの算術項の定義を考えてみます。:

```
abstract class Term[T]
case class Lit(x: Int) extends Term[Int]
case class Succ(t: Term[Int]) extends Term[Int]
case class IsZero(t: Term[Int]) extends Term[Boolean]
case class If[T](c: Term[Boolean],
                 t1: Term[T],
                 t2: Term[T]) extends Term[T]
```

数値リテラル、インクリメント、ゼロテスト、条件を表す項があります。 各項には、型パラメータとして、それが表す式の型 (Int あるいは Boolean) のいずれかが伴っています。 そのような項に対する型安全な評価子は、次のように書けます。

```
def eval[T](t: Term[T]): T = t match {
  case Lit(n)        => n
  case Succ(u)       => eval(u) + 1
  case IsZero(u)     => eval(u) == 0
  case If(c, u1, u2) => eval(if (eval(c)) u1 else u2)
}
```

評価子が、取り囲むメソッドの型パラメータがパターンマッチングを通して新しい境界を獲得できることを深く利用することに注意してください。 たとえば、2 つめのケース中のパターン Succ(u) の型は Int です。 それは、T に対して Int の上・下限境界を仮定する場合のみ、セレクタ型 T に適合します。 Int <: T <: Int の仮定の下、2 つめのケースの右側の型 Int が その要請型 T に適合することも確かめることができます。

8.5 パターンマッチング無名関数 (Pattern Matching Anonymous Functions)

構文:

```
BlockExpr ::= '{' CaseClauses '}'
```

無名関数は次のようなケースの並びで定義できます。

```
{ case p1 => b1 ... case pn => bn }
```

これは match が前につかずに、式として現われます。 このような式の要請型は、ある程度定義されなければなりません。 それは、ある k > 0 に対して scala.Functionk[S1,...,Sk , R] であるか、あるいは、scala.PartialFunction[S1, R] でなければなりません。 ここで 引数型 S1,...,Sk は完全に決まっていなければなりませんが、しかしその結果型 R は未決定でも構いません。

もしその要請型が scala.Functionk[S1,...,Sk, R] なら、式は次の無名関数に等価とみなされます。

```
(x1 : S1,...,xk : Sk) => (x1,...,xk) match {
  case p1 => b1 ... case pn => bn
}
```

ここで、各 xi は新規の名前です。 (§6.23)で見たように、この無名関数は今度は、次のインスタンス生成式に等価です。 ここで T は、すべての bi の型の弱い最少の上限境界です。

```
new scala.Functionk[S1,...,Sk, T] {
  def apply(x1 : S1,...,xk : Sk): T = (x1,...,xk) match {
    case p1 => b1 ... case pn => bn
  }
}
```

もしその要請型が scala.PartialFunction[S, R] なら、式は次のインスタンス生成式に等価とみなされます。

```
new scala.PartialFunction[S, T] {
    def apply(x : S): T = x match {
      case p1 => b1 ... case pn => bn
    }
    def isDefinedAt(x : S): Boolean = {
      case p1 => true ... case pn => true
      case _ => false
    }
}
```

ここで x は新規の名前、T はすべての bi の型の弱い最少の上限境界です。 isDefinedAt メソッド中の最後のデフォルトケースは、もしパターン p1,...,pn の 1 つがすでに変数あるいはワイルドカードパターンなら、除かれます。

Example 8.5.1次は、左畳み込み操作 /: を使うメソッドで、2 つのベクトルのスカラ積を計算します。:

```
def scalarProduct(xs: Array[Double], ys: Array[Double]) =
  (0.0 /: (xs zip ys)) {
    case (a, (b, c)) => a + b * c
  }
```

このコード中のケース節は、次の無名関数に等価です:

```
(x, y) => (x, y) match {
  case (a, (b, c)) => a + b * c
```

```
}
```

9 トップレベル定義（Top-Level Definitions）

9.1 コンパイル単位（Compilation Units）

構文:

```
CompilationUnit   ::=   {'package' QualId semi} TopStatSeq
TopStatSeq        ::=   TopStat {semi TopStat}
TopStat           ::=   {Annotation} {Modifier} TmplDef
                    |   Import
                    |   Packaging
                    |   PackageObject
                    |
QualId            ::=   id {'.' id}
```

コンパイル単位は、パッケージング、インポート節、クラスとオブジェクト定義、等の並びからなり、先行してパッケージ節があっても構いません。

次の、1 つ以上のパッケージ節で始まるコンパイル単位

```
package p1 ;
...
package pn ;
stats
```

は、次のパッケージングからなるコンパイル単位に等価です。

```
package p1 { ...
  package pn {
      stats
    } ...
}
```

すべてのコンパイル単位中に暗黙のうちにインポートされるものは、順に、パッケージ java.lang、パッケージ scala とオブジェクト scala.Predef（§12.5）です。 この順番の中で、後でインポートしたメンバーは、前でインポートしたメンバーを隠します。

9.2 パッケージング（Packagings）

構文:

```
Packaging        ::=   'package' QualId [nl] '{' TopStatSeq '}'
```

パッケージはメンバークラス、オブジェクトとパッケージの集合を定義する特別なオブジェクトです。 他のオブジェクトと異なり、パッケージは定義では導入されません。 その代わりに、パッケージのメンバーの集合は、パッケージングによって決定されます。

パッケージング `package p { ds }` は、`ds` 中のすべての定義をその限定修飾された名前が `p` であるパッケージに、メンバーとして注入します。 パッケージのメンバーは**トップレベル**定義と呼ばれます。 もし `ds` 中の定義が `private` と印されていれば、そのパッケージ中の他のメンバーからのみ可視となります。

パッケージング内では、パッケージ `p` のすべてのメンバーについて、それらの単純名が可視となります。 しかし、この規則は、パス `p` の前置子によって指定される、`p` が取り囲むパッケージのメンバーには拡張されません。

Example 9.2.1次のパッケージングが与えられているとします。

```
package org.net.prj {
  ...
}
```

パッケージ `org.net.prj` のすべてのメンバーは、その単純名が可視です。 しかしパッケージ `org` あるいは `org.net` のメンバーは、明示的な限定修飾あるいはインポートを必要とします。

`p` からの選択 `p.m` は、`p` からのインポートと同様、オブジェクトについてうまく機能します。しかし、他のオブジェクトと異なり、パッケージは値として使用できません。 モジュールあるいはクラス名と、完全修飾された同じ名前のパッケージは不正です。

パッケージング外でのトップレベル定義は、特別な空パッケージに注入されるとみなされます。 このパッケージは、名前を付けることはできず、したがってインポートできません。 しかし、空パッケージのメンバーは互いに限定修飾なしで可視です。

9.3 パッケージオブジェクト（Package Objects）

構文:

```
PackageObject ::=   'package' 'object' ObjectDef
```

パッケージオブジェクト `package object p extends t` は、テンプレート `t` のメンバーをパッケージ `p` に加えます。 パッケージ毎に、ただ 1 つのパッケージオブジェクトが可能です。 標準的な命名規則では、上記の定義を、パッケージ `p` に直接対応するディレクトリ中の `package.scala` という名前のファイルに置きます。

パッケージオブジェクトは、パッケージ `p` 中で定義されたトップレベルオブジェクトあるいはクラスの 1 つと同じ名前のメンバーを定義すべきではありません。 名前の衝突がある場合、プ

ログラムの振る舞いは現在未定義です。 この制限は Scala の将来のバージョンで取り上げられることが期待されます。

9.4 パッケージ参照（Package References）

構文:

```
QualId              ::=   id {'.' id}
```

パッケージへの参照は、限定修飾された識別子の形をとります。 他のすべての参照と同じように、パッケージ参照は相対的です。 すなわち、名前 p で始まるパッケージ参照は、名前 p のメンバーを定義する、最も近く取り囲むスコープが検索されます。

事前定義された特別な名前 _root_ は、すべてのトップレベルのパッケージを含む、最も外側のルートパッケージを参照します。

Example 9.4.1 次のプログラムを考えます。:

```
package b {
  class B
}

package a.b {
  class A {
    val x = new _root_.b.B
  }
}
```

ここで、参照 _root_.b.B は、トップレベルパッケージ b 中のクラス B を参照します。 もし _root_ 前置子が省略されたなら、名前 b は代わりに、パッケージ a.b へ変換されます。 そしてそのパッケージもクラス B を含まないなら、コンパイルエラーとなります。

9.5 プログラム（Programs）

プログラムは、型 (Array[String])Unit のメンバーメソッド main を持つトップレベルのオブジェクトです。 プログラムはコマンドシェルから実行できます。 プログラムのコマンド引数は、型 Array[String] のパラメータとして main メソッドへ渡されます。

プログラムの main メソッドは、オブジェクト中で直接定義するか、あるいは、それを継承できます。 Scala ライブラリでは、空の継承された main メソッドを定義する、クラス scala.Application を定義しています。 このクラスを継承するオブジェクト m は 1 つのプログラムとなり、オブジェクト m の 初期化コードを実行します。

Example 9.5.1

次の例は、モジュール test.HelloWorld の中でメソッド main を定義し、hello world プログラムを生成します。

このプログラムはコマンドで起動できます。

```
scala test.HelloWorld
```

Java 環境では、次のコマンド

```
java test.HelloWorld
```

は、同様にうまく働くでしょう。

HelloWorld は main メソッドがなくても、その代わりに Application を継承することで同様に定義できます。

```
package test
object HelloWord extends Application {
  println("hello world")
}
```

10 XML 式とパターン (XML Expressions and Patterns)

By Burak Emir

この章では XML 式とパターンの文法構造を記述します。 可能な限りしっかりと XML 1.0 仕様書[W3C]および、今後の Scala コードの埋め込みで要求される変更に従います。

10.1 XML 式 (XML Expressions)

XML 式は次の文法規則によって生成される式です。ここで最初の要素の 始め山括弧 '<' は、字句上の XML モード(§1.5)を開始できる位置になければなりません。

構文:

```
XmlExpr     ::= XmlContent {Element}
```

XML 仕様書の整形式制約が適用されます。 その意味は例えば、開始タグと終了タグはマッチしなければならず、属性は一度だけ定義できる、ただしエンティティ解決にかかわる制約の例外がある、ということです。

次の文法規則は Scala の XML を記述しており、W3C XML 標準規格に可能な限り近く設計されています。 属性値と文字データの文法規則だけは変えられています。 Scala は、宣言、CDATA セクションあるいは処理命令をサポートしていません。 エンティティ参照は、実行時には解決されません。

構文:

```
Element         ::=      EmptyElemTag
                  |      STag Content ETag

EmptyElemTag    ::=     '<' Name {S Attribute} [S] '/>'

STag            ::=     '<' Name {S Attribute} [S] '>'
ETag            ::=     '</' Name [S] '>'
Content         ::=     [CharData] {Content1 [CharData]}
Content1        ::=     XmlContent
                  |     Reference
                  |     ScalaExpr
XmlContent      ::=     Element
                  |     CDSect
                  |     PI
                  |     Comment
```

もし XML 式がただ 1 つの要素なら、その値は XML ノード(scala.xml.Node のサブクラスのインスタンス)の実行時表現です。 もし XML 式が 1 つより多くの要素から成るなら、その値は X

ML ノードのシーケンス (scala.Seq[scala.xml.Node] のサブクラスのインスタンス)の実行時表現です。

もし XML 式がエンティティ参照、CDATA セクション、処理命令あるいはコメントなら、それは対応する Scala 実行時クラスのインスタンスによって表現されます。

デフォルトでは、要素コンテンツ中の始まりとお終いの空白文字は削除され、連続する空白文字はただ 1 つのスペース文字 \u0020 で置き換えられます。 この振る舞いはコンパイラ・オプションで、すべての空白文字を維持するように変えることができます。

構文:

```
Attribute   ::=   Name Eq AttValue

AttValue    ::=   '"' {CharQ | CharRef} '"'
                | ''' {CharA | CharRef} '''
                | ScalaExpr

ScalaExpr   ::=   Block

CharData    ::=  { CharNoRef } without {CharNoRef}'{'CharB {CharNoRef}
                                 and without {CharNoRef}']]>'{CharNoRef}
```

XML 式は、Scala 式を属性値としてあるいはノード内に含めることができます。 後者の場合、それらは 1 つの左波括弧 '{' を使って埋め込まれ、右波括弧 '}' で終わります。 CharData で生成された時の XML テキスト中で 1 つの左波括弧を表すには、それを 2 重にします。 ですから、'{{' は XML テキスト '{' を表し、埋め込まれた Scala 式を導入しません。

構文:

```
BaseChar, Char, Comment, CombiningChar, Ideographic, NameChar, S, Reference
                ::= "W3C XML と同様"

Char1           ::=   Char without '<' | '&'
CharQ           ::=   Char1 without '"'
CharA           ::=   Char1 without '''
CharB           ::=   Char1 without '{'

Name            ::=   XNameStart {NameChar}

XNameStart      ::=   '_' | BaseChar | Ideographic
                   (W3C XMLと同じだが ':'なし
```

10.2 XML パターン (XML Patterns)

XML パターンは次の文法規則によって生成されるパターンです。 ここで要素パターンの始め山括弧 '<' は、字句上の XML モード(§1.5)を開始できる位置になければなりません。

構文:

```
XmlPattern ::= ElementPattern
```

XML 仕様書の整形式制約が適用されます。

XML パターンは、ただ 1 つの要素パターンでなければなりません。 それは、パターンによって記述されたのと同じ構造を持つ XML ツリーの実行時表現と正確にマッチします。 XML パターンは Scala パターン(§8.4)を含んでいても構いません。

空白文字は XML 中と同じ方法で扱われます。 エンティティ参照、CDATA セクション、処理命令とコメント等のパターンは、同じ実行時表現とマッチします。

デフォルトでは、要素コンテンツ中の始まりとお終いの空白文字は削除され、連続する空白文字はただ 1 つのスペース文字 \u0020 で置き換えられます。 この振る舞いはコンパイラ・オプションで、すべての空白文字を維持するように変えることができます。

構文:

```
ElemPattern   ::=    EmptyElemTagP
                |    STagP ContentP ETagP

EmptyElemTagP ::=    '<' Name [S] '/>'
STagP         ::=    '<' Name [S] '>'
ETagP         ::=    '</' Name [S]'>'
ContentP      ::=    [CharData] {(ElemPattern|ScalaPatterns) [CharData]}

ContentP1     ::=   ElemPattern
                |   Reference
                |   CDSect
                |   PI
                |   Comment
                |   ScalaPatterns
ScalaPatterns ::=   '{' Patterns '}'
```

11 ユーザー定義アノテーション（User-Defined Annotations）

構文:

```
Annotation       ::=       '@' SimpleType {ArgumentExprs}
ConstrAnnotation ::=       '@' SimpleType ArgumentExprs
```

ユーザー定義アノテーションは、定義とメタ情報を関連づけます。 単純なアノテーションは、@c あるいは @c(a1,...,an) の形をしています。 ここで、c はクラス C のコンストラクタで、クラス scala.Annotation に適合しなくてはなりません。

アノテーションは、定義または宣言、型、あるいは式に適用できます。 定義または宣言のアノテーションは、その定義の前に現われます。 型のアノテーションは、その型の後に現れます。式 e のアノテーションは、式 e の後にコロンによって分離されて現われます。 1 つ以上のアノテーション節をエンティティに適用できます。 与えられたアノテーションの順番は重要ではありません。

例:

```
@serializable class C { ... }       //   クラスアノテーション
@transient @volatile var m: Int     //   変数アノテーション
String @local                       //   型アノテーション
(e: @unchecked) match { ... }       //   式アノテーション
```

アノテーション節の意味は処理系依存です。 Java プラットフォームでは、次のアノテーションは標準的な意味を持っています。

`@transient`

フィールドが非永続的であるとマークします。; これは Java の transient 修飾子に等価です。

`@volatile`

プログラムの制御外でそのフィールド値が変化する可能性があるとマークします。; これは、Java の volatile 修飾子に等価です。

`@serializable`

クラスがシリアライズ可能であるとマークします。; これは Java の java.io.Serializable インターフェースの継承に等価です。

`@SerialVersionUID(<longlit>)`

シリアルバージョン識別子(長い定数)をクラスに付属させます。 これは Java の次のフィールド定義に等価です。:

```
private final static  SerialVersionUID = <longlit>
```

`@throws(<classlit>)`

Java コンパイラは、メソッド/コンストラクタの実行で生じる可能性のある チェック例外を分析し、 プログラムがそれらチェック例外に対するハンドラーを含んでいるか確かめます。 起きる可能性のある各チェック例外に対して、 メソッド/コンストラクタの throw 節では、 その例外クラスあるいはその例外クラスのスーパークラスの 1 つに言及しなくてはなりません。

`@deprecated(<stringlit>)` (廃止)

定義が廃止であるとマークします。 定義されたエンティティへのアクセスで、コンパイラはメッセージ <stringlit> に言及する廃止警告を発行します。 廃止警告は、deprecated と印された定義自身に属するコードでは発行されません。

`@scala.reflect.BeanProperty`

このアノテーションがある変数 X の定義の前に置かれたときは、Java bean スタイルのゲッターおよびセッターメソッド getX、setX を、 変数 X を含むクラスに付け加えます。 変数の最初の文字は大文字化されて get/set の後に現れます。 このアノテーションがイミュータブルな値定義 X の定義に付加されたときは、 ゲッターだけが生成されます。 これらメソッドの組立はコード生成の一部です。; ですからそれらメソッドは、それを含むクラスを生成するための classfile にただ一度見えるだけです。

`@scala.reflect.BooleanBeanProperty`

このアノテーションは、生成されたゲッターメソッドが getX の代わりに isX と名付けられる点を除けば、scala.reflect.BeanProperty と同じです。

`@unchecked`

この属性は、マッチ式のセレクタに適用されたとき、 そうでなければ発せられるはずの、非網羅的パターンマッチに関するあらゆる警告を止めます。 例えば、次のメソッド定義に対しては警告はないでしょう。

```
def f(x: Option[Int]) = (x: @unchecked) match {
  case Some(y) => y
}
```

@unchecked アノテーションがなければ、 Scala コンパイラはパターンマッチが非網羅的であると推論し、 Option が sealed クラスなので警告を発します。

`@uncheckedStable`

値宣言/定義に適用すると、たとえその型が volatile(§3.6)であっても、 パス中でその定義された値を使えます。例えば、次のメンバー定義は正しいです:

```
type A { type T }
type B
@uncheckedStable val x: A with B // volatile型
val y: x.T                       // OK since 'x' s still a path
```

@uncheckedStable アノテーションがなければ、型 A with B は volatile なので、 指定子 x はパスではありません。ですから、参照 x.T は不正となります。

このアノテーションを非 volatile 型を持つ値宣言/定義に適用しても効果はありません。

`@specialized`

このアノテーションを型パラメータの定義に適用すると、 コンパイラはプリミティブ型に特化した定義を生成します。 プリミティブ型のオプションリストが与えることができ、その場合、特化は、 それらの型だけについて考慮されます。 例えば、次のコードは Unit、Int と Double に対して、特化したトレイトを生成します。

```
trait Function0[@specialized(Unit, Int, Double) T] {
  def apply: T
}
```

式の静的な型が、定義の特化した変位指定(variant)と一致するときはいつでも、 コンパイラは代わりに、特化したバージョンを使います。 実装のさらなる詳細については [Dra10] を見てください。

他のアノテーションは、プラットフォームあるいはアプリケーション依存のツールによって解釈されるでしょう。 クラス scala.Annotation には、これらアノテーションの保持方法を示すのに使われる 2 つのサブトレイトがあります。

トレイト scala.ClassfileAnnotation を継承するアノテーションクラスのインスタンスは、生成されたクラスファイル内に格納されます。 トレイト scala.StaticAnnotation を継承するアノテーションクラスのインスタンスは、アノテーション付きのシンボルをアクセスするすべてのコンパイル単位中で、Scala の型チェックから可視です。 アノテーションクラスは、scala.ClassfileAnnotation と scala.StaticAnnotation の両方を継承できます。 もしアノテーションクラスが scala.ClassfileAnnotation と scala.StaticAnnotation のいずれも継承しないなら、そのインスタンスは、それらを解析するコンパイル実行中だけローカルに可視です。

scala.ClassfileAnnotation を継承するクラスは、それらをホスト環境にマップできることを保証するために、さらに制限を受けるかもしれません。 特に、Java と.NET プラットフォームの両方において、そのようなクラスはトップレベルでなければなりません。; つまり、それらを他のクラス/オブジェクトに含めることはできません。 さらに、Java と.NET 両方において、すべてのコンストラクタ引数は定数式でなければなりません。

12 Scala 標準ライブラリ（The Scala Standard Library）

Scala 標準ライブラリは、多数のクラスとモジュールを備えた scala パッケージからなります。 以下で、それらのいくつかについて説明します。

12.1 ルートクラス（Root Classes）

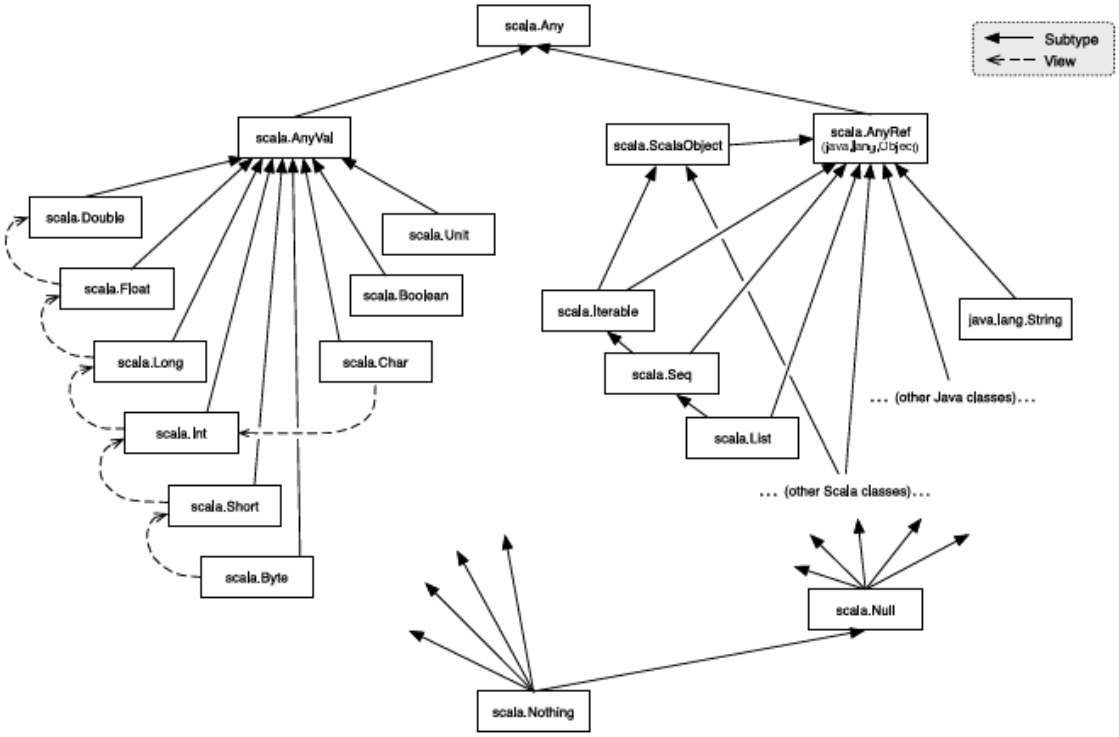


図 12.1 は Scala のクラス階層を示しています。

この階層構造のルートは、クラス Any です。 Scala 実行環境中のすべてのクラスは、このクラスを直接あるいは間接に継承しています。 クラス Any は、直接のサブクラスを 2 つ持っています: AnyRef と AnyVal です。

サブクラス AnyRef は、ホストシステム中でオブジェクトとして表されるすべての値を表現します。 ユーザーが定義するすべての Scala クラスは、直接あるいは間接にこのクラスを継承します。 さらに、すべてのユーザー定義 Scala クラスは、トレイト scala.ScalaObject も継承します。 他の言語で書かれたクラスは、scala.ScalaObject ではなく、scala.AnyRef を継承します。

クラス AnyVal は、ホストシステム中でオブジェクトとして実装されない値を記述する、固定数のサブクラスを持っています。

クラス AnyRef と AnyVal は、クラス Any 中で宣言されたメンバーだけを提供する必要がありますが、しかし実装では、ホスト特有のメソッドをそれらクラスに加えることがあります(例えば、実装では、クラス AnyRef を自身のオブジェクトのルートクラスと同一視するかもしれません)。

これらルートクラスのシグニチャは、次の定義で記述されます。

```
package scala
/** The universal root class */
/** 普遍的なルートクラス*/
abstract class Any {

  /** equality定義; ここでは抽象*/
  def equals(that: Any): Boolean

    /** Semantic equality between values */
    /** 値の間のセマンティックな等価性 */
    final def == (that: Any): Boolean =
      if (null eq this) null eq that else this equals that

    /** Semantic inequality between values */
    /** 値の間の、セマンティックな非等価性 */
    final def != (that: Any): Boolean = !(this == that)

    /** Hash code; abstract here */
    /** ハッシュコード;ここでは抽象*/
    def hashCode: Int = ...

    /** Textual representation; abstract here */
    /** テキスト表現;ここでは抽象*/
    def toString: String = ...

    /** Type test; needs to be inlined to work as given */
    /** 型テスト;与動作にはインライン展開要 */
    def isInstanceOf[a]: Boolean

    /** Type cast; needs to be inlined to work as given */ */
    /** 型キャスト;与動作にはインライン展開要 */
    def asInstanceOf[A]: A = this match {
      case x: A => x
      case _ => if (this eq null) this
                else throw new ClassCastException()
    }
}

/** すべての値型のルートクラス */

final class AnyVal extends Any

/** 全ての参照型のルートクラス */
class AnyRef extends Any {
  def equals(that: Any): Boolean      = this eq that
  final def eq(that: AnyRef): Boolean = ... // 参照等価性
```

```
  final def ne(that: AnyRef): Boolean = !(this eq that)

    def hashCode: Int = ...    // hashCode は割当てられたアドレスから計算
    def toString: String = ... // toString は hashCode とクラス名から計算

/** すべてのユーザ定義 Scala クラスに対するミックスインクラス */
trait ScalaObject extends AnyRef
```

型テスト x.isInstanceOf[T] は、次の型付きパターンマッチに等価です。

```
x match {
  case _: T´ => true
  case _ => false
}
```

where the type T´ is the same as T except if T is of the form D or D[tps] , where D is a type member of some outer class C . In this case T´ is C#D (or C#D[tps], respectively), whereas T itself would expand to C.this.D[tps]. In other words, an isInstanceOf test does not check for the

ここで 型 T´ は、T が D あるいは D[tps] の形（D はある外側のクラス C の型メンバー）である場合を除き、T と同じ。 この場合 T´は C#D(あるいは、D[tps]に対しては C#D[tps])であるのに対して、T 自身は C.this.D[tps] に拡張されます。 言い替えれば、isInstanceOf テストは...をチェックしません。

もし T が数値型(§12.2)なら、テスト x.asInstanceOf[T] は特別に扱われます。 この場合、キャストは変換メソッド x.toT(§12.2.1)の適用に翻訳されます。 非数値 x に対する操作は Class CastException を引き起こします。

12.2 値クラス（Value Classes）

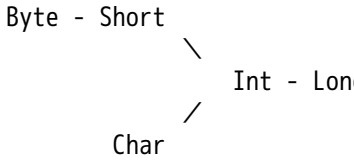
値クラスは、そのインスタンスがホストシステムではオブジェクトとして表されないクラスです。 すべての値クラスはクラス AnyVal を継承します。 Scala の実装は、値クラス Unit、Boolean、Double、Float、Long、Int、Char、 Short とByte を提供する必要があります（しかし、同様に他のものを供給するのは自由です)。 これらクラスのシグニチャは、以降で定義されています。

12.2.1 数値型（Numeric Value Types）

Numeric value types are ranked in the following partial order:

クラス Double、Float、Long、Int、Char、Short と Byte はまとめて**数値型**と呼ばれます。クラス Byte、Short、あるいは Char は、**部分領域型**と呼ばれます。Int や Long と同様、部分領域型は**整数型**と呼ばれ、他方、Float と Double は**浮動小数点型**と呼ばれます。

数値型は、次の半順序でランク付けされます:



Byte と Short はこの順序中の最も低いランク型であるのに対して、 Double は最も高いランクです。ランク付けは適合関係 (§3.5.2)を意味**しません。**; たとえば、インスタンス Int は Long のサブ型ではありません。 しかし、オブジェクト Predef(§12.5)では、すべての数値型から、すべてのより高いランクの数値型へのビュー(§7.3)を定義しています。 ですから、コンテキストによって必要とされるとき、より低いランクの型は暗黙のうちにより高いランクの型へ変換されます(§6.26)。

与えられた 2 つの数値型 S と T に対し、S と T の**演算型**は次のように定義されます。:もし S と T の両方が部分領域型なら、 S と T の演算型は Int です。 そうでなければ S と T の演算型は、2 つの型のうちランクに関してより高い方です。 与えられた 2 つの数値 v と w に対して、v と w の演算型は、それらの実行時型の演算型です。

すべての数値型 T は、次のメソッドをサポートします。

- 等しい (==)、等しくない(!=)、小さい (<)、大きい(>)、 小さいか等しい (<=)、大きいか等しい (>=) 等のための比較メソッド。 それぞれに 7 つのオーバーロードされた代替物があります。 各代替物は、何らかの数値型パラメータをとります。 その結果型は Boolean 型です。演算は、 レシーバとその引数をそれらの演算型へ変換し、 その型の与えられた比較演算を実行することで、評価されます。

- 算術演算メソッドの、加算 (+)、減算 (-)、乗算 (*)、除算 (/) と 剰余 (%)。 それぞれに 7 つのオーバーロードされた代替物があります。 各代替物は、何らかの数値型 U をもつパラメータをとります。 その結果型は T と U の演算型です。演算は、 レシーバとその引数をそれらの演算型へ変換し、 その型の与えられた算術演算を実行することで、評価されます。

- パラメータなしの算術演算メソッド、同値 (+) と符号反転 (-)。 結果型は T です。 1 つめのメソッドは変わらないレシーバを返すのに対し、 2 つめはその符号反転を返します。

- 変換メソッドの、toByte、toShort、toChar、toInt、toLong、toFloat、toDouble。 これらは、Java の数値型キャスト操作規則を使って、 レシーバオブジェクトをターゲット型に変換します。 変換は数値を切り落とすかもしれません(Long から Int、 あるいは Int から Byte へ変換する時)。 あるいはまた、変換により精度が失われるかもしれません (Double から Float、あるいは Long と Float 間で変換するとき)。

整数値型は、さらに次の演算をサポートしています。:

- Bit manipulation methods bitwise-and (&), bitwise-or |, and bitwiseexclusive -or (^), which each exist in 5 overloaded alternatives . Each alternative takes a parameter of some integer numeric value type . Its result type is the operation type of T and U . The operation is evaluated by converting the receiver and its argument to their operation type and performing the given bitwise operation of that type .
- ビット操作メソッドの、ビット単位 AND (&)、ビット単位 OR (|)、 ビット単位 XOR (^)。 それぞれに 5 つのオーバーロードされた代替物があります。 各代替物はいくつかの整数値型パラメータをとります。 その結果型は T と U の演算型です。演算は、 レシーバとその引数をそれらの演算型へ変換し、 その型の与えられたビット単位演算を実行することで、評価されます。

- パラメータなしの、ビット反転メソッド(~)。 その結果型は、レシーバの型 T あるいは Int の、どちらか大きい方です。 演算は、レシーバをその結果型へ変換し、 その値中の各ビットを反転させることで、評価されます。

- ビットシフトメソッドの、左シフト (<<)、算術右シフト (>>)、 符号無し右シフト(>>>)。 これらの各メソッドは、 それぞれ 2 つのオーバーロードされた代替物を持っていて、 Int 型あるいは Long型のパラメータ n をとります。 演算の結果型は、レシーバの型 T あるいは Int の、どちらか大きい方です。 演算は、レシーバを結果型へ変換し、 指定された n ビットシフトを実行することで、評価されます。

数値型は、クラス Any の操作 equals、hashCode、toString も実装しています。

equals メソッドは、引数が数値型であるかどうかテストします。 もしそれが真なら、その型に適した == 演算を実行します。 すなわち、数値型の equals メソッドは、次のように定義されていると考えられます。

```
def equals(other: Any): Boolean = other match {
  case that: Byte   => this == that
  case that: Short  => this == that
  case that: Char   => this == that
  case that: Int    => this == that
  case that: Long   => this == that
  case that: Float  => this == that
  case that: Double => this == that
  case _ => false
}
```

hashCode メソッドは、等しい数値を等しい結果にマップする整数 hashcode を返します。 それは、Int 型とすべての部分領域型に対して同一性(identity)を保証します。

toString メソッドは、そのレシーバを整数あるいは浮動小数点数として表示します。

Example 12.2.1 例として、数値型 Int のシグニチャを示します。

```
package scala
abstract sealed class Int extends AnyVal {
    def == (that: Double): Boolean // double equality
    def == (that: Float): Boolean   // float equality
    def == (that: Long): Boolean    // long equality
    def == (that: Int): Boolean     // int equality
    def == (that: Short): Boolean   // int equality
    def == (that: Byte): Boolean    // int equality
    def == (that: Char): Boolean    // int equality
    /* analogous for !=, <, >, <=, >= */

    def + (that:   Double): Double     //   double addition
    def + (that:   Float): Double      //   float addition
    def + (that:   Long): Long         //   long addition
    def + (that:   Int): Int           //   int addition
    def + (that:   Short): Int         //   int addition
    def + (that:   Byte): Int          //   int addition
    def + (that:   Char): Int          //   int addition
    /* analogous   for -, *, /, % */
```

```
    def & (that:   Long): Long         //   long bitwise and
    def & (that:   Int): Int           //   int bitwise and
    def & (that:   Short): Int         //   int bitwise and
    def & (that:   Byte): Int          //   int bitwise and
    def & (that:   Char): Int          //   int bitwise and
    /* analogous   for |, ^ */

    def << (cnt: Int): Int             // int left shift
    def << (cnt: Long): Int            // long left shift
    /* analogous for >>, >>> */

    def unary_+ : Int                  // int identity
    def unary_- : Int                  // int negation
    def unary_~ : Int                  // int bitwise negation

    def   toByte: Byte                 //   convert   to   Byte
    def   toShort: Short               //   convert   to   Short
    def   toChar: Char                 //   convert   to   Char
    def   toInt: Int                   //   convert   to   Int
    def   toLong: Long                 //   convert   to   Long
    def   toFloat: Float               //   convert   to   Float
    def   toDouble: Double             //   convert   to   Double
}
```

12.2.2 クラス Boolean (Class Boolean)

クラス Boolean は、ただ 2 つの値を持っています。: true と false です。 それは、次のクラス定義で与えられるのと同等の演算を実装しています。

```
package scala
abstract sealed class Boolean extends AnyVal {
  def && (p: => Boolean): Boolean =   // boolean and
    if (this) p else false
  def || (p: => Boolean): Boolean =   // boolean or
    if (this) true else p
  def & (x: Boolean): Boolean =       // boolean strict and
    if (this) x else false
  def | (x: Boolean): Boolean =       // boolean strict or
    if (this) true else x
  def == (x: Boolean): Boolean =      // boolean equality
    if (this) x else x.unary_!
  def != (x: Boolean): Boolean        // boolean inequality
    if (this) x.unary_! else x
  def unary_!: Boolean                // boolean negation
    if (this) false else true
}
```

このクラスはクラス Any の操作 equals、hashCode と toString も実装します。

equals メソッドは、もし引数がレシーバと同じ boolean 値なら true を返し、そうでなければ false を返します。 hashCode メソッドは、true上で起動されたときには、 固定された、実装固有のハッシュコードを返し、false上で起動されたときには、異なる、固定された、実装固有のハッシュコードを返します。 toString メソッドは、文字列に変換されたレシーバを返します、すなわち、"true" あるいは "false" のいずれかを返します。

12.2.3 クラス Unit (Class Unit)

クラス Unit は、ただ 1 つの値を持っています。: ()です。 これは、クラス Any の equals、hashCode、と toStringt のただ 3 つのメソッドを実装しています。

equals メソッドは、もし引数が unit 値 () なら true を返し、そうでなければ false を返します。 hashCode メソッドは、固定された、実装固有のハッシュコードを返します。 toString メソッドは、"()" を返します。

12.3 標準的な参照クラス (Standard Reference Classes)

この節では、Scala コンパイラ中で特別な方法で扱われる、いくつかの標準的な Scala 参照クラスを紹介します。 Scala はそれらのために糖衣構文を提供するか、あるいは、Scala コンパイラは操作ための特別なコードを生成します。 標準 Scala ライブラリ中の他のクラスは、Scala ライブラリドキュメント中に HTML ページの形で文書化されています。

12.3.1 クラス String (Class String)

Scala の文字列クラスは通常、ホストシステムの標準的な String クラスから派生されます(それと同一視されているかもしれません)。 Scala クライアントに対して、クラスはそれぞれの場合に応じて、次のメソッドをサポートするとされます。

```
def + (that: Any): String
```

これは、その左オペランドと右オペランドのテキスト表現を連結します。

12.3.2 タプルクラス (The Tuple Classes)

Scala は、n = 2,...,9 に対して、タプルクラス Tuplen を定義しています。 それらは次のように定義されています。

```
package scala
case class Tuplen[+a_1,..., +a_n](_1: a_1,..., _n : a_n) {
  def toString = "(" ++ _1 ++ "," ++ ... ++ "," ++ _n ++ ")"
}
```

暗黙のうちにインポートされた predef(事前定義済み)オブジェクト (§12.5)は、Pair という名前の Tuple2 のエイリアスと、 Tripleという名前の Tuple3 のエイリアスを定義しています。

12.3.3 関数クラス (The Function Classes)

Scala は、n = 1,...,9 に対して、関数クラス Functionn を定義しています。 それらは次のように定義されています。

```
package scala
trait Functionn[-a_1,..., -a_n , +b] {
  def apply(x_1: a_1,..., x_n : a_n): b
  def toString = "<function>"
}
```

Function1 のサブクラスは部分関数を表し、そのドメインのいくつかの点で未定義です。 関数の apply メソッドに加えて、部分関数も isDefined メソッドを持っており、それは、与えられた引数で関数が定義されているかどうかを答えます。:

```
class PartialFunction[-A, +B] extends Function1[A, B] {
  def isDefinedAt(x: A): Boolean
}
```

暗黙のうちにインポートされる predefオブジェクト (§12.5)は、名前 Function を Function1 のエイリアスと定義しています。

12.3.4 クラス Array (Class Array)

ジェネリックな配列クラスは、次のように与えられています。

```
final   class Array[A](len: Int) extends Seq[A] {
  def   length: Int = len
  def   apply(i: Int): A = ...
  def   update(i: Int, x: A): Unit = ...
  def   elements: Iterator[A] = ...
  def   subArray(from: Int, end: Int): Array[A] = ...
  def   filter(p: A => Boolean): Array[A] = ...
  def   map[B](f: A => B): Array[B] = ...
  def   flatMap[B](f: A => Array[B]): Array[B] = ...
}
```

もし T が型パラメータあるいは抽象型でないなら、型 Array[T] はホストシステム中のネイティブな配列型 []T として表現されます。 そのような場合、length は配列の長さを返し、apply は添え字指定を意味し、update は要素の更新を意味します。

apply と update 操作(§6.26)の糖衣構文のおかげで、配列 xs の操作について、Scala と Java/C# のコード間に次の対応があります。:

```
Scala               Java/C#
xs.length           xs.length
xs(i)               xs[i]
xs(i) = e           xs[i] = e
```

配列は、イテレータ中の配列のすべての要素を返す elements メソッドを定義することで、シーケンストレイト scala.Seq も実装しています。

Scala におけるパラメータ化された型と、ホスト言語における配列のアドホックな実装間の緊張関係のため、配列を取り扱うとき、いくつかの微妙な点を考慮する必要があります。 それらを次に説明します。

最初に、Java あるいは C# 中の配列と異なり、Scala 中の配列は共変**ではありません。**;すなわち、Scalaでは、 S <: T が Array[S] <: Array[T] を意味しません。 しかし、S の配列をキャストして T の配列にすることは、もしそのようなキャストがホスト環境で許されるなら、可能です。

たとえば、String は Objectに適合しますが、インスタンス Array[String] は Array[Object] に適合しません。 しかし、型 Array[String] の式を型 Array[Object] へキャストすることは可能であり、このキャストは ClassCastException を引き起こすことなく成功します。 例:

```
val xs = new Array[String](2)
// val ys: Array[Object] = xs   // **** error: 非互換な型
val ys: Array[Object] = xs.asInstanceOf[Array[Object]] // OK
```

第二に、その要素型として型パラメータあるいは抽象型 T を持つ**多相的配列**に対して、[]T と異なる表現が使われることがあります。 しかし、isInstanceOf と asInstanceOf があたかも配列が単相的配列の標準的な表現を使うかのように、それでも動作することは保証されています。:

```
val ss = new Array[String](2)

def f[T](xs: Array[T]): Array[String] =
  if (xs.isInstanceOf[Array[String]]) xs.asInstanceOf[Array[String])
  else throw new Error("not an instance")

f(ss)                                          // ss を返す
```

多相的配列用に選ばれた表現は、多相的配列の生成が期待どおり動作することも保証しています。次の例はメソッド mkArray の実装で、その要素を定義する T のシーケンスを与えられ、任意の型 T の配列を生成します。

```
def mkArray[T](elems: Seq[T]): Array[T] = {
  val result = new Array[T](elems.length)
  var i = 0
  for (elem <- elems) {
    result(i) = elem
    i += 1
  }
}
```

Java の配列の型消去モデルの下では、上のメソッドが期待通りには動作しない --- 実際、それは常に Object の配列を返す --- ことに注意してください。

第三に、Java 環境ではメソッド System.arraycopy があります。 それは、2 つのオブジェクトをパラメータにとると共に、開始インデックスと長さの引数を もち、オブジェクトは互換の要素型からなる配列であるとして、1 つのオブジェクトから他方へ要素をコピーします (訳注:arraycopy(Object src, int srcPos, Object dest, int destPos, int length))。 System.arraycopy は、Scala の多相的配列に対しては機能しません。 なぜなら、それらの表現が異なるからです。 その代わりに Array.copy を使うべきです。 それはクラス Array のコンパニオンオブジェクト中で定義されています。 このコンパニオンオブジェクトも、配列上のパターンマッチングを可能にする抽出子メソッド unapplySeq(§8.1.8)と同様、配列のために種々のコンストラクタメソッドを定義しています。

```
package scala
object Array {
  /** 'src' から 'dest'へ配列要素をコピー */
  def copy(src: AnyRef, srcPos: Int,
           dest: AnyRef, destPos: Int, length: Int): Unit = ...

    /**  1 つの配列へ、すべての引数配列を連結 */
    def concat[T](xs: Array[T]*): Array[T] = ...

    /** 連続した整数からなる配列を生成 */
    def range(start: Int, end: Int): Array[Int] = ...

    /** 与えられた要素の配列を生成 */
    def apply[A <: AnyRef](xs: A*): Array[A] = ...

    /**   Analogous   to above. */
    def   apply(xs: Boolean*): Array[Boolean]   =   ...
    def   apply(xs: Byte*)   : Array[Byte]      =   ...
    def   apply(xs: Short*)  : Array[Short]     =   ...
    def   apply(xs: Char*)   : Array[Char]      =   ...
    def   apply(xs: Int*)    : Array[Int]       =   ...
    def   apply(xs: Long*)   : Array[Long]      =   ...
    def   apply(xs: Float*)  : Array[Float]     =   ...
    def   apply(xs: Double*) : Array[Double]    =   ...
    def   apply(xs: Unit*)   : Array[Unit]      =   ...

    /** 要素の複数のコピーを含む配列の生成 */
    def make[A](n: Int, elem: A): Array[A] = {

    /** 配列上のパターンマッチングを可能とする */
    def unapplySeq[A](x: Array[A]): Option[Seq[A]] = Some(x)
}
```

Example 12.3.1次のメソッドは、与えられた引数配列を複写し、オリジナルと複写したものからなる、ペアを返します:

```
def duplicate[T](xs: Array[T]) = {
  val ys = new Array[T](xs.length)
  Array.copy(xs, 0, ys, 0, xs.length)
  (xs, ys)
```

```
}
```

12.4 クラス Node (Class Node)

```
package scala.xml

trait Node {

  /** このノードのラベル */
  def label: String

  /** axis 属性 */

  def attribute: Map[String, String]

  /** 子 axis (このノードの全ての子) */
  def child: Seq[Node]

  /** 子孫 axis (このノードの全ての子孫) */
  def descendant: Seq[Node] = child.toList.flatMap {
    x => x::x.descendant.asInstanceOf[List[Node]]
  }

  /** 子孫 axis (このノードの全ての子孫) */
  def descendant_or_self: Seq[Node] = this::child.toList.flatMap {
    x => x::x.descendant.asInstanceOf[List[Node]]
  }

  override def equals(x: Any): Boolean = x match {
    case that:Node =>
      that.label == this.label &&
        that.attribute.sameElements(this.attribute) &&
          that.child.sameElements(this.child)
    case _ => false
  }

  /** XPath スタイルの射影関数(projection function)。'that'と印された、
   *  このノードの全ての子を返す。ドキュメントの順番は維持される。
   */
     def \(that: Symbol): NodeSeq = {
       new NodeSeq({
         that.name match {
           case "_" => child.toList
           case _ =>
             var res:List[Node] = Nil
             for (x <- child.elements if x.label == that.name) {
               res = x::res
             }
             res.reverse
         }
       })
     }

  /** XPath スタイルの射影関数。'descendat_or_self'(子孫または自身) axis
   * から、'that'と印されたこのノードの全ての子を返す。
```

```
   * ドキュメントの順番は維持される。
   */
   def \\(that: Symbol): NodeSeq = {
     new NodeSeq(
       that.name match {
        case "_" => this.descendant_or_self
        case _ => this.descendant_or_self.asInstanceOf[List[Node]].
        filter(x => x.label == that.name)
      })
  }

  /** この XML ノードに対する ハッシュコード */
  override def hashCode =
    Utility.hashCode(label, attribute.toList.hashCode, child)

  /** このノードの文字列表現 */
  override def toString = Utility.toXML(this)

}
```

12.5 事前定義済みオブジェクト（The Predef Object）

事前定義済みオブジェクトは Scala プログラムの標準関数と型エイリアスを定義します。 これは常に暗黙のうちにインポートされるので、その定義されたすべてのメンバーは修飾なしで利用可能です。 JVM 環境での定義は、次のシグニチャに一致します:

```
package scala
object Predef {

  // classOf ---------------------------------------------------------

  /** クラス型の実行時表現を返す */

  def classOf[T]: Class[T] = null
  // これはダミー。classOf はコンパイラによって処理される。
  // 標準の型エイリアス -----------------------------------------

  type   byte      =   scala.Byte
  type   short     =   scala.Short
  type   char      =   scala.Char
  type   int       =   scala.Int
  type   long      =   scala.Long
  type   float     =   scala.Float
  type   double    =   scala.Double
  type   boolean   =   scala.Boolean
  type   unit      =   scala.Unit
```

```
type String          = java.lang.String
type Class[T]        = java.lang.Class[T]
type Runnable        = java.lang.Runnable

type Throwable = java.lang.Throwable
type Exception = java.lang.Exception
type Error     = java.lang.Error

type RuntimeException = java.lang.RuntimeException
type NullPointerException = java.lang.NullPointerException
type ClassCastException = java.lang.ClassCastException
type IndexOutOfBoundsException = java.lang.IndexOutOfBoundsException
type ArrayIndexOutOfBoundsException =
     java.lang.ArrayIndexOutOfBoundsException
type StringIndexOutOfBoundsException =
     java.lang.StringIndexOutOfBoundsException
type UnsupportedOperationException =
       java.lang.UnsupportedOperationException
type IllegalArgumentException = java.lang.IllegalArgumentException
type NoSuchElementException = java.util.NoSuchElementException
type NumberFormatException = java.lang.NumberFormatException

// その他 ---------------------------------------------------

type Function[-A, +B] = Function1[A, B]

type Map[A, B] = collection.immutable.Map[A, B]
type Set[A] = collection.immutable.Set[A]

val Map = collection.immutable.Map
val Set = collection.immutable.Set

// エラーとアサーション---------------------------------------

def error(message: String): Nothing = throw new Error(message)

def exit: Nothing = exit(0)

def exit(status: Int): Nothing = {
  java.lang.System.exit(status)
  throw new Throwable()
}

def assert(assertion: Boolean) {
  if (!assertion)
    throw new java.lang.AssertionError("assertion failed")
}

def assert(assertion: Boolean, message: Any) {
  if (!assertion)
    throw new java.lang.AssertionError("assertion failed: " + message)
}

def assume(assumption: Boolean) {
```

```
  if (!assumption)
    throw new IllegalArgumentException("assumption failed")
}

def assume(assumption: Boolean, message: Any) {
  if (!assumption)
    throw new IllegalArgumentException(message.toString)
}

  // tupling ------------------------------------------------------

type Pair[+A, +B] = Tuple2[A, B]
object Pair {
  def apply[A, B](x: A, y: B) = Tuple2(x, y)
  def unapply[A, B](x: Tuple2[A, B]): Option[Tuple2[A, B]] = Some(x)
}

type Triple[+A, +B, +C] = Tuple3[A, B, C]
object Triple {
  def apply[A, B, C](x: A, y: B, z: C) = Tuple3(x, y, z)
  def unapply[A, B, C](x:Tuple3[A, B, C]): Option[Tuple3[A, B, C]] = Some(x)
}

class ArrowAssoc[A](x: A) {
  def -> [B](y: B): Tuple2[A, B] = Tuple2(x, y)
}
implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A] = new ArrowAssoc(x)

// 印字と読み込み ------------------------------------------------

def   print(x: Any) = Console.print(x)
def   println() = Console.println()
def   println(x: Any) = Console.println(x)
def   printf(text: String, xs: Any*) = Console.printf(text, xs: _*)
def   format(text: String, xs: Any*) = Console.format(text, xs: _*)

def   readLine(): String = Console.readLine()
def   readLine(text: String, args: Any*) = Console.readLine(text, args)
def   readBoolean() = Console.readBoolean()
def   readByte() = Console.readByte()
def   readShort() = Console.readShort()
def   readChar() = Console.readChar()
def   readInt() = Console.readInt()
def   readLong() = Console.readLong()
def   readFloat() = Console.readFloat()
def   readDouble() = Console.readDouble()
def   readf(format: String) = Console.readf(format)
def   readf1(format: String) = Console.readf1(format)
def   readf2(format: String) = Console.readf2(format)
def   readf3(format: String) = Console.readf3(format)

// The ''catch-all'' implicit -----------------------------------------

implicit def identity[A](x: A): A = x
```

```
  // クラス Ordered へのビュー

%% @@@KSW what is ''Proxy''? It's not defined anywhere.
%% @@@MO It's a Java interface.
  implicit def int2ordered(x: Int):Ordered[Int] = new Ordered[Int] with Proxy{
    def self: Any = x
    def compare[B >: Int <% Ordered[B]](y: B): Int = y match {
      case y1: Int =>
        if (x < y1) -1
        else if (x > y1) 1
        else 0
      case _ => -(y compare x)
    }
  }

  // 次のメソッド実装は、最後の一つに類似
  implicit   def   char2ordered(x: Char): Ordered[Char] = ...
  implicit   def   long2ordered(x: Long): Ordered[Long] = ...
  implicit   def   float2ordered(x: Float): Ordered[Float] = ...
  implicit   def   double2ordered(x: Double): Ordered[Double] = ...
  implicit   def   boolean2ordered(x: Boolean): Ordered[Boolean] = ...

  implicit def seq2ordered[A <% Ordered[A]](xs: Array[A]): Ordered[Seq[A]] =
    new Ordered[Seq[A]] with Proxy {
      def compare[B >: Seq[A] <% Ordered[B]](that: B): Int = that match {
        case that: Seq[A] =>
          var res = 0
          val these = this.elements
          val those = that.elements
          while (res == 0 && these.hasNext)
            res = if (!those.hasNext) 1 else these.next compare those.next
        case _ => - (that compare xs)
    }

  implicit def string2ordered(x: String): Ordered[String] =
    new Ordered[String] with Proxy {
      def self: Any = x
      def compare[b >: String <% Ordered[b]](y: b): Int = y match {
        case y1: String => x compare y1
        case _ => -(y compare x)
      }
    }


  implicit def tuple2ordered[a1 <% Ordered[a1], a2 <% Ordered[a2]]
                           (x: Tuple2[a1, a2]): Ordered[Tuple2[a1, a2]] =
    new Ordered[Tuple2[a1, a2]] with Proxy {
      def self: Any = x
      def compare[T >: Tuple2[a1, a2] <% Ordered[T]](y: T): Int = y match {
        case y: Tuple2[a1, a2] =>
          val res = x._1 compare y._1
          if (res == 0) x._2 compare y._2
```

```
        else res
      case _ => -(y compare x)
    }
  }

// Tuple3 〜 Tuple9 についても同様

// クラス Seq へのビュー

implicit def string2seq(str: String): Seq[Char] = new Seq[Char] {
  def length = str.length()
  def elements = Iterator.fromString(str)
  def apply(n: Int) = str.charAt(n)
  override def hashCode: Int = str.hashCode
  override def equals(y: Any): Boolean = (str == y)
  override protected def stringPrefix: String = "String"
}

// プリミティブ型から Java のボックス型へのビュー

implicit   def   byte2Byte(x: Byte) = new java.lang.Byte(x)
implicit   def   short2Short(x: Short) = new java.lang.Short(x)
implicit   def   char2Character(x: Char) = new java.lang.Character(x)
implicit   def   int2Integer(x: Int) = new java.lang.Integer(x)
implicit   def   long2Long(x: Long) = new java.lang.Long(x)
implicit   def   float2Float(x: Float) = new java.lang.Float(x)
implicit   def   double2Double(x: Double) = new java.lang.Double(x)
implicit   def   boolean2Boolean(x: Boolean) = new java.lang.Boolean(x)

// 数値変換ビュー

implicit   def   byte2short(x: Byte): Short = x.toShort
implicit   def   byte2int(x: Byte): Int = x.toInt
implicit   def   byte2long(x: Byte): Long = x.toLong
implicit   def   byte2float(x: Byte): Float = x.toFloat
implicit   def   byte2double(x: Byte): Double = x.toDouble

implicit   def   short2int(x: Short): Int = x.toInt
implicit   def   short2long(x: Short): Long = x.toLong
implicit   def   short2float(x: Short): Float = x.toFloat
implicit   def   short2double(x: Short): Double = x.toDouble

implicit   def   char2int(x: Char): Int = x.toInt
implicit   def   char2long(x: Char): Long = x.toLong
implicit   def   char2float(x: Char): Float = x.toFloat
implicit   def   char2double(x: Char): Double = x.toDouble

implicit def int2long(x: Int): Long = x.toLong
implicit def int2float(x: Int): Float = x.toFloat
implicit def int2double(x: Int): Double = x.toDouble

implicit def long2float(x: Long): Float = x.toFloat
implicit def long2double(x: Long): Double = x.toDouble
```

```
  implicit def float2double(x: Float): Double = x.toDouble
}
```

Bibliography

[Dra10] Iulian Dragos. Scala specialization, 2010. SID-9.

[KP07] Andrew J. Kennedy and Benjamin C. Pierce. On Decidability of Nominal Subtyping with Variance, January 2007. FOOL-WOOD '07.

[Oa04] Martin Odersky and al. An Overview of the Scala Programming Language . Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[OCRZ03] Martin Odersky, Vincent Cremet, Christine Rockl, and Matthias Zenger. A Nominal Theory of Objects with Dependent Types. In Proc. ECOOP'03, Springer LNCS, July 2003.

[Ode06] Martin Odersky. The Scala Experiment - Can We Provide Better Language Support for Component Systems? In Proc. ACM Symposium on Principles of Programming Languages, 2006.

[OZ05a] Martin Odersky and Matthias Zenger. Independently Extensible Solutions to the Expression Problem. In Proc. FOOL 12, January 2005. http://homepages.inf.ed.ac.uk/wadler/fool .

[OZ05b] Martin Odersky and Matthias Zenger. Scalable Component Abstractions. In Proc. OOPSLA, 2005.

[W3C] W3C. Extensible Markup Language (XML). http://www.w3.org/TR/REC-xml .

Chapter A Scala 文法要約

(訳注:前半は日本語訳、後半は英語原文を記載。 資料にはないがドキュメント本文には出てくる hexDigit を追加。 クラスパラメータリストの区切りは ','に修正して掲載)

次は、Scala の字句構文を EBNF記法で記述している。

```
大文字             ::=  'A' | … | 'Z' | '$' | '_' と ユニコードカテゴリ Lu
小文字             ::=  'a' | … | 'z' と ユニコードカテゴリ Ll
文字               ::=  大文字 | 小文字 と ユニコードカテゴリ Lo, Lt, Nl
数字               ::=  '0' | … | '9'
演算子文字         ::=  "\u0020-007F 中の他の全ての文字と、ユニコードカテゴリ
                        Sm、So。ただし括弧([])およびピリオドを除く" (*1)

演算子             ::=  演算子文字 {演算子文字}
変数識別子         ::=  小文字 識別子文字列
一般識別子         ::=  大文字 識別子文字列
                     |  変数識別子
                     |  記号
識別子             ::=  一般識別子
                     |  '\`' stringLit '\`'
識別子文字列       ::=  {文字 | 数字} ['_' 記号]

整数リテラル       ::=  (10進数値 | 16進数値 | 8進数値) ['L' | 'l']
10進数値           ::=  '0' | 非0数字 {数字}
16進数値           ::=  '0' 'x' 16進数字 {16進数字}
8進数値            ::=  '0' 8進数字 {8進数字}
数字               ::=  '0' | 非0数字
非0数字            ::=  '1' | … | '9'
8進数字            ::=  '0' | … | '7'
16進数字           ::=  '0' | … | '9' | 'A' | … | 'F' | 'a' | … | 'f' |

浮動小数点リテラル ::=  数字 {数字} '.' {数字} [指数部] [浮動型]
                     |  '.' 数字 {数字} [指数部] [浮動型]
                     |  数字 {数字} 指数部 [浮動型]
                     |  数字 {数字} [指数部] 浮動型
指数部             ::=  ('E' | 'e') ['+' | '-'] 数字 {数字}
浮動型             ::=  'F' | 'f' | 'D' | 'd'

ブーリアンリテラル ::=  'true' | 'false'

文字リテラル       ::=  '\'' printableChar '\''
                     |  '\'' charEscapeSeq '\''

文字列リテラル     ::=  '"' {文字列要素} '"'
                     |  '"""' 複数行文字 '"""'
文字列要素         ::=  printableCharNoDoubleQuote
                     |  charEscapeSeq
複数行文字         ::=  {['"'] ['"'] charNoDoubleQuote} {'"'}

シンボルリテラル   ::=  ''' 一般識別子
```

```
コメント          ::=  '/*' "文字の任意の並び" '*/'
                     |  '//' "行末までの、文字の任意の並び"

改行区切り        ::=  "改行文字"
複文区切り        ::=  ';' | 改行区切り {改行区切り}
```

次は、Scala の自由文脈文法を EBNF記法で記述している。

```
リテラル          ::=  ['-'] 整数リテラル
                     |  ['-'] 浮動小数点リテラル
                     |  ブーリアンリテラル
                     |  文字リテラル
                     |  文字列リテラル
                     |  シンボルリテラル
                     |  'null'

限定識別子        ::=  識別子 {'.' 識別子}
識別子リスト      ::=  識別子 {',' 識別子}

パス              ::=  安定識別子
                     |  [識別子 '.'] 'this'
安定識別子        ::=  識別子
                     |  パス '.' 識別子
                     |  [識別子 '.'] 'super' [クラス明示子] '.' 識別子
クラス明示子      ::=  '[' 識別子 ']'

型                ::=  関数引数型 '=>' 型
                     |  中置型 [存在節]
関数引数型        ::=  中置型
                     |  '(' [ パラメータ型 {',' パラメータ型 } ] ')'
存在節            ::=  'forSome' '{' 存在宣言 {複文区切り 存在宣言} '}'
存在宣言          ::=  'type' 型宣言
                     |  'val' Val宣言
中置型            ::=  複合型 {識別子 [改行区切り] 複合型}
複合型            ::=  アノテーション型 {'with' アノテーション型} [細別]
                     |  細別
アノテーション型  ::=  単純型 {アノテーション}
単純型            ::=  単純型 型引数
                     |  単純型 '#' 識別子
                     |  安定識別子
                     |  パス '.' 'type'
                     |  '(' 型リスト ')'

型引数          ::=  '[' 型リスト ']'
型リスト        ::=  型 {',' 型}
細別            ::=  [改行区切り] '{' 細別文 {複文区切り 細別文} '}'
細別文          ::=  宣言
                   |  'type' 型定義
                   |
型パターン      ::=  型

帰属            ::=  ':' 中置型
                   |  ':' アノテーション {アノテーション}
                   |  ':' '_' '*'
```

```
式              ::=  (名前束縛リスト | ['implicit'] 識別子 | '_') '=>' 式
                 |   式1
式1             ::=  'if' '(' 式 ')' {改行区切り} 式 [[複文区切り] else 式]
                 |   'while' '(' 式 ')' {改行区切り} 式
                 |   'try' '{' ブロック '}' ['catch' '{' ケース節リスト '}']
                     ['finally' 式]
                 |   'do' 式 [複文区切り] 'while' '(' 式 ')'
                 |   'for' ('(' 列挙子リスト ')' | '{' 列挙子リスト '}')
                     {改行区切り} ['yield'] 式
                 |   'throw' 式
                 |   'return' [式]
                 |   [単純式 '.'] 識別子 '=' 式
                 |   単純式1 引数式リスト '=' 式
                 |   後置式
                 |   後置式 帰属
                 |   後置式 'match' '{' ケース節リスト '}'
後置式          ::=  中置式 [識別子 [改行区切り]]
中置式          ::=  前置式
                 |   中置式 識別子 [改行区切り] 中置式
前置式          ::=  ['-' | '+' | '~' | '!'] 単純式
単純式          ::=  'new' (クラステンプレート | テンプレート本体)
                 |   ブロック式
                 |   単純式1 ['_']
単純式1         ::=  リテラル
                 |   パス
                 |   ' '
                 |   '(' [式リスト] ')'
                 |   単純式 '.' 識別子
                 |   単純式 型引数
                 |   単純式1 引数式リスト
                 |   XmlExpr
式リスト        ::=  式 {',' 式}
引数式リスト    ::=  '(' [式リスト] ')'
                 |   [改行区切り] ブロック式
ブロック式      ::=  '{' ケース節リスト '}'
                 |   '{' ブロック '}'
ブロック        ::=  {ブロック文 複文区切り} [結果式]
ブロック文      ::=  インポート
                 |   {アノテーション} ['implicit' | 'lazy'] 定義
                 |   {アノテーション} {ローカル修飾子} テンプレート定義
                 |   式1
                 |
結果式          ::=  式1
                 |   (名前束縛リスト | (識別子 | '_') ':' 複合型) '=>' ブロック

列挙子リスト    ::=  生成子 {複文区切り 列挙子}
列挙子          ::=  生成子
                 |   ガード
                 |   'val' パターン1 '=' 式
生成子          ::=  パターン1 '<-' 式 [ガード]

ケース節リスト  ::=  ケース節 { ケース節 }
ケース節        ::=  'case' パターン [ガード] '=>' ブロック
```

```
ガード          ::=   'if' 後置式

パターン        ::=   パターン1 { '|' パターン1 }
パターン1       ::=   変数識別子 ':' 型パターン
                  |   '_' ':' 型パターン
                  |   パターン2
パターン2       ::=   変数識別子 ['@' パターン3]
                  |   パターン3
パターン3       ::=   単純パターン
                  |   単純パターン { 識別子 [改行区切り] 単純パターン }
単純パターン
          ::=   '_'
              |  変数識別子
              |  リテラル
              |  安定識別子
              |  安定識別子 '(' [パターンリスト ')'
              |  安定識別子 '(' [パターンリスト ','] [変数識別子 '@'] '_' '*' ')'
              |  '(' [パターンリスト] ')'
              |  XmlPattern
パターンリスト      ::=   パターン [',' パターンリスト]
                    |   '_' *

型パラメータ節      ::=   '[' 変位型パラメータ {',' 変位型パラメータ} ']'
関数型パラメータ節  ::=   '[' 型パラメータ {',' 型パラメータ} ']'
変位型パラメータ    ::=   {アノテーション} ['+' | '-'] 型パラメータ
型パラメータ        ::=   (識別子 | '_') [型パラメータ節] ['>:' 型] ['<:' 型]
                          {'<%' 型} {':' 型}

パラメータ節リスト  ::=   {パラメータ節}
                          [[改行区切り] '(' 'implicit' パラメータリスト ')']
パラメータ節        ::=   [改行区切り] '(' [パラメータリスト] ')'
パラメータリスト    ::=   パラメータ {',' パラメータ}
パラメータ          ::=   {アノテーション} 識別子 [':' パラメータ型] ['=' 式]
パラメータ型        ::=   型
                    |   '=>' 型
                    |   型 '*'
クラスパラメータ節リスト
                ::=  {クラスパラメータ節}
                     [[改行区切り] '(' 'implicit' クラスパラメータリスト ')']
クラスパラメータ節  ::=   [改行区切り] '(' [クラスパラメータリスト] ')'
クラスパラメータリスト
                    ::=   クラスパラメータ {',' クラスパラメータ}
クラスパラメータ    ::=   {アノテーション} [{修飾子} ('val' | 'var')]
                          識別子 ':' パラメータ型 ['=' 式]
名前束縛リスト      ::=   '(' 名前束縛 {',' 名前束縛 ')'
名前束縛            ::=   (識別子 | '_') [':' 型]

修飾子              ::=   ローカル修飾子
                    |   アクセス修飾子
                    |   'override'
ローカル修飾子      ::=   'abstract'
                    |   'final'
                    |   'sealed'
                    |   'implicit'
```

```
                  |   'lazy'
アクセス修飾子     ::=   ('private' | 'protected') [アクセス限定子]
アクセス限定子     ::=   '[' (識別子 | 'this') ']'

アノテーション                ::=   '@' 単純型 {引数式リスト}
コンストラクタアノテーション   ::=   '@' 単純型 引数式リスト
名前値ペア         ::=   'val' 識別子 '=' 前置式

テンプレート本体   ::=   [改行区切り] '{' [自己型] テンプレート文
                         {複文区切り テンプレート文} '}'
テンプレート文     ::=   インポート
                  |   {アノテーション [改行区切り]} {修飾子} 定義
                  |   {アノテーション [改行区切り]} {修飾子} 宣言
                  |   式
                  |
自己型             ::=   識別子 [':' 型] '=>'
                  |   'this' ':' 型 '=>'

インポート    ::=   'import' インポート式 {',' インポート式}
インポート式  ::=   安定識別子 '.' (識別子 | '_' | インポートセレクタリスト)
インポートセレクタリスト
              ::=   '{' {インポートセレクタ ','} (インポートセレクタ | '_') '}'
インポートセレクタ ::=   識別子 ['=>' 識別子 | '=>' '_']

宣言               ::=   'val' Val宣言
                  |   'var' 変数宣言
                  |   'def' 関数宣言
                  |   'type' {改行区切り} 型宣言

Val宣言            ::=   識別子リスト ':' 型
変数宣言           ::=   識別子リスト ':' 型
関数宣言           ::=   関数シグニチャ [':' 型]
関数シグニチャ     ::=   識別子 [関数型パラメータ節] パラメータ節リスト
型宣言             ::=   識別子 [型パラメータ節] ['>:' 型] ['<:' 型]

パターン変数定義   ::=   'val' パターン定義
                  |   'var' 変数定義
定義               ::=   パターン変数定義
                  |   'def' 関数定義
                  |   'type' {改行区切り} 型定義
                  |   テンプレート定義
パターン定義       ::=   パターン2 {',' パターン2} [':' 型] '=' 式
変数定義           ::=   パターン定義
                  |   識別子リスト ':' 型 '=' '_'
関数定義      ::=   関数シグニチャ [':' 型] '=' 式
                |   関数シグニチャ [改行区切り] '{' ブロック '}'
                |   'this' パラメータ節 パラメータ節リスト
                    ('=' コンストラクタ式 | [改行区切り] コンストラクタブロック)
型定義             ::=   識別子 [型パラメータ節] '=' 型

テンプレート定義   ::=   ['case'] 'class' クラス定義
                  |   ['case'] 'object' オブジェクト定義
                  |   'trait' トレイト定義
クラス定義         ::=   識別子 [型パラメータ節] {コンストラクタアノテーション}
```

```
                        [アクセス修飾子] クラスパラメータ節リスト
                        クラステンプレートオプション
トレイト定義       ::=  識別子 [型パラメータ節] トレイトテンプレートオプション
オブジェクト定義   ::=  識別子 クラステンプレートオプション
クラステンプレートオプション
          ::= 'extends' クラステンプレート | [['extends'] テンプレート本体]
トレイトテンプレートオプション
          ::= 'extends' トレイトテンプレート | [['extends'] テンプレート本体]
クラステンプレート ::=  [事前定義リスト] 親クラスリスト [テンプレート本体]
トレイトテンプレート ::=  [事前定義リスト] 親トレイトリスト [テンプレート本体]
親クラスリスト     ::=  コンストラクタ {'with' アノテーション型}
親トレイトリスト   ::=  アノテーション型 {'with' アノテーション型}
コンストラクタ     ::=  アノテーション型 {引数式リスト}
事前定義リスト     ::=  '{' [事前定義 {複文区切り 事前定義}] '}' 'with'
事前定義           ::=  {アノテーション [改行区切り]} {修飾子} パターン変数定義
コンストラクタ式   ::=  自己呼出し
                     |  コンストラクタブロック
コンストラクタブロック ::=  '{' 自己呼出し {複文区切り ブロック文} '}'
自己呼出し         ::=  'this' 引数式リスト {引数式リスト}

トップレベル文並び ::= トップレベル文 {複文区切り トップレベル文}
トップレベル文     ::= {アノテーション [改行区切り]} {修飾子} テンプレート定義
                     | インポート
                     | パッケージング
                     | パッケージオブジェクト
                     |

パッケージング     ::=  'package' 限定識別子 [改行区切り]
                        '{' トップレベル文並び '}'
パッケージオブジェクト ::=  'package' 'object' オブジェクト定義

コンパイル単位     ::=  {'package' 限定識別子 複文区切り} トップレベル文並び
```

訳注:(*1) 第 1 章の記述と合致していない。

Scala Syntax Summary

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

```
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'
opchar            ::=  "all other characters in \u0020-007F and Unicode
                       categories Sm, So except parentheses ([]) and periods"

op                ::=  opchar {opchar}
varid             ::=  lower idrest
plainid           ::=  upper idrest
                     |  varid
                     |  op
id                ::=  plainid
                     |  '\`' stringLit '\`'
idrest            ::=  {letter | digit} ['_' op]

integerLiteral    ::=  (decimalNumeral | hexNumeral | octalNumeral) ['L' | 'l']
decimalNumeral    ::=  '0' | nonZeroDigit {digit}
hexNumeral        ::=  '0' 'x' hexDigit {hexDigit}
octalNumeral      ::=  '0' octalDigit {octalDigit}
digit             ::=  '0' | nonZeroDigit
nonZeroDigit      ::=  '1' | · · · | '9'
octalDigit        ::=  '0' | · · · | '7'

floatingPointLiteral
                  ::=  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  ::=  '\'' printableChar '\''
                     |  '\'' charEscapeSeq '\''

stringLiteral     ::=  '"' {stringElement} '"'
                     |  '"""' multiLineChars '"""'
stringElement     ::=  printableCharNoDoubleQuote
                     |  charEscapeSeq
multiLineChars    ::=  {['"'] ['"'] charNoDoubleQuote} {'"'}

symbolLiteral     ::=  ''' plainid

comment           ::=  '/*' "any sequence of characters" '*/'
```

```
                      |   '//' "any sequence of characters up to end of line"

   nl                ::=   "new line character"
   semi              ::=   ';' |   nl {nl}
```

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 '}' ['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] ')'
                    |    [nl] BlockExpr
BlockExpr           ::=  '{' CaseClauses '}'
                    |    '{' Block '}'
Block               ::=  {BlockStat semi} [ResultExpr]
BlockStat           ::=  Import
                    |    {Annotation} ['implicit' | 'lazy'] Def
                    |    {Annotation} {LocalModifier} TmplDef
                    |    Expr1
                    |
ResultExpr          ::=  Expr1
                    |    (Bindings | (id | '_') ':' CompoundType) '=>' Block

Enumerators         ::=  Generator {semi Enumerator}
Enumerator          ::=  Generator
                    |    Guard
                    |    'val' Pattern1 '=' Expr
Generator           ::=  Pattern1 '<-' Expr [Guard]

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}       (訳注:','の誤りか?)
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
NameValuePair     ::=  'val' id '=' PrefixExpr

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
```

Chapter B 変更履歴

バージョン 2.8.0 中の変更

お終いのカンマ (Trailing commas)

式、引数、型あるいはパターンシーケンス中のお終いのカンマは、もうサポートされません。

バージョン 2.8 中の変更(開発中)

ネストしたパッケージの可視性規則を変更(どこでした?)

§2 中の可視性規則を、パッケージが特別扱いされないように変更した。

§3.5.3 に弱い適合性に関する節を追加。 最少の上限境界を使って結果型を計算するために、条件、マッチ式、try 式の型規則を緩めた。 引数型が、推論された形式上のパラメータ型に弱く適合することだけが必要であるように、ローカルな型推論に対する型規則を緩めた。 弱適合性をサポートするために、§6.26 中に数値拡張に関する節を追加。

§5.1.4 中の想定外のオーバライドを避けるために規則を強化。

クラスリテラル削除。

コンテキスト境界に関する節を §7.4 に追加。

isInstanceOf とパターンマッチの相違を明確化(§12.1)。

ただ 1 つのパラメータをもつ関数リテラルに対する implicit 修飾子を許可(§6.23)。

(訳注:------- これより以前の履歴は、見出し、体裁以外は原文のままです ------)

バージョン 2.7.2 (2008-11-10) 中の変更

代入演算子の優先順位

The precedence of assignment operators has been brought in line with Java's (§6.12). From now on, += has the same precedence as =.

関数パラメータとしてのワイルドカード

A formal parameter to an anonymous fucntion may now be a wildcard represented by an underscore (§6.23). Example:

```
_ => 7   // The function that ignores its argument
         // and always returns 7.
```

左矢印のユニコード代替物

The Unicode glyph \u2190 '←' is now treated as a reserved identifier, equivalent to the ASCII symbol '<-'.

バージョン 2.7.1 (2008-4-09) 中の変更

型におけるワイルドカードプレースホルダに関するスコープ規則変更

A wildcard in a type now binds to the closest enclosing type application. For example List[List[_]] is now equivalent to the existential type

```
List[List[t] forSome { type t }] .
```

In version 2.7.0, the type expanded instead to

```
List[List[t]] forSome { type t } .
```

The new convention corresponds exactly to the way wildcards in Java are interpreted .

No Contractiveness Requirement for Implicits

The contractiveness requirement for implicit method definitions has been dropped. Instead it is checked for each implicit expansion individually that the expansion does not result in a cycle or a tree of infinitely growing types (§7.2).

バージョン 2.7.0 (2008-2-07) 中の変更

Java ジェネリック(総称型)

Scala now supports Java generic types by default:

- A generic type in Java such as ArrayList<String> is translated to a generic type in Scala: ArrayList[String].
- A wildcard type such as ArrayList<? extends Number> is translated to ArrayList[_ <: Number]. This is itself a shorthand for the existential type ArrayList[T] forSome { type T <: Number }.
- A raw type in Java such as ArrayList is translated to ArrayList[_], which is a shorthand for ArrayList[T] forSome { type T }.

This translation works if -target:jvm-1.5 is specified, which is the new default. For any other target, Java generics are not recognized. To ensure upgradability of Scala c

odebases, extraneous type parameters for Java classes under -target:jvm-1.4 are simply ignored. For instance, when compiling with -target:jvm-1.4, a Scala type such as ArrayList[String] is simply treated as the unparameterized type ArrayList.

ケースクラスへの変更

The Scala compiler generates now for every case class a companion extractor object (§5.3.2). For instance, given the case class:

```
case class X(elem: String)
```

the following companion object is generated:

```
object X {
  def unapply(x: X): Some[String] = Some(x.elem)
  def apply(s: String): X = new X(s)
}
```

If the object exists already, only the apply and unapply methods are added to it. Three restrictions on case classes have been removed.

1. Case classes can now inherit from other case classes.
2. Case classes may now be abstract.
3. Case classes may now come with companion objects.

バージョン 2.6.1 (2007-11-30) 中の変更

パターン束縛によるミュータブル変数の導入

Mutable variables can now be introduced by a pattern matching definition (§4.2), just like values can. Examples:

```
var (x, y) = if (positive) (1, 2) else (-1, -3)
var hd :: tl = mylist
```

自己型(Self-types)

Self types can now be introduced without defining an alias name for this (§5.1). Example:

```
class C {
  type T <: Trait
  trait Trait { this: T => ... }
}
```

バージョン 2.6 (2007-7-27) 中の変更

存在型(Existential types)

It is now possible to define existential types (§3.2.10). An existential type has the form T forSome {Q} where Q is a sequence of value and/or type declarations. Given the class definitions

```
class Ref[T]
abstract class Outer { type T }
```

one may for example write the following existential types

```
Ref[T] forSome { type T <: java.lang.Number }
Ref[x.T] forSome { val x: Outer }
```

遅延評価Val(Lazy values)

It is now possible to define lazy value declarations using the new modifier lazy (§4.1) . A lazy value definition evaluates its right hand side e the first time the value is accessed. Example:

```
import compat.Platform._
val t0 = currentTime
lazy val t1 = currentTime

val t2 = currentTime

println("t0 <= t2: " + (t0 <= t2))        //true
println("t1 <= t2: " + (t1 <= t2))        //false (lazy evaluation of t1)
```

構造的型(Structural types)

It is now possible to declare structural types using type refinements (§3.2.7). For example:

```
class File(name: String) {
  def getName(): String = name
  def open() { /*..*/ }
  def close() { println("close file") }
}
def test(f: { def getName(): String }) { println(f.getName) }

test(new File("test.txt"))
test(new java.io.File("test.txt"))
```

There's also a shorthand form for creating values of structural types. For instance,

```
new { def getName() = "aaron" }
```

is a shorthand for

```
new AnyRef{ def getName() = "aaron" }
```

バージョン 2.5 (2007-5-02) 中の変更

型コンストラクタの多相性(Type constructor polymorphism (*1))

Type parameters (§4.4) and abstract type members (§4.3) can now also abstract over type constructors (§3.3.3). This allows a more precise Iterable interface:

```
trait Iterable[+T] {
  type MyType[+T] <: Iterable[T] // MyType is a type constructor

     def filter(p: T => Boolean): MyType[T] = ...
     def map[S](f: T => S): MyType[S] = ...
}
```

(*1) Implemented by Adriaan Moors

```
abstract class List[+T] extends Iterable[T] {
  type MyType[+T] = List[T]
}
```

This definition of Iterable makes explicit that mapping a function over a certain structure (e.g., a List) will yield the same structure (containing different elements).

オブジェクトの事前初期化(Early object initialization)

It is now possible to initialize some fields of an object before any parent constructors are called (§5.1.6). This is particularly useful for traits, which do not have normal constructor parameters. Example:

```
trait Greeting {
  val name: String
  val msg = "How are you, "+name
}
class C extends {
  val name = "Bob"
} with Greeting {
  println(msg)
}
```

In the code above, the field name is initialized before the constructor of Greeting is called. Therefore, field msg in class Greeting is properly initialized to "How are you, Bob".

For内包表記、再掲

The syntax of for-comprehensions has changed (§6.19). In the new syntax, generators do not start with a val anymore, but filters start with an if (and are called guards). A

semicolon in front of a guard is optional. For example:

```
for (val x <- List(1, 2, 3); x % 2 == 0) println(x)
```

is now written

```
for (x <- List(1, 2, 3) if x % 2 == 0) println(x)
```

The old syntax is still available but will be deprecated in the future.

暗黙の無名関数(Implicit anonymous functions)

It is now possible to define anonymous functions using underscores in parameter position (§Example 6.23.1). For instance, the expressions in the left column are each function values which expand to the anonymous functions on their right.

```
_ + 1                          x => x + 1
_ * _                          (x1, x2) => x1 * x2
(_: int) * 2                   (x: int) => (x: int) * 2
if (_) x else y                z => if (z) x else y
_.map(f)                       x => x.map(f)
_.map(_ + 1)                   x => x.map(y => y + 1)
```

As a special case (§6.7), a partially unapplied method is now designated m _ instead of the previous notation &m. The new notation will displace the special syntax forms .m() for abstracting over method receivers and &m for treating an unapplied method as a function value. For the time being, the old syntax forms are still available, but they will be deprecated in the future.

パターンマッチング無名関数、再掲

It is now possible to use case clauses to define a function value directly for functions of arities greater than one (§8.5). Previously, only unary functions could be defined that way. Example:

```
def scalarProduct(xs: Array[Double], ys: Array[Double]) =
  (0.0 /: (xs zip ys)) {
    case (a, (b, c)) => a + b * c
  }
```

バージョン 2.4 (2007-3-09) 中の変更

オブジェクトローカルな private と protected (Object-local private and protected)

The private and protected modifiers now accept a [this] qualifier (§5.2). A definition M which is labelled private[this] is private, and in addition can be accessed only from within the current object. That is, the only legal prefixes for M are this or C.this. Analogously, a definition M which is labelled protected[this] is protected , and in addition can be accessed only from within the current object.

タプル、再掲

The syntax for tuples has been changed from {...} to (...) (§6.9). For any sequence of types T1,...,Tn , (T1,...,Tn) is a shorthand for Tuplen[T1,...,Tn].

Analogously, for any sequence of expressions or patterns x1,...,xn , (x1,...,xn) is a shorthand for Tuplen(x1,...,xn).

基本コンストラクタへのアクセス修飾子

The primary constructor of a class can now be marked private or protected (§5.3). If such an access modifier is given, it comes between the name of the class and its value parameters. Example:

```
class C[T] private (x: T) { ... }
```

アノテーション(Annotations)

The support for attributes has been extended and its syntax changed (§11). Attributes are now called annotations. The syntax has been changed to follow Java's conventions, e.g. @attribute instead of [attribute]. The old syntax is still available but will be deprecated in the future. Annotations are now serialized so that they can be read by compile-time or runtime tools. Class scala.Annotation has two sub-traits which are used to indicate how annotations are retained. Instances of an annotation class inheriting from trait scala.ClassfileAnnotation will be stored in the generated class files. Instances of an annotation class inheriting from trait scala.StaticAnnotation will be visible to the Scala type-checker in every compilation unit where the annotated symbol is accessed.

決定可能なサブ型付け(Decidable subtyping)

The implementation of subtyping has been changed to prevent infinite recursions. Termination of subtyping is now ensured by a new restriction of class graphs to be finitary (§5.1.5).

ケースクラスは抽象ではない(Case classes cannot be abstract)

It is now explicitly ruled out that case classes can be abstract (§5.2). The specification was silent on this point before, but did not explain how abstract case classes were treated. The Scala compiler allowed the idiom.

自己エイリアス、自己型に対する新文法

It is now possible to give an explicit alias name and/or type for the self reference this (§5.1). For instance, in

```
class C { self: D =>
  ...
}
```

the name self is introduced as an alias for this within C and the self type (§5.3) of C is assumed to be D. This construct is introduced now in order to replace eventually both the qualified this construct C.this and the requires clause in Scala.

代入演算子

It is now possible to combine operators with assignments (§6.12.4). Example:

```
var x: int = 0
x += 1
```

バージョン 2.3.2 (2007-1-23) 中の変更

抽出子(Extractors)

It is now possible to define patterns independently of case classes, using unapply methods in extractor objects (§8.1.8). Here is an example:

```
object Twice {
  def apply(x:Int): int = x*2
  def unapply(z:Int): Option[int] = if (z%2==0) Some(z/2) else None
}
val x = Twice(21)
x match { case Twice(n) => Console.println(n) } // prints 21
```

In the example, Twice is an extractor object with two methods:

- The apply method is used to build even numbers.
- The unapply method is used to decompose an even number; it is in a sense the reverse of apply. unapply methods return option types: Some(...) for a match that succeeds, None for a match that fails. Pattern variables are returned as the elements of Some. If there are several variables, they are grouped in a tuple.

In the second-to-last line, Twice's apply method is used to construct a number x. In the last line, x is tested against the pattern Twice(n). This pattern succeeds for even numbers and assigns to the variable n one half of the number that was tested. The pattern match makes use of the unapply method of object Twice. More details on extractors can be found in the paper "Matching Objects with Patterns" by Emir, Odersky and Williams.

タプル(Tuples)

A new lightweight syntax for tuples has been introduced (§6.9). For any sequence of types T1,...,Tn , {T1,...,Tn} is a shorthand for Tuplen[T1,...,Tn].

Analogously, for any sequence of expressions or patterns x1,...,xn , {x1,...,xn} is a shorthand for Tuplen(x1,...,xn).

多項の中置演算子(Infix operators of greater arities)

It is now possible to use methods which have more than one parameter as infix operators (§6.12). In this case, all method arguments are written as a normal parameter list in parentheses. Example:

```
class C {
  def +(x: int, y: String) = ...
}
val c = new C
```

```
c + (1, "abc")
```

廃棄属性(Deprecated attribute)

A new standard attribute deprecated is available (§11). If a member definition is mark ed with this attribute, any reference to the member will cause a "deprecated" warning message to be emitted.

バージョン 2.3 (2006-11-23) 中の変更

手続き(Procedures)

A simplified syntax for functions returning unit has been introduced (§4.6.3). Scala n ow allows the following shorthands:

```
def f(params)           for     def f(params): unit
def f(params) { ... }   for def f(params): unit = { ... }
```

型パターン(Type Patterns)

The syntax of types in patterns has been refined (§8.2). Scala now distinguishes betwe en type variables (starting with a lower case letter) and types as type arguments in p atterns. Type variables are bound in the pattern. Other type arguments are, as in prev ious versions, erased. The Scala compiler will now issue an "unchecked" warning at pla ces where type erasure might compromise type-safety.

標準の型(Standard Types)

The recommended names for the two bottom classes in Scala's type hierarchy have change d as follows:

```
All        ==>       Nothing
AllRef     ==>       Null
```

The old names are still available as type aliases.

バージョン 2.1.8 (2006-8-23) 中の変更

protectedに対する可視修飾子(Visibility Qualifier for protected)

Protected members can now have a visibility qualifier (§5.2), e.g. protected[<qualifier >]. In particular, one can now simulate package protected access as in Java writing

```
protected[P] def X ...
```

where P would name the package containing X.

privateアクセスの緩和(Relaxation of Private Acess)

Private members of a class can now be referenced from the companion module of the clas s and vice versa (§5.2)

暗黙の検索(Implicit Lookup)

The lookup method for implicit definitions has been generalized (§7.2). When searching for an implicit definition matching a type T , now are considered

1. all identifiers accessible without prefix, and
2. all members of companion modules of classes associated with T .

(The second clause is more general than before). Here, a class is associated with a ty pe T if it is referenced by some part of T , or if it is a base class of some part of T . For instance, to find implicit members corresponding to the type

```
HashSet[List[Int], String]
```

one would now look in the companion modules (aka static parts) of HashSet, List, Int, and String. Before, it was just the static part of HashSet.

厳しくなったパターンマッチ(Tightened Pattern Match)

A typed pattern match with a singleton type p.type now tests whether the selector valu e is reference-equal to p (§8.1). Example:

```
val p = List(1, 2, 3)
val q = List(1, 2)
val r = q
r match {
  case _: p.type => Console.println("p")
  case _: q.type => Console.println("q")
}
```

This will match the second case and hence will print "q". Before, the singleton types were erased to List, and therefore the first case would have matched, which is nonsens ical .

バージョン 2.1.7 (2006-7-19) 中の変更

複数行文字列リテラル(Multi-Line string literals)

It is now possible to write multi-line string-literals enclosed in triple quotes (§1.3. 5). Example:

```
"""this is a
   multi-line
   string literal"""
```

No escape substitutions except for unicode escapes are performed in such string literals.

クロージャ構文(Closure Syntax)

The syntax of closures has been slightly restricted (§6.23). The form

```
x: T => E
```

is valid only when enclosed in braces, i.e. { x: T => E }. The following is illegal, because it might be read as the value x typed with the type T => E:

```
val f = x: T => E
```

Legal alternatives are:

```
val f = { x: T => E }
val f = (x: T) => E
```

バージョン 2.1.5 (2006-5-24) 中の変更

クラスリテラル(Class Literals)

There is a new syntax for class literals (§6.2): For any class type C , classOf[C] designates the run-time representation of C .

バージョン 2.0 (2006-3-12) 中の変更

Scala in its second version is different in some details from the first version of the language. There have been several additions and some old idioms are no longer supported. This appendix summarizes the main changes.

新しいキーワード(New Keywords)

The following three words are now reserved; they cannot be used as identifiers (§1.1)

```
implicit       match       requires
```

文区切りとしての改行(Newlines as Statement Separators)

Newlines can now be used as statement separators in place of semicolons (§1.2)

構文の制限(Syntax Restrictions)

There are some other situations where old constructs no longer work:

Pattern matching expressions. The match keyword now appears only as infix operator bet
ween a selector expression and a number of cases, as in:

```
expr match {
  case Some(x) => ...
  case None => ...
}
```

Variants such as expr.match {...} or just match {...} are no longer supported .

"With" in extends clauses. . The idiom

```
class C with M { ... }
```

is no longer supported. A with connective is only allowed following an extends clause.
For instance, the line above would have to be written

```
class C extends AnyRef with M { ... }
```

However, assuming M is a trait (see 5.3.3), it is also legal to write

```
class C extends M { ... }
```

The latter expression is treated as equivalent to

```
class C extends S with M { ... }
```

where S is the superclass of M.

Regular Expression Patterns. The only form of regular expression pattern that is curre
ntly supported is a sequence pattern, which might end in a sequence wildcard _*. Examp
le:

```
case List(1, 2, _*) => ... // will match all lists starting with \code{1,2}.
```

It is at current not clear whether this is a permanent restriction. We are evaluating
the possibility of re-introducing full regular expression patterns in Scala.

自己型アノテーション(Selftype Annotations)

The recommended syntax of selftype annotations has changed.

```
class C: T extends B { ... }
```

becomes

```
class C requires T extends B { ... }
```

That is, selftypes are now indicated by the new requires keyword. The old syntax is st
ill available but is considered deprecated.

For内包表記(For-comprehensions)

For-comprehensions (§6.19) now admit value and pattern definitions. Example:

```
for {
  val x <- List.range(1, 100)
  val y <- List.range(1, x)
  val z = x + y
  isPrime(z)
} yield Pair(x, y)
```

Note the definition val z = x + y as the third item in the for-comprehension.

変換(Conversions)

The rules for implicit conversions of methods to functions (§6.26) have been tightened . Previously, a parameterized method used as a value was always implicitly converted to a function. This could lead to unexpected results when method arguments where forgotten. Consider for instance the statement below:

```
show(x.toString)
```

where show is defined as follows:

```
def show(x: String) = Console.println(x) .
```

Most likely, the programmer forgot to supply an empty argument list () to toString. The previous Scala version would treat this code as a partially applied method, and expand it to:

```
show(() => x.toString())
```

As a result, the address of a closure would be printed instead of the value of s. Scala version 2.0 will apply a conversion from partially applied method to function value only if the expected type of the expression is indeed a function type. For instance , the conversion would not be applied in the code above because the expected type of show's parameter is String, not a function type. The new convention disallows some previously legal code. Example:

```
def sum(f: int => double)(a: int, b: int): double =
  if (a > b) 0 else f(a) + sum(f)(a + 1, b)

val sumInts    =   sum(x => x)    // error: missing arguments
```

The partial application of sum in the last line of the code above will not be converted to a function type. Instead, the compiler will produce an error message which states that arguments for method sum are missing. The problem can be fixed by providing an expected type for the partial application, for instance by annotating the definition of sumInts with its type:

```
val sumInts: (int, int) => double    =   sum(x => x)    // OK
```

On the other hand, Scala version 2.0 now automatically applies methods with empty parameter lists to () argument lists when necessary. For instance, the show expression above will now be expanded to

```
show(x.toString()) .
```

Scala version 2.0 also relaxes the rules of overriding with respect to empty parameter lists. The revised definition of matching members (§5.1.3) makes it now possible to override a method with an explicit, but empty parameter list () with a parameterless method, and vice versa. For instance, the following class definition is now legal:

```
class C {
  override def toString: String = ...
}
```

Previously this definition would have been rejected, because the toString method as inherited from java.lang.Object takes an empty parameter list.

クラスパラメータ(Class Parameters)

A class parameter may now be prefixed by val or var (§5.3).

private修飾子(Private Qualifiers)

Previously, Scala had three levels of visibility: private, protected and public. There was no way to restrict accesses to members of the current package, as in Java. Scala 2 now defines access qualifiers that let one express this level of visibility, among others. In the definition

```
private[C] def f(...)
```

access to f is restricted to all code within the class or package C (which must contain the definition of f) (§5.2)

ミックスインモデルにおける変更

The model which details mixin composition of classes has changed significantly. The main differences are:

1. We now distinguish between traits that are used as mixin classes and normal classes. The syntax of traits has been generalized from version 1.0, in that traits are now allowed to have mutable fields. However, as in version 1.0, traits still may not have constructor parameters.
2. Member resolution and super accesses are now both defined in terms of a class linearization.
3. Scala's notion of method overloading has been generalized; in particular, it is now possible to have overloaded variants of the same method in a subclass and in a superclass, or in several different mixins. This makes method overloading in Scala conceptually the same as in Java.

The new mixin model is explained in more detail in §5.

暗黙のパラメータ(Implicit Parameters)

Views in Scala 1.0 have been replaced by the more general concept of implicit parameters (§7)

パターンマッチングの柔軟な型付け

The new version of Scala implements more flexible typing rules when it comes to patter n matching over heterogeneous class hierarchies (§8.4). A heterogeneous class hierarch y is one where subclasses inherit a common superclass with different parameter types. With the new rules in Scala version 2.0 one can perform pattern matches over such hier archies with more precise typings that keep track of the information gained by compari ng the types of a selector and a matching pattern (§Example 8.4.1). This gives Scala c apabilities analogous to guarded algebraic data types.