Bits and pieces of information about the parser, namer, and typer phases that turn out to be necessary just to be able to unparse Scala ASTs

© Miguel Garcia, LAMP, EPFL
`http://lamp.epfl.ch/~magarcia`

January 2nd, 2011

Abstract

It all started as an API migration tool (`jdk2ikvm`) which in turn required a pretty-printer for Scala ASTs (`scala.tools.unparse`) and soon details were needed about (1) the shape of trees built by the `scalac` parser; and (2) the synthetics added by `namer` and `typer`. This write-up zooms into some of those details (those useful when unparsing). In a nutshell, dealing with synthetic trees is a cumbersome aspect of unparsing. Printing type references is another. The sections in this write-up shift their focus from parser-related aspects (roughly, Sec. 2 to Sec. 7), to synthetics (Sec. 8). How type references are unparsed is discussed in a dedicated write-up `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2011Q1/TypesScaladocWay.pdf`.

Contents

1 Background

Build and run instructions for the compiler plugin described in these notes can be found in Sec. 1 of `http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q4/Unparsing.pdf`

We focus on a subset of the following:

```
protected def computeInternalPhases() {
  // Note: this fits -Xshow-phases into 80 column width, which it is
  // desirable to preserve.
  val phs = List(
    syntaxAnalyzer        -> "parse source into ASTs, perform simple desugaring", /*- mostly about this phase */
    analyzer.namerFactory -> "resolve names, attach symbols to named trees", /*- only enterSyntheticSym */
    analyzer.packageObjects -> "load package objects",
    analyzer.typerFactory -> "the meat and potatoes: type the trees", /*- hopefully only until here */
    superAccessors        -> "add super accessors in traits and nested classes",
    pickler               -> "serialize symbol tables",
    refchecks             -> "reference/override checking, translate nested objects",

  . . .
```

Regarding Scala ASTs, a good intro is the report[1] (especially App. D) by Mirko Stocker about the *Scala Refactoring library*, `http://scala-refactoring.org`

2 The shape of `Trees` delivered by `parser`

We focus mostly on the section in `nsc/ast/parser/Parsers.scala` tagged:

```
/* ---------- TREE CONSTRUCTION ------------------------------------------ */
```

2.1 Sample: `This`

Nota bene: If you're a wiz kid please skip this subsection.

At unparse time each node type is visited, e.g. `This`. To find out what textual syntax gave origin to it, one may (must?) check in `Parsers.scala` all places where the node's case class is instantiated.

1. As part of

```
    /** Path      ::= StableId
     *             |  [Ident '.'] this
     *  AnnotType ::= Path ['.' type]
     */
    def path(thisOK: Boolean, typeOK: Boolean): Tree = {
```

   a `This` node is returned in the following cases:

   (a) `t = atPos(start) { This(tpnme.EMPTY) }`

   (b) `t = atPos(start) { This(name.toTypeName) }`

2. As part of

[1] `http://scala-refactoring.org/wp-content/uploads/scala-refactoring.pdf`

```
/** ImportExpr ::= StableId '.' (Id | '_' | ImportSelectors)
 */
def importExpr(): Tree = {
```

a This node is returned in the following case:

(a)
```
loop(in.token match {
  case THIS  => thisDotted(tpnme.EMPTY)
  case _     =>
    val id = atPos(start)(Ident(ident()))
    accept(DOT)
    if (in.token == THIS) thisDotted(id.name.toTypeName)
    else id
})
```

Correspondingly we should unparse a This node follows:

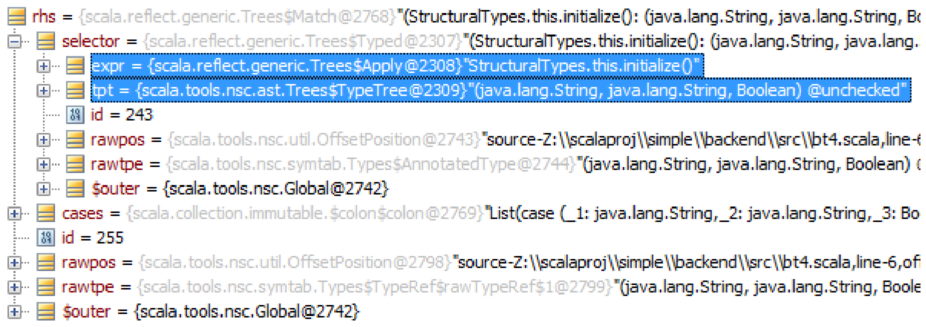
```
/*- sketch, final version appears in the next listing */
    override def This(tree: This, qual: Name): Tile = {
      if(qual.toString == "") "this" // specifically if qual eq tpnme.EMPTY
      else qual.asIdentifier + "." + "this"
    }
```

However reality is more complex, in that the phaser is not the only phase getting its hands at Trees (namer and typer add synthetics) and thus to cope with those synthetics the final version looks like:

```
override def This(tree: This, qual: Name): Tile = {
  if(qual.toString == "") "this" // specifically if qual eq tpnme.EMPTY
  else {
    val sym = tree.symbol
    // skip the 'this' in 'scala.this', same for 'java.this.lang'
    val isScalaThis = (sym eq definitions.ScalaPackage) || (sym eq definitions.ScalaPackageClass)
    val isJavaThis = tree.toString == "java.this" // TODO get hold of module class symbol 'package java'
    val isScalaCollectionImmutableThis = tree.toString == "immutable.this" // TODO get hold of module class symb
    if(isScalaThis) "scala"
    else if(isJavaThis) "java"
    else if(isScalaCollectionImmutableThis) "scala.collection.immutable"
    else if(tree.symbol.isInstanceOf[ModuleClassSymbol])
      qual.asIdentifier // eg., reflect.this.Manifest
    else
      qual.asIdentifier ~ "." ~ "this" // eg., C.this with C a class
  }
}
```

3 Unparsing pattern definitions, with synthetics and all

As you can imagine, other occurrences of synthetic nodes are awaiting. In fact it's the only cumbersome aspect of unparsing. The procedure to deal with them is exemplified in more detail in this section, in the context of pattern definitions.



```
rhs = {scala.reflect.generic.Trees$Match@2768}"(StructuralTypes.this.initialize(): (java.lang.String, java.lang.String, B
  selector = {scala.reflect.generic.Trees$Typed@2307}"(StructuralTypes.this.initialize(): (java.lang.String, java.lang.
    expr = {scala.reflect.generic.Trees$Apply@2308}"StructuralTypes.this.initialize()"
    tpt = {scala.tools.nsc.ast.Trees$TypeTree@2309}"(java.lang.String, java.lang.String, Boolean) @unchecked"
    id = 243
    rawpos = {scala.tools.nsc.util.OffsetPosition@2743}"source-Z:\\scalaproj\\simple\\backend\\src\\bt4.scala,line-
    rawtpe = {scala.tools.nsc.symtab.Types$AnnotatedType@2744}"(java.lang.String, java.lang.String, Boolean)
    $outer = {scala.tools.nsc.Global@2742}
  cases = {scala.collection.immutable.$colon$colon@2769}"List(case (_1: java.lang.String,_2: java.lang.String,_3: Bo
  id = 255
  rawpos = {scala.tools.nsc.util.OffsetPosition@2798}"source-Z:\\scalaproj\\simple\\backend\\src\\bt4.scala,line-6,of
  rawtpe = {scala.tools.nsc.symtab.Types$TypeRef$rawTypeRef$1@2799}"(java.lang.String, java.lang.String, Boole
  $outer = {scala.tools.nsc.Global@2742}
```

Figure 1: Pattern definition (Sec. 3.1)

3.1 What the spec says

Quoting from the SLS §4.1:

> *Value definitions can alternatively have a pattern (§8.1) as left-hand side. If p is some pattern other than a simple name or a name followed by a colon and a type, then the value definition val p = e is expanded as follows:*
>
> 1. *If the pattern p has bound variables $x_1, \ldots, x_n$, where $n > 1$:*
>
> ```
> val $x = e match {case p => {x1, . . . , xn}}
> val x1 = $x._1
> . . .
> val xn = $x._n .
> ```
>
> *Here, $x is a fresh name.*
>
> 2. *If p has a unique bound variable x:*
>
> ```
> val x = e match { case p => x }
> ```
>
> 3. *If p has no bound variables:*
>
> ```
> e match { case p => ()}
> ```

3.2 Tree shape to unparse: an example

```
   val (settings, sourceFiles, javaOnly) = initialize
   def initialize() = ("a", "b", true)
```

results in a `Match` node reaching our plugin (Figure 1) whose `Typed` selector contains way too much information than we want to unparse.

```
 /** Type annotation, eliminated by explicit outer */
 case class Typed(expr: Tree, tpt: Tree)
     extends TermTree
```

The `expr` part of the `Typed` node in Figure 1 is all right for unparsing in full, but its `tpt` part (a `TypeTree`) duplicates in its `orig` field (which contains an

Annotated) information about the selector. We want just the annot part of that orig value.

3.3 For comparison, -Yshow-trees output

```
object detailedTrees extends {
  val global: JDK2IKVM.this.global.type = JDK2IKVM.this.global
} with nsc.ast.NodePrinters {
  infolevel = InfoLevel.Normal /*- InfoLevel.Normal is enough */
  def cunit2string(unit: CompilationUnit): String = {
    Option(unit.body) map (x => nodeToString(x) + "\n") getOrElse "<null>"
  }
}
```

and then

```
val detailTile = row(StrTile("/* "), StrTile(detailedTrees.cunit2string(unit)), StrTile(" */"))
```

3.4 Sidenote: where pattern definitions are rewritten

3.4.1 The ValDefs that parser delivers

In TreeBuilders.scala:

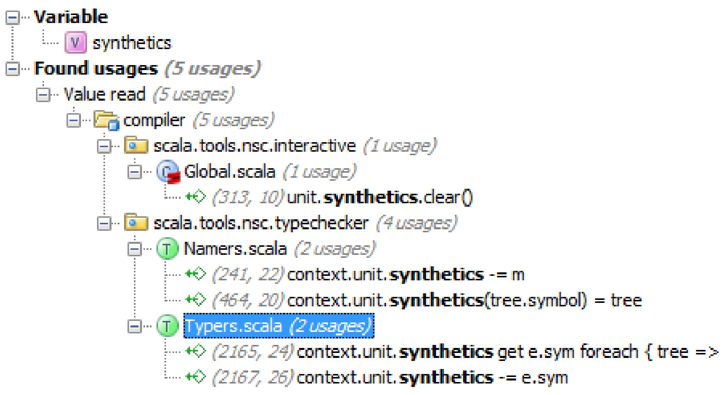
```
/** Create tree for pattern definition <val pat0 = rhs> */
def makePatDef(pat: Tree, rhs: Tree): List[Tree] =
  makePatDef(Modifiers(0), pat, rhs)
```

The above is called from the actual rule in Parsers.scala:

```
/** PatDef ::= Pattern2 {‘,’ Pattern2} [‘:’ Type] ‘=’ Expr
 *  ValDcl ::= Id {‘,’ Id} ‘:’ Type
 *  VarDef ::= PatDef | Id {‘,’ Id} ‘:’ Type ‘=’ ‘_’
 */
def patDefOrDcl(pos : Int, mods: Modifiers): List[Tree] = {
  /*- returns a List[Tree] for what appears syntactically as a single ValDef */
 . . .
```

Details about makePatDef can be found at Listing 1. For the example at hand, it returns trees that looks as follows:

```
private val x$1 = (
  (StructuralTypes.this.initialize : @scala.unchecked)
  match {
    case Tuple3(settings, sourceFiles, javaOnly) =>
       scala.Tuple3.apply[String, String, Boolean](settings, sourceFiles, javaOnly)
  }
)
val settings : String = StructuralTypes.this.x$1._1
val sourceFiles : String = StructuralTypes.this.x$1._2
val javaOnly : Boolean = StructuralTypes.this.x$1._3
```



```
Variable
  synthetics
Found usages (5 usages)
  Value read (5 usages)
    compiler (5 usages)
      scala.tools.nsc.interactive (1 usage)
        Global.scala (1 usage)
          (313, 10) unit.synthetics.clear()
      scala.tools.nsc.typechecker (4 usages)
        Namers.scala (2 usages)
          (241, 22) context.unit.synthetics -= m
          (464, 20) context.unit.synthetics(tree.symbol) = tree
        Typers.scala (2 usages)
          (2165, 24) context.unit.synthetics get e.sym foreach { tree =>
          (2167, 26) context.unit.synthetics -= e.sym
```

Figure 2: Adding the `Trees` for synthetics (Sec. 3.4.2)

3.4.2 What `namer` does with those `ValDefs`

The `namer` phase performs no rewriting for the trees in question. But now that we're talking about `namer`, here's what it does when it does add trees.

As a rule, when `namer` creates synthetic `Trees` (for just created symbols) it does not hang them right away to their future parents. Instead, it keeps that association by means of a `[Symbol, Tree]` map, `CompilationUnit.synthetics`:

```
def enterSyntheticSym(tree: Tree): Symbol = {
  enterSym(tree)
  context.unit.synthetics(tree.symbol) = tree
  tree.symbol
}
```

Afterwards, in `Typers.scala`, those trees are added as children (Figure 2).

3.5 Solving the mystery why pattern definitions didn't unparse properly

The call to `makeUnchecked` (highlighted in Listing 1) indicates the synthesized selector to be an `Annotated` node:

```
  private def makeUnchecked(expr: Tree): Tree = atPos(expr.pos) {
    Annotated( /*- annot == */ New(scalaDot(definitions.UncheckedClass.name), List(Nil)),
               /*- arg   == */ expr)
  }
```

However, after `typer`, we get a `Typed` node for the selector (as we saw in Figure 1). The original `Annotated` node has been re-packaged as the `orig` in the selector's `tpt`.

For unparsing purposes, we have to unearth the annotation (only) of that `Annotated` node. We do that not by intercepting `Match` but in the handler for `Typed` (thus it will be more generally applicable) as follows:

```
override def Typed(tree: Typed, expr: Tree, tpt: Tree): Tile = {
  /* sample usage outside pattern matching:
   *      Console.printf(text, args: _*)
   */
  val exprPart = xform(expr)
  val tptPart = tpt match {
    case tt : TypeTree if (tt.original.isInstanceOf[Annotated]) =>
      /* see discussion on unparsing pattern definitions at
         http://lamp.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/2010Q4/parser.pdf
       */
      // discard the annotation's arg
      val (_, annotationPart) = unparseAnnotated(tt.original.asInstanceOf[Annotated])
      annotationPart
    case _ =>
      xform(tpt)
  }
  forceParens(exprPart ~ ": " ~ tptPart)
}
```

4 Modifiers in formal params of the primary constructor

4.1 What should be unparsed

Quoting from the SLS §5.3:

- *If a formal parameter declaration x : T is preceded by a* `val` *or* `var` *keyword, an accessor (getter) definition (§4.2) for this parameter is implicitly added to the class . . .*
- *The formal parameter declaration may contain modifiers, which then carry over to the accessor definition(s).*
- *A formal parameter prefixed by* `val` *or* `var` *may not at the same time be a call-by-name parameter (§4.6.1).*

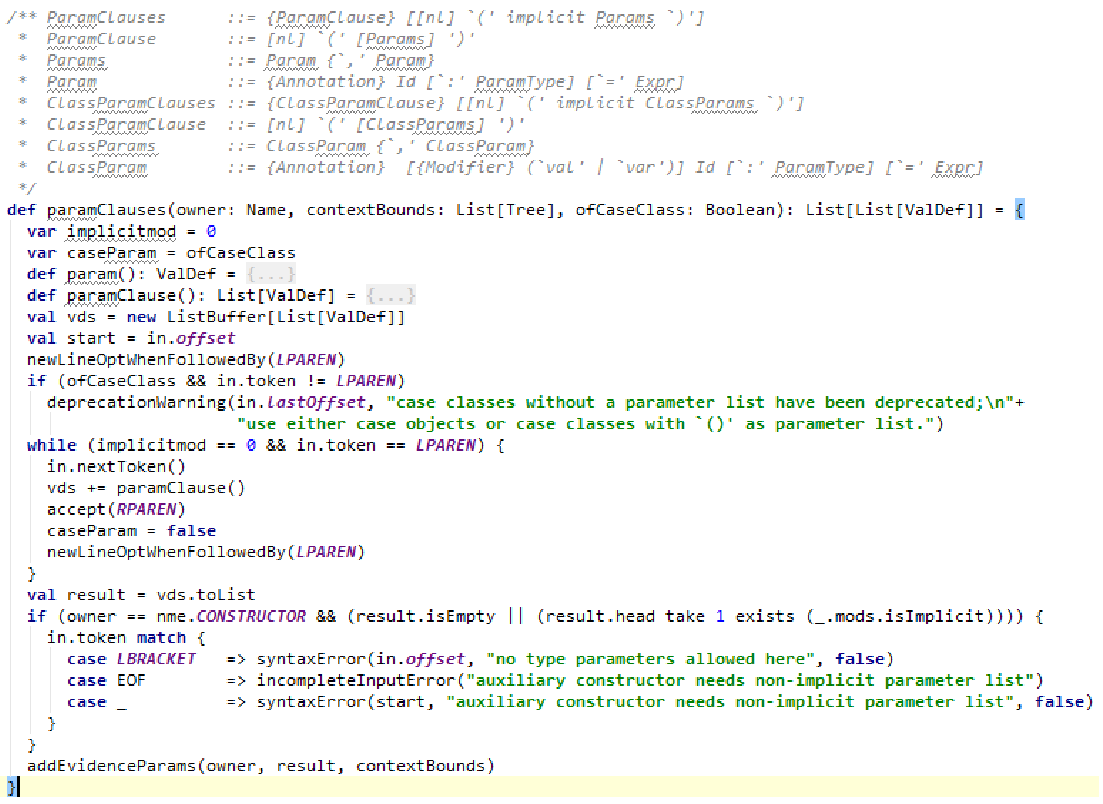
4.2 What the parser does about value param clauses

All things value-param-clauses (be they for the primary constructor of a class, or a `funDefOrDecl`) are the domain of `paramClauses` (Figure 3) which for example adds *evidence params* for context bounds:

```
  /** Append implicit parameter section if 'contextBounds' nonempty */
  def addEvidenceParams(owner: Name, vparamss: List[List[ValDef]], contextBounds: List[Tree]): List[List[ValDef]
    if (contextBounds.isEmpty) vparamss
    else {
      val mods = Modifiers(if (owner.isTypeName) PARAMACCESSOR | LOCAL | PRIVATE else PARAM)
      def makeEvidenceParam(tpt: Tree) = ValDef(mods | IMPLICIT, freshName(nme.EVIDENCE_PARAM_PREFIX), tpt, Empt
      vparamss ::: List(contextBounds map makeEvidenceParam)
  }
```

The following snippet shows where the param's modifiers and `val`/`var` keywords end up:

```
var mods = Modifiers(Flags.PARAM)
if (owner.isTypeName) {
  mods = modifiers() | Flags.PARAMACCESSOR
  /*- mods of all params of the primary constructor of a class or case class are tagged PARAMACCESSOR */
  if (mods.isLazy) syntaxError("lazy modifier not allowed here. Use call-by-name parameters instead", false)
```



```
/** ParamClauses      ::= {ParamClause} [[nl] `(' implicit Params `)']
 *  ParamClause       ::= [nl] `(' [Params] ')'
 *  Params            ::= Param {`,' Param}
 *  Param             ::= {Annotation} Id [`:' ParamType] [`=' Expr]
 *  ClassParamClauses ::= {ClassParamClause} [[nl] `(' implicit ClassParams `)']
 *  ClassParamClause  ::= [nl] `(' [ClassParams] ')'
 *  ClassParams       ::= ClassParam {`,' ClassParam}
 *  ClassParam        ::= {Annotation}  [{Modifier} (`val' | `var')] Id [`:' ParamType] [`=' Expr]
 */
def paramClauses(owner: Name, contextBounds: List[Tree], ofCaseClass: Boolean): List[List[ValDef]] = {
  var implicitmod = 0
  var caseParam = ofCaseClass
  def param(): ValDef = {...}
  def paramClause(): List[ValDef] = {...}
  val vds = new ListBuffer[List[ValDef]]
  val start = in.offset
  newLineOptWhenFollowedBy(LPAREN)
  if (ofCaseClass && in.token != LPAREN)
    deprecationWarning(in.lastOffset, "case classes without a parameter list have been deprecated;\n"+
                       "use either case objects or case classes with `()' as parameter list.")
  while (implicitmod == 0 && in.token == LPAREN) {
    in.nextToken()
    vds += paramClause()
    accept(RPAREN)
    caseParam = false
    newLineOptWhenFollowedBy(LPAREN)
  }
  val result = vds.toList
  if (owner == nme.CONSTRUCTOR && (result.isEmpty || (result.head take 1 exists (_.mods.isImplicit)))) {
    in.token match {
      case LBRACKET   => syntaxError(in.offset, "no type parameters allowed here", false)
      case EOF        => incompleteInputError("auxiliary constructor needs non-implicit parameter list")
      case _          => syntaxError(start, "auxiliary constructor needs non-implicit parameter list", false)
    }
  }
  addEvidenceParams(owner, result, contextBounds)
}
```

Figure 3: The parser and param clauses (Sec. 4.2)

```
  in.token match {
    case v @ (VAL | VAR) =>
      mods = mods withPosition (in.token, tokenRange(in)) /*- only way to recover whether a VAL was there */
      if (v == VAR) mods |= Flags.MUTABLE
      in.nextToken()
    case _ =>
      if (mods.flags != Flags.PARAMACCESSOR) accept(VAL)
      if (!caseParam) mods |= Flags.PRIVATE | Flags.LOCAL
  }
  if (caseParam) mods |= Flags.CASEACCESSOR
}
```

Summing up:

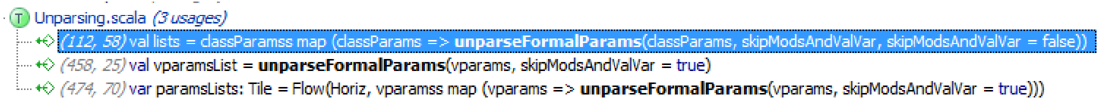
- `var` is recovered by simply checking for `Flags.MUTABLE`
- `val` is recovered by checking in the `mods` positions whether a `Tokens.VAL` long appears as key.
- other modifiers grab them from `mods` as usual

4.3 Solution

Solution: when unparsing a `ClassDef`:

```
def unparseFormalParam(t: ValDef, skipModsAndValVar: Boolean, printTPEIfNoOriginalTPT: Boolean) = {

  val ValDef(mods, name, tpt, rhs) = t

  // TODO mods.annotations

  val modsPart =
    if(skipModsAndValVar) ""
    else {
      // as per paramClauses() in Parsers.scala
      val nonPrivMods = mods.&~(Flags.PRIVATE)
      /* the ValDef mods comes from was grabbed from a template body where isParamAccessor,
         and by that time had been marked PRIVATE */
      val m = unparseFlags(nonPrivMods.flags, mods.privateWithin.asIdentifier.toString)
      val v = if(mods.isMutable) "var "
              else if (mods.positions.contains(nsc.ast.parser.Tokens.VAL)) "val "
              else ""
      if(m.isEmpty) v else (m + " " + v)
    }

  val (rhsPart, tptPart) = unparseValDefRHSAndTPT(t, printTPEIfNoOriginalTPT)

  modsPart ~ name.asIdentifier ~ tptPart ~ rhsPart
}
```

Neither a `DefDef` node nor a `Function` node will have their formal params unparsed with modifiers and/or `val`/`var`, thus the `skipModsAndValVar = true`:



```
Unparsing.scala (3 usages)
  (112, 58) val lists = classParamss map (classParams => unparseFormalParams(classParams, skipModsAndValVar, skipModsAndValVar = false))
  (458, 25) val vparamsList = unparseFormalParams(vparams, skipModsAndValVar = true)
  (474, 70) var paramsLists: Tile = Flow(Horiz, vparamss map (vparams => unparseFormalParams(vparams, skipModsAndValVar = true)))
```

5 Miscellanea

5.1 By-name and repeated params

Another example of figuring out a cause-effect relationship by starting from the effect. The error we get:

```
c:\temp\src\library\scala\DelayedInit.scala:6: error: not found: type byname
   def delayedInit(x : _root_.scala.<byname>[Unit]) : Unit
                                      ^
```

we notice that `TreeBuilders` has:

```
  def byNameApplication(tpe: Tree): Tree =
    AppliedTypeTree(rootScalaDot(tpnme.BYNAME_PARAM_CLASS_NAME), List(tpe))

  def repeatedApplication(tpe: Tree): Tree =
    AppliedTypeTree(rootScalaDot(tpnme.REPEATED_PARAM_CLASS_NAME), List(tpe))
```

which is only invoked from:

```
/** ParamType ::= Type | '=>' Type | Type '*'
 */
def paramType(): Tree = paramType(useStartAsPosition = false)
def paramType(useStartAsPosition: Boolean): Tree = {
  val start = in.offset
  in.token match {
    case ARROW =>
      in.nextToken()
      atPos(start)(byNameApplication(typ()))
    case _     =>
      val t = typ()
      if (isRawStar) {
        in.nextToken()
        if (useStartAsPosition) atPos(start)(repeatedApplication(t))
        else atPos(t.pos.startOrPoint, t.pos.point)(repeatedApplication(t))
      }
      else t
  }
}
```

Correspondingly, unparsing looks like:

```
override def AppliedTypeTree(tree: AppliedTypeTree, tpt: Tile, args: List[Tile]): Tile = {
  tree.tpt match {
    case Select(_, tpnme.BYNAME_PARAM_CLASS_NAME) =>
      assert(args.size == 1)
      "=> " ~ args.head
    case Select(_, tpnme.REPEATED_PARAM_CLASS_NAME) =>
      assert(args.size == 1)
      forceParens(args.head) ~ " *"
    case _ =>
      tpt ~ brackets(args: _*)
  }
}
```

5.2 `package object`

From `Parsers.scala`:

```
/** Create a tree representing a package object, converting
 *    package object foo { ... }
 *  to
 *    package foo {
 *      object `package` { ... }
 *    }
 */
def makePackageObject(start: Int, objDef: ModuleDef): PackageDef = objDef match {
  case ModuleDef(mods, name, impl) =>
    makePackaging(
      start, atPos(o2p(objDef.pos.startOrPoint)){ Ident(name) }, List(ModuleDef(mods, nme.PACKAGEkw, impl)))
}

/** Create a tree representing a packaging */
  def makePackaging(start: Int, pkg: Tree, stats: List[Tree]): PackageDef = pkg match {
    case x: RefTree => atPos(start, pkg.pos.point)(PackageDef(x, stats))
  }
```

Thus the shape to detect when visiting a `PackageDef` node involves:

- `stats` contains a single `ModuleDef` m
- the name of m is `nme.PACKAGEkw`

5.3 Annotations and access modifier for the primary constructor

Quoting from the SLS §5.3:

> *The most general form of class definition is*
>
> `class c[tps] as m(ps1). . .(psn) extends t (n >= 0)`
>
> - *as is a possibly empty sequence of annotations (§11). If any annotations are given, they apply to the primary constructor of the class.*
> - *m is an access modifier (§5.2) such as private or protected, possibly with a qualification. If such an access modifier is given it applies to the primary constructor to the class.*

```
TODO
```

5.4 `implicit` modifier for named param of anonymous function

Quoting from SLS §6.23 ("Anonymous Functions"):

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

. . .

A named parameter of an anonymous function may be optionally preceded by an `implicit` *modifier. In that case the parameter is labeled implicit (§7); however the parameter section itself does not count as an implicit parameter section in the sense of (§7.2). Hence, arguments to anonymous functions always have to be given explicitly.*

TODO

5.5 Simplifications purposely left out

`TreePrinters.scala` shows a number of code reductions that could be applied, but so far we've desisted from doing so (this is an unparser, not a pretty-printer).

```
    // if a Block only contains one actual statement, just print it.
    case Block(stats, expr) =>
      allStatements(tree) match {
        case List(x)          => printRaw(x)
        case xs               => s()
      }

    // We get a lot of this stuff
    case If( IsTrue(), x, _)     => printRaw(x)
    case If(IsFalse(), _, x)     => printRaw(x)

    case If(cond, IsTrue(), elsep) => printLogicalOr(cond -> true, elsep -> true)
    case If(cond, IsFalse(), elsep) => printLogicalAnd(cond -> false, elsep -> true)
    case If(cond, thenp, IsTrue()) => printLogicalOr(cond -> false, thenp -> true)
    case If(cond, thenp, IsFalse()) => printLogicalAnd(cond -> true, thenp -> true)

    // If thenp or elsep has only one statement, it doesn't need more than one line.
    case If(cond, thenp, elsep) =>
```

```
// the unparser also performs block flattening, only that differently

// drill down through Blocks and pull out the real statements.
def allStatements(t: Tree): List[Tree] = t match {
  case Block(stmts, expr) => (stmts flatMap allStatements) ::: List(expr)
  case _                  => List(t)
}

. . .

// if a Block only continues one actual statement, just print it.
case Block(stats, expr) =>
  allStatements(tree) match {
    case List(x)          => printRaw(x)
    case xs               => s()
  }
```

6 Recovering original member modifiers from `flags`

For those of us who haven't delved yet into the parser of `scalac` (that's borderline illiteracy some say).

6.1 Local vs. access vs. other modifiers

First there's the very reasonable distinction between local, access (i.e, `private`, `protected`), and "other" modifiers (the lonely `override`):

```
def isLocalModifier: Boolean = in.token match {
  case ABSTRACT | FINAL | SEALED | IMPLICIT | LAZY => true
  case _ => false
}

/** AccessModifier ::= (private | protected) [AccessQualifier]
 */
def accessModifierOpt(): Modifiers = normalize {
  in.token match {
    case m @ (PRIVATE | PROTECTED) => in.nextToken() ; accessQualifierOpt(Modifiers(flagTokens(m)))
    case _                         => NoMods
  }
}

def isModifier: Boolean = in.token match {
  case ABSTRACT | FINAL | SEALED | PRIVATE |
       PROTECTED | OVERRIDE | IMPLICIT | LAZY => true
  case _ => false
}
```

6.2 Double representation of modifiers (flags and positions)

Then there's the double representation of modifiers, as flags (look at `(mods | mod)` in the snippet below) and as part of a `Modifier`'s `positions: Map[Long, Position]`.

```
private def addMod(mods: Modifiers, mod: Long, pos: Position): Modifiers = {
  if (mods hasFlag mod) syntaxError(in.offset, "repeated modifier", false)
  in.nextToken()
  (mods | mod) withPosition (mod, pos)
}
```

6.3 Disappearing `PRIVATE` and contracting `ABSOVERRIDE`

You might also want to know why one flag mysteriously disappears (`Flags.PRIVATE`) while another mysteriously comes out of nowhere (`Flags.ABSOVERRIDE`):

```
/** Drop 'private' modifier when followed by a qualifier.
 *  Contract 'abstract' and 'override' to ABSOVERRIDE
 */
private def normalize(mods: Modifiers): Modifiers =
  if (mods.isPrivate && mods.hasAccessBoundary)
    normalize(mods &~ Flags.PRIVATE)
  else if (mods hasAllFlags (Flags.ABSTRACT | Flags.OVERRIDE))
    normalize(mods &~ (Flags.ABSTRACT | Flags.OVERRIDE) | Flags.ABSOVERRIDE)
  else
    mods
```

6.4 Recovering

For now done as follows:

```
private def unparseFlags(flags: Long, privateWithin: String): String = {
  import Flags._
  // we include only those flags for which flagsToString already delivers printable output
  val maskLocal = ABSTRACT | FINAL | SEALED | IMPLICIT | LAZY
  val maskAccess = PROTECTED | PRIVATE
  val maskOther = OVERRIDE
  val mask = maskLocal | maskAccess | maskOther
  val ao = if((flags & ABSOVERRIDE) != 0L) "abstract override "
         else ""
  val res0 = flagsToString(flags & mask, privateWithin)
  ao + res0
}
```

7 Type params: Variance and Bounds (upper, lower, context, view)

```
Regarding the encoding in ASTs of the syntax:

/** TypeParamClauseOpt  ::= [TypeParamClause]
 *  TypeParamClause     ::= '[' VariantTypeParam {',' VariantTypeParam} ']']
 *  VariantTypeParam    ::= {Annotation} ['+' | '-'] TypeParam
 *  FunTypeParamClauseOpt ::= [FunTypeParamClause]
 *  FunTypeParamClause  ::= '[' TypeParam {',' TypeParam} ']']
 *  TypeParam           ::= Id TypeParamClauseOpt TypeBounds {<% Type} {":" Type}
 */
```

Some quick lessons from Listing 2 are:

- `+` is represented with `Flags.COVARIANT`
- `-` is represented with `Flags.CONTRAVARIANT`
- invariance sets no dedicated flag
- view bound specs are tracked in `contextBoundBuf`, by inserting a `makeFunctionTypeTree(List(Ident(pname)), typ())`
- context bound specs are also tracked in `contextBoundBuf`, by inserting a `AppliedTypeTree(typ(), List(Ident(pname)))`
- those context bounds are turned into evidence parameters by `paramClauses`

8 Unparsing case classes and case objects

8.1 Where they get their synthetics

8.1.1 `extends scala.Product() with scala.Serializable`

Added by the parser, as shown below. However, by the time `typer` is over, `parents` contains not the `Select`s added below, but `TypeTree`s whose `orig` field wraps those `Select`s.

```
/** ClassTemplateOpt ::= 'extends' ClassTemplate | [['extends'] TemplateBody]
 *  TraitTemplateOpt ::= TraitExtends TraitTemplate | [['extends'] TemplateBody] | '<:' TemplateBody
 *  TraitExtends    ::= 'extends' | `<:'
 */
def templateOpt(mods: Modifiers, name: Name, constrMods: Modifiers, vparamss: List[List[ValDef]], tstart: Int):
  val (parents0, argss, self, body) =
    if (in.token == EXTENDS || settings.YvirtClasses && mods.hasTraitFlag && in.token == SUBTYPE) {
      in.nextToken()
      template(mods.hasTraitFlag)
    } else if ((in.token == SUBTYPE) && mods.hasTraitFlag) {
      in.nextToken()
      template(true)
    } else {
      newLineOptWhenFollowedBy(LBRACE)
      val (self, body) = templateBodyOpt(false)
      (List(), List(List()), self, body)
    }
  var parents = parents0
  if (!isInterface(mods, body) && !isScalaArray(name))
    parents = parents :+ scalaScalaObjectConstr
  if (parents.isEmpty)
    parents = List(scalaAnyRefConstr)
  if (mods.isCase) parents ++= List(productConstr, serializableConstr) /*- <--- here <<<<<<< */
  val tstart0 = if (body.isEmpty && in.lastOffset < tstart) in.lastOffset else tstart
  atPos(tstart0) {
    Template(parents, self, constrMods, vparamss, argss, body, o2p(tstart))
  }
}
```

8.1.2 Auxiliary methods

At `SyntheticMethods.scala`. Quoting from the source comments:

- *`productArity`, `element` implementations added to case classes*
- *`equals`, `hashCode` and `toString` methods are added to case classes, unless they are defined in the class or a baseclass different from `java.lang.Object`*
- *`toString` method is added to case objects, unless they are defined in the class or a baseclass different from `java.lang.Object`*

8.2 Where we don't unparse those synthetics

8.2.1 `extends scala.Product() with scala.Serializable`

We skip this at `TemplateMethods.explicitParents`:

```
/** Parents except those for ObjectClass and ScalaObjectClass.
 *  Also skips scala.Product and scala.Serializable from case classes and case objects. */
def explicitParents(isCase: Boolean) : List[Tree] = {

    def isScalaDot(t: Tree, name: Name) = t match {
      case tt : TypeTree if (tt.original != null) => tt.original match {
        case Select(Ident(nme.scala_), n) if (n == name) =>
          true // not checking symbol just in case the tree was built manually
        case _ => false
      }
      case _ => false
    }
    def isProductConstr(t: Tree)   = isScalaDot(t, tpnme.Product)
    def isSerializableConstr(t: Tree) = isScalaDot(t, tpnme.Serializable)

  val parents0 = t.parents filterNot (_.isEmpty) filterNot
                          (_.symbol eq definitions.ScalaObjectClass) filterNot
                          (_.symbol eq definitions.ObjectClass)

  if(!isCase) parents0 else {
    parents0 filterNot (p => isProductConstr(p) || isSerializableConstr(p))
  }

}
```

8.2.2 Auxiliary methods

We skip them at `TemplateMethods.originalBody`:

```
/** Statements, except:
 *    (a) the synthetic primary constructor and the fields for its parameters,
 *    (b) the auxiliary methods of case classes and case objects. */
def originalBody(isCase: Boolean) : List[Tree] = {
  val primaryConstructor = primaryConstructorOpt getOrElse EmptyTree
  val res0 = t.body filterNot ((primaryConstructor :: flattenedClassParams) contains) filterNot (_.isEmpty)
  // filter (s => s.symbol != null && s.symbol.isSourceMethod)
  val res1 = res0 filterNot (s => s.symbol != null && s.symbol.isSynthetic)

  val auxCaseMethNames =
    List("hashCode", "toString", "equals", "productPrefix", "productArity", "productElement", "canEqual", "readRe
  def isAuxCaseMethod(t: Tree) = t match {
      case dd : DefDef => auxCaseMethNames contains dd.name.toString.trim // TODO waiting for better way to dete
      case _ => false
    }
  if(!isCase) res1 else res1 filterNot (t => isAuxCaseMethod(t))
}
```

Listing 1: Some details about `makePatDef`, Sec. 3.4.1

```
/** Create tree for pattern definition <mods val pat0 = rhs> */
def makePatDef(mods: Modifiers, pat: Tree, rhs: Tree): List[Tree] = matchVarPattern(pat) match {
  case Some((name, tpt)) =>
    List(atPos(pat.pos union rhs.pos) {
      ValDef(mods, name, tpt, rhs)
    })

  case None =>
    //  in case there is exactly one variable x_1 in pattern
    //  val/var p = e ==> val/var x_1 = e.match (case p => (x_1))
    //
    //  in case there are zero or more than one variables in pattern
    //  val/var p = e ==> private synthetic val t$ = e.match (case p => (x_1, ..., x_N))
    //                 val/var x_1 = t$._1
    //                 ...
    //                 val/var x_N = t$._N
    val pat1 = patvarTransformer.transform(pat)
    val vars = getVariables(pat1)
    val matchExpr = atPos((pat1.pos union rhs.pos).makeTransparent) {
      Match(
        makeUnchecked(rhs), /*- check this call to makeUnchecked */
        List(
          atPos(pat1.pos) {
            CaseDef(pat1, EmptyTree, makeTupleTerm(vars map (_._1) map Ident, true))
          }
        ))
    }
    vars match {
      case List((vname, tpt, pos)) =>
        List(atPos(pat.pos union pos union rhs.pos) {
          ValDef(mods, vname, tpt, matchExpr)
        })
      case _ =>
        val tmp = freshName()
        val firstDef =
          atPos(matchExpr.pos) {
            ValDef(Modifiers(PRIVATE | LOCAL | SYNTHETIC | (mods.flags & LAZY)),
                   tmp, TypeTree(), matchExpr)
          }
        var cnt = 0
        val restDefs = for ((vname, tpt, pos) <- vars) yield atPos(pos) {
          cnt = cnt + 1
          ValDef(mods, vname, tpt, Select(Ident(tmp), newTermName("_" + cnt)))
        }
        firstDef :: restDefs
    }
}
```

Listing 2: Type params: Variance and Bounds (upper and lower, context and view), Sec. 7

```
/** TypeParamClauseOpt  ::= [TypeParamClause]
 *  TypeParamClause     ::= '[' VariantTypeParam {',' VariantTypeParam} ']']
 *  VariantTypeParam    ::= {Annotation} ['+' | '-'] TypeParam
 *  FunTypeParamClauseOpt ::= [FunTypeParamClause]
 *  FunTypeParamClause  ::= '[' TypeParam {',' TypeParam} ']']
 *  TypeParam           ::= Id TypeParamClauseOpt TypeBounds {<% Type} {":" Type}
 */
def typeParamClauseOpt(owner: Name, contextBoundBuf: ListBuffer[Tree]): List[TypeDef] = {
  def typeParam(ms: Modifiers): TypeDef = {
    var mods = ms | Flags.PARAM
    val start = in.offset
    if (owner.isTypeName && isIdent) {
      if (in.name == raw.PLUS) {
        in.nextToken()
        mods |= Flags.COVARIANT
      } else if (in.name == raw.MINUS) {
        in.nextToken()
        mods |= Flags.CONTRAVARIANT
      }
    }
    val nameOffset = in.offset
    // TODO AM: freshName(o2p(in.skipToken()), "_$$"), will need to update test suite
    val pname: TypeName = wildcardOrIdent().toTypeName
    val param = atPos(start, nameOffset) {
      val tparams = typeParamClauseOpt(pname, null) // @M TODO null --> no higher-order context bounds for now
      TypeDef(mods, pname, tparams, typeBounds())
    }
    if (contextBoundBuf ne null) {
      while (in.token == VIEWBOUND) {
        contextBoundBuf += atPos(in.skipToken()) {
          makeFunctionTypeTree(List(Ident(pname)), typ())
        }
      }
      while (in.token == COLON) {
        contextBoundBuf += atPos(in.skipToken()) {
          AppliedTypeTree(typ(), List(Ident(pname)))
        }
      }
    }
    param
  }
  newLineOptWhenFollowedBy(LBRACKET)
  if (in.token == LBRACKET) inBrackets(commaSeparated(typeParam(NoMods withAnnotations annotations(true))))
  else Nil
}

/** TypeBounds ::= ['>:' Type] ['<:' Type]
 */
def typeBounds(): TypeBoundsTree = {
  val t = TypeBoundsTree(
    bound(SUPERTYPE, tpnme.Nothing),
    bound(SUBTYPE, tpnme.Any)
  )
  t setPos wrappingPos(List(t.hi, t.lo))
}

def bound(tok: Int, default: TypeName): Tree =
  if (in.token == tok) { in.nextToken(); typ() }
  else atPos(o2p(in.lastOffset)) { rootScalaDot(default) }
```