

Untangling Scala ASTs

© Miguel Garcia, STS, Hamburg University of Technology
<http://www.sts.tu-harburg.de/people/mi.garcia>

December 22, 2009

Contents

1	Trees, Symbols, and Types	1
1.1	Slicing the API on a need-to-know basis	1
1.2	During parsing	1
1.2.1	Tree shapes after parsing, syntax extensions	1
1.2.2	“The parser does too much”	3
1.2.3	For-comprehensions	3
2	Tree shapes after typing	6
2.1	What is reachable from what	7
2.1.1	Looking up declarations from usages	7
3	A code walkthrough: how Scala X-Ray works	11
4	Comparing ASTs across phases	15

Abstract

At each compilation phase, ASTs are expected to comply with certain well-formedness conditions (which embody the static semantics of the language fragment allowed at the end of that phase). Problem is, those conditions have not been documented in a machine-processable manner. These notes cannot fill that gap, they summarize instead some points useful to know when performing (simple) manipulations of Scala ASTs. Broadly speaking, AST processing can occur (a) during parsing, (b) after typechecking but before `ICode` generation, and (c) during bytecode or MSIL generation. These notes focus on (b), which comprises the compilation phases after `typer` and before `icode`.

Chapter 1

Trees, Symbols, and Types

1.1 Slicing the API on a need-to-know basis

The API exposed by `Tree` and `Symbol` can be understood more easily by noticing that not all of its methods and fields are used throughout all phases. For example, comments in the typechecker mention that symbols are typed over a number of iterations, along the lines of “a (compiler) run involves several (typing) phases which in turn comprise one or more periods”. The associated data structures are however *not accessed directly* once the `typer` phase has run. Therefore, developers of compiler-plugins operating after that phase need not understand in full things like:

```
// SymbolTable.scala
final def period(rid: RunId, pid: Phase#Id): Period =
  (currentRunId << 8) + pid

// Symbols.scala
private sealed case class TypeHistory(var validFrom: Period, info: Type, prev: TypeHistory) {
  assert((prev eq null) || phaseId(validFrom) > phaseId(prev.validFrom), this)
  assert(validFrom != NoPeriod)
  override def toString() =
    "TypeHistory(" + phaseOf(validFrom) + ": " + runId(validFrom) + ", " + info + ", " + prev + ")"
}
```

However, one does need to know (a) how to create and configure symbols, (b) which API to use to enter them into name tables, (c) how to create and configure trees for those symbols, and (d) how to invoke the `typer` API on an updated or new subtree. We’ll cover some aspects of this (unforgiving?) choreography in following sections.

1.2 During parsing

1.2.1 Tree shapes after parsing, syntax extensions

A few very specific use cases call for extending the syntax of Scala. If at all unavoidable, that can be achieved by (1) manually patching the lexer and/or parser; or by (2) replacing them with others supporting declarative syntax ex-

Listing 1.1: Trees the parser delivers (`scala.tools.nsc.ast.parser.SyntaxAnalyzer`)

```
abstract class SyntaxAnalyzer extends SubComponent
with Parsers with MarkupParsers with Scanners
with JavaParsers with JavaScanners {

  val phaseName = "parser"

  def newPhase(prev: Phase): StdPhase = new ParserPhase(prev)

  class ParserPhase(prev: scala.tools.nsc.Phase) extends StdPhase(prev) {
    override val checkable = false
    def apply(unit: global.CompilationUnit) {
      global.informProgress("parsing " + unit)
      unit.body =
        if (unit.source.fileName.endsWith(".java")) new JavaUnitParser(unit).parse()
        else if (!global.reporter.incompleteHandled) new UnitParser(unit).smartParse()
        else new UnitParser(unit).parse()
      if (global.settings.Yrangepos.value && !global.reporter.hasErrors)
        global.validatePositions(unit.body)
    }
  }
}
```

tensions (e.g., the *Rats!*-based parser used in the NetBeans¹ plugin for Scala). In both cases, knowing the tree shapes expected by later compilation phases is necessary. These trees can be visualized by placing a breakpoint after the assignment to `unit.body` in the source shown in Listing 1.1.

Broad-brush summary of the shapes immediately after parsing: the `rawpos` of each tree has been initialized, unlike its `symbol` (i.e. for those trees having symbols, subtypes of `SymTree`) which is invariably set to `NoSymbol`. The type of each tree, `rawtpe`, is `null`.

Given that each tree's symbol is `NoSymbol` at this stage, that symbol's `rawannots`, `rawpos`, and `rawowner` are meaningless. Beware of the similarly named `rawpos` in `Tree` and `Symbol`. Also beware that the type history of a symbol, maintained in field `infos`, is displayed by the debugger under a longish, mangled name.

Regarding annotations, those tree instances subtyping `MemberDef` sport field `mods: Modifiers` which in turn sports a field `annotations: List[Tree]`. The representation of annotations varies before and after typechecking, as covered in detail in SID 5 and summarized next:

Before type-checking, annotations are part of the AST; note that an annotation is simply a constructor call. Symbol annotations are stored in the definition's `Modifiers` ... During type-checking annotations are removed from the AST and attached to the corresponding symbol or type as instances of the class `AnnotationInfo` ... Symbol annotations can be accessed through the symbol of the corresponding definition using the method `annotations: List[AnnotationInfo]` ... Types with type annotations are represented in the compiler as instances of `AnnotatedType`

¹<http://wiki.netbeans.org/ScalaImpl>

1.2.2 “The parser does too much”

That’s an eye-catching title for a subsection, isn’t it? (Almost all) on-the-fly concrete-syntax-tree conversions performed by the parser are documented in `Parsers.scala` and reproduced next:

1. Places all pattern variables in Bind nodes. In a pattern, for identifiers x:

```
x => x @ _
x:T => x @ (_ : T)
```

2. Removes pattern definitions (PatDef’s) as follows: If pattern is a simple (typed) identifier:

```
val x = e    ==> val x = e
val x: T = e ==> val x: T = e
```

if there are no variables in pattern

```
val p = e ==> e match (case p => ())
```

if there is exactly one variable in pattern

```
val x_1 = e match (case p => (x_1))
```

if there is more than one variable in pattern

```
val p = e ==> private synthetic val t$ = e match (case p => (x_1, ..., x_N))
                val x_1 = t$._1
                ...
                val x_N = t$._N
```

3. Removes function types as follows:

```
(argtpes) => restpe ==> scala.Function_n[argtpes, restpe]
```

4. Wraps naked case definitions in a match as follows:

```
{ cases } ==> (x => x.match {cases})
```

, except when already argument to match

1.2.3 For-comprehensions

Further examples of context-free rewritings during parsing include:

- by-name argument evaluated too early when occurring after `-`-ending operator, <http://lampsvn.epfl.ch/trac/scala/ticket/1980>
- desugaring of for-comprehensions into invocations to `map`, `filter`, and `flatMap`, as documented in §6.19 of the SLS. A synopsis of its realization in `Parsers.scala` and `TreeBuilder.scala` follows.

Listing 1.2: Desugaring of for-comprehensions, part A

```
case FOR =>
  atPos(in.skipToken()) {
    val (open, close) = if (in.token == LBRACE) (LBRACE, RBRACE) else (LPAREN, RPAREN)
    val enums = surround(open, close)(enumerators(), Nil)
    newLinesOpt()
    if (in.token == YIELD) {
      in.nextToken()
      makeForYield(enums, expr())
    } else {
      makeFor(enums, expr())
    }
  }
}
```

Scala for-comprehensions follow the syntax

```
for ((' Enumerators ') | '{ Enumerators }') {nl} [yield] Expr
```

where each *enumerator* can be a *generator*, a *filter*, or a *val-declaration*. The production is handled by method `def expr(): Tree = expr(Local)` in `Parsers.scala` as shown in Listing 1.2.

After the AST has been built, depending on whether the keyword `yield` is present, one of two on-the-fly rewritings will be performed (`makeForYield` or `makeFor`), which are defined (in `TreeBuilder.scala`) as follows:

```
/** Create tree for for-do comprehension &lt;for (enums) body&gt;; */
def makeFor(enums: List[Enumerator], body: Tree): Tree =
  makeFor(nme.foreach, nme.foreach, enums, body)

/** Create tree for for-yield comprehension &lt;for (enums) yield body&gt;; */
def makeForYield(enums: List[Enumerator], body: Tree): Tree =
  makeFor(nme.map, nme.flatMap, enums, body)
```

The documentation for `TreeBuilder.makeFor` is shown in Listing 1.3.

In case you were asking, no, constant folding is *not* performed during parsing ;-)) but during typechecking: look for `ConstantFolder.scala` (that file is just 173 lines long and thus serves to test your teeth at AST processing). Conveniently enough, the ascription of types to updated trees involves only types of literals.

Listing 1.3: Desugaring of for-comprehensions, part B

```
/** Create tree for for-comprehension <math>\text{for (enums) do body}</math>; or
 * <math>\text{for (enums) yield body}</math>; where mapName and flatMapName are chosen
 * corresponding to whether this is a for-do or a for-yield.
 * The creation performs the following rewrite rules:
 *
 * 1.
 *   for (P <- G) E ==> G.foreach (P => E)
 *
 *   Here and in the following (P => E) is interpreted as the function (P => E)
 *   if P is a variable pattern and as the partial function { case P => E } otherwise.
 *
 * 2.
 *   for (P <- G) yield E ==> G.map (P => E)
 *
 * 3.
 *   for (P_1 <- G_1; val P_2 <- G_2; ...) ...
 *   ==>
 *   G_1.flatMap (P_1 => for (P_2 <- G_2; ...) ...)
 *
 * 4.
 *   for (P <- G; E; ...) ...
 *   =>
 *   for (P <- G.filter (P => E); ...) ...
 *
 * 5. For N < MaxTupleArity:
 *   for (P_1 <- G; val P_2 = E_2; val P_N = E_N; ...)
 *   ==>
 *   for (TupleN(P_1, P_2, ... P_N) <-
 *     for (x_1 @ P_1 <- G) yield {
 *       val x_2 @ P_2 = E_2
 *       ...
 *       val x_N @ P_N = E_N
 *       TupleN(x_1, ..., x_N)
 *     } ...)
 *
 *   If any of the P_i are variable patterns, the corresponding 'x_i @ P_i' is not generated
 *   and the variable constituting P_i is used instead of x_i
 */
```

Chapter 2

Tree shapes after typing

As shown below (from `Global.scala`), what we call “naming and typing” involves three compilation phases (i.e., instances of `SubComponent`).

```
protected def computeInternalPhases() {
  phasesSet += syntaxAnalyzer           // The parser
  phasesSet += analyzer.namerFactory    // note: types are there because otherwise
  phasesSet += analyzer.packageObjects  // consistency check after refchecks would fail.
  phasesSet += analyzer.typerFactory
  phasesSet += superAccessors           // add super accessors
  phasesSet += pickler                  // serialize symbol tables
  phasesSet += refchecks                // perform reference and override checking, translate nested objects
  ...
}
```

By the time they are done, each tree’s `rawtpe` and `symbol` fields have been set (among others). Additionally, some minor massaging of the ASTs has been performed. After the `refchecks` phase, major massaging can rely on the per-tree-node type information and on that collected in `SymbolTable`. By the way, `SymbolTable` makes visible a number of things:

```
abstract class SymbolTable extends Names
  with Symbols
  with Types
  with Scopes
  with Definitions
  with Constants
  with BaseTypeSeqs
  with InfoTransformers
  with StdNames
  with AnnotationInfos
  with AnnotationCheckers
  with Trees
  with Positions
  with DocComments
```

This chapter summarizes the shape of these trees after these phases (as needed for inserting a custom compiler plugin after parsing and before `superAccessors`). To recap from Sec. 1.2.1. Just after parsing, the `rawpos` of each tree has been initialized, unlike its `symbol` (i.e. for those trees having symbols, subtypes of `SymTree`) which is invariably set to `NoSymbol`. The type of each tree, `rawtpe`, is `null`.

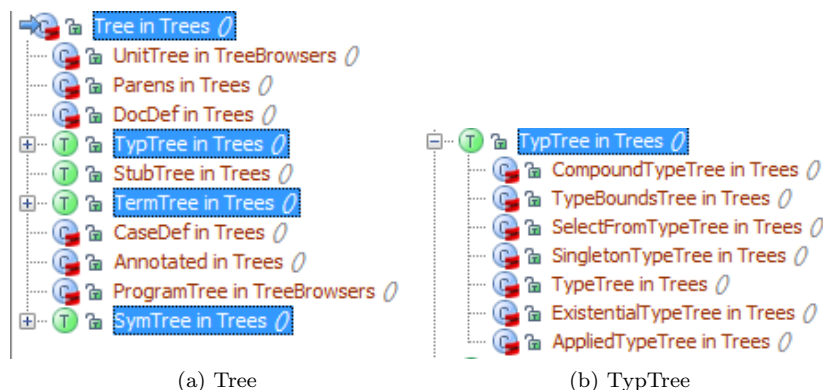


Figure 2.1: Tree hierarchy, 1 of 2

2.1 What is reachable from what

An overview of the `Tree` type hierarchy is shown in Figure 2.1 and Figure 2.2. Those subtypes labelled with “in `TreeBrowsers`” in Figure 2.1a are not to be understood as part of the hierarchy, as they are artefacts used in `TreeBrowsers` (to denote artificial “root nodes” for Swing browsers).

Two observations: (1) not all `Tree` subtypes are used across all phases; and (2) in a few cases (TODO: which ones), a subtype has more than one trait parent. Examples:

```

/** Type annotation, eliminated by explicit outer */
case class Typed(expr: Tree, tpt: Tree)
  extends TermTree
...
/** Dynamic value application.
 * In a dynamic application q.f(as)
 * - q is stored in qual
 * - as is stored in args
 * - f is stored as the node's symbol field.
 */
case class ApplyDynamic(qual: Tree, args: List[Tree])
  extends TermTree with SymTree
  // The symbol of an ApplyDynamic is the function symbol of 'qual',
  // or NoSymbol, if there is none.

```

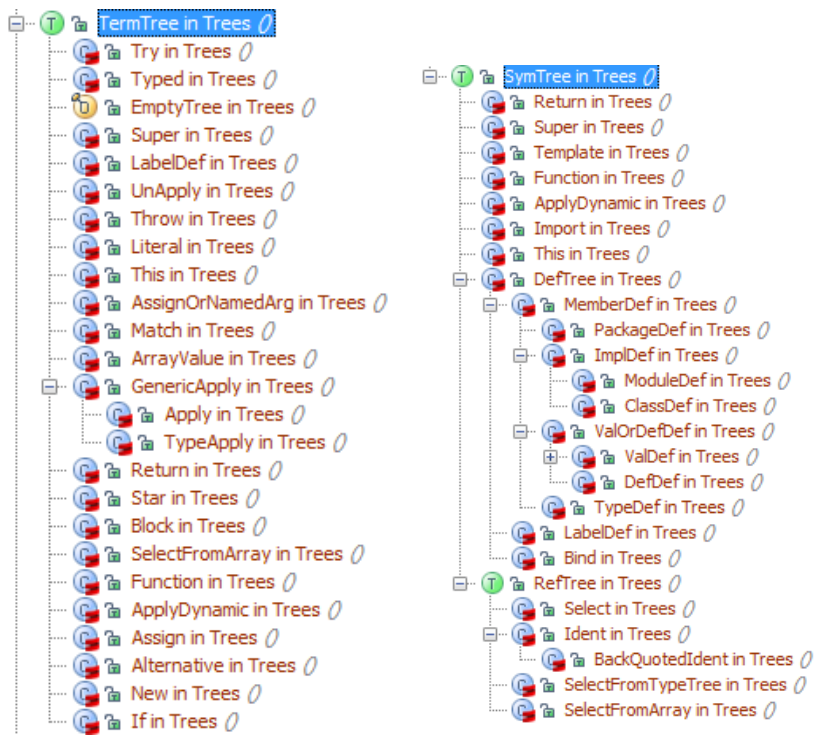
Some insight into the purpose of each node kind can be gleaned from “A standard pattern match” in `Trees.scala`, reproduced in Listing 2.1 and Listing 2.2. The LAMP videos at <http://www.scala-lang.org/node/598> also cover trees, symbols, and types.

2.1.1 Looking up declarations from usages

From <http://comments.gmane.org/gmane.comp.lang.scala.internals/2472>

My approach in NetBeans plugin is: (1) traverse AST to gather all declaration, record (symbol \rightarrow tree); (2) for a given position, tree or symbol usage (reference), query the recorded information.

Detailed explanation: TODO.



(a) TermTree

(b) SymTree

Figure 2.2: Tree hierarchy, 2 of 2

Listing 2.1: A standard Tree pattern match, 1 of 2

```

/* A standard pattern match
case EmptyTree =>
case PackageDef(pid, stats) =>
    // package pid { stats }
case ClassDef(mods, name, tparams, impl) =>
    // mods class name [tparams] impl where impl = extends parents { defs }
case ModuleDef(mods, name, impl) => (eliminated by refcheck)
    // mods object name impl where impl = extends parents { defs }
case ValDef(mods, name, tpt, rhs) =>
    // mods val name: tpt = rhs
    // note missing type information is expressed by tpt = TypeTree()
case DefDef(mods, name, tparams, vparams, tpt, rhs) =>
    // mods def name[tparams](vparams_1)...(vparams_n): tpt = rhs
    // note missing type information is expressed by tpt = TypeTree()
case TypeDef(mods, name, tparams, rhs) => (eliminated by erasure)
    // mods type name[tparams] = rhs
    // mods type name[tparams] >: lo <: hi, where lo, hi are in a TypeBoundsTree,
    // and DEFERRED is set in mods
case LabelDef(name, params, rhs) =>
    // used for tailcalls and like
    // while/do are desugared to label defs as follows:
    // while (cond) body ==> LabelDef($L, List(), if (cond) { body; L$() } else ())
    // do body while (cond) ==> LabelDef($L, List(), body; if (cond) L$() else ())
case Import(expr, selectors) => //(eliminated by typecheck)
    // import expr.{selectors}
    // Selectors are a list of pairs of names (from, to).
    // The last (and maybe only name) may be a nme.WILDCARD
    // for instance
    // import qual.{x, y => z, _} would be represented as
    // Import(qual, List(("x", "x"), ("y", "z"), (WILDCARD, null)))
case DocDef(comment, definition) => //(eliminated by typecheck)
    // /** comment */ definition
case Template(parents, self, body) =>
    // extends parents { self => body }
    // if self is missing it is represented as emptyValDef
case Block(stats, expr) =>
    // { stats; expr }
case CaseDef(pat, guard, body) => //(eliminated by transmatch/explicitouter)
    // case pat if guard => body
case Alternative(trees) => //(eliminated by transmatch/explicitouter)
    // pat1 | ... | patn
case Star(elem) => //(eliminated by transmatch/explicitouter)
    // pat*
case Bind(name, body) => //(eliminated by transmatch/explicitouter)
    // name @ pat
case UnApply(fun: Tree, args)
    // (introduced by typer, eliminated by transmatch/explicitouter)
    // used for unapply's
case ArrayValue(elempt, trees) => //(introduced by uncurry)
    // used to pass arguments to vararg arguments
    // for instance, printf("%s%d", foo, 42) is translated to after uncurry to:
    // Apply(
    //   Ident("printf"),
    //   Literal("%s%d"),
    //   ArrayValue(<Any>, List(Ident("foo"), Literal(42))))

```

Listing 2.2: A standard Tree pattern match, 2 of 2

```

case Function(vparams, body) => (eliminated by lambdaLift)
  // vparams => body where vparams:List[ValDef]
case Assign(lhs, rhs) =>
  // lhs = rhs
case AssignOrNamedArg(lhs, rhs) => (eliminated by typecheck)
  // lhs = rhs
case If(cond, thenp, elsep) =>
  // if (cond) thenp elsep elsep
case Match(selector, cases) =>
  // selector match { cases }
case Return(expr) =>
  // return expr
case Try(block, catches, finalizer) =>
  // try block catch { catches } finally finalizer where catches: List[CaseDef]
case Throw(expr) =>
  // throw expr
case New(tpt) =>
  // new tpt always in the context: (new tpt).<init>[targs](args)
case Typed(expr, tpt) => (eliminated by erasure)
  // expr: tpt
case TypeApply(fun, args) =>
  // fun[args]
case Apply(fun, args) =>
  // fun(args)
  // for instance fun[targs](args)
  // is expressed as Apply(TypeApply(fun, targs), args)
case ApplyDynamic(qual, args)
  // (introduced by erasure, eliminated by cleanup)
  // fun(args)
case Super(qual, mix) =>
  // qual.super[mix]
  // if qual and/or mix is empty, then are nme.EMPTY.toTypeName
case This(qual) =>
  // qual.this
case Select(qualifier, selector) =>
  // qualifier.selector
case Ident(name) =>
  // name
  // note: type checker converts idents that refer to
  // enclosing fields or methods to selects; name ==> this.name
case Literal(value) =>
  // value
case TypeTree() => (introduced by refcheck)
  // a type that's not written out, but given in the tpe attribute
case Annotated(annot, arg) => (eliminated by typer)
  // arg @annot for types, arg: @annot for exprs
case SingletonTypeTree(ref) => (eliminated by uncurry)
  // ref.type
case SelectFromTypeTree(qualifier, selector) => (eliminated by uncurry)
  // qualifier # selector, a path-dependent type p.T is expressed as p.type # T
case CompoundTypeTree(templ: Template) => (eliminated by uncurry)
  // parent1 with ... with parentN { refinement }
case AppliedTypeTree(tpt, args) => (eliminated by uncurry)
  // tpt[args]
case TypeBoundsTree(lo, hi) => (eliminated by uncurry)
  // >: lo <: hi
case ExistentialTypeTree(tpt, whereClauses) => (eliminated by uncurry)
  // tpt forSome { whereClauses }

```

Chapter 3

A code walkthrough: how Scala X-Ray works

We have just scratched the surface of APIs and data structures relevant for AST processing. Before delving deeper, it pays off to do a reality check in terms of a code walkthrough, this time for a third-party tool (not the compiler) that digs out information from ASTs. We're talking about *Scala X-Ray* (SXR in what follows), <http://github.com/harrah/browse>, a documentation tool for Scala (it generates HTML files showing Scala programs in their textual syntax, with some usages linked to their definitions).

As a sidenote, there are at least two other ways to visualize ASTs: (a) on a Swing window, `-Ybrowse:<phase>`; and (b) using `NodePrinters`, i.e. with the compiler option `-Yshow-trees`.

The `scala-plugin.xml` file for SXR states that the start class is `sxr.BrowsePlugin`, which contains the plumbing code to connect the plugin to the compiler, and extends `sxr.Browse`. The interplay between both classes follows a pattern by which plumbing details are encapsulated in `BrowsePlugin`, while `Browse` declares them as abstract, thus avoiding clutter.

Leaving aside HTML-related aspects, the ASTs are visited in `Browse.generateOutput()`. Besides visiting the AST of each compilation unit proper, its underlying source file (`unit.source.file.file`) is lexed (once more) to obtain the following book-keeping information (we won't need this):

```
// from Browse.scala
/** Tokenizes the given source. The tokens are put into an ordered set by the start position of the token.
 * Symbols will be mapped back to these tokens by the offset of the symbol.*/
private def scan(unit: CompilationUnit) = {
    ...
    tokens
}

// from Token.scala
/** Represents a token at the lexer level with associated type information.
 * 'start' is the offset of the token in the original source file.
 * 'length' is the length of the token in the original source file
 * 'code' is the class of the token (see Tokens in the compiler)*/
private case class Token(start: Int, length: Int, code: Int, text: String)
extends NotNull with Ordered[Token] with Comparable[Token]
```

With the list of tokens, the top-level `Tree` of each compilation unit (`unit.body`) is traversed as shown (partially) in Listing 3.2. The case distinction is as follows:

1. `case ValDef(_, _, _, rhs) => ...`
2. `case Template(parents, self, body) => ...`
3. all other cases: `process(tree); super.traverse(tree)`

In `Browse.Traverse.process(t: Tree)` (not shown), for different kinds of tree nodes `t`, a distinction is made to either

- invoke “default-wise” `processSymbol(t, token, source.file.file)` (definition not shown)
- or to invoke instead `processSimple()` whose definition reads

```
def processSimple() { token.tpe = TypeAttribute(typeString(t.tpe), None) }  
...  
/** Holds type information. ... */  
private case class TypeAttribute(name: String, definition: Option[Link])  
extends NotNull
```

Usage examples in `process()`:

```
case _: Match => processSimple() // this will annotate the 'match' keyword  
// with the type returned by the associated pattern match  
  
case _: CaseDef => processSimple() // this will annotate the 'case' keyword  
// with the type returned by that particular case statement  
  
case _: Throw => processSimple()  
  
case ta: TypeApply => processSimple() // this fills in type parameters for methods
```

In a nutshell, `processSimple()` assigns a textual representation of a type (given by `t.tpe`) to the token for `t`. There is more to types than the code in Listing 3.1 might suggest, but it serves as introduction. (I guess more complex type and AST exploration is done by `Scaladoc2`, which certainly would deserve its own chapter too :-). Other ways to “print types on the console” include `-Xprint-types` and `NodePrinters`.

Listing 3.1: SXR way of stringifying types

```
/** Produces a string for the given type that should be informative, but brief.*/
private def typeString(t: Type): String = {
  t match {
    case ct: CompoundType => compoundTypeString(ct, "")
    // tries to reduce size of some type strings
    case pt: PolyType =>
      import pt._
      if(typeParams.isEmpty) "> " + typeString(resultType)
      else {
        val typeParameters = typeParams.map(_.defString).mkString("[", ", ", "]")
        resultType match
          { case ct: CompoundType => compoundTypeString(ct, typeParameters)
            case _ => typeParameters + typeString(resultType)
          }
      }
    case _ => if(t == null) "" else t.toString
  }
}

/** Converts the given compound type to a string.
 * 'mainPostfix' is copied after the main type symbol
 * but before any parents or refinements*/
private def compoundTypeString(ct: CompoundType, mainPostfix: String) = {
  import ct._
  typeSymbol.toString + mainPostfix +
  { if(ct.typeSymbol.isPackageClass) ""
    else if(parents.isEmpty)
      {
        if(decls.isEmpty) ""
        else decls.mkString("{", "; ", "}")
      }
    else parents.mkString(" extends ", " with ", "")
  }
}
}
```

Listing 3.2: SXR entry method for traversing a tree

```
private class Traverse(tokens: wrap.SortedSetWrapper[Token], source: SourceFile) extends Traverser
{
  // magic method #1
  override def traverse(tree: Tree)
  {
    def handleDefault()
    {
      process(tree)
      super.traverse(tree)
    }
    tree match
    {
      case ValDef(_, _, _, rhs) =>
        // tests for synthetic val created for the x in x :: Nil,
        // which would associate the wrong type with ::
        // because the synthetic val is associated with the :: token
        if(tree.symbol != null && tree.symbol.hasFlag(Flags.SYNTHETIC))
          traverse(rhs)
        else
          handleDefault()
      case Template(parents, self, body) =>
        // If the first parent in the source is a trait,
        // the first parent in parents will be AnyRef and it will
        // use the trait's token, bumping the trait.
        // So, this hack processes traits first
        val (traits, notTraits) = parents.partition(_.symbol.isTrait)
        traverseTrees(traits)
        traverseTrees(notTraits)
        if (!self.isEmpty) traverse(self)
        traverseStats(body, tree.symbol)
      case _ =>
        handleDefault()
    }
  }
}
```

Chapter 4

Comparing ASTs across phases

This chapter gives a very, very initial treatment of the process by which an AST is progressively decorated (and mutated). The coverage is example-based. It's a start, you know.

The program versions in what follows were obtained by compiling with option `-Yshow-trees` and choosing different phases (`superaccessors`, `liftcode`, ..., `flatten`). For example, `-Xprint:explicitouter -Yshow-trees`.

The program of interest is simply the following. It started its life as a quick way to see what LambdaLift does.

```
object LLTest {
  def f(x: Int) = {
    class C {
      def cAge = 99 + java.lang.System.currentTimeMillis() + 1;
      class D {
        val y = x + cAge
      }
    }
  }
}
```

An example of the differences (limited to a few AST subtrees across `superaccessors-liftcode-uncurry`) can be seen in Figure 4.1.

Of course, it would be great to see at this point (with AST-level examples) comments on the transformations performed by each phase (I originally planned to do that). Feel free to contribute a detailed account of that ...

