UNIVERSITY OF APPLIED SCIENCES RAPPERSWIL

Spring Semester 2010

Scala Abstract Syntax Trees

scala-refactoring.org

AUTHOR
Mirko Stocker

SUPERVISOR
Prof. Peter Sommerlad

This document is part of my Scala Refactoring thesis' documentation. A nightly build of the documentation can be found on its website at http://scala.ifs.hsr.ch.

For feedback, suggestions, and corrections, please drop me an email to me@misto.ch

Scala AST

This chapter describes the abstract syntax tree classes that are used in the Scala compiler; the implementations can be found in the scala.reflect.generic.Trees trait (not to be confused with scala.reflect.Tree, which provides an undocumented representation of the code at runtime). Note that the list is not complete – some tree classes are only used during parsing and have already been eliminated at the point tools typically see the code, which is after the typer phase.

We start with the root class Tree, some of the more interesting traits and abstract classes and then describe the concrete trees. Scala constructs – syntactic sugar – that are not represented as trees like parallel assignment and for-comprehensions are described in the last section.

Some remarks on the presentation: on the right of the tree class' name are its ancestor classes and traits. All concrete trees are case classes, so their parameters are listed below the class name. In two-column listings, the one on the left is the original source code and the one on the right is the (cleaned up) result of the Tree's toString method.

Base Classes and Traits

Figure 0.1 on the following page shows a inheritance diagram of the various tree classes in the compiler.

Tree

The Tree class is the root of all other trees in the AST. It provides some common functionality for all others, for example the position (the *pos*: Position attribute), the type (*tpe*: Type), and the symbol (*symbol*: Symbol). Not all subtrees have symbols or types, so these attributes might return null.

More operations of the Tree class are defined in TreeOps, for example to filter trees or find elements in subtrees.

SymTree Tree

The SymTree trait is extended by all trees that can have a symbol, but it returns NoSymbol by default.

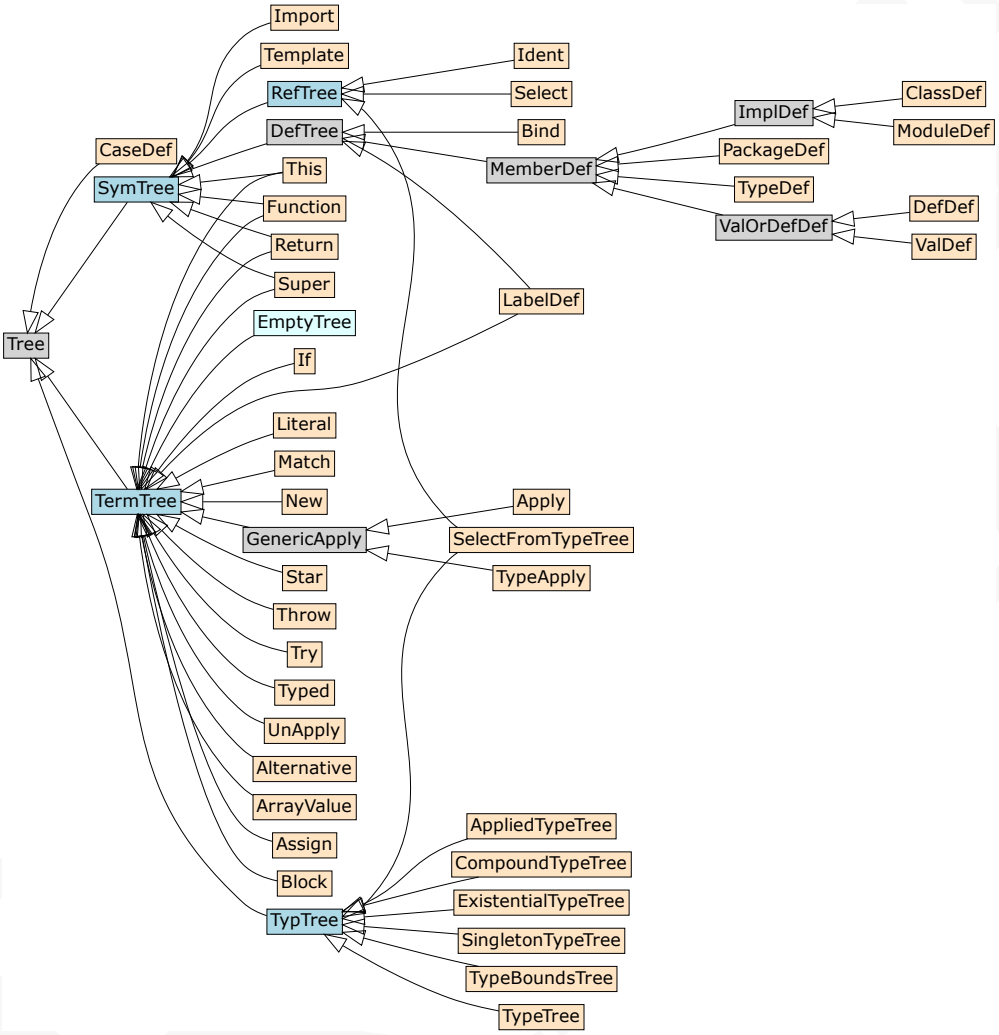

Figure 0.1: The Tree class with some if its subclasses (some have been omitted for the sake of readability). The gray colored classes are abstract, blue ones are traits and the bisque colored leafs of the tree are case classes that can be used in pattern matching. The light cyan EmptyTree is an object (from [**?**]).

DefTree SymTree <: Tree

The DefTree class is extended by all trees that define or introduce a new entity into the program. Each DefTree also has a name and introduces a symbol.

RefTree SymTree <: Tree

RefTrees represent references to DefTrees. They also have a name and their symbol is the same as their corresponding DefTree's – i.e. it can be compared using ==.

Symbol Symbols provide another view on the program. Symbols are introduced by DefTrees and referenced by RefTrees. Symbols provide a lot more information about the program than the Trees – there are several dozen isXy methods defined on symbol to query information. In contrast to Trees, Symbols are much more connected to each other: this allows us to resolve class hierarchies or find the enclosing method or class for a symbol. Martin Odersky recorded three videos [**?**] that walk through the compiler, explaining many Symbol-related concepts.

Position The pos attribute of the trees is very important for us during refactoring. Unless the compiler runs in IDE mode, it generates OffsetPositions, which have only a single offset – the point – that indicates where a tree comes from; this is sufficient for non-interactive usage. When running in interactive – IDE – mode, the compiler generates RangePositions to represent the source range a tree originally comes from and OffsetPositions for those trees that were compiler-generated. Ranges have a start-, a point- and an end offset, where the end offset points to the position after the last character.

Range positions satisfy the following invariants, as specified in the Position.scala source's documentation:

1. A tree with an offset position never contains a child with a range position.
2. If the child of a tree with a range position also has a range position, then the child's range is contained in the parent's range.
3. Opaque range positions of children of the same node are non-overlapping (this means their overlap is at most a single point).

Due to the first invariant, compiler generated trees with an OffsetPosition cannot have children with original source code. Sometimes, it is still necessary for the compiler to generate trees that have children with RangePositions; in these cases, a Transparent position instead of an OffsetPosition is used.

There's also a NoPosition object that is assign to trees that have no origin in a source file. In the refactorings, we use this to indicate newly generated trees.

Concrete Trees

EmptyTree TermTree <: Tree

A Tree object that can stand in for most other trees, it has no position, no type and no symbol. For ValDefs, the equivalent is the emptyValDef object.

PackageDef MemberDef <: DefTree <: SymTree <: Tree

pid: RefTree, *stats*: List[Tree]

Describes a package clause with a package identifier and a list of statements. The package identifier is either an instance of Ident for a package like package a or an instance of Select for a package name like package a.b. The compilation unit root is always a package, even if there is no explicit package declaration present. In this case, the identifier is simply <empty>.

According to the Scala Language Specification, the two different notations are equal:

```
package a
package b.c
```

```
package a {
  package b.c {
  }
}
```

If there exists a top level package definition, its position does not necessarily enclose the whole source file, everything that lies before the package keyword or after the last statement in the package is not contained in the position. In a package that contains no explicit package declaration and only one statement, the package definition has the same start and end position as the statement, but a different point, which makes them distinguishable.

ClassDef ImplDef <: MemberDef <: DefTree <: SymTree <: Tree

mods: Modifiers, *name*: Name, *tparams*: List[TypeDef], *impl*: Template

The definition for all kinds of classes and traits (objects are defined in ModuleDef). The definition contains all modifiers, the name and the type parameters. The class' constructor arguments, super classes and its body are all defined in the impl Template.

Modifiers are a set of abstract, final, sealed, private, protected, trait, case. Note that the class keyword is not contained in the modifiers. If the class is anonymous (this can be queried with isAnonymousClass on the class' symbol), the name is of the form $anon.

ModuleDef ImplDef <: MemberDef <: DefTree <: SymTree <: Tree

mods: Modifiers, *name*: Name, *impl*: Template

The definition of a singleton object, similar to the ClassDef except that a module does not take type parameters.

Template SymTree <: Tree

parents: List[Tree], *self*: ValDef, *body*: List[Tree]

The implementation of either a ModuleDef or ClassDef; also contains early definitions, super types, the self type annotation, and the statements in the class body. In the case of a ClassDef, it also contains the class' constructor parameters.

The following example illustrates into what constructor parameters and super constructor calls are desugared:

```
class B(i: Int) extends A(i)
```

```
class B extends A with ScalaObject {
  <paramaccessor>
    private[this] val i: Int = _
  def this(i: Int): B = {
    B.super.this(i)
  }
}
```

To identify the parameters from the list of body statements, we can check the modifiers of all ValDefs for the PARAMACCESSOR and CASEACCESSOR flags. In the same way, values and types from the early definition are identified by their PRESUPER flag. To check whether a value or type belongs to the early definitions, the compiler's treeInfo.isEarlyDef method can be used.

The super call parameters can be identified as follows: find the constructor DefDef (symbol.isConstructor is true) and then check its body Block for the following pattern: Apply(Select(Super(_, _), _), args). Because only super classes and not traits can have constructor arguments, there can be at most one such super call.

If the self type is not specified, it is the emptyValDef object. Otherwise, there are several different kinds of self type annotations:

```
trait Trait {
}
trait ATrait {
  self =>
}
trait BTrait {
  self: ATrait =>
}
trait CTrait {
  self: BTrait with ATrait =>
}
```

```
abstract trait Trait extends scala.AnyRef {
}
abstract trait ATrait extends scala.AnyRef {
  self: ATrait =>
}
abstract trait BTrait extends scala.AnyRef {
  self: BTrait with ATrait =>
}
abstract trait CTrait extends scala.AnyRef {
  self: CTrait with BTrait with ATrait =>
}
```

We see that a self type annotation automatically intersects the current trait type with all explicitly named types. Extracting the exact positions of all type names is not trivial and involves searching the value's position for the occurrences of the names.

It is also allowed to use this for the self type's name. This introduces no alias and the name of the ValDef is just _.

ValDef ValOrDefDef <: MemberDef <: DefTree <: SymTree <: Tree

mods: Modifiers, *name*: Name, *tpt*: Tree, *rhs*: Tree

Value definitions are all definitions of vals, vars (identified by the MUTABLE flag) and parameters (identified by the param flag).

The modifiers also contain the other properties a value can have: override, abstract, final, implicit, lazy, private, protected. Whether a modifier is applicable depends on the context where a value is used. A value can also be synthetic, i.e. compiler-generated (identified by the SYNTHETIC flag) – for example in the following listing of two equivalent statements, a synthetic value is passed to println:

```
List(1, 2) foreach println
List(1, 2) foreach (println _)
```

Even though the value is compiler generated, it sometimes still has a name. In these examples, it is x, which is the name of println's formal parameter. Sometimes, a name of the form x$1 is used.

Note that not every val in the source code is necessarily also represented by a ValDef. The following listing shows how the abstract value in the trait on the left is actually represented by the compiler:

```
trait A {
  val a: Int
}
```

```
abstract trait A extends scala.AnyRef {
  <stable> <accessor> def a: Int
}
```

In general, values are always private to the class. For external access, stable accessors are generated, as the following listing illustrates.

```
class A {
  val a = 42

}
```

```
class A extends Object with ScalaObject {
  private[this] val a: Int = 42;
  <stable><accessor> def a: Int = A.this.a
}
```

Several methods defined on Symbol can be used to cross-reference between the getters, setters and their underlying value. The accessed method on a getter or setter symbol returns the underlying value's symbol. To get the corresponding setter or getter from a value, the methods getter and setter can be used.

DefDef ValOrDefDef <: MemberDef <: DefTree <: SymTree <: Tree

mods: Modifiers, *name*: Name, *tparams*: List[TypeDef], *vparamss*: List[List[ValDef]], *tpt*: Tree, *rhs*: Tree

The DefDef trees represent method definitions. Methods can have modifiers that further describe the implementation or constrain its visibility. Every method also has a name, but note that symbolic names are stored in their alphabetic form, to get the original name, the symbol's nameString method can be used.

In contrast to a ValDef, a method can be parametrized with types and may have several argument lists. Each argument is represented by a ValDef.

Abstract methods have the DEFERRED flag and an EmptyTree right hand side child.

Finding methods in sub- or super classes requires the use of their symbols. Super classes can be found via the ancestors method on the class' symbol. In contrast, moving down the inheritance hierarchy is more expensive. To find all subclasses of a class C one has to collect all other classes in the universe and test each's ancestors for the presence of C. Once the class hierarchy is assembled, the definition symbol's overriddenSymbol method can be used on each class in the hierarchy to gather all overrides.

TypeDef MemberDef <: DefTree <: SymTree <: Tree

mods: Modifiers, *name*: Name, *tparams*: List[TypeDef], *rhs*: Tree

TypeDef trees are definitions of types. The following listing shows three occurrences – A, B, C – of TypeDefs:

```
class Types {
  type A = Int
  type B >: Nothing <: AnyRef
  def d[C] ...
}
```

Just as the other member definitions trees (ValDef and DefDef), type definitions can have modifiers.

LabelDef DefTree <: SymTree <: Tree ∧ TermTree <: Tree

name: Name, params: List[Ident], rhs: Tree

The LabelDef tree is used to represent while and do ... while loops. The name holds the name of the label.

The Scala language specification [**?**] says

> The while loop expression while (e_1) e_2 is typed and evaluated as if it was an application of whileLoop (e_1) (e_2) where the hypothetical function whileLoop is defined as follows.
>
> ```
> def whileLoop(cond: => Boolean)(body: => Unit): Unit =
> if (cond) { body ; whileLoop(cond)(body) } else {}
> ```

We can also see this when we print the LabelDef:

```
while (true != false) println("loop")
```

```
while$1(){
  if (true != false) {
    println("loop")
    while$1()
  }
}
while$2(){
  if (true)
  {
    {
      println("loop");
      println("loop")
    }
    while$2()
  }
}
```

```
while(true) {
  println("loop")
  println("loop")
}
```

We can see that multiple statements in the body create an additional block that wraps the statements. The do ... while loops are represented in a similar way:

```
do println("loop") while (true)
```

```
doWhile$1(){
  println("loop")
  if (true)
    doWhile$1()
}
```

Thanks to pattern matching, extracting the relevant parts is easy:

```
case LabelDef(_, _, If(cond, body, _)) // while with single expression
case LabelDef(_, _, If(cond, Block((body: Block) :: Nil, _) // while
case LabelDef(_, _, Block(body, If(cond, _, _))) // do While
```

Import

SymTree <: Tree

expr: Tree, *selectors*: List[ImportSelector]

An import statement imports one or many names – the selectors – from a package or object expr. An ImportSelector has two name-position pairs, the first one stands for the imported name and the second one is an optional renaming. Wildcard imports are also represented with an ImportSelector.

Import trees can also be comma separated, in this case, only the first import includes the import keyword in its position.

Block

TermTree <: Tree

stats: List[Tree], *expr*: Tree

A Block encloses a list of statements in { ... } and returns the value of its expr child. Block trees are only generated when needed – for example, the right hand side of a DefDef with a single expression is not a Block but the expression itself, even when the expression is enclosed in { ... }.

The expr is usually the last line of a block, with regards to their positions, but this is not always the case. For example, when creating an anonymous class, the class is introduced with a compiler generated name and then instantiated:

```
val a = new {
}
```

```
val a: java.lang.Object = {
  final class $anon extends scala.AnyRef {
    ...
  }
  new $anon()
}
```

CaseDef

Tree

pat: Tree, *guard*: Tree, *body*: Tree

The body of a Match tree contains a number of CaseDefs trees. The guard can be an empty tree if it is not present. Note that even though the if keyword is used, the tree is not an If tree.

Patterns can be of different form, the catch-all _ is simply an Ident tree, whereas extractors are represented through the UnApply trees. Patterns that use an @ binding or are restricted by type with : are Bind trees. The body can again be an arbitrary tree.

Alternative

TermTree <: Tree

trees: List[Tree]

Alternative trees are used in case definitions to match on alternative clauses, they are separated by |.

Star

TermTree <: Tree

elem: Tree

Patterns can choose to match the whole remainder of an extracted sequence using the _* pattern, as in the following example:

```
"abcde".toList match {
  case Seq(car, cadr, _*) => car
}
```

The wildcard-star is represented by the Star tree.

Bind

DefTree <: SymTree <: Tree

name: Name, *body*: Tree

The Bind tree binds a name to an expression and is used in the patterns of CaseDef. We can see from some examples that several seemingly different syntax variations are all represented in a uniform way in the AST:

```
list match {
  case i => ...
  case i: Int => ...
  case a @ i: Int => ...
}
```

```
list match {
  case (i @ _) => ...
  case (i @ (_: Int)) => ...
  case (a @ (i @ (_: Int))) => ...
}
```

UnApply

TermTree <: Tree

fun: Tree, *args*: List[Tree]

When an extractor object is used in the pattern of a case definition, an UnApply tree is used. The arguments of UnApply can then be more patterns.

```
case Ex(i) => i

case a @ Ex(i) => i

case a @ Ex(i: Int) => i
```

```
case Ex.unapply(<unapply-selector>)
  <unapply> ((i @ _)) => i
case (a @ Ex.unapply(<unapply-selector>)
  <unapply> ((i @ _))) => i
case (a @ Ex.unapply(<unapply-selector>)
  <unapply> ((i @ (_: Int)))) => i
```

Function

TermTree <: Tree ∧ SymTree <: Tree

vparams: List[ValDef], body: Tree

The Function tree contains a single list of parameters and a body for the implementation. The following listing shows various usages of the Function tree and how their trees look like.

```
list foreach println

list foreach (println _)

list foreach (i => println(i))
list foreach ((i: Int) => println(i))
list foreach {
  case i => println(i)
}
```

```
list foreach ({
  ((x: Any) => println(x))
})
list foreach ({
  ((x: Any) => println(x))
})
list foreach (((i: Int) => println(i)))
list foreach (((i: Int) => println(i)))
list foreach (((x0$1: Int) => x0$1 match {
  case (i @ _) => println(i)
}))
```

In the first two examples, the functions are encapsulated in an additional Block

– hence the curly braces. When the function parameter does not have a name, the compiler generates one and marks it with the SYNTHETIC flag. In the last example, we see that the pattern matching on the parameter is made explicit in the AST.

Assign TermTree <: Tree

lhs: Tree, *rhs*: Tree

Assign trees are not used for all = calls, only for non-initial assignments to variables. Calls to setter methods are not represented by Assign trees but are regular method calls.

If TermTree <: Tree

cond: Tree, *thenp*: Tree, *elsep*: Tree

An If expression consists of three parts: the condition, the then part and the else part. If the else part is omitted, the literal () of type Unit is generated and the type of the conditional is set to an upper bound of Unit and the type of the then expression, usually Any.

else if terms are implemented using nested if conditionals. We can see this in the following listing.

```
if (a)
  b
else if (c)
  d
else
  e
```

```
if (a)
  b
else
  if (c)
    d
  else
    e
```

Note that the if used in pattern matching guards is not an If tree but a designated member of the CaseDef tree.

Match TermTree <: Tree

selector: Tree, *cases*: List[CaseDef]

A match tree is used to represent a pattern match, with the selector being the tree that is matched against. When a pattern matching expression is used as the body of a function, the selector is a synthetic value:

```
list foreach {
  case i => println(i)
}
```

```
list foreach (((x0$1: Int) => x0$1 match {
  case (i @ _) => println(i)
}))
```

Return TermTree <: Tree ∧ SymTree <: Tree

expr: Tree

The Return tree contains an expression that constitutes the return value. For return statements without an expression, the compiler generates a () literal.

Try TermTree <: Tree

block: Tree, *catches*: List[CaseDef], *finalizer*: Tree

The Try tree represents try ... catch expressions. Both the catches and the finalizer are optional.

Throw TermTree <: Tree

expr: Tree

The Throw tree stands for the throw keyword and its expression.

New TermTree <: Tree

tpt: Tree

The New tree represents new statements, the tpt member is the type that is being instantiated.

Typed TermTree <: Tree

expr: Tree, *tpt*: Tree

The Typed tree is used whenever an expression is annotated with a type. For example, in the following listing, the second and third occurrences of Int are Typed trees:

```
val a: Int = 42: Int
println(a: Int)
```

Typed trees are also used in pattern matching when the match checks the type of the underlying object.

TypeApply

GenericApply <: TermTree <: Tree

fun: Tree, *args*: List[Tree]

TypeApply trees are used whenever a type is applied to a generic method. For example, in the following listing, both expressions on the left are represented by the same AST on the right.

```
List(1,2,3)

List[Int](1,2,3)
```

```
Apply(
  TypeApply(
    Select(..., "apply"),
    List(Int)),
  List(1,2,3))
```

Apply

GenericApply <: TermTree <: Tree

fun: Tree, *args*: List[Tree]

Function application is represented with Apply trees. The fun is often a Select tree that specifies the function name and args are the actual parameters.

Super

TermTree <: Tree ∧ SymTree <: Tree

qual: Name, *mix*: Name

The Super tree represents a super call, with optional qualifier and super class specifier:

```
trait A {
  def x = 42
}
trait B extends A {
  override def x = 43
}
class C extends A with B {
  println(super[A].x)
}
```

This

TermTree <: Tree ∧ SymTree <: Tree

qual: Name

The This tree represents the this reference, with an optional qualifier:

```
class Outer {
  class Inner {
    val outer = Outer.this
  }
}
```

Select RefTree <: Symtree <: Tree

qualifier: Tree, *name*: Name

The Select tree occurs on places that select a name from a qualifier, e.g. in method calls. Note that the typer fully qualifies references as illustrated in the following listing.

```
class A {
  val a = ...
  val b = a
}
```

```
class A {
  val a = ...
  val b = A.this.a
}
```

As usual, these generated trees then have an OffsetPosition.

Ident RefTree <: Symtree <: Tree

name: Name

Holds a Name, which can be generated (check with symbol.isSynthetic) by the compiler. Note that the name is in its alphabetic form; the real name can be found via the tree's symbol.

Literal TermTree <: Tree

value: Constant

All literals are represented by Literal trees. All possible kinds of constants are listed in the Constant trait.

TypeTree AbsTypeTree <: TypTree <: Tree

original: Tree

From the Scala compiler's documentation [**?**]:

> A synthetic term holding an arbitrary type. Not to be confused with with TypTree, the trait for trees that are only used for type trees. TypeTrees are inserted in several places, but most notably in RefCheck, where the arbitrary type trees are all replaced by TypeTrees.

The original type tree is still accessible via the TypeTree's original member. Note that the standard tree Traverser and Transformer visitors do not traverse into the original subtree.

SingletonTypeTree TypTree <: Tree

ref: Tree

Whenever the .type expression is used, the tree is represented by a SingletonTypeTree tree.

SelectFromTypeTree TypTree <: Tree ∧ RefTree <: SymTree <: Tree

qualifier: Tree, *name*: Name

Type selection of the form qualifier#name is represented with the SelectFromTypeTree tree.

CompoundTypeTree TypTree <: Tree

templ: Template

An intersection type is represented by a CompoundTypeTree. Note that the tree contains a Template, this is because the compound type can have an optional refinement:

```
trait A
trait B
... A with B {
  ...
}
```

AppliedTypeTree TypTree <: Tree

tpt: Tree, *args*: List[Tree]

When a type is applied to a polymorphic function, a TypeApply tree is used. When a type is applied to an other type, an AppliedTypeTree is used.

TypeBoundsTree TypTree <: Tree

lo: Tree, *hi*: Tree

Whenever a type is constrained to lower or upper bounds, TypeBoundsTree represents these bounds. If one of the bounds is omitted, the compiler inserts Nothing respectively Any for the missing lower or upper bound. This is illustrated in the following example:

```
type B >: Nothing <: AnyRef
type C >: String
type D <: AnyRef
```

```
type B >: Nothing <: AnyRef
type C >: String <: Any
type D >: Nothing <: AnyRef
```

ExistentialTypeTree TypTree <: Tree

tpt: Tree, *whereClauses*: List[Tree]

Existential types are represented with a ExistentialTypeTree. In Scala, there exist two notations for existentials:

```
List[_]
List[T] forSome { type T }
```

Both are represented the same way in the AST, with their full notation:

```
List[_$1] forSome {
  <synthetic> type _$1 >: _root_.scala.Nothing <: _root_.scala.Any
}
List[T] forSome {
  type T >: _root_.scala.Nothing <: _root_.scala.Any
}
```

Note that the existential type tree is stored in another TypeTree's orig member, which is not traversed and transformed by the Scala compiler's Traverser and Transformer classes.

Other AST Constructs

For Comprehensions

For or sequence comprehensions are a purely syntactic construct; in the AST, they are represented with foreach, withFilter, map, and flatMap calls. The following listing shows some examples.

```
val xs = 1 to 10 toList
val ys = 1 to 10 toList

val coordinates1: List[(Int, Int)] = for(x ← xs; y ← ys) yield (x → y)
// is equal to
val coordinates2: List[(Int, Int)] = xs.flatMap(x ⇒ ys.map(y ⇒ (x → y)))

for(x ← xs if x % 2 == 0) println(x)
// is equal to
xs.withFilter(x ⇒ x % 2 == 0).foreach(x ⇒ println(x))
```

Multiple Assignment

Scala's multiple or parallel assignment syntax is just an abbreviation for a more complex pattern match expression. The following listing shows the desugared form for the call val (a, b) = getPair():

```
def getPair() = (1, 2)

val x$1 = getPair() match {
  case (a, b) ⇒ (a, b)
}

val a = x$1._1
val b = x$1._2
```

The same transformation is also performed when extractors are involved in the assignment:

```
val MyRegex = """(\w)(.*)""".r
val MyRegex(firstGroup, secondGroup) = "Hello"
```

becomes:

```
val MyRegex = """(\w)(.*)""".r
val x$1 = "Hello" match {
  case MyRegex(firstGroup, secondGroup) ⇒ (firstGroup, secondGroup)
}

val firstGroup = x$1._1
val secondGroup = x$1._2
```

Named Arguments

Named arguments are desugared into a series of local values that are then passed in the right order to the method. That is, the following code:

```
def p(first: String, second: Int) = ()

p(second = 42, first = "-")
```

is represented as:

```
def p(first: String, second: Int): Unit = ()
{
  val x$1 = 42
  val x$2 = "-"
  Account.this.p(x$2, x$1)
}
```

This is described in the first Scala Improvement Document [**?**].