TASTY Reference Manual

Version 0.06
9 April 2016
Martin Odersky
Dmitry Petrashko
Eugene Burmako

1 Preface

This reference manual describes the TASTY serialization format for typed syntax trees representing Scala programs. A motivation and a short summary of the format is found in the companion document A TASTY Alternative.

Notation:

We use BNF notation. Terminal symbols start with at least two consecutive upper case letters. Each terminal is represented as a single byte tag. Non-terminals are mixed case. Prefixes of the form ***lower case letter*_*** are for explanation of semantic content only, they can be dropped without changing the grammar.

2 Overall Layout and Vocabulary

```
File          = Header majorVersion_Nat minorVersion_Nat UUID
                nameTable_Length Name* Section*
```

A Tasty *file* consists of a header, a name table and a number of sections.

```
Section       = NameRef Length Bytes
Length        = Nat                   // length of rest of entry in bytes
```

Each section consists of:

- A reference to the name of the section.
- A length field indicating the number of bytes that follow.
- The data making up the section contents.

Numbers use a variable length encoding, defined as follows:

```
LongInt       = Digit* StopDigit      // big endian 2's complement, fits in a Long w/o overflow
Int           = LongInt               // big endian 2's complement, fits in an Int w/o overflow
Nat           = LongInt               // non-negative value, fits in an Int without overflow
Digit         = 0 | ... | 127
```

```
StopDigit     = 128 | ... | 255          // value = digit - 128
```

That is, numbers are represented using base 128 digits with a stop bit indicating the end of a number. Numbers can be signed (**Int**) or unsigned (**Nat**).

Some numbers are used for specific purposes. In particular, we distinguish:

```
Length        = Nat      // A number indicating the number of bytes that follow in the current entry.
NameRef       = Nat      // A number serving as an index into the name table.
ASTRef        = Nat      // A number serving as a byte address which identifies the start of a
                         // node in the typed trees section.
```

3 The Header

```
Header         = 5CA1AB1F
UUID           = Byte*16
```

The header consists of three parts:

1. 4 leading bytes with hex codes 5C, A1, AB, 1F.
2. The major and minor version of the file. Files with different major versions are treated as incompatible. Formats with the same major version are required to be backwards compatible. That is, a processor for version x.y is required to also understand Tasty files with version x.z for z < y.
3. The UUID. This is a random UUID which identifies a Tasty file uniquely.

4 The Name Table

The nametable defines all names used in a Tasty file. It implicitly gives an index to each defined name according to its position in the name table. Indices start from 1. Everywhere else, names are represented by their index in the name table.

A name in a name table is encoded in one of several formats, which are described in the next sections.

4.1 Simple Names

```
Name          = UTF8          Length UTF8-CodePoint*
```

A **UTF8** entry encoded a simple name in unicode format. It comprises:

- A length field indicating the number of bytes that follow.
- A sequence of UTF8 codepoints.

Names defined by UTF8 entries are plain strings. Based on the context we can decide whether a name is a type-name or a term-name. The same string can represent both.

4.2 Qualified Names

```
Name            = QUALIFIED     Length qualified_NameRef selector_NameRef
```

A **QUALIFIED** entry represents a qualified name <prefix>.<selector>. It comprises:

- The index of the name <prefix>. This can be another qualified name.
- The index of the name <selector>.

4.3 Expanded Names

```
Name           = EXPANDED       Length prefix_NameRef member_NameRef
```

An **EXPANDED** entry represents an expanded name, which makes a member name unique by prefixing it with an encoding of the member's location. It comprises the indices of the prefix and the member name.

4.4 Signed Names

```
Name            = SIGNED        Length original_NameRef resultSig_NameRef paramSig_NameRef*
```

A **SIGNED** entry represents a name and a type signature. It comprises:

- The *original*_**NameRef** index of the name proper.
- The *resultSig*_**NameRef** index of the result type signature. For a normal type T, the result type signature is the fully qualified name of the class that T erases to. For an array type T[], it is the "S[]", where S is the result type signature of T. For a method type MT, it is the result type signature of MT's final result type, skipping all type and value parameter sections.
- The *paramSig*_**NameRef** indices of the parameter type signatures. For a method type MT, the parameter type signatures are the result type signatures of all its value parameters, where it does not matter whether these parameters appear in curried form or in a single parameter section. For all other types, the list of parameter type signatures is empty.

As an example, a reference to the method **f** defined as

```
def f(x: Int => String, y: =>Long, z: Byte*): Unit
```

would have an original name **f**, a result signature **scala.Unit,** and parameter signatures **scala.Function1**, **scala.Function0**, and **scala.collection.immutable.Seq**.

4.5 Shadowed Names

```
Name            = SHADOWED      Length original_NameRef
```

A **SHADOWED** entry represents the shadowed name of a class member which is shadowed by a **private** member with the same name in a subclass. It comprises the original name of the class member.

Here is an example motivating this category of names. Say we have:

```
class C { def x: Int }
class D extends C { val d: D; private def x: Int }
```

Then the reference **d.x** depends on the position of the observer. If the observer is inside **D**, it refers to the private member **x** of **D**. If the observer is outside **D**, it refers to the public member of **C**. To obtain absolute references that do not depend on the position of the observer, we use the reference **d."SHADOWED x"** to refer to the public member instead of the private one.

4.6 Other Forms of Names

Further names will be defined in the future. In particular, we plan to add:

```
Name          = OBJECTCLASS   Length object_NameRef
Name          = SUPERACCESSOR Length accessed_NameRef
Name          = DEFAULTGETTER Length method_NameRef paramNumber_Nat
Name          = MANGLED       Length mangle_NameRef name_NameRef
```

5 Standard-Section: Typed Trees

The typed trees section starts with a name reference to the string "ASTs", and contains a list of top-level statements, represented by **TopLevelStat** productions.

```
TopLevelStat  = PACKAGE        Length Path TopLevelStat*
                Stat
```

A top-level statement is either a normal statement or a package definition. A package definition comprises

- A path representing the fully qualified name of the package as a **TERMREF** node.
- The top-level statements making up the contents of that package in the current file (this means that packages can be nested).

5.1 Statements

Statements are represented by **Stat** productions.

5.1.1 Expression Statements

```
Stat          = Term
```

A statement can be an expression.

5.1.2 Value Definitions

```
Stat          = VALDEF         Length NameRef Type rhs_Term? Modifier*
```

A **VALDEF** entry serializes a value definition **<mods> val x: T = E** or a variable definition **<mods> var x: T = E**. It comprises

- The name **x** of the defined value.
- The type **T** of the defined value.
- The right hand side **E**, which is omitted for abstract value definitions **val x: T**.
- The modifiers **<mods>** given for the definition. The modifier **MUTABLE** indicates a var.

5.1.3 Method Definitions

```
Stat          = DEFDEF         Length NameRef TypeParam* Params* return_Type rhs_Term?
                                      Modifier*
Params        = PARAMS         Length Param*
Param         = PARAM          Length NameRef Type rhs_Term? Modifier*
```

A **DEFDEF** entry serializes a method definition **<mods> def x[...](...)...(...) : T = E**. It comprises:

- The name of the defined method x.
- The type parameters of the method.
- The value parameter sections of the method.
- The return type T.
- The right hand side E. Omitted for abstract method definitions def x ...: T.
- The modifiers <mods>.

5.1.4 Type Parameters

```
TypeParam     = TYPEPARAM      Length NameRef Type Modifier*
```

A **TYPEPARAM** entry serializes a type parameter **<mods> T <bounds>**. It comprises

- The name **T** of the type parameter.
- The bounds **<bounds>** of the type parameter, given as a **TYPEBOUNDS** type.
- The modifiers **<mods>.**

5.1.5 Parameter Sections

```
Params        = PARAMS         Length Param*
Param         = PARAM          Length NameRef Type rhs_Term? Modifier*
```

A `PARAMS` entry serializes a parameter section `(P1, … Pn)` consisting of a list of parameters. Each parameter `<mods> x: T` is given in a `PARAM` entry. It comprises:

- The name `x` of the parameter.
- The type `T` of the parameter.
- The modifiers `<mods>` given for the parameter.
- Possibly a right hand side. The right hand side is given only for a value parameter in a template (see below). If it is present, it indicates that the parameter is an alias of another parameter in a superclass that is referenced by the right-hand side. Here is an example:

  ```
  class A(val x: Int)
  class B(y: Int) extends A(y)
  ```

  In this case, the parameter `y` in class `B` is known to be an alias of parameter `x` in class `A`. It can be represented internally as

  ```
  def y: Int = super.x
  ```

  The serialized information of this parameter is as a `Param` with a right hand side of `super.x`.

5.1.6 Type Definitions

```
Stat           = TYPEDEF        Length NameRef (Type | Template) Modifier*
```

A `TYPEDEF` entry serializes a type definition `<mods> type T <rhs>` or a class or trait definition `<mods> class T <template>` or `<mods> trait T <template>`. It comprises

- The name `T` of the defined type.
- If it is a type definition, its right hand-side given as a `TYPEBOUNDS` type for an abstract type or `TYPEALIAS` type for an alias type. Otherwise, for a class or trait, its template.
- The modifiers `<mods>` given for the type.

5.1.7 Templates

```
Template       = TEMPLATE       Length TypeParam* Param* Parent* Self? Stat*
Parent         = Application
                 Type
Self           = SELFDEF               selfName_NameRef selfType_Type
```

A template represents the body of a class, trait, or object. It comprises:

- A list of type parameters. For objects, this is always the empty list.
- A list of value parameters. These parameters are not grouped into sections but appear as a single list. For objects, that list is always empty. Value parameters carry the modifiers given in the class definition. They can be aliases of superclass parameters as explained previously.

- A list of parents of the template. Parents can be terms (indicating a constructor call) or types (indicating a parent type without an associated passing of arguments). Term parents are always of the form **`new T.<init>[targs](args)`**, that is, they are selections of a constructor from an instance creation node NEW, applied to zero or more type- and value arguments.
- An optional **`SELFDEF`** entry representing a self reference, indicating the name and type of the self reference of the template.
- A sequence of definitions making up the body of the template. The first definition in the sequence is always the primary constructor of the class.

The *primary constructor* of a class **`C`** is a **`DEFDEF`** entry representing a definition of the form

```
def <init>[tparams](params1)...(paramsN): C
```

Here:

- The name of the constructor is always **`<init>.`**
- The type parameters of the constructor are those of the class.
- The value parameters of the constructor are those of the class.
- The result type is the fully parameterized type of the class.

A primary constructor has no right hand side.

5.1.8 Import Clauses

```
Stat          = IMPORT        Length qual_Term Selector*
Selector      = IMPORTED             name_NameRef
                RENAMED       Length from_NameRef to_NameRef
```

An **`IMPORT`** entry serializes an import clause **`import E <selectors>`**. It comprises

- A qualifier term **`E`** from which is imported.
- A list of selectors. A selector is either *simple* or *renaming*.

Import entries are redundant for the interpretation and code generation of Tasty trees because all import references have been resolved before. They are retained for the benefit of source-level frameworks such as scala.meta.

A **`IMPORTED`** entry serializes a simple selector consisting of the name of the imported entity. Wildcard imports are represented by the name “**`_`**”.

A **`RENAMED`** entry serializes a renaming selector **`x => y`**. Again, wildcards are represented by “**`_`**”.

5.2 Constants

A **`Constant`** refers to a literal value. It is of one of the following forms.

```
Constant      = UNITconst
```

```
                    FALSEconst
                    TRUEconst
                    BYTEconst           Int
                    SHORTconst          Int
                    CHARconst           Nat
                    INTconst            Int
                    LONGconst           LongInt
                    FLOATconst          Int
                    DOUBLEconst         LongInt
                    STRINGconst         NameRef
                    NULLconst
                    CLASSconst          Type
                    ENUMconst           Path
```

Notes:

- **UNITconst** represents the value ()
- The value represented by a **FLOATconst** is stored as a signed Int. The floating point value can be recovered from it by a **java.lang.Float.intBitsToFloat** conversion.
- The value represented by a **DOUBLEconst** is stored as a signed long Int. The floating point value can be recovered from it by a **java.lang.Double.longBitsToFloat** conversion.
- The value represented by a **CLASSconst** is a class literal which is referenced by a **TYPEREF** type.
- The value represented by an **ENUMconst** is an enumeration value referenced by a **TERMREF** type.

5.3 Paths

A **Path** refers to a single value; it can be used as a term and as a type.

5.3.1 Constant Paths

```
 Path          = Constant
```

A path can refer to a constant value.

5.3.2 Term references

```
  Path          = TERMREFdirect        sym_ASTRef
                  TERMREFsymbol        sym_ASTRef qual_Type
                  TERMREFpkg           fullyQualified_NameRef
                  TERMREF              possiblySigned_NameRef qual_Type
```

There are four kinds of references to terms: By-name (**TERMREF)**, symbolic (**TERMREFsymbol)**, direct (**TERMREFdirect)**, and package (**TERMREFpkg)**.

A **TERMREF** reference indicates a member of a qualifier type with a given name. A **SIGNED** name is used to identify one of several possible overloaded alternatives uniquely.

A **TERMREFsymbol** reference indicates a member of a qualifier type with a given symbol. The symbol is represented by an **ASTRef** pointing to the start address of the definition entry that defines the symbol. A **TERMREFsymbol** entry should not be used as a way to refer to definitions in other compilation units because such definitions can change upon recompilation.

A **TERMREFdirect** reference indicates a value represented directly by a local definition which is not a member of any object.

A **TERMREFpkg** reference indicates a package with the given fully qualified name.

5.3.3 this References

```
Path          = THIS                   clsRef_Type
```

A **THIS** entry serializes a reference **C.this**. It comprises a type reference indicating the class **C**.

5.3.4 Skolem Types

```
Path          = SKOLEMtype             refinedType_ASTRef
```

A **SKOLEMtype** entry serializes a this reference inside a refined type. It comprises the reference to the enclosing refined type indicated by the this.

5.3.5 Shared Paths

```
Path          = SHARED                 path_ASTRef
```

A **SHARED** entry serializes a path by referring to a previously generated entry of the same path. **SHARED** can also be used to alias a type or term.

5.4 Types

```
Type          = Path
                 ...
```

A **Type** summarizes statically known information about a term or definition. **Path** is a subclass of **Type.** Other forms of **Type** are listed below.

5.4.1 Type References

```
Type          = TYPEREFdirect        sym_ASTRef
                TYPEREFsymbol        sym_ASTRef qual_Type
                TYPEREFpkg           fullyQualified_NameRef
                TYPEREF              possiblySigned_NameRef qual_Type
```

As is the case for term references, there are four kinds of references to types: By-name (**TYPEREF)**, symbolic (**TYPEREFsymbol)**, direct (**TYPEREFdirect)**, and package (**TYPEREFpkg)**.

A **TYPEREF** reference indicates a type member of a qualifier type with a given name. That name is never a **SIGNED** name.

A **TYPEREFsymbol** reference indicates a member of a qualifier type with a given symbol. The symbol is represented by an **ASTRef** pointing to the start address of the definition entry that defines the symbol. A **TYPEREFsymbol** entry should not be used as a way to refer to definitions in other compilation units because such definitions can change upon recompilation.

A **TYPEREFdirect** reference indicates a value represented directly by a local definition which is not a member of any object.

A **TYPEREFpkg** reference indicates the companion class of a package with the given fully qualified name.

5.4.2 Super Types

```
Type          = SUPERtype      Length this_Type underlying_Type
```

A **SUPERtype** entry serializes a super reference **C.super** or **C.super[M]** that is used as part of a type. It comprises:

- the serialization of **C.this.**,
- the underlying type of the super reference. If a mixin qualifier **M** is given, the underlying type is the trait referred to by **M**. Otherwise the underlying type is the intersection of all parent types of the class **C**.

5.4.3 Refined types

```
Type          = REFINEDtype    Length underlying_Type refinement_NameRef info_Type
```

A **REFINEDtype** entry serializes a refined type **T { val x: U }**, **T {def x…: T }**, **or T { type x >: L <: U }**. It comprises:

- The parent type **T**.
- The name **x** of a member of type **T**
- A refinement type **U** which provides specific type information for x. If **U** is a **TYPEBOUNDS** or **TYPEALIAS** type, **x** is taken as a type name, and **U** provides a bound constraint or alias for **x**. If **U** is some other type, **x** is taken as a term name, and **U** represents the refined type for **x**.

5.4.4 Applied types

```
Type          = APPLIEDtype   Length tycon_Type arg_Type*
```

An **APPLIEDtype** entry serializes an applied type **T[U1, …, Un]**. It comprises

- The type constructor **T.**
- The type arguments **U1**, …, **Un**.

A wildcard argument **_ >: S <: U** is represented with a **TYPEBOUNDS** entry that indicates the bounds **S**, **U**. Applied types can be seen as short forms of refinement types, where the type constructor is the parent type, each concrete type argument **U** leads to a refinement **{ t = U }** of the corresponding type parameter **t** and each wildcard argument **_ >: S <: U** leads to a refinement **{ t >: S <: U }**.

5.4.5 Type Bounds

```
Type          = TYPEBOUNDS    Length low_Type high_Type
```

A **TYPEBOUNDS** entry serializes the bounds **>: S <: U** of a type parameter, type definition, or wildcard argument. It comprises:

- The lower bound **S**. If none is given in the original source expression, **scala.Nothing** is used for the serialization.
- The upper bound **U**. If none is given in the original source expression, **scala.Any** is used for the serialization.

5.4.6 Type Aliases

```
Type          = TYPEALIAS     Length alias_Type (COVARIANT | CONTRAVARIANT)?
```

A **TYPEALIAS** entry serializes an alias **= U**. It comprises:

- the type **U**.
- optionally, a flag **COVARIANT** indicating that the alias binds a covariant parameter, or a flag **CONTRAVARIANT** indicating that the alias binds a contravariant parameter.

5.4.7 Annotated Types

```
Type          = ANNOTATED     Length underlying_Type fullAnnotation_Term
```

An **ANNOTATED** entry serializes an annotated type **T @annot**. It comprises:

- The annotated type **T**.
- An instance creation expression that when executed creates a new instance of the given annotation **@annot**.

5.4.8 And and Or Types

```
Type          = ANDtype       Length left_Type right_Type
                ORtype        Length left_Type right_Type
```

An `ANDtype` entry serializes an intersection type `T & U`. A `ORtype` entry serializes a union type `T | U.` Either entry comprises the operand types `T` and `U`.

5.4.9 Type Bindings

```
Type          = BIND          Length boundName_NameRef bounds_Type
```

When used as a type, a `BIND` entry serializes a type variable `t` which is defined in a type pattern. It comprises

- The name `t` of the type variable.
- The inferred type of `t`. This is always a `TYPEBOUNDS` or `TYPEALIAS` entry.

5.4.10 By-name Types

```
Type          = BYNAMEtype    Length underlying_Type
```

A `BYNAMEtype` entry serializes the `type => T` of a call-by-name parameter. It comprises the argument type `T`.

5.4.11 Method Types

```
Type          = POLYtype      Length result_Type NamesTypes
                METHODtype    Length result_Type NamesTypes
NamesTypes    = ParamType*
NameType      = paramName_NameRef typeOrBounds_ASTRef
```

`METHODtype` and `POLYtype` entries are used as types of refinements. A `METHODtype` represents the type of a method `(P1,...,Pn)R`. It comprises:

- The method result type `R` (which can be another method type).
- The names and types of the parameters `P1,...,Pn,` represented as a list of interleaved names and types.

A `POLYtype` represents the type of a polymorphic method `[TP1,...,TPn]R`. It comprises:

- The result type `R` (which can be a method type).
- The names and bounds of the type parameters `TP1,...,TPn`, represented as a list of interleaved names and `TYPEBOUNDS` entries.

5.4.12 Parameter types

```
Type          = PARAMtype     Length binder_ASTref paramNum_Nat
```

A `PARAMtype` entry refers to a type or value parameter of an enclosing method type. It comprises:

- A reference to the type which binds the referenced parameter. This type is always serialized as a **POLYtype** or a **METHODtype.**
- A natural number indicating the index of the referenced parameter in the list of defined parameters of the binding type. Indices start at 0.

5.4.13 Shared Types

```
Type          = SHARED                 path_ASTRef
```

A **SHARED** entry serializes a type by referring to a previously generated entry representing the same type.

5.5 Terms

```
Term          = Path
                Application
                ...
```

A **Term** represents an expression or pattern. A **Path** is special form of a **Term.** Another subclass of **Term** is **Application.** The possible forms of terms and applications are given in the following subsections.

5.5.1 Identifiers

```
Term          = IDENT                  NameRef Type
```

An **IDENT** node serializes an identifier **x**. It comprises the name **x** and the identifier's type. **IDENT** nodes are usually omitted if an identifier **x** refers to a definition and the type of the identifier is the declared type of the definition. In that case, a [Term reference] can be used instead.

5.5.2 Selections

```
Term              = SELECT               possiblySigned_NameRef qual_Term
```

A **SELECT** node serializes a selection **E.x**. It comprises the selector name **x** and the qualifier term **E**. A **SIGNED** name is used to identify one of several possible overloaded alternatives uniquely.

5.5.3 Applications

```
Application       = APPLY        Length fn_Term arg_Term*
```

An **APPLY** entry serializes an application **F(E1, …, En)**. It comprises:

- A function expression **F**.
- Argument expressions **E1, …, En**.

Functions in TASTY are always fully applied, and the order of arguments matches the order of formal parameters.

5.5.4 Type Applications

```
 Application        = TYPEAPPLY      Length fn_Term arg_Type*
```

A **TYPEAPPLY** entry serializes a type application **F[T1, …, Tn]**. It comprises:

- A function expression **F**.
- Argument types **T1, …, Tn**.

5.5.5 Instance Creation Expressions

```
 Term               = NEW                      cls_Type
```

A **NEW** node serializes an instance creation new **T**. It comprises the type of the node that's created. The node is always part of a constructor invocation; that is it is enclosed in a **SELECT** node, which in turn forms part of an **Application.**

5.5.6 Super References

```
 Term           = SUPER           Length this_Term mixinTrait_Type?
```

A **SUPER** node serializes a super reference **C.super[M]** where **C** may be implied and **[M] m**ay be missing. It comprises

- the serialization of **C.this**, and, optionally,
- a type referring to the class referenced by the mixin qualifier **M**.

5.5.7 Pairs

```
  Term          = PAIR            Length left_Term right_Term
```

A **PAIR** entry serializes an unboxed pair **(E1, E2)**, either as a term or as a pattern. It comprises the two halfs of the pair **E1 a**nd **E2**. (Unboxed pairs are not yet implemented.)

5.5.8 Type Ascriptions

```
  Term          = TYPED           Length expr_Term ascription_Type
```

A TYPED entry serializes a type ascription E: T. It comprises:

- The expression E.
- The ascribed type T.

5.5.9 Named Arguments

```
  Term          = NAMEDARG       Length paramName_NameRef arg_Term
```

A NAMEDARG entry serializes a named function argument x = E. It comprises

- The parameter name x.
- The argument expression E.

Named arguments are redundant for the interpretation and code generation of Tasty trees because all arguments are always given in same sequence as the formal parameters they correspond to. They are retained for the benefit of source-level frameworks such as scala.meta.

5.5.10 Assignments

```
  Term          = ASSIGN         Length lhs_Term rhs_Term
```

An ASSIGN entry serializes an assignment E1 = E2. It comprises:

- The left-hand side term E1.
- The right-hand side expression E2.

5.5.11 Blocks

```
  Term          = BLOCK          Length expr_Term Stat*
```

A BLOCK entry serializes a block { S1; … Sn; E }. It comprises:

- The list of statements S1, …, Sn.
- The result expression E.

5.5.12 Lambdas

```
5.5.13 Term          = LAMBDA       Length meth_Term target_Type?
```

5.5.14 A LAMBDA entry serializes a function literal. It comprises:

- 5.5.15 A reference to the method implementing the closure.
- 5.5.16 Optionally, a target type, which must be a SAM (single abstract method) type. If a target type it is given, it is the type of the closure. Otherwise, the type of the closure is the function type corresponding to the implementation method.

5.5.17 Conditional Expressions

```
  Term          = IF             Length cond_Term then_Term else_Term
```

An IF entry serializes a conditional expression if (E1) E2 else E3. It comprises:

- The condition E1.
- The "then" part E2.
- The "else" part E3. Single side ifs if (E1) E2 have the unit literal () as an implied else part.

5.5.18 Match Expressions

```
Term          = MATCH          Length sel_Term CaseDef*
CaseDef       = CASEDEF        Length pat_Tree rhs_Term guard_Term?
```

A MATCH entry serializes a match expression sel match { case1 … casen }. value. It comprises:

- A selector expression sel.
- A list of cases.

Each case case pat => rhs or pat if guard => rhs is serialized by a CASEDEF entry, which comprises:

- The pattern pat.
- The right-hand side expression rhs.
- Optionally, a guard expression guard.

5.5.19 Try Expressions

```
Term          = TRY            Length expr_Term CaseDef* finalizer_Term?
```

A TRY entry serializes a try expression try E1 catch { cases } finally E2 where the catch or finally parts may be missing. It comprises:

- The body of the try expression.
- The list of cases in the catch part of the try expression.
- Optionally, a finalizer expression.

5.5.20 Return Expressions

```
5.5.21 Term          = RETURN         Length meth_ASTRef expr_Term?
```

5.5.22 A RETURN entry serializes a return expression return or return E. It comprises:

- 5.5.23 A reference to the method from which is terminated by the return
- 5.5.24 Optionally a return expression E. If none is given, the unit value () is assumed.

5.5.25 Repeated Arguments

```
Term          = REPEATED       Length elem_Type elem_Term*
```

A REPEATED entry represents a list of arguments that is passed to a repeated formal parameter of type T*. It comprises:

- The assumed type of the elements of the list (this is relevant if the list is empty)

- The list of arguments passed.

5.5.26 Variable Binding Patterns

```
Term          = BIND           Length boundName_NameRef patType_Type pat_Term
```

A BIND entry in term position serializes a variable binding x @ P. It comprises:

- The name x of the defined variable.
- The inferred type of x.
- The pattern P to which x is bound.

5.5.27 Pattern Alternatives

```
Term          = ALTERNATIVE    Length alt_Term*
```

An ALTERNATIVE entry serializes an pattern alternative P1 | … | Pn. It comprises the alternative patterns P1, …, Pn.

5.5.28 Unapply Patterns

```
Term          = UNAPPLY        Length fun_Term UnapplyArg* pat_Type pat_Term*
UnapplyArg    = UNAPPYarg      arg_Term
```

An UNAPPLY pattern serializes an extractor call to an unapply or unapplySeq method. The most general form of such a call is

prefix.unapp[T1, …, Tm](_)(E1, …, Em),

where

- prefix is a reference to an extractor
- unapp is an unapply or unapplySeq method,
- T1, …, Tm are type arguments,
- the wildcard “_” is a placeholder for the selector against which the pattern is matched, and
- E1, …, En are additional arguments to the call (in Scala syntax, such arguments are necessarily arguments to implicit parameters).

The entry comprises:

- The unapply call prefix.unapp[T1, …, Tm]. This is always a selection to a unapply or unapplySeq method, possibly applied to type arguments.
- Any additional arguments E1, …, En represented as UNAPPLYarg entries.
- The inferred type of the pattern.
- The list of patterns matched by the unapply.

5.5.29 Shared Terms

```
Term         = SHARED         term_ASTRef
```

A **SHARED** entry serializes a term by referring to a previously generated entry representing the same term.

5.6 Annotations

```
Annotation   = ANNOTATION     Length tycon_Symbol fullAnnotation_Term
```

An **ANNOTATION** entry serializes an annotation of some definition. It comprises:

- A reference to the definition of the annotation class.
- An instance creation expression that when executed creates a new instance of the given annotation.

The reason for having both the class and the full annotation expression as parts of the entry is that full annotations are read lazily, but the annotation class has to be available before the rest of the annotation is read in order to avoid cycles.

5.7 Modifiers

```
Modifier     = Annotation
               …
```

A modifier gives some additional information of a definition. Annotations are a subclass of modifiers. Other modifiers have fixed entry tags, which are explained below.

Tag	Explanation
PRIVATE	The **private** modifier is given for the definition
INTERNAL	The **internal** modifier is given for the definition. This is currently reserved for future usage.
PROTECTED	The **protected** modifier is given for the definition.
ABSTRACT	The **abstract** modifier is given for the class definition. Note that **ABSTRACT** is only used for classes, never for fields, methods, or traits.
FINAL	The **final** modifier is given for the definition.
SEALED	The **sealed** modifier is given for the class definition.
CASE	The **case** modifier is given for the class definition, or the definition is a **ValDef** representing a **case** object.
IMPLICIT	The **implicit** modifier is given for the definition.
LAZY	The **lazy** modifier is given for the definition.

OVERRIDE The **override** modifier is given for the definition.

INLINE The **inline** modifier is given for the definition.

STATIC The definition should be implemented as a static member in Java.

OBJECT The definition is a **ValDef** representing an **object** or a **ClassDef** representing the class of an **object**.

TRAIT The definition is a **TypeDef** representing a trait.

LOCAL Always used in conjunction with **PRIVATE** or **PROTECTED.** If set, the definition has a qualified **private[this]** or **protected[this]** modifier.

SYNTHETIC The definition is generated by the Scala compiler, it has no counterpart in the original source.

ARTIFACT The definition should be tagged as ACC_SYNTHETIC when implemented in Java.

MUTABLE The definition is a ValDef representing a mutable var.

LABEL The definition is a DefDef representing a label method. Label methods are used internally by the compiler to express control flow.

FIELDaccessor The definition is a compiler-generated getter or setter for a field.

CASEaccessor The definition is a getter for a parameter in the first parameter list of a case class.

COVARIANT The definition is a covariant type parameter or abstract type (marked "+")

CONTRAVARIANT The definition is a contravariant type parameter or abstract type (marked "-")

SCALA2X The definition was produced by a 2.xy compatible Scala compiler.

DEFAULTparameterized The definition is a method that has some default parameters. The default values for any default parameters are given by separate methods with DEFAULTGETTER names.

INSUPERCALL The definition is located in the argument of a constructor supercall

STABLE The definition is a stable method

The following two modifiers each take a qualifier type. It is foreseen that they will be removed and replaced with **internal.**

PRIVATEqualified *qualifier*_**Type**
PROTECTEDqualified *qualifier*_**Type**

The modifiers are given for definitions marked private[Q] or protected[Q] for a qualifier Q. The entry comprises the type of Q (which in the case where Q is a package or object is the associated class).

5.8 Encoding of Tree Tags

Tree tags are grouped into 5 categories that determine what follows, and thus allow to compute the size of the tagged tree in a generic way.

Category 1 (tags 0-63):	**`tag`**
Category 2 (tags 64-95):	**`tag Nat`**
Category 3 (tags 96-111):	**`tag AST`**
Category 4 (tags 112-127):	**`tag Nat AST`**
Category 5 (tags 128-255):	**`tag Length <payload>`**

Here **`tag`** represents a one-byte entry, **`Nat`** represents a natural number, and **`AST`** represents a serialized tree.

6 Standard Section: Positions

```
PositionsSection = filesize_Nat Assoc*
```

The positions section starts with a name reference to the string “Positions”. It contains

- the total length of the source file as a natural number, and
- a list of associations **`Assoc,`** which are defined as follows:

```
Assoc        = addr_Delta offset1_Delta offset2_Delta?
Delta        = Int
```

An association encodes the position of a tree node relative to its parent node. Positions are ranges consisting of a start offset and an end offset. An association consists of 2 or 3 numbers (“deltas”), which are encoded as variable length signed integers.

The first number, *addr*_Delta, records the difference of byte address of the node referenced by the current association relative to the node addressed by the immediately preceding association. *addr*_Delta is always a non-negative number and can be 0 only for the first recorded association.

The third number, *offset2*_Delta, if it is given, records the difference of the end offset of the referenced node relative to the end offset of its parent node.

The second number, *offset1*_Delta, has one of two possible meanings, depending on its sign and whether *offset2*_Delta exists.

- If *offset2*_Delta exists or the number is >= 0, it records the difference of the start offset of the referenced node relative to the start offset of its parent node.
- Otherwise, if *offset2*_Delta does not exist and the number is < 0, it records the difference of the end offset of the referenced node relative to the end offset of its parent node.

The format is unambiguous as long as the delta of the end offset of a node relative to its parent node is always negative or 0. This property needs to be assured by the serializer. Where positions do not fit this scheme, (i.e. a node's position ends later than the position of its parent node), one of the two end positions has to be adapted before serializing.