Overture Technical Report Series
No. TR-002

August 2011

VDM-10 Language Manual

by

Kenneth Lausdah Augusto Ribeiro

Document history

Month	Year	Version
April	2010	

Contents

Chapter 1

Introduction

In order to achieve an ideal common AST three goals have been identified:

1. *Development must be supported both at specification and implementation level, allowing the VDM language itself to be used for tool specification.*

2. *The AST must be extensible while extensions must be kept isolated within a plug-in.*

3. *Sufficient navigation machinery must exist for an editor so that features like e.g. completion and re-factoring can be implemented easily.*

The following subsections will explain the essential principles for the new AST and what changes are required to a generator in order to produce an AST that complies with the identified goals.

It is essential that a new AST is easy to maintain and easy to extend, thus it must only contain functionality essential for the tree structure. To achieve both easy maintenance and support for specification and implementation level development an abstract tree definition can be used like in the `ASTGen` tool. However, extendibility is another matter that need attention; This can be handled by allowing one tree to extend another, by adding new nodes and fields or even refining a type of an existing field.

1.1 Syntax

The new AST that has been developed has an improved structure compared to both the existing Overture and VDMJ trees. The main addition made here is the ability to extend an AST while keeping the changes isolated in a plug-in architecture, explained in detail in section **??**. Secondly, the AST is specified using a grammar file inspired by SableCC[1], and can generated to both VDM and Java as supported by `ASTGen`. The main structural change compared to the Overture AST is the ability to add fields to super classes.

The syntax is divided into a number of sections:

`Packages` This section allowed the default packages to be defined for `base` that forms the basis for all nodes if nothing else is specified and `analysis` where all visitors will be generated under:

```
Packages
base org.overture.ast.node;
analysis org.overture.ast.analysis;
```

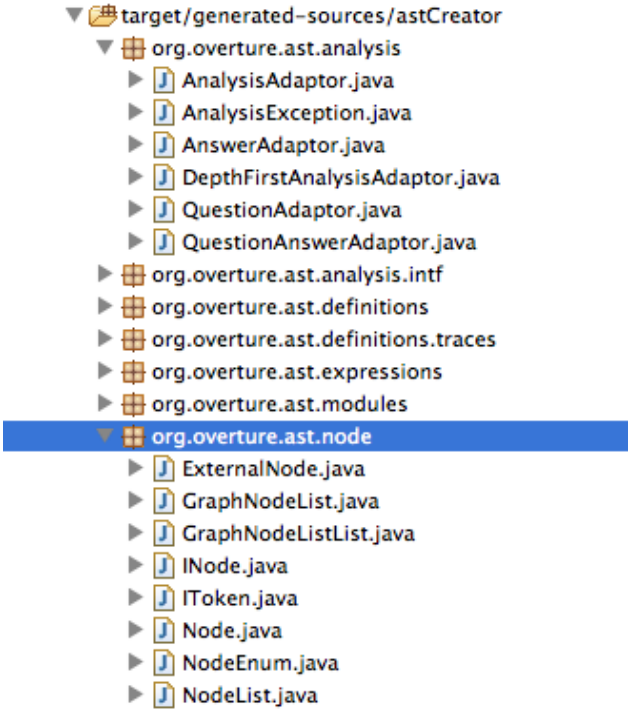


Figure 1.1: The package structure of the generated example.

`Tokens` This section defines tokens or external nodes that can be used as fields in the AST. *The important part here is that they must be clonable*. Tokens are non-generated classes which

[1] `http://www.sablecc.org/`

are used by the generated nodes. Tokens can be used as fields of AST Nodes and are Java classes. The following types of tokens can be specified:

Token A token is an instance of the class `Token`, it is automatically generated from a name plus the text it represents.

Standard Java Types Standard Java types (basic types) can be used in the class form. These tokens start with the keyword `java:`. The use of those relies on them being passed by value when cloning is done. E.g. `Integer, Boolean, Long, Character, ...`

External Java enum This is like the standard Java type where the type given is a Java `enum`. These tokens start with the keyword `java:enum:`

External classes Any external class can be used as a field in the AST but it must implement the interface `ExternalNode` that provides a handle to the AST for proper cloning for the node. These tokens start with the keyword `java:`

External defined Node This enables a node to be specified outside the AST creator but still included in the analysis. The class must extend `Node` and do a proper implementation of the analysis methods and kind methods returning the correct enumeration. These tokens start with the keyword `java:node:`

```
Tokens
//Token
bool = 'bool';
//Standard Java Type
java_Integer = 'java:java.lang.Integer';
//External Java enum
nameScope = 'java:enum:org.overturetool.vdmj.typechecker.NameScope';
//External Java type
location = 'java:org.overturetool.vdmj.lex.LexLocation';
//External defined node
LexToken = 'java:node:org.overturetool.vdmj.lex.LexToken';
```

`Abstract Syntax Tree` This section describes the tree. Names at the top level are considered roots and # are considered sub roots. Sub roots must be specified as a root and a child.

```
Abstract Syntax Tree
exp {-> package='org.overture.ast.expressions'}
    =   {binary} [left]:exp [op]:binop [right]:exp
    |   ...
    ;

binop {-> package='org.overture.ast.expressions'}
    = {and}
    |   {or}
    |   ...
    ;
```

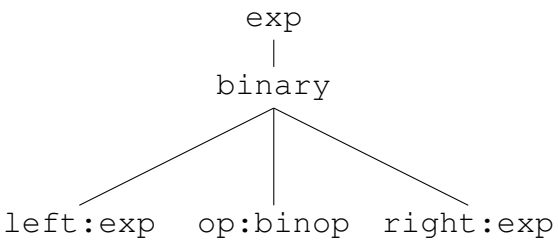


Figure 1.2: AST Example.

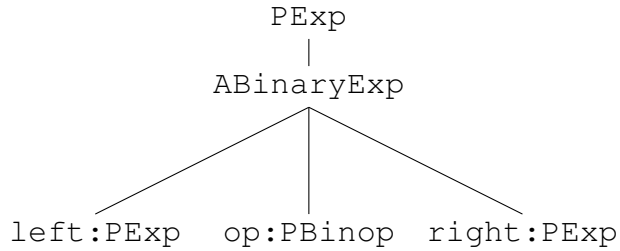


Figure 1.3: AST Java Generated code Example.

The above example will generate a tree like:

The type of field : Fields can be specified in two different ways:

Tree : Tree fields are fields that belong to a single node in tree and only that one. A tree node cannot be a child of any other node in the tree. Thus if an instance is assigned to a tree field the parent it may belong to is disconnected such that the child/parent relation ship is preserved.
Tree fields are syntactically specified as: `[feld]:type`, where `field` is the name and `type` is the type name.

Graph : Graph fields are reference fields thus the get parent will not deterministically return a single parent but just the first parent the instance was added as a reference field of. This type of node can e.g. be used to add type information to nodes where the type is a reference to a shared type.
Tree fields are syntactically specified as: `(field):type`, where `field` is the name and `type` is the type name.

Field types : Fields must define the type of the field as a name reference to a tree element like shown in listing **??** above. The syntax is `:type` where the type can be any of the below shown references:

Simple types : Simple types is just the type name like `:exp` or if a nested type is given `exp.#unary.abs`

Lists : Lists are specified as simple types but appended with a star `:exp*`

Double lists : List simple types but with two stars `:exp**`

Sub-classing : A sub class can be made by specifying a root as an alternative of another root. The naming convention dictates that the sub class roots must begin with #. See Figures 1.4 and 1.5

`Aspect Declaration` This is a feature that allows any fields to be added to base classes (root classes in the AST grammar, the nodes where a a package can be specified). One example could be to add a `location` field to all nodes of the `exp` type. The field will be the first declared field in any of the sub classes of `exp`. The syntax looks like this:

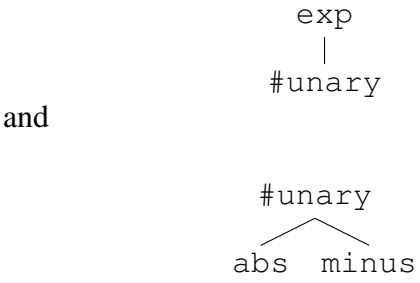


Figure 1.4: AST Example.

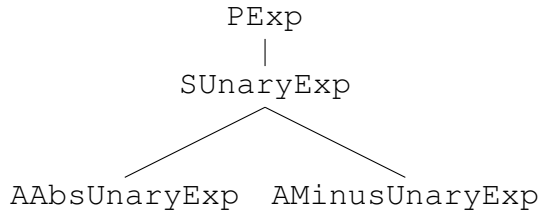


Figure 1.5: AST Java Generated code Example.

```
Abstract Declaration
%exp = [location]:location
    ;
```

Where % prefixes the root node name. The name can also be a composite name like `%exp.#unary`.

1.2 Node's "toString"

The toString macro-processing enables nodes to have a custom toString to make debugging easier. The AST is generated with a toString function that prints all fields one after another. However, in some cases debugging might be eased by printing the node in a more suitable format (e.g. in the syntax of the language at hand). To allow such customization a macro-processor is available together with a small language.

The base princiles is that all fields of a node is accessed as defined in the AST e.g. field 1 is accessed through [field1] and strings can be added around the fields by a quoted test "example test". The custom toString is defined per node by %node-¿sub1 = and can only be defined for leafs of the tree (**A**-nodes). Furthermore, it is possible to embed java withing the toString by using $ to escape to java. If external java classes are to be used withing the toString any imports that might be used must be specified in the begining of the file with the **`import`** keyword as shown below:

```
To String Extensions
// import packages used by external $$ java code
import org.overture.ast.util.Utils;
import org.overture.ast.util.ToStringUtil;


//  Expressions

%exp->apply = [root] "(" + $Utils.listToString($ [args] $)$ + ")"
```

1.3 Analysis

The generator automatically generated a number of visitors that can be applied to any node in the ast, those are:

- Analysis: Simple visitor.
- Answer: Simple visitor that allows a return value.
- Question: Simple visitor that allows an argument to be parsed along.
- QuestionAnswer: Simple visitor that allows an argument to be parsed along and a return value to be returned.
- DepthFirstAnalysis: Same as Analysis but does a depth first visit of the tree. All fields are visited for each node in the specified order from the syntax, base classes first.

All visitors define methods that can be overridden if needed. The methods are named based on the type of node:

- Simple alternatives: `caseA` followed by the name of the alternative.
- Sub roots (prefixed in the gramme with # and `S` in the generated class): `defaultS` the name of the sub root
- Roots (prefixed `P` in the generated class): `defaultP` the name of the root

The following listing illustrates how an overridden method for an unary interpreter van be implemented. It makes use of the numeration kind of the unary operator identifying the sub class of operator hold by the `node.getOperator()` field. This way all subclasses can be auto inserted in a switch statement by e.g. Eclipse.

```
@Override
public Value caseAUnaryExp(AUnaryExp node)
{
        Value val = node.getExp().apply(this);
        switch (node.getOperator().kindPUnop())
        {
                case ABS:
                        return new DoubleValue(val);
                case MINUS:
                        return new DoubleValue(val);
                }
        return new BooleanValue(false);
}
```

1.4 Usage

The ast generator can be invokes like:

```
java -cp astCreator.jar com.lausdahl.ast.creator.CmdMain ast.astV2 .
```

1.4.1 Invoking AST Creator using Maven

The AST Creator is made available as a maven plug-in that enables an AST to be automatically build as part of the build process. The current plug-is in avaliable in the overture repo at: `http://build.overturetool.org/builds/mt4e-m2repo-eclipse3.7.2/`

The plug-in is:

groupId	org.overturetool.tools
artifactId	astcreatorplugin
version	1.0.6

The astcreatorplugin plug-in has the goal **generate** for AST generation and the following configuration properties:

deletePackageOnGenerate A list of java packages that should be deleted before generation

ast A path to the ast file starting from src/main/resources

outputDirectory an option to define an alternative output location for the generated AST

The pom configuration is as shown below in listing **??**.

```
<build>
    <plugins>
      <plugin>
        <groupId>org.overturetool.tools</groupId>
        <artifactId>astcreatorplugin</artifactId>
        <version>1.0.6</version>
        <executions>
          <execution>
            <id>java</id>
            <phase>generate-sources</phase>
            <goals>
              <goal>generate</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <deletePackageOnGenerate>
            <param>vdm.generated.node</param>
            <param>org.overture.ast</param>
          </deletePackageOnGenerate>
          <ast>overtureII.astv2</ast>
          <!--outputDirectory>${project.basedir}/src/main/java</outputDirectory -->
```

```
      </configuration>
    </plugin>
  </plugins>
```

1.5 Extensions

In this subsection we describe a new way to specify and generate a tree that extends a base tree and preserves backwards compatibility with its base tree. Extensions are only visible to components that depend on the extended tree otherwise only the base tree structure will be accessible. In figure 1.6 examples of extensions required by Overture is shown. Each box defines a plug-in feature that needs its own extensions like a type field to store the derived type information from a type checker and a proof obligation generator needs a place to attach proof obligations. The figure also illustrates that an extended tree may be further extended; In this case the derived type information is needed by the interpreter.

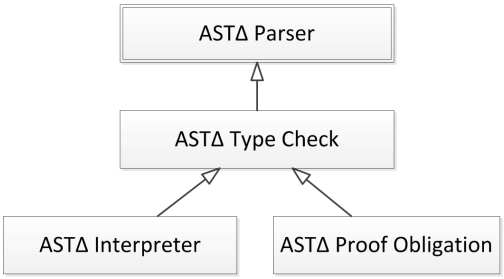


Figure 1.6: Illustration of AST extensions in Overture.

The extension principle is based on sub-classing from a class-hierarchy where each extended node sub-classes the corresponding node in the base hierarchy. This allows a new extended AST to be used by any implementation that supports its base tree e.g from figure 1.6 any feature that uses a typed AST can also use an interpreter AST. To achieve isolation between extensions we require extensions to be declared in a separate file so that a generator (illustrated in figure 1.7) can combine them into a single tree and generate a converter from the base tree to the extended tree. To illustrate the type check extension listing 1.1 shows how expressions are extended with a field for the derived type information.

Listing 1.1: Example showing how a type can be added to all expressions.

```
Aspect Declaration
exp = [type]:type;
```

1.5.1 Inside view of the extension idea

The idea is to have two ast files one with the base tree and one with the additions to the base tree. The tree will then be generated using interfaces for base classes to allow multiple inheritance in

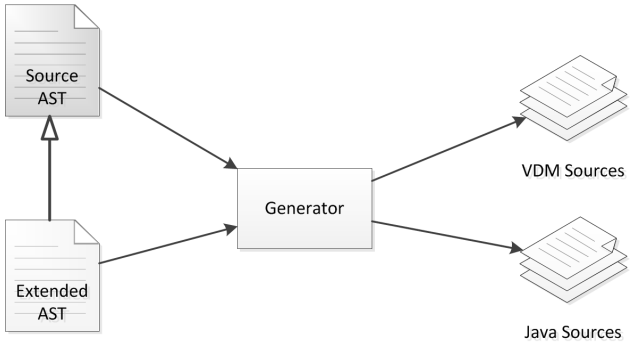


Figure 1.7: Extending an existing AST.

Java. This way a new node can be added as a sub class of an existing root (P.. or S .. class), the new node will then implement the new interface and extend an existing base node. The new interface for the extended tree will then define new enumerations to identify the new node, whereas the base tree enumeration returned for identification will be set to a extended type.

1.6 Guidelines and Cornercases

To make the use of the AST v2 Extensions more sensibly, some rules are defined:

- it is ONLY possible to use ”base visitors” in the extended AST if there is no new nodes in the AST instance. So if ”Base” tree is cloned to ”Extended” tree and the visitor is applied immediately, then it should work. If any ”Extended” node is added to the extended tree, the same visitor will not work and throw a ”RuntimeException” a an unknown node is hit.
- Calling the ”base” kind method of an ”extended” node will result in a ”RuntimeException”
- The generated ”Extended” visitor will contain the new cases for the added nodes, the cases of the nodes that already existed but were changed in the extension (added fields) and will not contain the ones that are unchanged.
- Setting the ”Base” parent in the ”Extended” tree will result in a ”RuntimeException”
- When cloning a tree, extra care should be taken to preserve the graph fields connections. A solution might be to clone the tree first without graph fields and keep a map of ”object $\rightarrow$ cloned object” and in the second pass fill the graph fields gap.
- If an extra field was added to a node existing in the base case the visitor case needs to overwritten in the extended visitor.
- It is not advised to mix nodes from base tree with the extended tree.

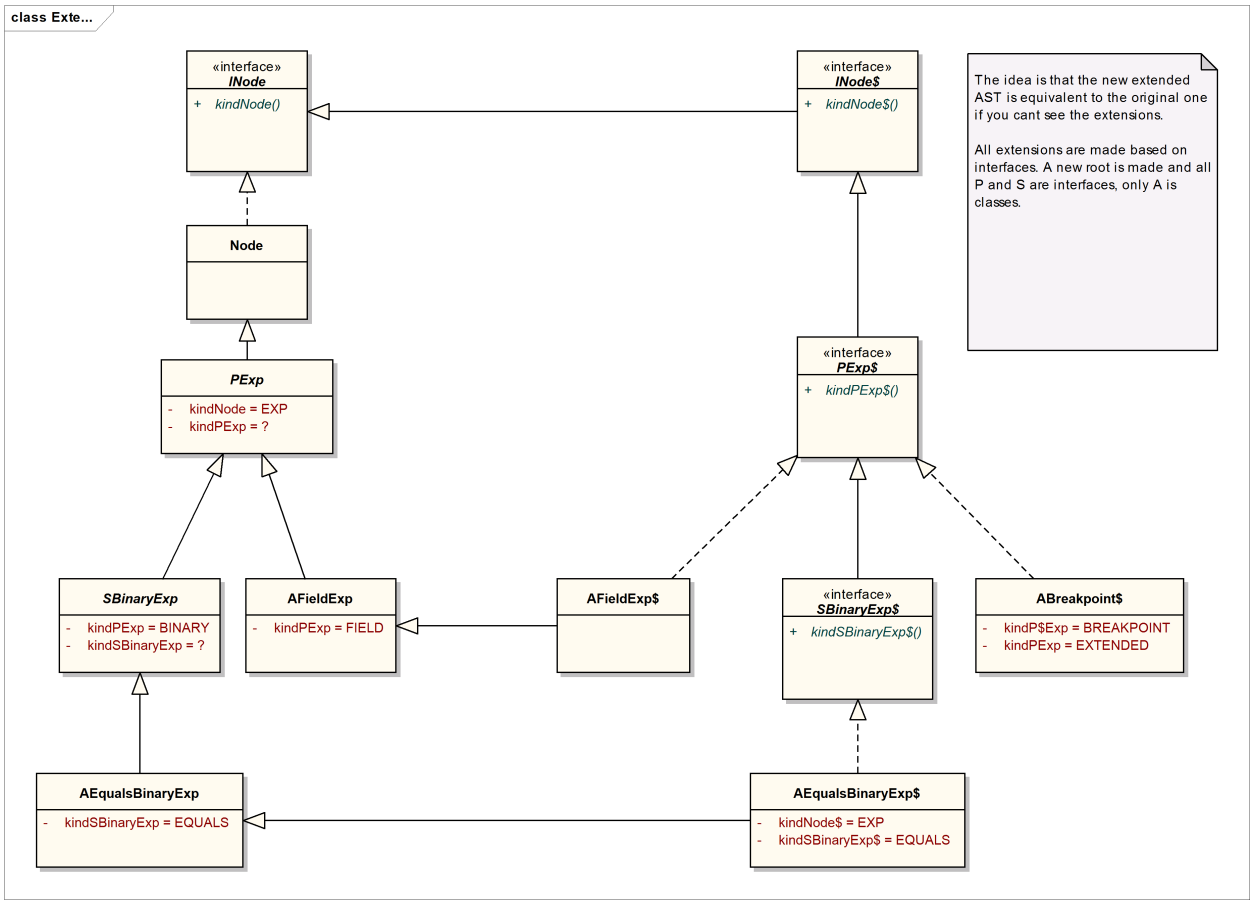


Figure 1.8: Illustration of AST extensions in Overture as a UML diagram.

Appendix A

Class diagrams of the Extended AST

A.1 Base

A.2 Extended

A.3 Extended AST Test Class

```
import junit.framework.TestCase;

import org.overture.ast.expressions.AE1Exp;
import org.overture.ast.expressions.AE1ExpInterpreter;
import org.overture.ast.expressions.AE2ExpInterpreter;
import org.overture.ast.expressions.AE3Exp;
import org.overture.ast.expressions.AE3ExpInterpreter;
import org.overture.ast.expressions.AE4ExpInterpreter;
import org.overture.ast.expressions.EExp;
import org.overture.ast.expressions.EExpInterpreter;
import org.overture.ast.node.NodeEnum;
import org.overture.ast.node.NodeEnumInterpreter;

public class TestCase1 extends TestCase
{
  AE1Exp base = null;
  AE2ExpInterpreter extended = null;

  @Override
  protected void setUp() throws Exception
  {
    base = new AE1Exp();
    extended = new AE2ExpInterpreter();
  }

  public void testenum()
  {
```

```
    base.kindNode();
    extended.kindNode();

    base.kindPExp();
    try
    {
      extended.kindPExp();
      fail("kindPExp successeded but should have failed");
    } catch (RuntimeException e)
    {

    }

    extended.kindPExpInterpreter();
  }

  public void testBaseVisitor()
  {
    System.out.println("testBaseVisitor");
    base.apply(new BaseVisitor());

    try
    {
      extended.apply(new BaseVisitor());
      fail("apply with BaseVisitor successeded but should have failed since this exp is not
    } catch (RuntimeException e)
    {

    }

  }

  @SuppressWarnings("deprecation")
  public void testAddedFieldToNode()
  {

    System.out.println("Base visit E3");
    AE1Exp field1Base = new AE1Exp();
    AE3Exp rootBase = new AE3Exp(field1Base);
    DBaseVisitor vBase = new DBaseVisitor();
    rootBase.apply(vBase);
    assertTrue("Root must be visited first", vBase.visitedNodes.get(0).equals(rootBase));
    assertTrue("Field 1 of root must be visited second", vBase.visitedNodes.get(1).equals(f
    assertEquals(vBase.visitedNodes.size(), 2);

    // System.out.println("Mixed visit E3");
    // AE1ExpInterpreter field1_ = new AE1ExpInterpreter();
    // AE3Exp mixtedRoot = new AE3Exp(field1_);
    // try
    // {
    // mixtedRoot.apply(vBase);
```

```
    // fail("E1 interpreter can not be visited with a base visitor. To allow this the exter
    // } catch (RuntimeException e)
    // {
    //
    // }
    //
    // System.out.println("Mixed Extended visit E3");
    // DExtendedDelegateVisitor extendedDelegateVisitor = new DExtendedDelegateVisitor();
    // mixtedRoot.apply(extendedDelegateVisitor);
    // System.out.println(extendedDelegateVisitor.visitedNodes);

    System.out.println("Extended visit E3");
    AE1ExpInterpreter field1_ = new AE1ExpInterpreter();
    AE2ExpInterpreter field2_ = extended;
    AE3ExpInterpreter root = new AE3ExpInterpreter(field1_, field2_);

    DExtendedVisitor vExtended = new DExtendedVisitor();
    root.apply(vExtended);

    assertTrue("Root must be visited first", vExtended.visitedNodes.get(0).equals(root));
    assertTrue("Field 1 of root must be visited second", vExtended.visitedNodes.get(1).equa
    assertTrue("Field 2 of root must be visited second", vExtended.visitedNodes.get(2).equa
    System.out.println(vExtended.visitedNodes);
  }

  public void testExtendVisitor()
  {
    System.out.println("testExtendVisitor");
    base.apply(new ExtendedVisitor());
    extended.apply(new ExtendedVisitor());

  }

  public void testPExtendedEnums()
  {
    for (NodeEnum e1 : NodeEnum.values())
    {
      try
      {
        Enum.valueOf(NodeEnumInterpreter.class, e1.toString());
      } catch (IllegalArgumentException e)
      {
        fail(e1 + " should exist in both ASTs");
      }
    }

    for (EExp e1 : EExp.values())
    {
      try
      {
        Enum.valueOf(EExpInterpreter.class, e1.toString());
```

```
      } catch (IllegalArgumentException e)
      {
        fail(e1 + " should exist in both ASTs");
      }
    }

    try
    {
      if (Enum.valueOf(NodeEnum.class, NodeEnumInterpreter.STM.toString()) != null)
      {
        fail("STM should exist in both ASTs");
      }
      fail("STM should exist in both ASTs");
    } catch (IllegalArgumentException e)
    {

    }

    try
    {
      if (Enum.valueOf(EExp.class, EExpInterpreter.E2.toString()) != null)
      {
        fail("E2 should exist in both ASTs");
      }
    } catch (IllegalArgumentException e)
    {

    }
  }

  @SuppressWarnings("deprecation")
  public void testclone()
  {
    AE2ExpInterpreter a=  extended.clone();
    System.out.println(a);

    AE1ExpInterpreter field1_ = new AE1ExpInterpreter();
    AE2ExpInterpreter field2_ = extended;
    AE3ExpInterpreter root = new AE3ExpInterpreter(field1_, field2_);

    AE3ExpInterpreter  r2 = root.clone();
    if(r2==root)
    {
      fail("Clone did not work");
    }

    if(r2.getField1()==field1_)
    {
      fail("Clone did not work");
    }
```

```
    if(r2.getField2()==field2_)
    {
      fail("Clone did not work");
    }
  }

  @SuppressWarnings("deprecation")
  public void testParent()
  {
    AE1ExpInterpreter field1_ = new AE1ExpInterpreter();
    AE2ExpInterpreter field2_ = extended;
    AE3ExpInterpreter root = new AE3ExpInterpreter(field1_, field2_);
    if(field1_.parent()!=root)
    {
      fail("Parent not set correctly");
    }
    if(field2_.parent()!=root)
    {
      fail("Parent not set correctly");
    }
    if(root.parent()!=null)
    {
      fail("Parent not set correctly");
    }

    AE3ExpInterpreter root2 = new AE3ExpInterpreter(field1_, field2_);

    if(field1_.parent()!=root2)
    {
      fail("Parent not set correctly");
    }
    if(field2_.parent()!=root2)
    {
      fail("Parent not set correctly");
    }
    if(root2.parent()!=null)
    {
      fail("Parent not set correctly");
    }
  }


  @SuppressWarnings("deprecation")
  public void testGraphField()
  {
    AE1ExpInterpreter field1_ = new AE1ExpInterpreter();
    if(field1_.parent()!=null)
    {
      fail("Parent not set correctly");
    }
    AE4ExpInterpreter e4 = new AE4ExpInterpreter(field1_);
```

```
    if(field1_.parent()!=e4)
    {
      fail("Parent not set correctly");
    }

    AE4ExpInterpreter e4_1 = new AE4ExpInterpreter(field1_);

    if(field1_.parent()!=e4)
    {
      fail("Parent not set correctly");
    }

    AE3ExpInterpreter e3 = new AE3ExpInterpreter(field1_, null);

    if(field1_.parent()!=e3)
    {
      fail("Parent not set correctly");
    }

    if(e4.getField2()==null || e4_1.getField2()==null)
    {
      fail("Graph nodes should not be removed when used in other nodes");
    }

  }
}
```

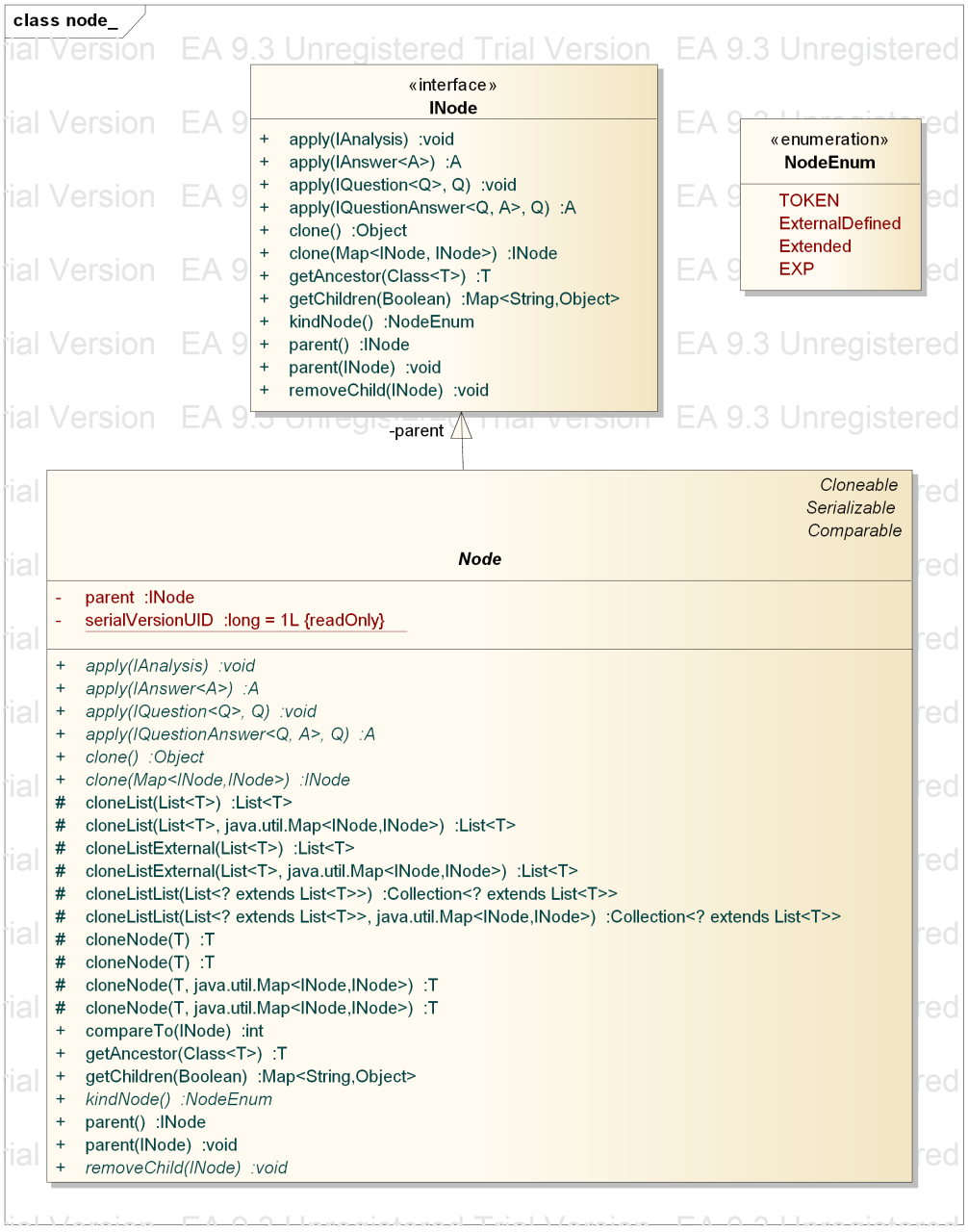


Figure A.1: Base node package

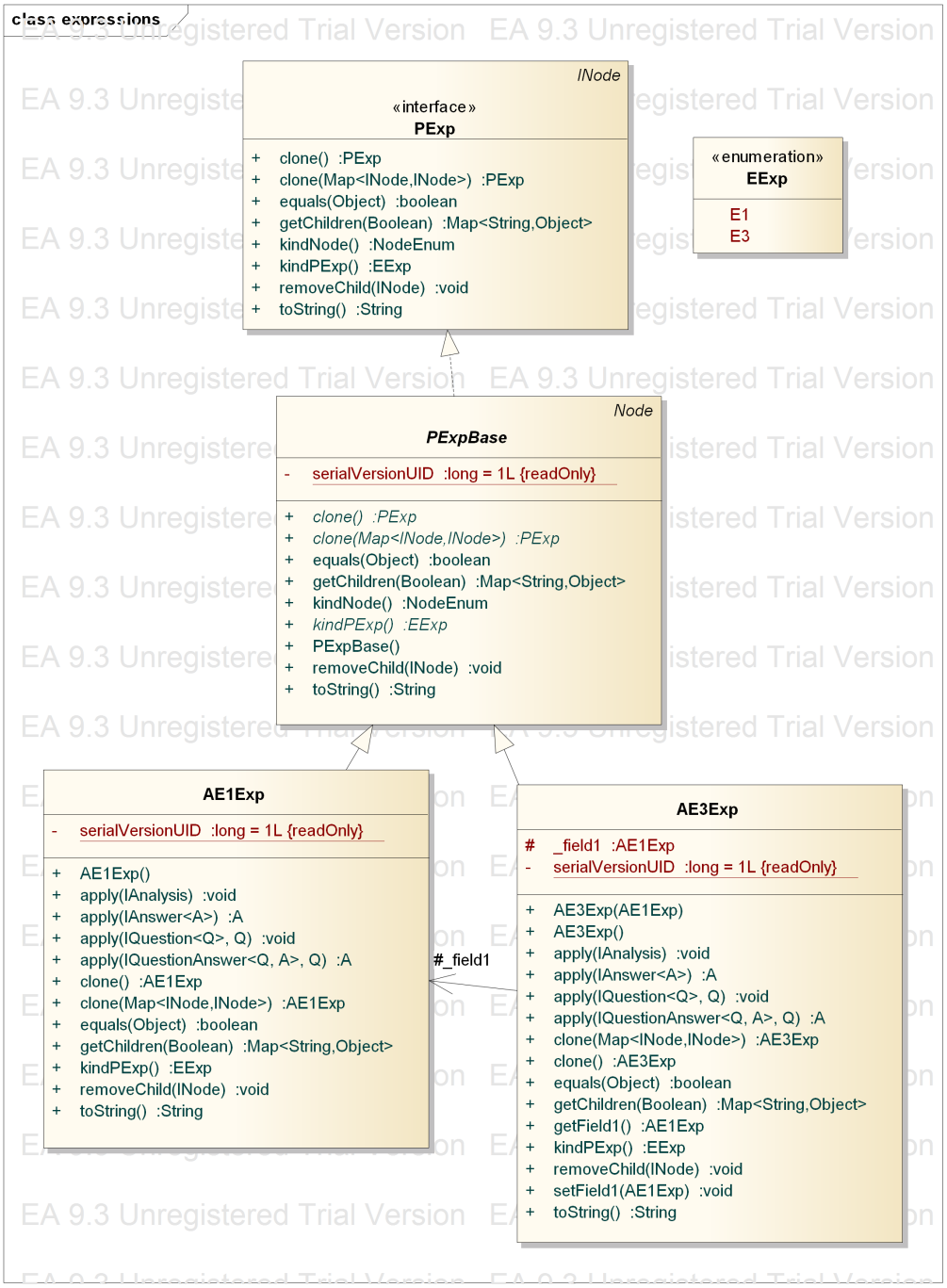


Figure A.2: Base expression package

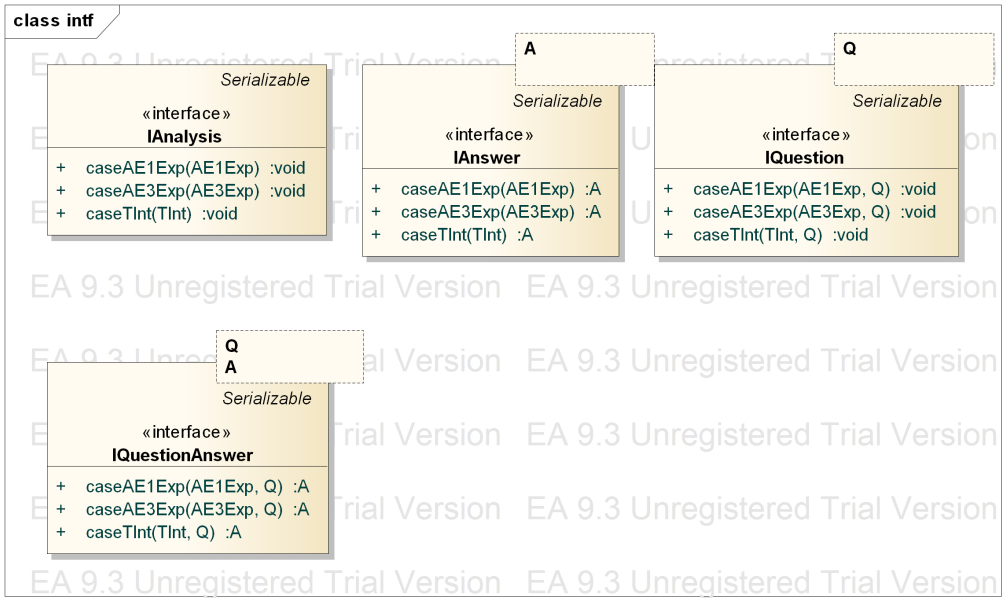


Figure A.3: Base analysis interfaces package

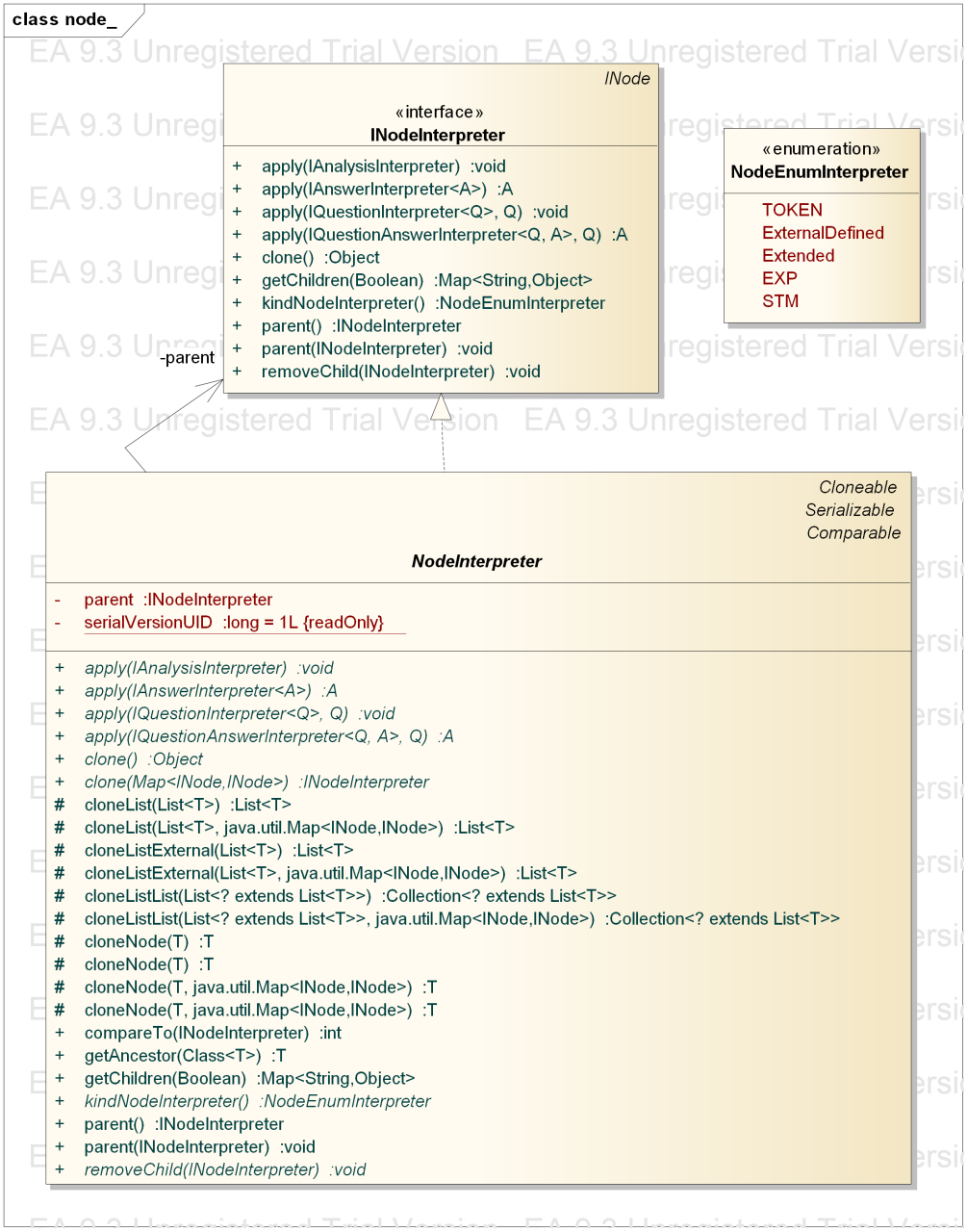


Figure A.4: Extended node package

Figure A.5: Extended expressions package

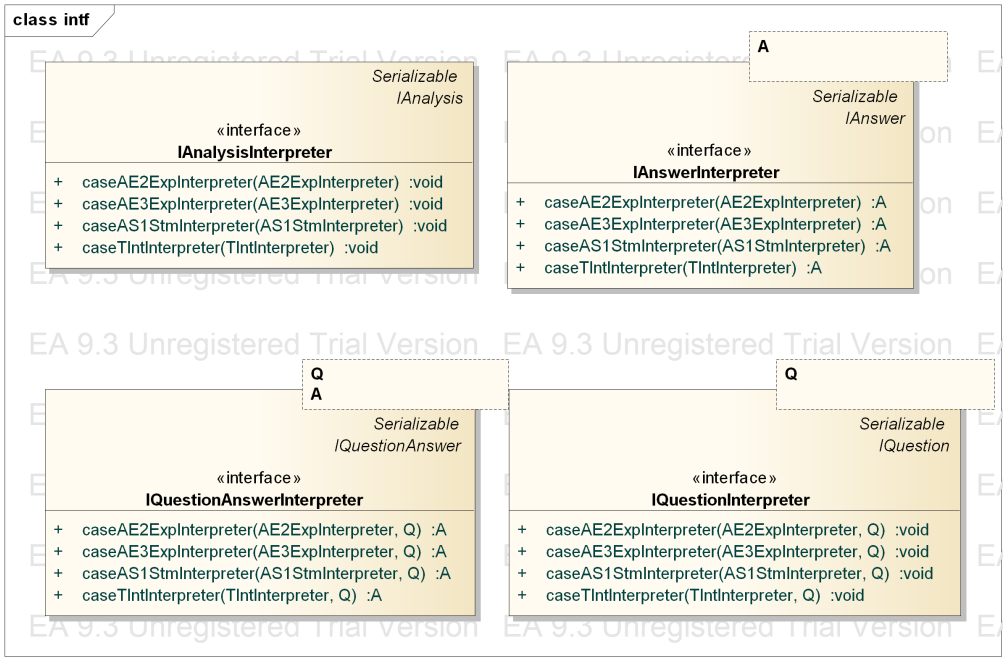
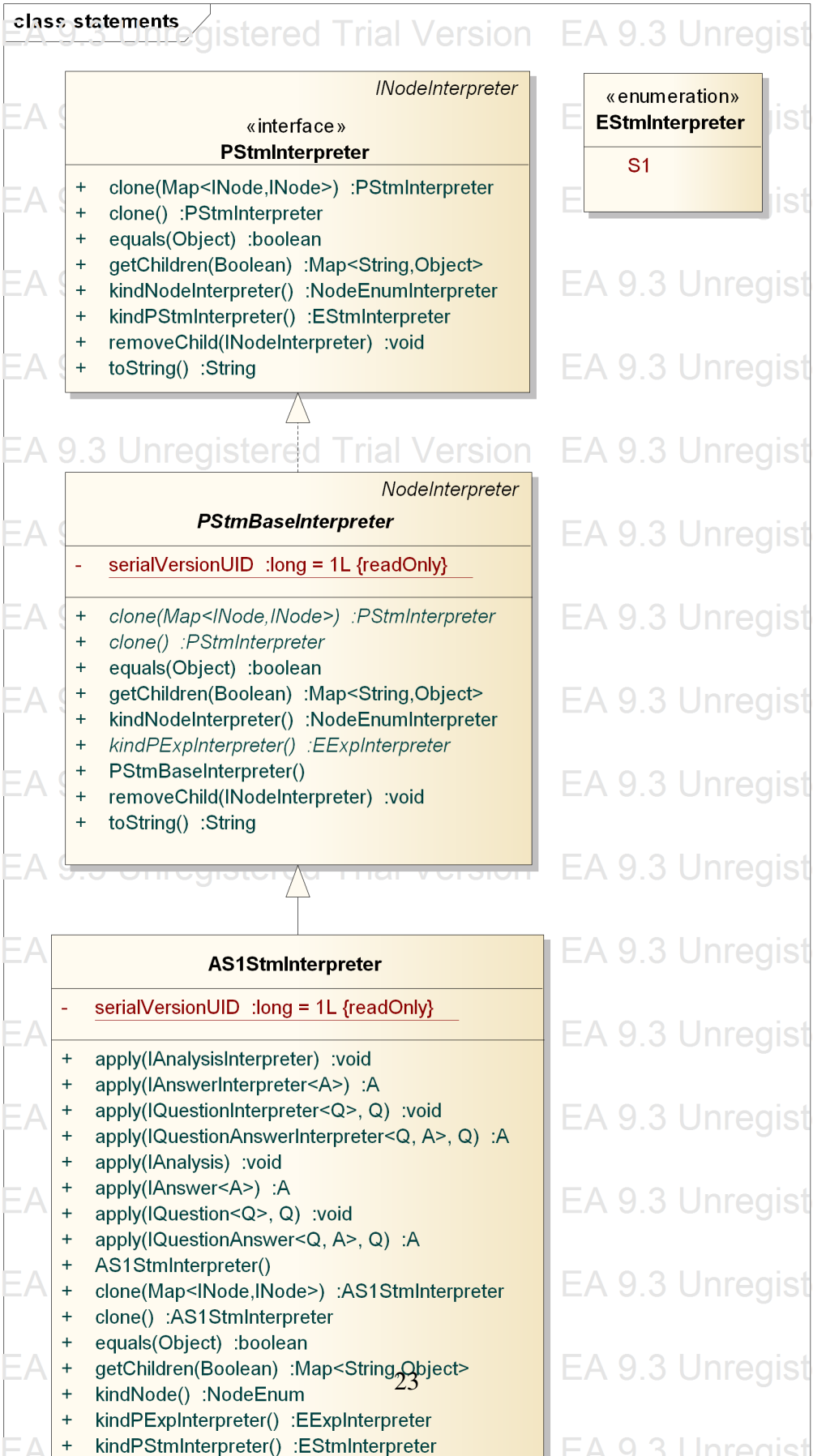


Figure A.6: Extended analysis interfaces package


class statements
INodeInterpreter
«interface»
PStmInterpreter
+ clone(Map<INode,INode>) :PStmInterpreter
+ clone() :PStmInterpreter
+ equals(Object) :boolean
+ getChildren(Boolean) :Map<String,Object>
+ kindNodeInterpreter() :NodeEnumInterpreter
+ kindPStmInterpreter() :EStmInterpreter
+ removeChild(INodeInterpreter) :void
+ toString() :String
«enumeration»
EStmInterpreter
S1
NodeInterpreter
PStmBaseInterpreter
- serialVersionUID :long = 1L {readOnly}
+ clone(Map<INode,INode>) :PStmInterpreter
+ clone() :PStmInterpreter
+ equals(Object) :boolean
+ getChildren(Boolean) :Map<String,Object>
+ kindNodeInterpreter() :NodeEnumInterpreter
+ kindPExpInterpreter() :EExpInterpreter
+ PStmBaseInterpreter()
+ removeChild(INodeInterpreter) :void
+ toString() :String
AS1StmInterpreter
- serialVersionUID :long = 1L {readOnly}
+ apply(IAnalysisInterpreter) :void
+ apply(IAnswerInterpreter<A>) :A
+ apply(IQuestionInterpreter<Q>, Q) :void
+ apply(IQuestionAnswerInterpreter<Q, A>, Q) :A
+ apply(IAnalysis) :void
+ apply(IAnswer<A>) :A
+ apply(IQuestion<Q>, Q) :void
+ apply(IQuestionAnswer<Q, A>, Q) :A
+ AS1StmInterpreter()
+ clone(Map<INode,INode>) :AS1StmInterpreter
+ clone() :AS1StmInterpreter
+ equals(Object) :boolean
+ getChildren(Boolean) :Map<String,Object>
+ kindNode() :NodeEnum
+ kindPExpInterpreter() :EExpInterpreter
+ kindPStmInterpreter() :EStmInterpreter

Appendix B

Challanges

- Tree consistency
- Tree automated analysis; visit in decent order
- Generating a tree to replace OO developed classed. Dificult because of functionality embedded in the classes where 1,2,4 super classes all might override some methods.
- Caching: converting a program which does OO caching/linking to use a generated tree. Not easy, solutions:
  - Hashmap
  - Add custum graph fields to the tree. Nodes in a graph field would be different because they are not a child of the node when are attached to. In relation to tree fields which can only have one parent and thus needs to disconnect from their previously attached parent before moving to a new,
- Handling OO embeded functions outside a tree. Developing assistants to handle public/private functions from OO tree. The difficulty is to keep the hierarchy intact.
- Display: For debugging it is important to have a decent toString of a tree, two solutions:
  - Generated toString including all fields of a node and it name.
  - Custom toString. Since a tree might represent (to some extend) the syntax of a language a toString representation close to the real syntax might be prefered. One solution would be to add a way to define the toString method of nodes based on their fields and strings plus possibly allowing external java methods to be called for more advanced display functions.