Overture Technical Report Series
No. TR-002

August 2011

VDM-10 Language Manual

by

Kenneth Lausdah Augusto Ribeiro

Overture – Open-source Tools for Formal Modelling

Document history

Month	Year	Version
April	2010	

Contents

Chapter 1

Introduction

1.1 Syntax

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;
```

`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.* 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. The use of those relies on them being parse 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`.

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.

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.

```
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}
    |   ...
    ;
```

The above example will generate a tree like:

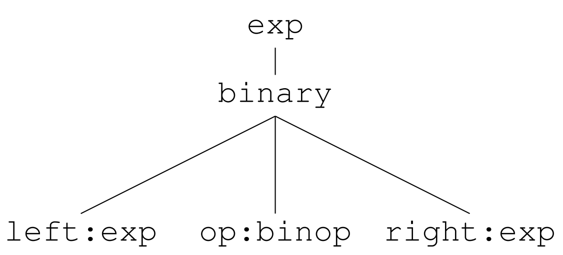


Figure 1.1: AST Example.

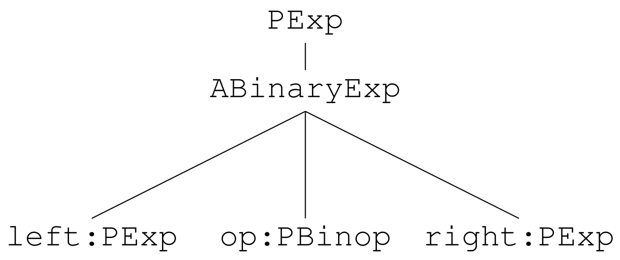


Figure 1.2: AST Java Generated code Example.

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: `(feld):type`, where `field` is the name and `type` is the type name.

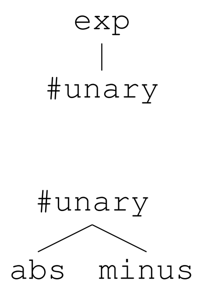
Field types : Fields must define the type of the field as a name reference to a tree element like shown in listing **??** above. The syntax is `:type` where the type can be any of the below shown references:

- **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 #. E.g.:



and

Figure 1.3: AST Example.

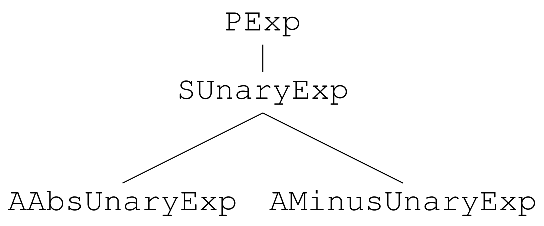


Figure 1.4: AST Java Generated code Example.

`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:

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

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

1.2 Usage

The ast generator can be invokes like:

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

1.3 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 1.5 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.

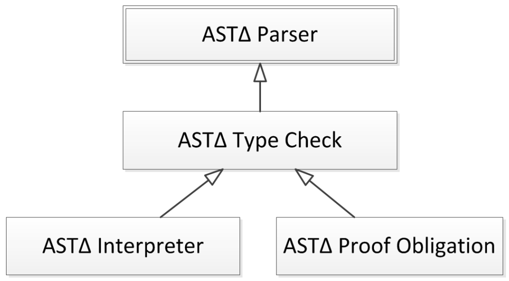


Figure 1.5: Illustration of AST extensions in Overture.

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 1.5 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 1.6) 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;
```

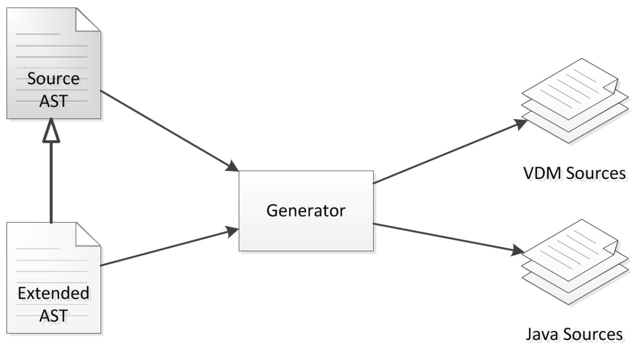
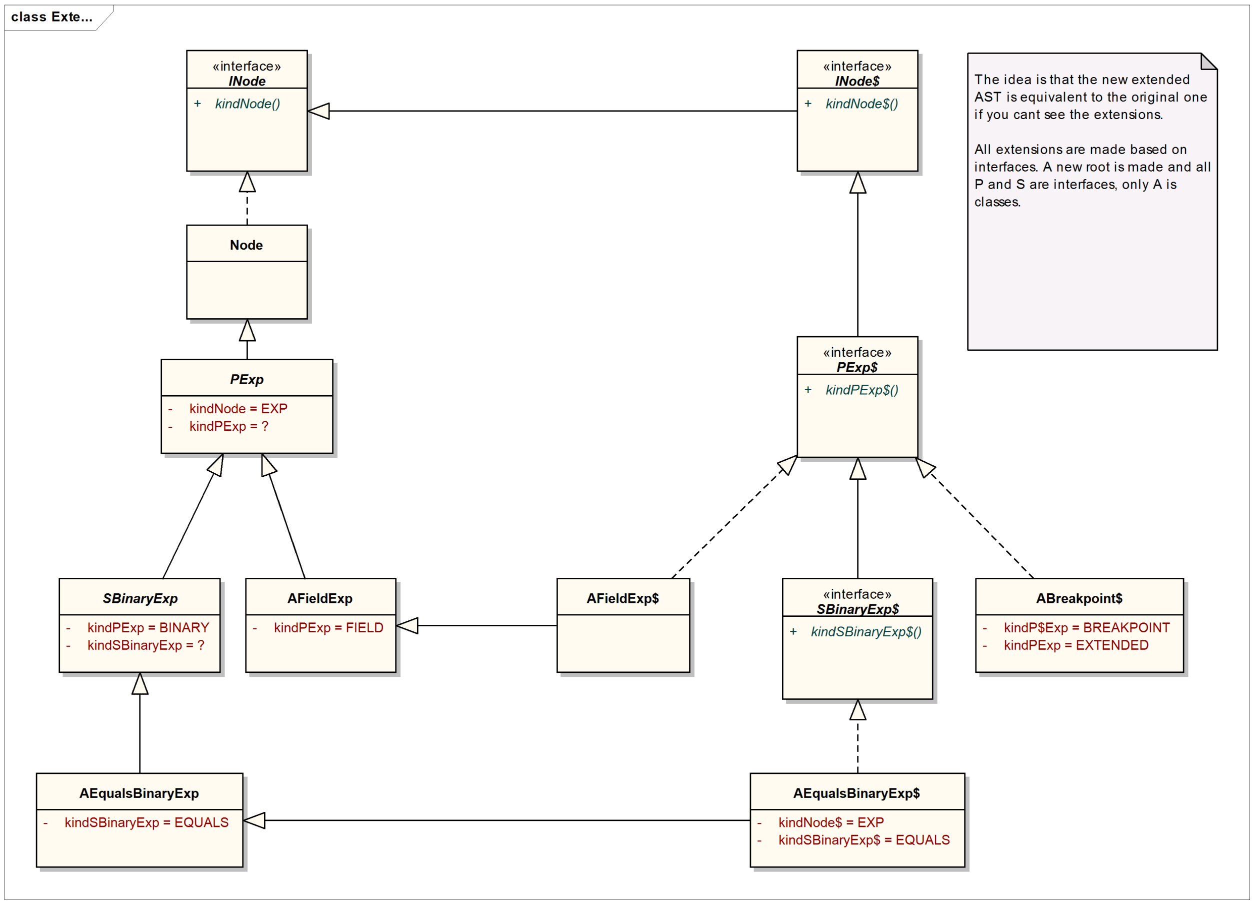


Figure 1.6: Extending an existing AST.

1.3.1 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 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.



The idea is that the new extended AST is equivalent to the original one if you cant see the extensions.

All extensions are made based on interfaces. A new root is made and all P and S are interfaces, only A is classes.

Figure 1.7: Illustration of AST extensions in Overture as a UML diagram.

1.4 Analysis

The generator automatically generated a number of automated 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);
}
```