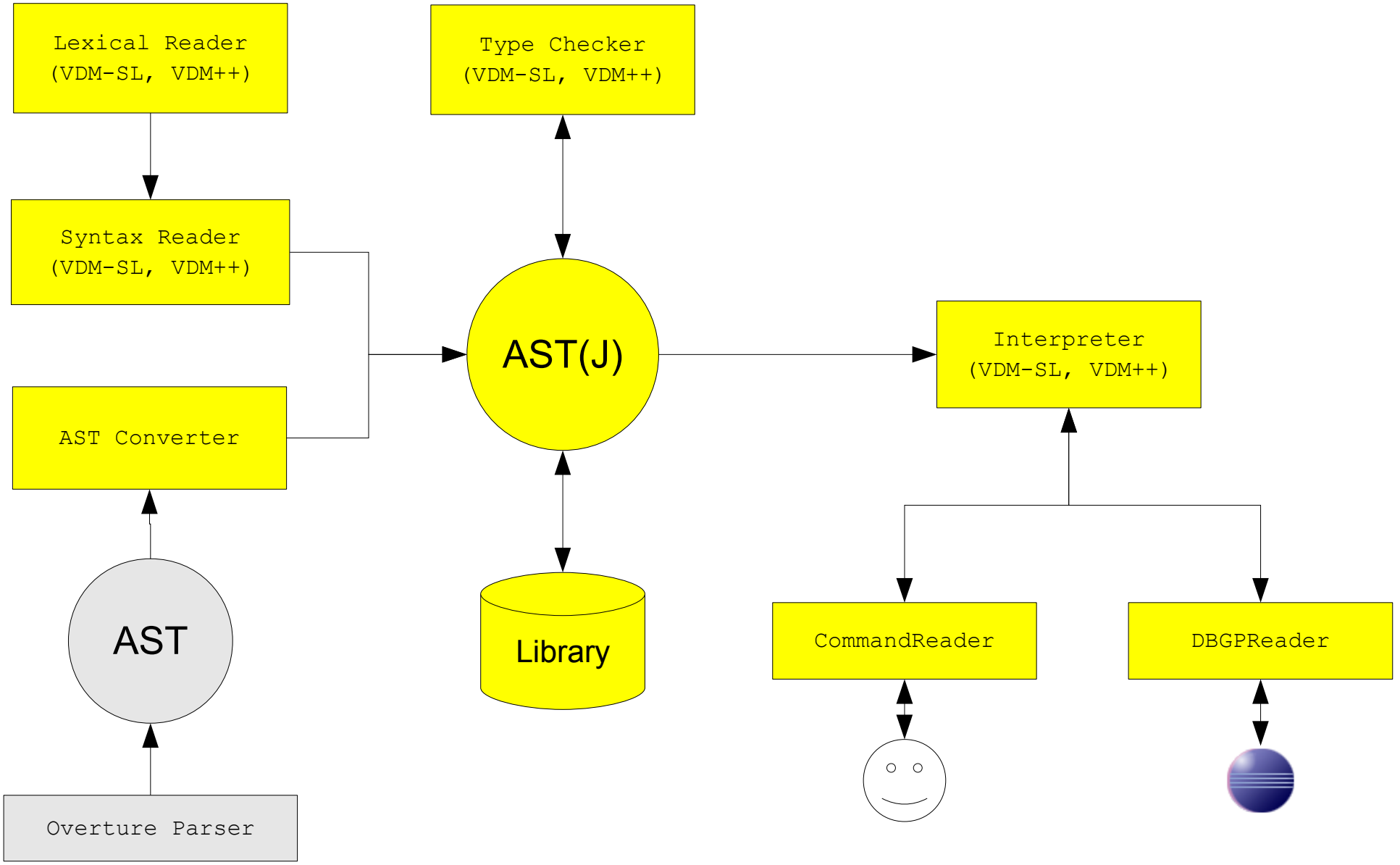
VDMJ

Nick Battle, Fujitsu
(nick.battle@uk.fujitsu.com)

VDMJ Overview

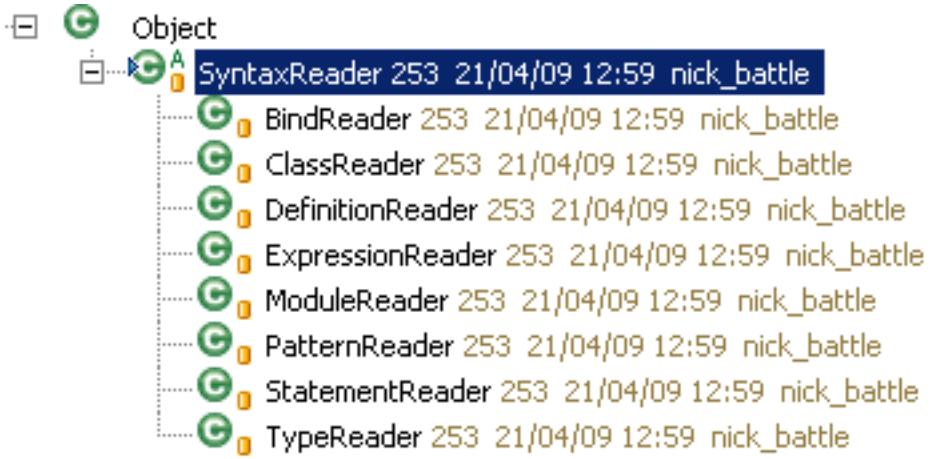
- Where did VDMJ come from...?
- Provides support for VDM-SL and VDM++ parsing, static type checking, interpreting/debugging, PO generation, test coverage and combinatorial testing
- Pure Java (5 or later), no external dependencies
- Released under GPLv3 by Fujitsu
- Command line interface only
- Informally developed (eg. not specified in VDM)
- User Guide and Design Specification docs available
- Passes CSK test suite (>3000 tests, converted to JUnit tests)
- Quite fast (3000 tests in ~20 seconds).

VDMJ Architecture



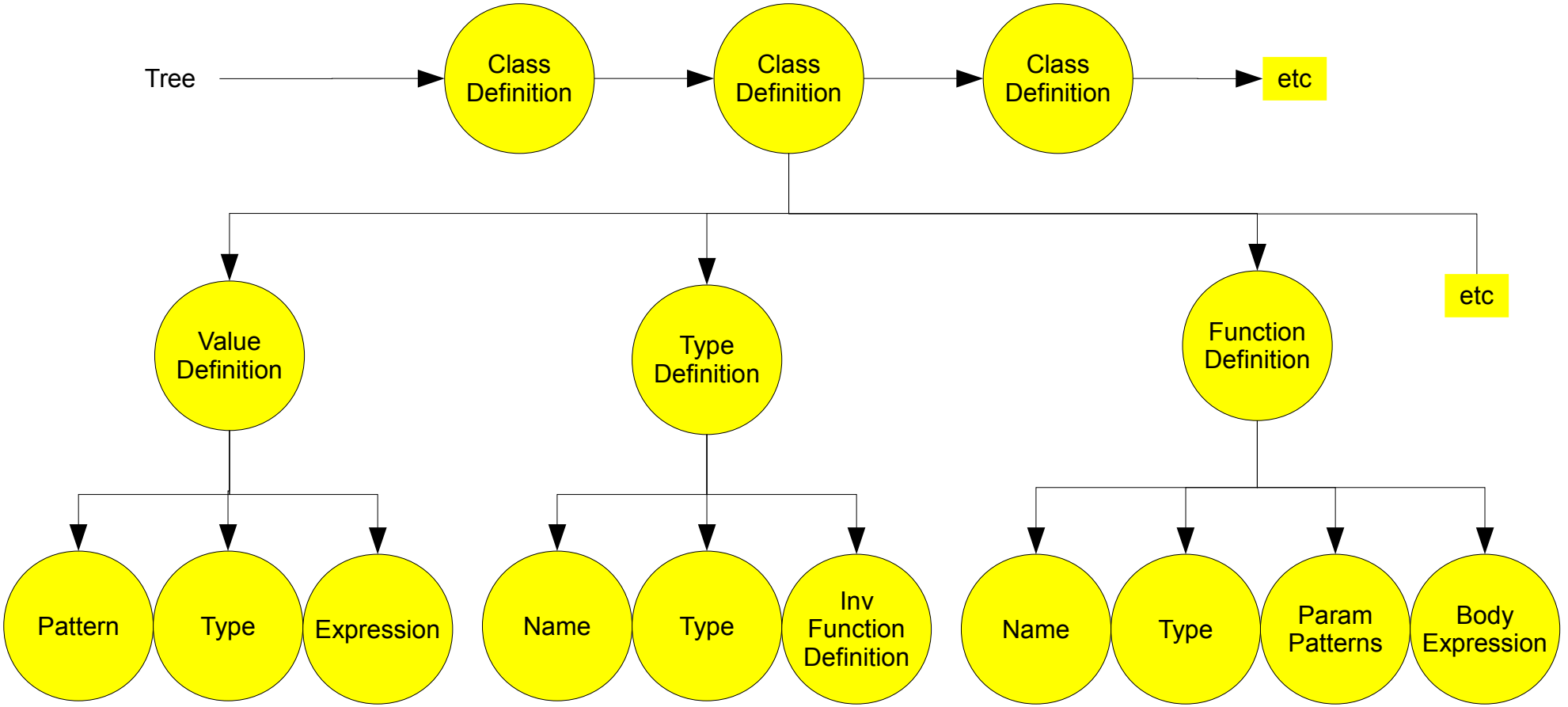
VDMJ Parser 1

VDMJ Parser 2



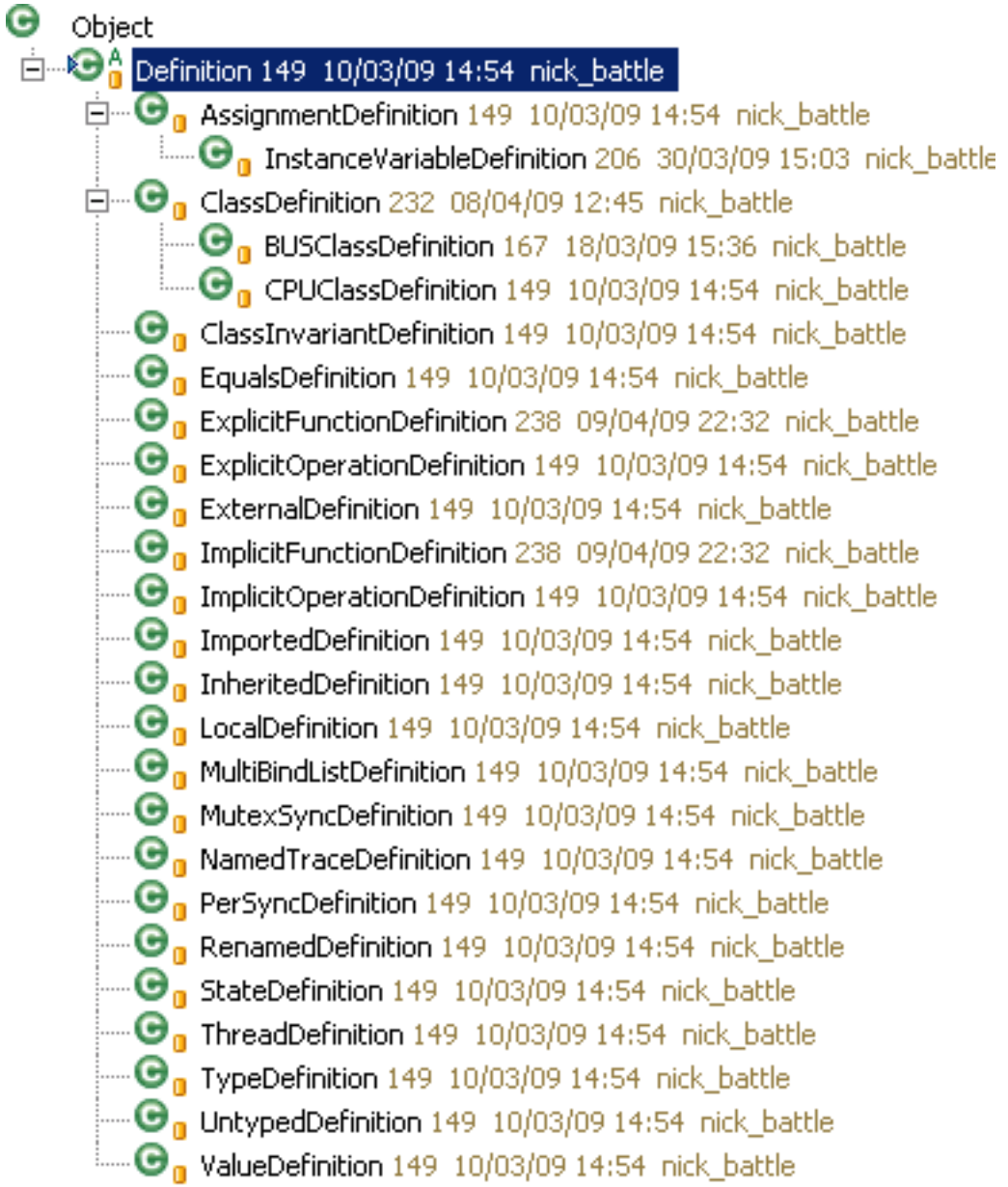
- One SyntaxReader subclass for each major group in the grammar
- Top level ModuleReader and ClassReader combine the others and return the AST (class list or module list)
- Each reader typically has one public method, like readPattern(), readType(), readClass() etc. and many private methods.
- All readers are constructed by being passed a LexTokenReader

VDMJ AST 1

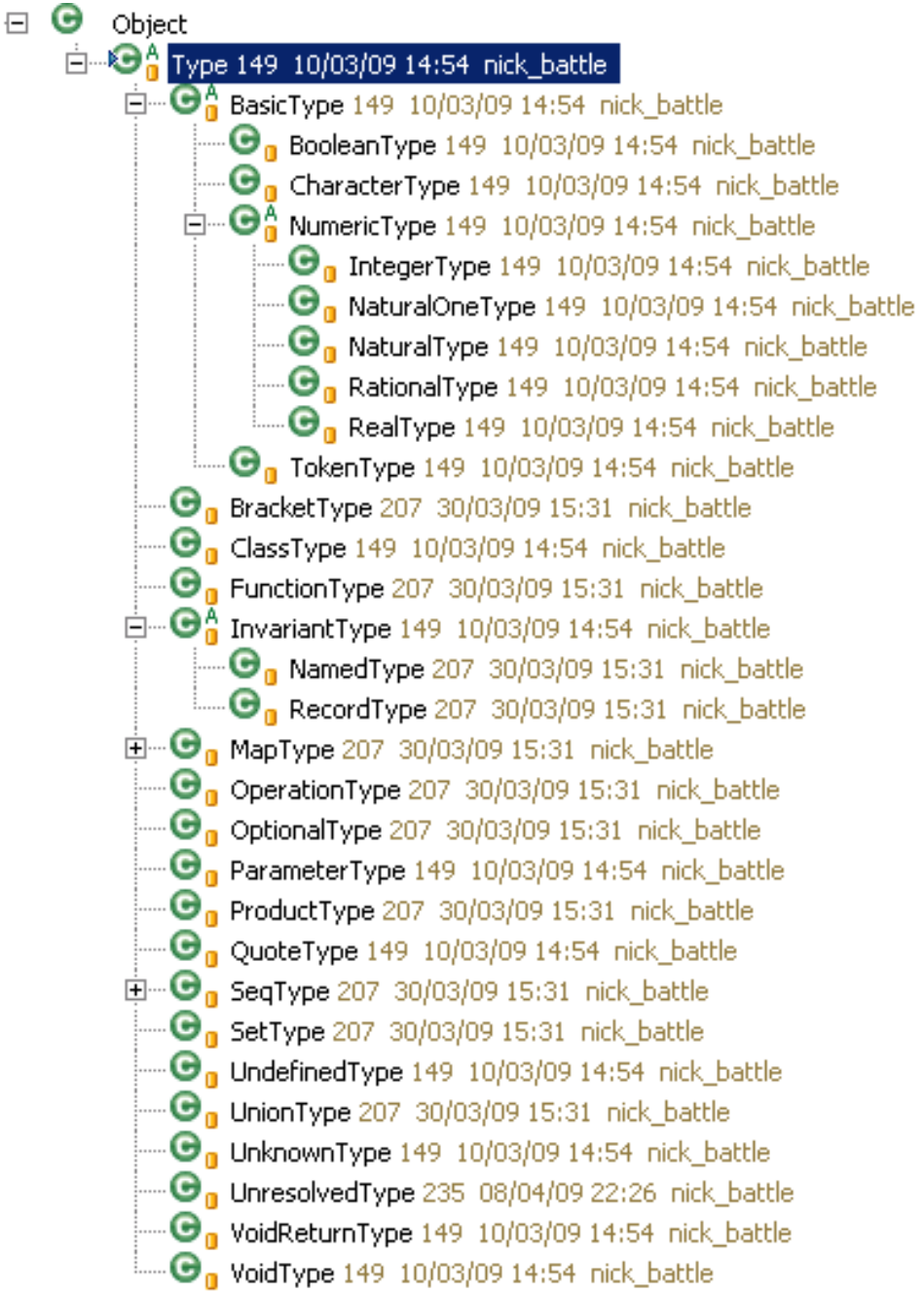


- AST is a "washing line" of classes or modules at the top level
- Classes and modules contain definitions
- Definitions can contain yet more definitions - "mk_(a,b) = tupval"

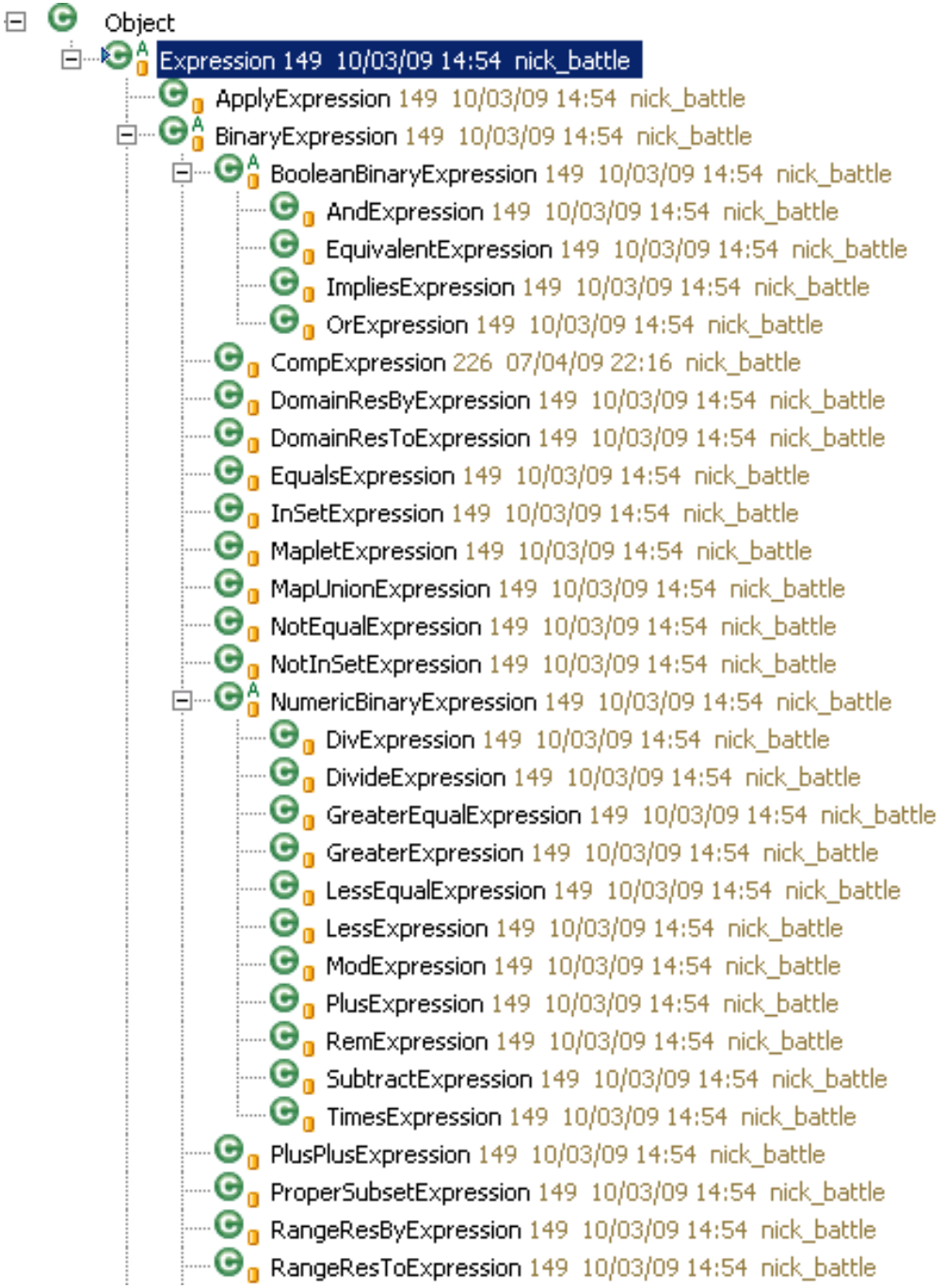
VDMJ AST 2


Object
Definition 149 10/03/09 14:54 nick_battle
AssignmentDefinition 149 10/03/09 14:54 nick_battle
InstanceVariableDefinition 206 30/03/09 15:03 nick_battle
ClassDefinition 232 08/04/09 12:45 nick_battle
BUSClassDefinition 167 18/03/09 15:36 nick_battle
CPUClassDefinition 149 10/03/09 14:54 nick_battle
ClassInvariantDefinition 149 10/03/09 14:54 nick_battle
EqualsDefinition 149 10/03/09 14:54 nick_battle
ExplicitFunctionDefinition 238 09/04/09 22:32 nick_battle
ExplicitOperationDefinition 149 10/03/09 14:54 nick_battle
ExternalDefinition 149 10/03/09 14:54 nick_battle
ImplicitFunctionDefinition 238 09/04/09 22:32 nick_battle
ImplicitOperationDefinition 149 10/03/09 14:54 nick_battle
ImportedDefinition 149 10/03/09 14:54 nick_battle
InheritedDefinition 149 10/03/09 14:54 nick_battle
LocalDefinition 149 10/03/09 14:54 nick_battle
MultiBindListDefinition 149 10/03/09 14:54 nick_battle
MutexSyncDefinition 149 10/03/09 14:54 nick_battle
NamedTraceDefinition 149 10/03/09 14:54 nick_battle
PerSyncDefinition 149 10/03/09 14:54 nick_battle
RenamedDefinition 149 10/03/09 14:54 nick_battle
StateDefinition 149 10/03/09 14:54 nick_battle
ThreadDefinition 149 10/03/09 14:54 nick_battle
TypeDefinition 149 10/03/09 14:54 nick_battle
UntypedDefinition 149 10/03/09 14:54 nick_battle
ValueDefinition 149 10/03/09 14:54 nick_battle

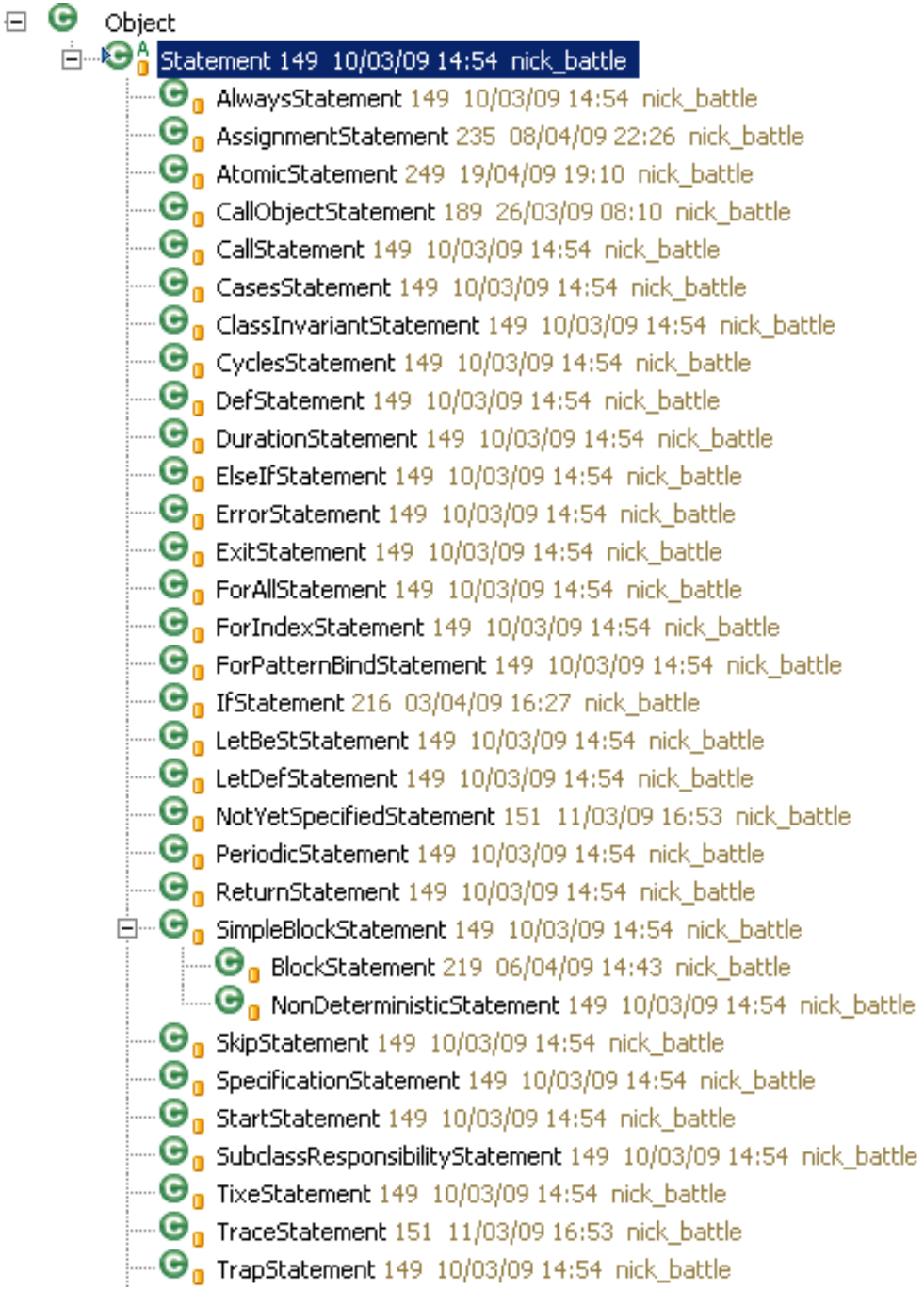
VDMJ AST 3


Object
Type 149 10/03/09 14:54 nick_battle
BasicType 149 10/03/09 14:54 nick_battle
BooleanType 149 10/03/09 14:54 nick_battle
CharacterType 149 10/03/09 14:54 nick_battle
NumericType 149 10/03/09 14:54 nick_battle
IntegerType 149 10/03/09 14:54 nick_battle
NaturalOneType 149 10/03/09 14:54 nick_battle
NaturalType 149 10/03/09 14:54 nick_battle
RationalType 149 10/03/09 14:54 nick_battle
RealType 149 10/03/09 14:54 nick_battle
TokenType 149 10/03/09 14:54 nick_battle
BracketType 207 30/03/09 15:31 nick_battle
ClassType 149 10/03/09 14:54 nick_battle
FunctionType 207 30/03/09 15:31 nick_battle
InvariantType 149 10/03/09 14:54 nick_battle
NamedType 207 30/03/09 15:31 nick_battle
RecordType 207 30/03/09 15:31 nick_battle
MapType 207 30/03/09 15:31 nick_battle
OperationType 207 30/03/09 15:31 nick_battle
OptionalType 207 30/03/09 15:31 nick_battle
ParameterType 149 10/03/09 14:54 nick_battle
ProductType 207 30/03/09 15:31 nick_battle
QuoteType 149 10/03/09 14:54 nick_battle
SeqType 207 30/03/09 15:31 nick_battle
SetType 207 30/03/09 15:31 nick_battle
UndefinedType 149 10/03/09 14:54 nick_battle
UnionType 207 30/03/09 15:31 nick_battle
UnknownType 149 10/03/09 14:54 nick_battle
UnresolvedType 235 08/04/09 22:26 nick_battle
VoidReturnType 149 10/03/09 14:54 nick_battle
VoidType 149 10/03/09 14:54 nick_battle

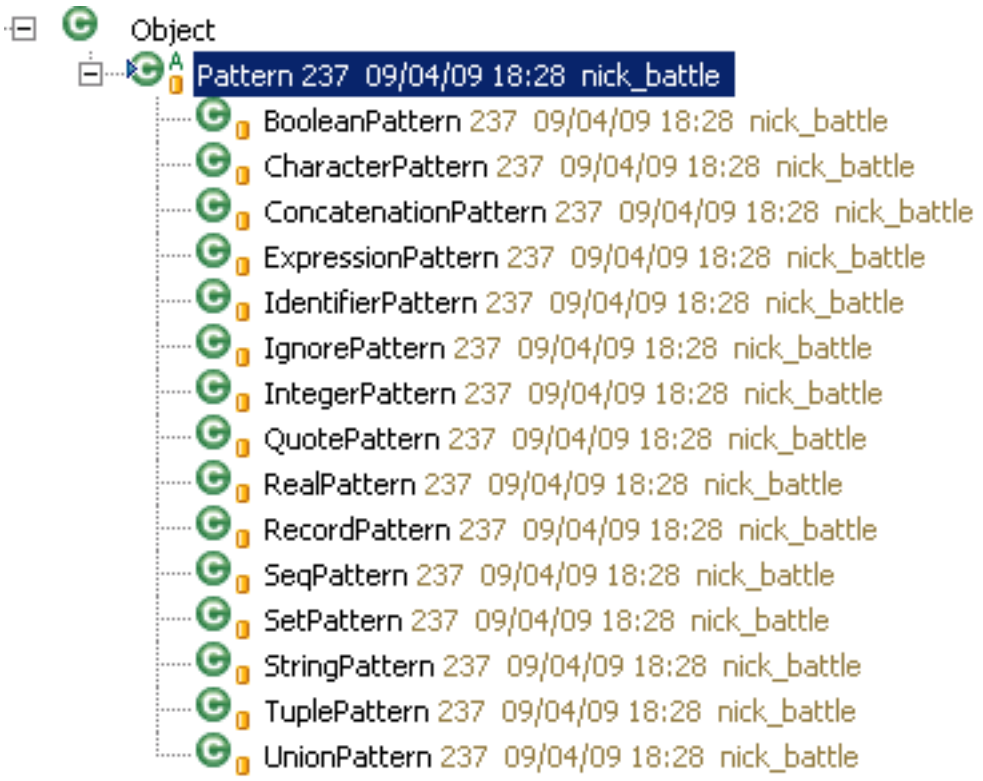
VDMJ AST 4


Object
Expression 149 10/03/09 14:54 nick_battle
ApplyExpression 149 10/03/09 14:54 nick_battle
BinaryExpression 149 10/03/09 14:54 nick_battle
BooleanBinaryExpression 149 10/03/09 14:54 nick_battle
AndExpression 149 10/03/09 14:54 nick_battle
EquivalentExpression 149 10/03/09 14:54 nick_battle
ImpliesExpression 149 10/03/09 14:54 nick_battle
OrExpression 149 10/03/09 14:54 nick_battle
CompExpression 226 07/04/09 22:16 nick_battle
DomainResByExpression 149 10/03/09 14:54 nick_battle
DomainResToExpression 149 10/03/09 14:54 nick_battle
EqualsExpression 149 10/03/09 14:54 nick_battle
InSetExpression 149 10/03/09 14:54 nick_battle
MapletExpression 149 10/03/09 14:54 nick_battle
MapUnionExpression 149 10/03/09 14:54 nick_battle
NotEqualExpression 149 10/03/09 14:54 nick_battle
NotInSetExpression 149 10/03/09 14:54 nick_battle
NumericBinaryExpression 149 10/03/09 14:54 nick_battle
DivExpression 149 10/03/09 14:54 nick_battle
DivideExpression 149 10/03/09 14:54 nick_battle
GreaterEqualExpression 149 10/03/09 14:54 nick_battle
GreaterExpression 149 10/03/09 14:54 nick_battle
LessEqualExpression 149 10/03/09 14:54 nick_battle
LessExpression 149 10/03/09 14:54 nick_battle
ModExpression 149 10/03/09 14:54 nick_battle
PlusExpression 149 10/03/09 14:54 nick_battle
RemExpression 149 10/03/09 14:54 nick_battle
SubtractExpression 149 10/03/09 14:54 nick_battle
TimesExpression 149 10/03/09 14:54 nick_battle
PlusPlusExpression 149 10/03/09 14:54 nick_battle
ProperSubsetExpression 149 10/03/09 14:54 nick_battle
RangeResByExpression 149 10/03/09 14:54 nick_battle
RangeResToExpression 149 10/03/09 14:54 nick_battle

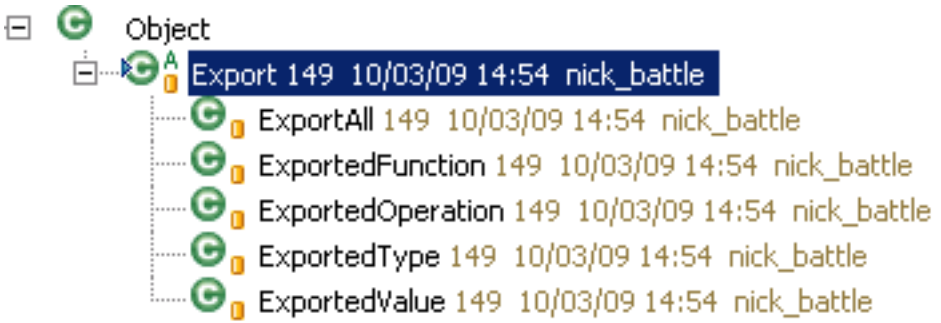
VDMJ AST 5

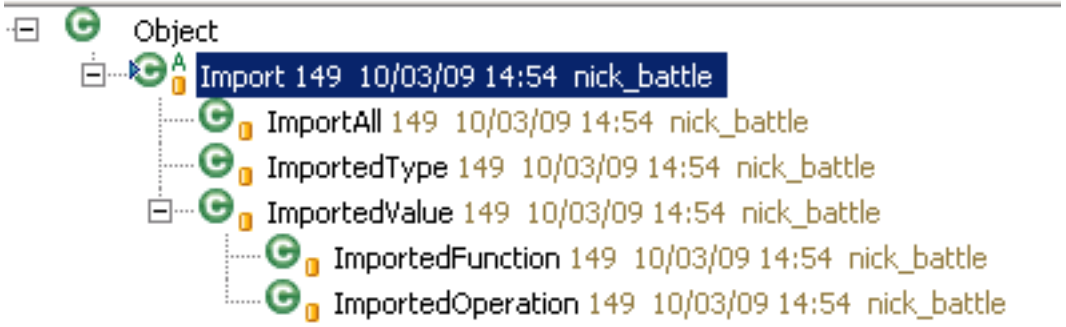

Object
Statement 149 10/03/09 14:54 nick_battle
AlwaysStatement 149 10/03/09 14:54 nick_battle
AssignmentStatement 235 08/04/09 22:26 nick_battle
AtomicStatement 249 19/04/09 19:10 nick_battle
CallObjectStatement 189 26/03/09 08:10 nick_battle
CallStatement 149 10/03/09 14:54 nick_battle
CasesStatement 149 10/03/09 14:54 nick_battle
ClassInvariantStatement 149 10/03/09 14:54 nick_battle
CyclesStatement 149 10/03/09 14:54 nick_battle
DefStatement 149 10/03/09 14:54 nick_battle
DurationStatement 149 10/03/09 14:54 nick_battle
ElseIfStatement 149 10/03/09 14:54 nick_battle
ErrorStatement 149 10/03/09 14:54 nick_battle
ExitStatement 149 10/03/09 14:54 nick_battle
ForAllStatement 149 10/03/09 14:54 nick_battle
ForIndexStatement 149 10/03/09 14:54 nick_battle
ForPatternBindStatement 149 10/03/09 14:54 nick_battle
IfStatement 216 03/04/09 16:27 nick_battle
LetBeStStatement 149 10/03/09 14:54 nick_battle
LetDefStatement 149 10/03/09 14:54 nick_battle
NotYetSpecifiedStatement 151 11/03/09 16:53 nick_battle
PeriodicStatement 149 10/03/09 14:54 nick_battle
ReturnStatement 149 10/03/09 14:54 nick_battle
SimpleBlockStatement 149 10/03/09 14:54 nick_battle
BlockStatement 219 06/04/09 14:43 nick_battle
NonDeterministicStatement 149 10/03/09 14:54 nick_battle
SkipStatement 149 10/03/09 14:54 nick_battle
SpecificationStatement 149 10/03/09 14:54 nick_battle
StartStatement 149 10/03/09 14:54 nick_battle
SubclassResponsibilityStatement 149 10/03/09 14:54 nick_battle
TixeStatement 149 10/03/09 14:54 nick_battle
TraceStatement 151 11/03/09 16:53 nick_battle
TrapStatement 149 10/03/09 14:54 nick_battle

VDMJ AST 6


Object
Pattern 237 09/04/09 18:28 nick_battle
BooleanPattern 237 09/04/09 18:28 nick_battle
CharacterPattern 237 09/04/09 18:28 nick_battle
ConcatenationPattern 237 09/04/09 18:28 nick_battle
ExpressionPattern 237 09/04/09 18:28 nick_battle
IdentifierPattern 237 09/04/09 18:28 nick_battle
IgnorePattern 237 09/04/09 18:28 nick_battle
IntegerPattern 237 09/04/09 18:28 nick_battle
QuotePattern 237 09/04/09 18:28 nick_battle
RealPattern 237 09/04/09 18:28 nick_battle
RecordPattern 237 09/04/09 18:28 nick_battle
SeqPattern 237 09/04/09 18:28 nick_battle
SetPattern 237 09/04/09 18:28 nick_battle
StringPattern 237 09/04/09 18:28 nick_battle
TuplePattern 237 09/04/09 18:28 nick_battle
UnionPattern 237 09/04/09 18:28 nick_battle

VDMJ AST 7

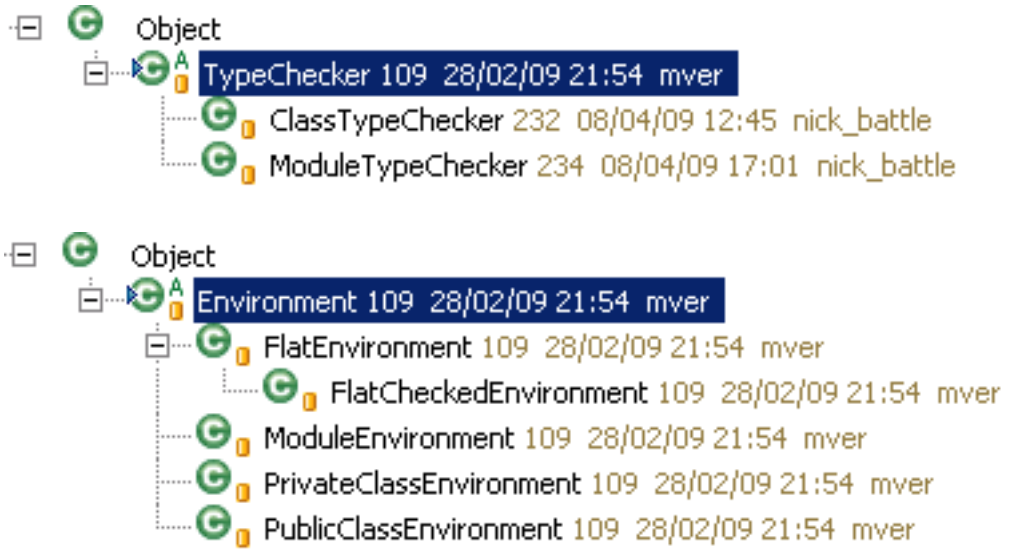

Object
Export 149 10/03/09 14:54 nick_battle
ExportAll 149 10/03/09 14:54 nick_battle
ExportedFunction 149 10/03/09 14:54 nick_battle
ExportedOperation 149 10/03/09 14:54 nick_battle
ExportedType 149 10/03/09 14:54 nick_battle
ExportedValue 149 10/03/09 14:54 nick_battle


Object
Import 149 10/03/09 14:54 nick_battle
ImportAll 149 10/03/09 14:54 nick_battle
ImportedType 149 10/03/09 14:54 nick_battle
ImportedValue 149 10/03/09 14:54 nick_battle
ImportedFunction 149 10/03/09 14:54 nick_battle
ImportedOperation 149 10/03/09 14:54 nick_battle

VDMJ Type Checking 1
Principles

- VDM syntax allows "3 + true", but type checking does not
- Concerned with types, which are ultimately defined by the types of literals and named definitions, and their combination via expressions, statements and further function/operation definitions.
- Also concerned with the scope of the names of definitions, the duplication of names (error) or when local named definitions are unused or hide outer names (warning).
- Syntax does not link type names to their definitions, so "x:T = 123" has an UnresolvedType called "T", but this may not even exist.
- Note that functions cannot see "state" definitions, and operation post conditions can see "old" variable values
- Classes are types (hence ClassDefinition), but modules are not.

VDMJ Type Checking 2
Principles



- Type checking is coordinated by subclasses of TypeCheck. The actual business of checking is performed by the AST elements
- An Environment refers to a list of definitions and allows them to be searched by name
- Environments chain together to add definitions to the current "scope" temporarily, such as parameters inside a function or new definitions from a let expression

VDMJ Type Checking 3
ModuleTypeChecker outline

- Check for duplicate module names in the list passed
- For each module, generate its definitions' implicit definitions (like pre and post functions)
- For each module, check the export definitions exist and are of the declared type, and make a list of exported definitions for the module.
- For each module, go through the import definitions and resolve against the exports.
- Create a list of all definitions from all modules (including their imports), create an Environment that contains them all, and attempt to perform type resolution on them – ie. find the type definition for every named type.
- In the pass order: [types, values, definitions], for each module, create a ModuleEnvironment representing the visible definitions, and type check the definitions of the given pass.
- Report any discrepancies between the final checked types of the modules' definitions and their explicit imported types elsewhere.
- Any definition names that have not been referenced or exported produce "unused" warnings.

VDMJ Type Checking 4
ClassTypeChecker outline

- Make sure there are no duplicate class definitions.
- For all classes and their definitions, generate the implicit definitions. This includes the construction of the class type hierarchy and the implicit local names for access to inherited definitions.
- Create a PublicClassEnvironment that can see all public class definitions.
- For each class, chain a PrivateClassEnvironment to the public environment, and perform type resolution on the definitions in the class.
- For each class, check for overloading and overriding of its definitions.
- In the pass order: [types, values, definitions], for each class, chain a PrivateClassEnvironment to the public environment, and type check the definitions of the given pass.
- Check for any definition names that have not been referenced, and produce "unused" warnings.

VDMJ Type Checking 5
Key Methods

```
public abstract class Definition implements Serializable
{
	public void typeResolve(Environment env) ...
	public void implicitDefinitions(Environment base) ...
	abstract public void typeCheck(Environment base, NameScope scope);
	public void unusedCheck() ...

public abstract class Expression implements Serializable
{
	abstract public Type typeCheck(Environment env, TypeList qualifiers, NameScope scope);


public abstract class Statement implements Serializable
{
	abstract public Type typeCheck(Environment env, NameScope scope);
```

- Type checking methods on Definition are all void, whereas those on Expression and Statement return the type of their content
- Type resolution is only performed on Definitions, not Expressions and Statements. Resolution of their types is delayed until their type check
- Expression's typeCheck has a list of "qualifier" types, used to resolve overloading with function apply expressions
- NameScope indicates whether state etc. is in scope or not

VDMJ Type Checking 6
TypeComparator

```
/**
 * A class for static type checking comparisons.
 */

public class TypeComparator
{
	public synchronized static boolean compatible(Type to, Type from)
	public synchronized static boolean compatible(TypeList to, TypeList from)
	public synchronized static boolean isSubType(Type sub, Type sup)
```

- TypeComparator is used to check whether one type is assignment compatible with another, and whether one type is a subtype of another
- VDMJ always does type checking with "possible semantics"
- The "compatible" method provides possible semantics for type conversion, eg. a real is possibly an int
- The "isSubType" method provides definite semantics for type comparison, eg. a real is not a subtype of int. This is used in PO generation

VDMJ Type Checking 7
Checking Functions (1)

- If there are any polymorphic type parameters for this function, check that the overall function type does not reference any type parameters except those named type parameters.
- For each type parameter, create a LocalDefinition of a ParameterType and add this to a local Environment.
- Check that the parameter patterns match the overall Type's parameters, and iterate through curried sets of parameters, using the return value from the overall Type (and its return value and so on for subsequent sets of parameters). Remember the expected result.
- Extend the local Environment with definitions for all the variables of all the patterns from all of the curried parameter sets.
- Type check the definitions this produced in the base environment (this will just do type resolution, if necessary).
- Label the local Environment as static (VDM++) if the definition's access specifier is static.
- If we are in VDM++ and the function is not static, add a "self" definition to the local Environment.
- *continued...*

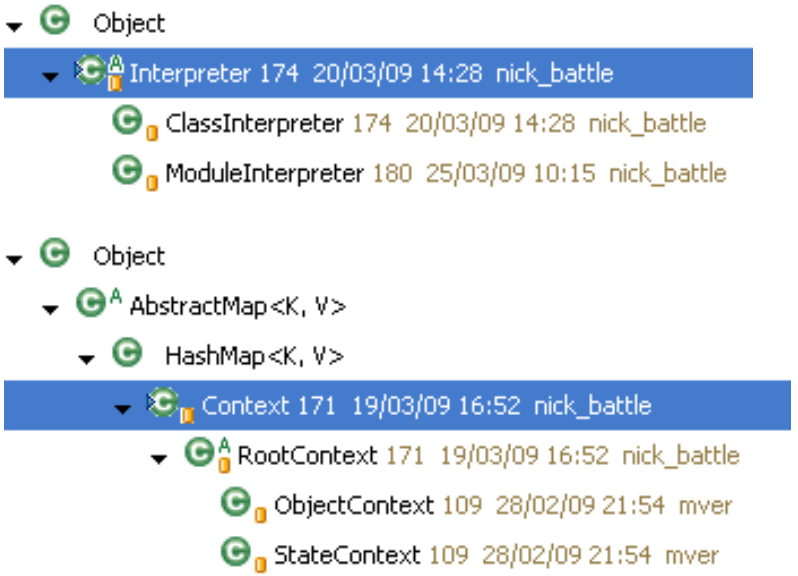
VDMJ Type Checking 8
Checking Functions (2)

- If there is a precondition expression, type check the definition for it.
- If there is a post condition expression, type check the definition for that too.
- Type check the body expression of the function, remembering the actual type returned.
- If the actual return type is not (possibly) assignable to the expected return type, raise an error.
- If the VDM++ accessibility of the expected return type is narrower than that of the definition itself, raise an error (eg. a public function cannot have a private or protected return type).
- If the function is recursive and does not define a "measure" function, raise a warning, else if there is a measure defined, check that it exists and has the correct type.
- Check that the parameter variables have been referenced in the local Environment, else raise an unused parameter warning. (This is suppressed for pre and post condition functions, which are permitted to not necessarily use their implicit parameters).

VDMJ Interpreter 1

- Interpreter evaluates expressions from their definitions in the AST
- Acts as the common “interface” for external user interaction classes (command line or IDE).
- Uses a default module or class. Used for simplified naming. Defaults to the first module/class on the AST list
- Contains a list of breakpoints, and methods to set/clear/list them, as well as to locate expressions/statements in AST by file/line number
- Contains a list of source file contents for source debugging
- Also acts as the initiator for PO generation, though POs are generated in the AST elements.
- Initialization method creates or restores an “initial context” representing the global static context for all subsequent evaluations
- Can evaluate expressions in the initial context, or in a local context when a breakpoint is reached

VDMJ Interpreter 2



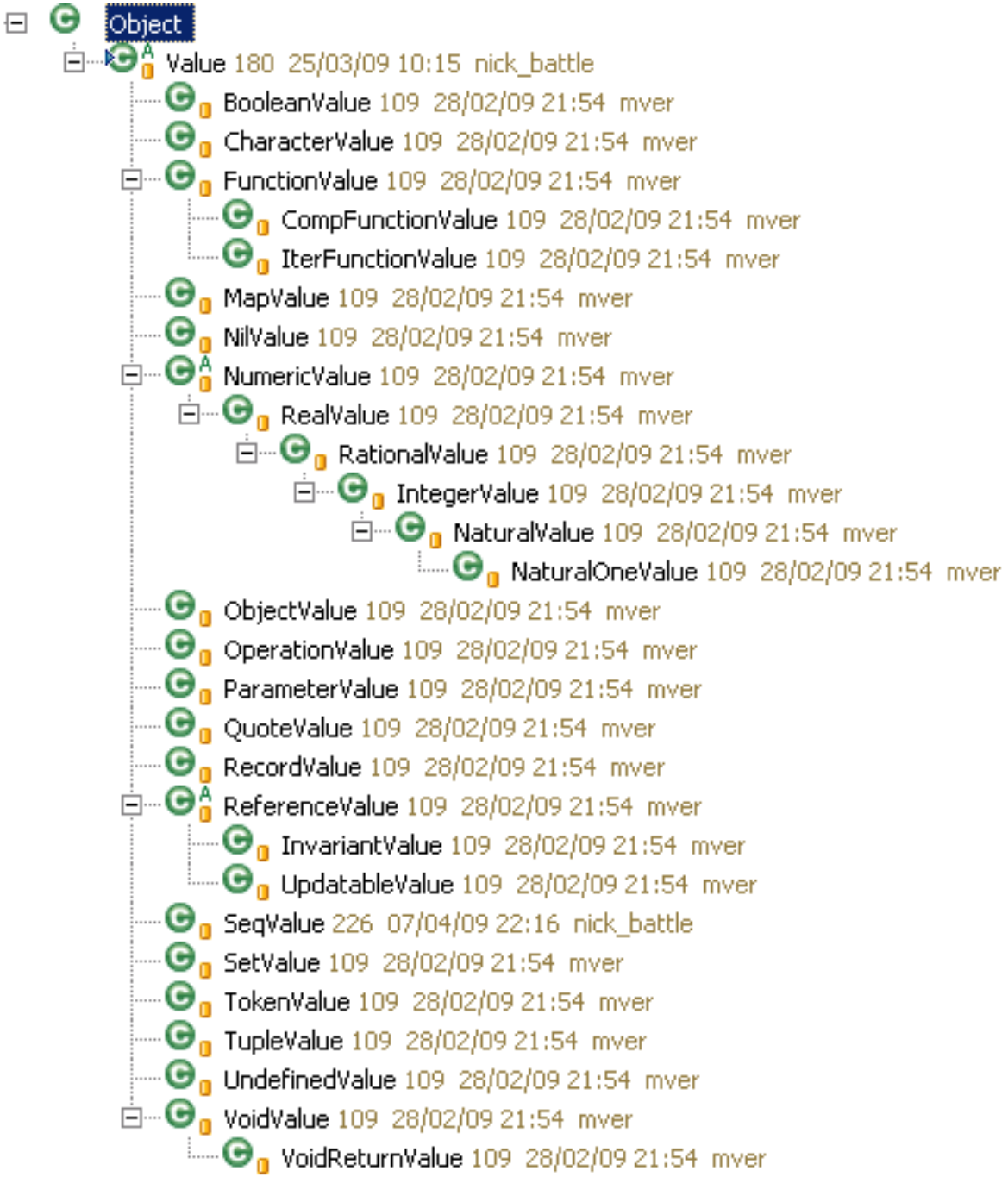
- Evaluation is coordinated by subclasses of Interpreter. The actual business of evaluation is performed by the AST elements
- Runtime name/value pairs held in Context subclasses which mirror the Environment subclasses used in type checking
- Contexts chain together to form the runtime stack. RootContext is the abstract base of a function/operation call. ObjectContext has a “self”, and StateContext points to any module state
- Global values (module state or class static fields) are held in ClassDefinition or Module definitions in AST

VDMJ Interpreter 3

```
class A
functions
    public f: int * int -> int
    f(a, b) ==
        new A().g(a + b);

    g: int -> int
    g(x) ==
        let r = x + 1, s = x - 1 in
            let a = r + s in
                a;

end A
```

```
[thread 1]> stack
Stopped at break [1] in 'A' (test.vpp) at line 11:17
        a = 6
In context of let expression in 'A' (test.vpp) at line 10:13
        r = 4
        s = 2
In context of let expression in 'A' (test.vpp) at line 9:9
        g(int) = (int -> int)
        x = 3
        self = A{#2}
In object context of g(x) in 'A' (test.vpp) at line 5:9
        self = A{#1}
        f(int, int) = (int * int -> int)
        b = 2
        a = 1
In object context of f(a, b) in 'A' (console) at line 1:1
In root context of global static scope
```

VDMJ Interpreter 4



VDMJ Interpreter 5
Key Methods

```
public abstract class Expression implements Serializable
{
	abstract public Value eval(Context ctxt);

public abstract class Statement implements Serializable
{
	abstract public Value eval(Context ctxt);
```

- Expression and Statement's eval methods return their Value, given the passed Context
- Evaluation recurses over the AST, evaluating and combining the Values of sub-expressions or contained Statements
- The Context is extended by expressions or statements which add variables (eg. let expressions)
- Evaluations can throw ContextExceptions – a runtime exception which includes the Context (stack) as well as a number, text and location

VDMJ Interpreter 6
TailExpression.eval

```
@Override
public Value eval(Context ctxt)
{
    breakpoint.check(location, ctxt);

    ValueList seq = null;

    try
    {
        seq = new ValueList(exp.eval(ctxt).seqValue(ctxt));
    }
    catch (ValueException e)
    {
        return abort(e);
    }

    if (seq.isEmpty())
    {
        abort(4033, "Tail sequence is empty", ctxt);
    }

    seq.remove(0);
    return new SeqValue(seq);
}
```

VDMJ Interpreter 7
BlockStatement.eval

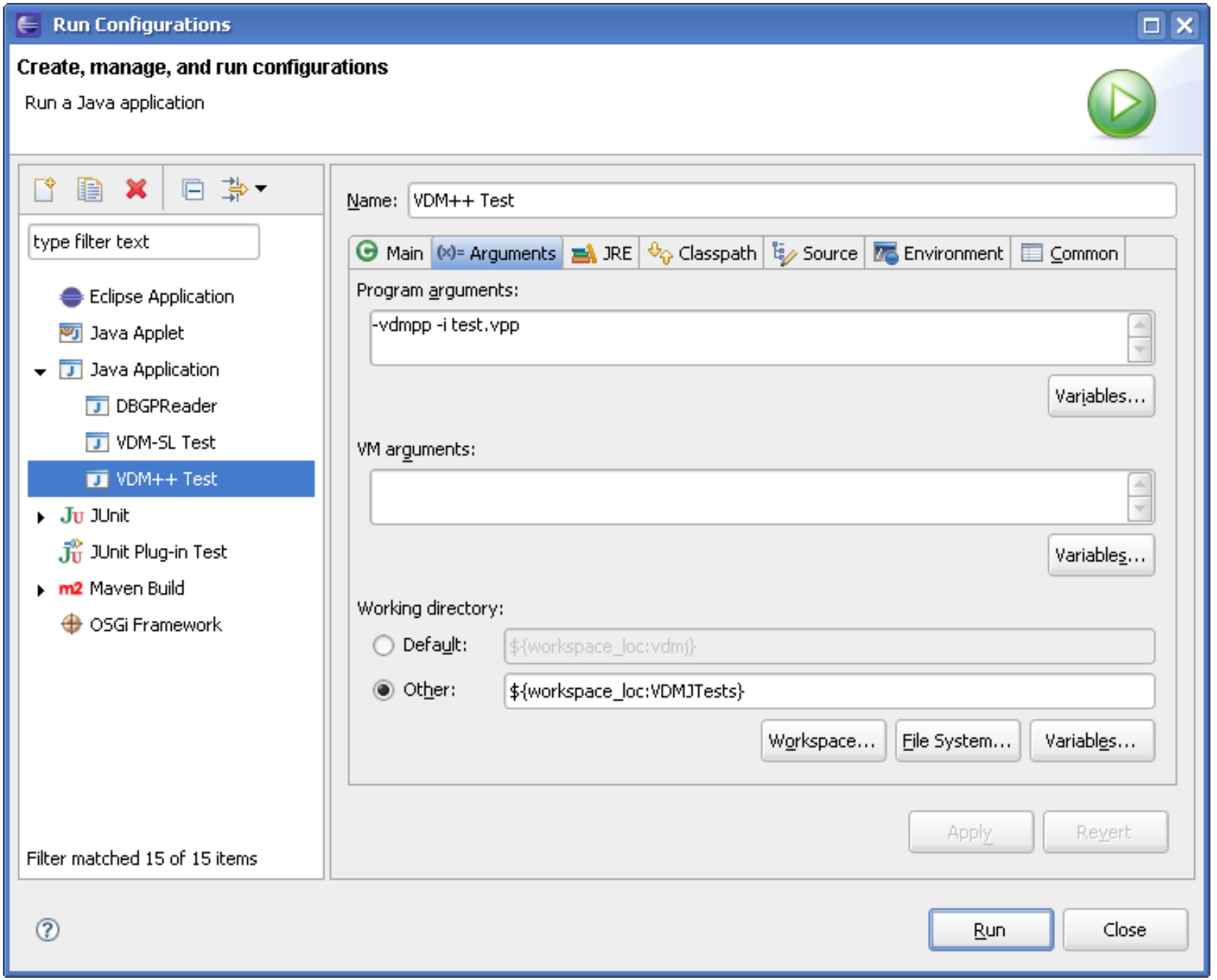
```
@Override
public Value eval(Context ctxt)
{
    breakpoint.check(location, ctxt);

    Context evalContext = new Context(location, "block statement", ctxt);

    for (Definition d: assignmentDefs)
    {
        evalContext.put(d.getNamedValues(evalContext));
    }

    for (Statement s: statements)
    {
        Value rv = s.eval(evalContext);

        if (!rv.isVoid())
        {
            return rv;
        }
    }

    return new VoidValue();
}
```

VDMJ Exercise 1

sequence for loop = ‘for’, pattern bind, ‘in’, [‘reverse’], expression, ‘do’, statement ;

- Currently, the "reverse" keyword is part of the sequence loop grammar
- It would be better to treat "reverse" as a new unary sequence operator, like hd, tl, len, elems, inds and conc.
- Grammar becomes: "for", pattern bind, "in", expression, "do", statement
- ReverseExpression.java is partly written in SVN. The VDMJ parser has been changed to build the AST already.
- Complete the eval method of ReverseExpression as an exercise – compare with TailExpression.java
- Test with "print rev [1,2,3]" – should give [3,2,1]. Check that the reverse for loop still works too. See whether you can set a breakpoint.
- The typeCheck method is also blank. Complete this as an exercise too – test with "rev 123". It should complain that the argument is not a sequence.
- What should the getProofObligations method look like?

VDMJ Exercise 2



VDMJ Exercise 3
Tests...

```
Interpreter started
> p rev 1234
Error 3292: Argument to 'rev' is not a sequence in 'A' (console) at line 1:1
> p rev [1,2,3,4]
= [4, 3, 2, 1]
Executed in 0.013 secs.
> p rev "hello"
= "olleh"
Executed in 0.0020 secs.
> p rev []
= []
Executed in 0.0010 secs.
>
```

VDMJ Exercise 4
Solutions...

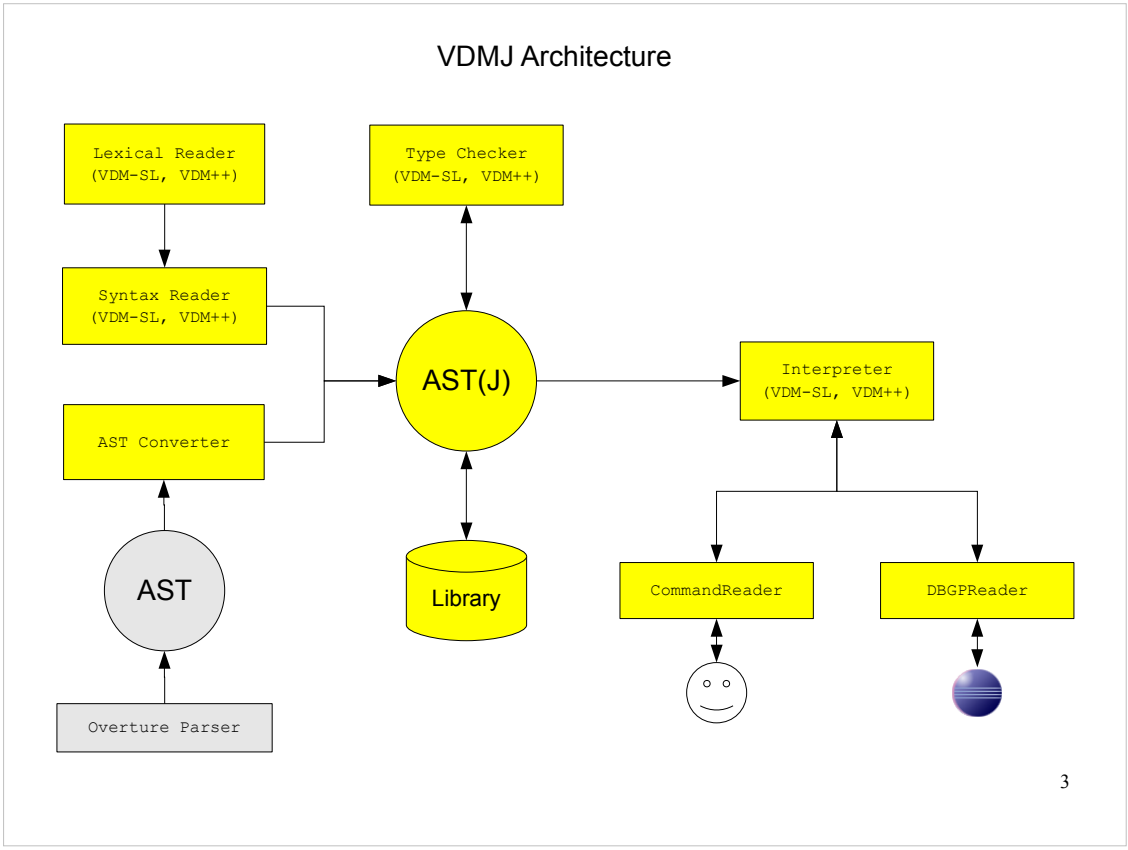
```
@Override
public Type typeCheck(Environment env, TypeList qualifiers, NameScope scope)
{
    Type etype = exp.typeCheck(env, null, scope);

    if (!etype.isSeq())
    {
        report(3292, "Argument to 'rev' is not a sequence");
        return new SeqType(location, new UnknownType(location));
    }

    return etype;
}

@Override
public Value eval(Context ctxt)
{
    breakpoint.check(location, ctxt);

    ValueList seq = null;

    try
    {
        seq = new ValueList(exp.eval(ctxt).seqValue(ctxt));
        Collections.reverse(seq);
    }
    catch (ValueException e)
    {
        return abort(e);
    }

    return new SeqValue(seq);
}
```

VDMJ

Nick Battle, Fujitsu
(nick.battle@uk.fujitsu.com)

VDMJ Overview

- Where did VDMJ come from...?
- Provides support for VDM-SL and VDM++ parsing, static type checking, interpreting/debugging, PO generation, test coverage and combinatorial testing
- Pure Java (5 or later), no external dependencies
- Released under GPLv3 by Fujitsu
- Command line interface only
- Informally developed (eg. not specified in VDM)
- User Guide and Design Specification docs available
- Passes CSK test suite (>3000 tests, converted to JUnit tests)
- Quite fast (3000 tests in ~20 seconds).



This is the overall architecture of VDMJ.

VDM source files are read by the Lexical Reader which uses a LaTeX filter to remove markup, and produces a stream of LexTokens. The reader is told which dialect it is reading (VDM-SL or VDM++, VICE is under development).

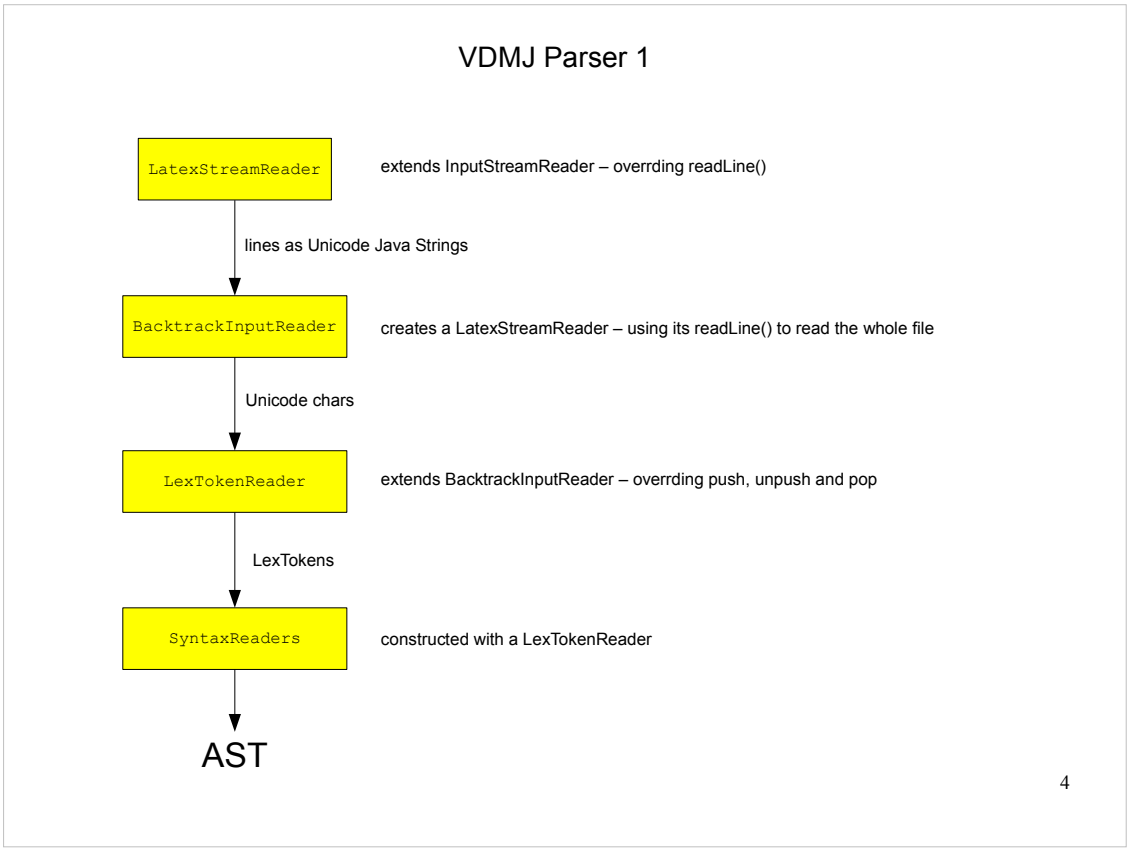
The LexTokens are used by the Syntax Reader to create the AST.

Alternatively, the Overture parser can be used to produce its own AST (different) and this can be translated to VDMJ's AST by an ASTConverter (fast).

The AST is examined by the Type Checker for static type errors, and enriched with extra type information. There are module and class specific checkers for checking the different dialects.

ASTs that have been type checked can be serialized and compressed, and saved to disk. Loading such ASTs again from disk is faster than re-parsing and re-checking them (at least for large specifications).

The Interpreter creates a runtime environment based on the classes or modules in the AST, then allows expressions to be evaluated in that environment. The Interpreter also allows the setting of breakpoints etc. The Interpreter is operated through the command line (console) or via the Xdebug remote debugging protocol.



These four class groups comprise the VDMJ parser.

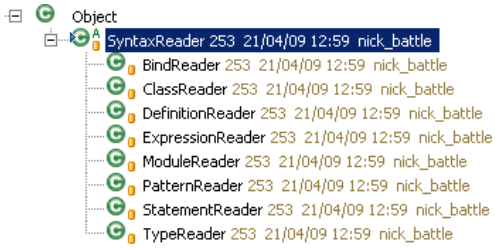
A LatexStreamReader extends InputStreamReader, replacing its readLine with a method that searches for LaTeX markup headers, turning such "non-content" lines into blank lines. This has the effect of preserving line number layout, but avoiding the syntax errors that LaTeX markup would otherwise produce.

A BacktrackInputReader is used to replace the InputReader's mark() and reset() methods with a more flexible stack of markers that can be pushed and popped. This allows arbitrary backtracking to occur which simplifies the token reader and syntax readers. To enable this, the entire source file is read in to memory (via the readLine of the LaTeX reader). The output is a sequence of Unicode Java chars.

The LexTokenReader extends BacktrackInputReader, replacing its push/pop methods with its own which push/pop the full lexical state – eg. including the current line number, character position and last token read. It provides two methods to the SyntaxReaders, in addition to push/pop, to read the next token from the stream or repeat the last token. The output is a stream of LexTokens.

Lastly, SyntaxReaders consume the LexTokens and build the AST. There are different SyntaxReader subclasses for each of the major syntactic groups in the grammar (definitions, expressions, statements, patterns, bindings and types), plus two top level readers for modules and classes respectively.

VDMJ Parser 2



- One SyntaxReader subclass for each major group in the grammar
- Top level ModuleReader and ClassReader combine the others and return the AST (class list or module list)
- Each reader typically has one public method, like readPattern(), readType(), readClass() etc. and many private methods.
- All readers are constructed by being passed a LexTokenReader

5

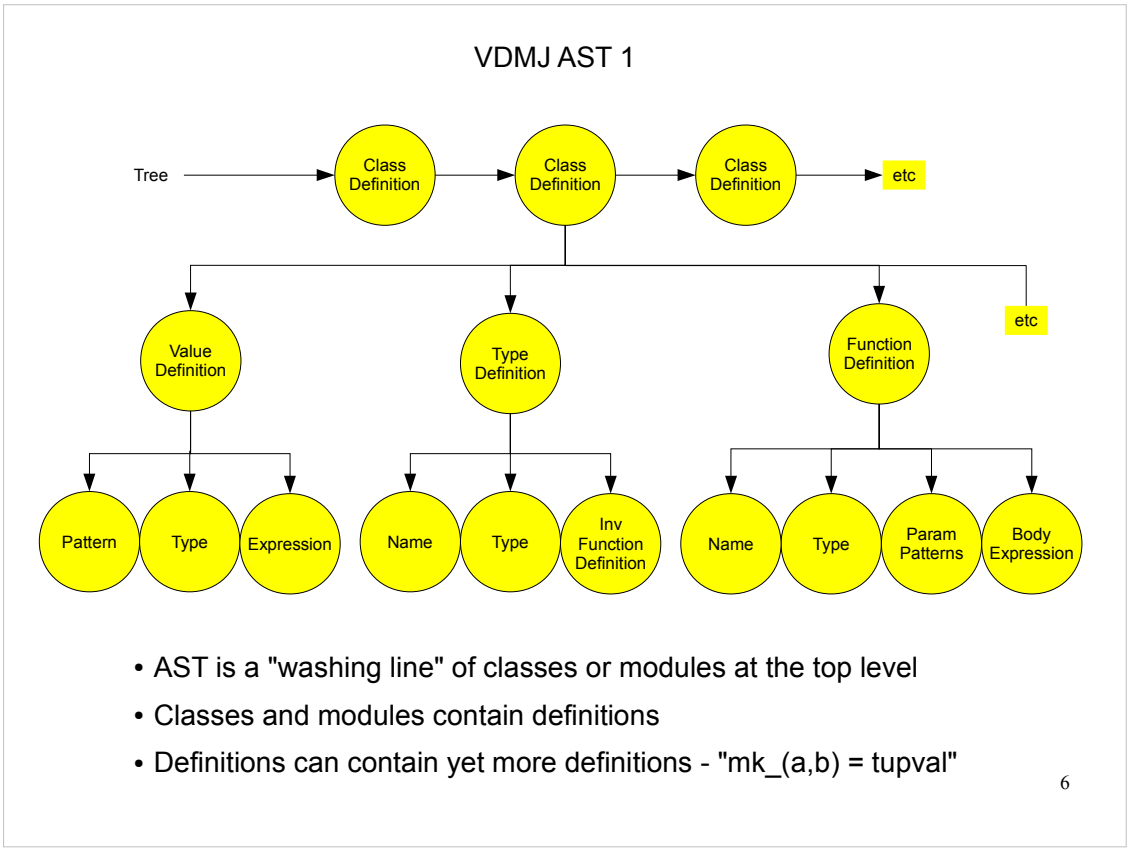
The SyntaxReaders comprise the syntax analyser of the VDMJ parser.

The readers use "recursive descent" parsing with backtracking, which means that the structure of the code closely resembles the structure of the grammar being parsed. This is intuitive and easy to debug, but laborious to write.

The readers create and attach new instances of other readers to their lexical stream when they need to read parts of the parse of a larger structure. For example, a DefinitionReader creates a TypeReader and PatternReader to parse the type/parameter signature of a function, and an ExpressionReader to parse the body expression, finally putting the parts together into an ExplicitFunctionDefinition object (part of the AST).

Syntax errors throw exceptions. The syntax recovery catches these exceptions and advances in the stream until a token in one of two lists is encountered: the first is a list of tokens which must be read up to and past; the second list is those which must be read up to and from which parsing will continue. For example, when parsing statements, immediately after a semi-colon would be a good place to recover, and at the next definition section ("functions" or "types" etc) would be a good place to recover from (ie. including that token).

In the case of grammar ambiguities, the parser uses backtracking to keep the structure of the code simple, at the cost of re-reading the source. For example, a lexical identifier at the start of a statement is either an assignment or an operation call. Assignments start with state designators and operation calls with object state designators (in VDM++), and the parser cannot tell which it is until more tokens have been read. To keep the parse of the two separate and clean, the statement parser "pushes" the position at the start; tries to parse one alternative, and if that fails tries to parse the other(s) after "popping" back.

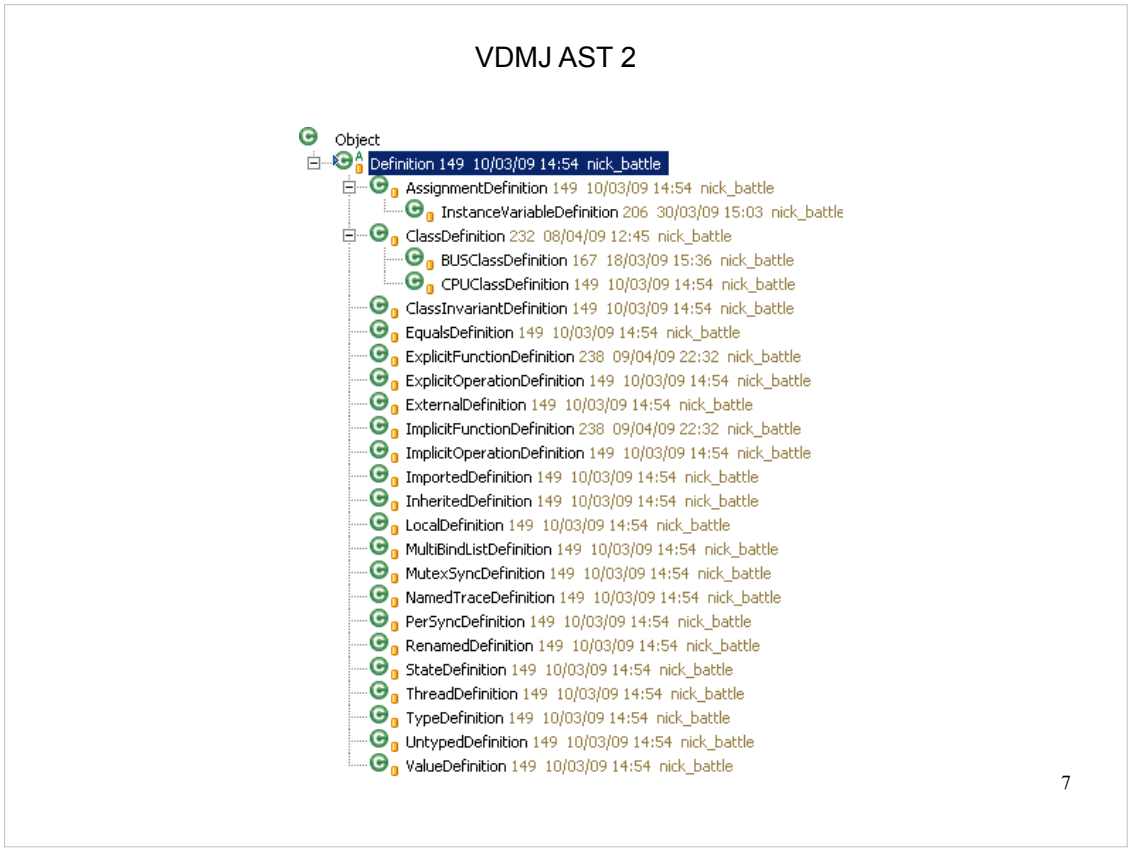


At the top level an AST is a list of class or module definitions, so it is more like a washing line than a "tree".

The top level definitions contain a list of definitions for the various subsections of the class or module (values, types, functions, operations, threads, sync, trace, instance variables).

Each definition contains a tree structure representing the parsed structure of the source file, including the patterns, types, binds, expressions, statements etc read by the various SyntaxReaders.

Definitions can contain definitions, for example because they include something that generates a function definition (eg. types with invariants generate a definition of inv_T with the body of that function being the expression parsed), or because they are defined with patterns that produce several named variables, such as a value definition defined as "mk_(a, b) = ..." which defines "a" and "b".



All AST definitions in VDMJ are subclasses of Definition.

Most of the subclass names relate directly to the names of the grammar for the corresponding definition types. Note that not all of them are top level grammar items, such as AssignmentDefinition, which appears in the "dcl" statement grammar, and is identical to the grammar for instance variable definitions. Similarly, EqualsDefinitions appear in "def" statements.

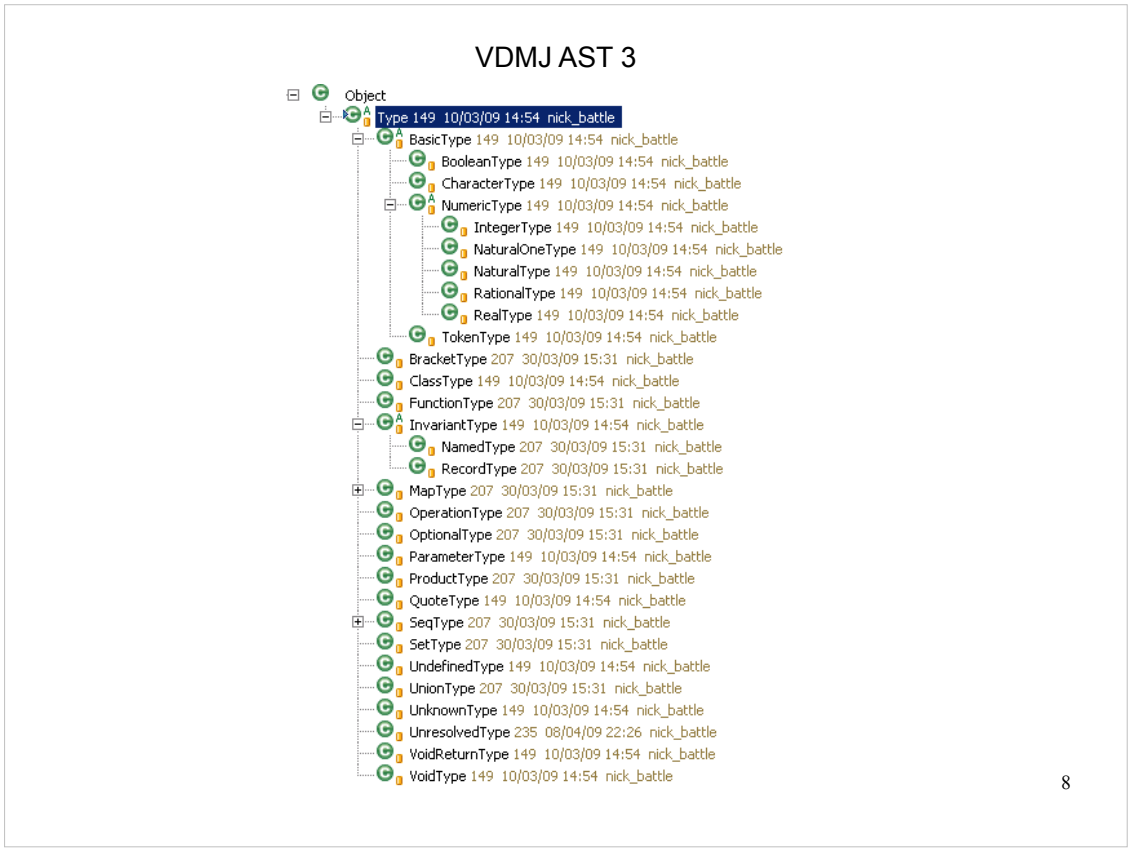
ExternalDefinitions are for the "ext" clauses that can be given with operation definitions.

ImportedDefinitions and RenamedDefinitions are used in module import clauses. They wrap another defintion, but indicate that the defintion is not within the module (or has been renamed within the module).

LocalDefinitions are used for local variables, such as those created from parameter patterns and "let" defintions. They just contain a name and a type.

MultiBindListDefinition is used for several grammatical constructs which use "bind lists", such as quantified forall and exists expressions.

UntypedDefinition is the odd one out. This is used as a placeholder when a value definition is given, but where there is no type information, such as "x = 1" rather than "x:nat = 1". The UntypedDefinition is subsequently replaced with a typed definition (usually a LocalDefinition) in the type checking phase.



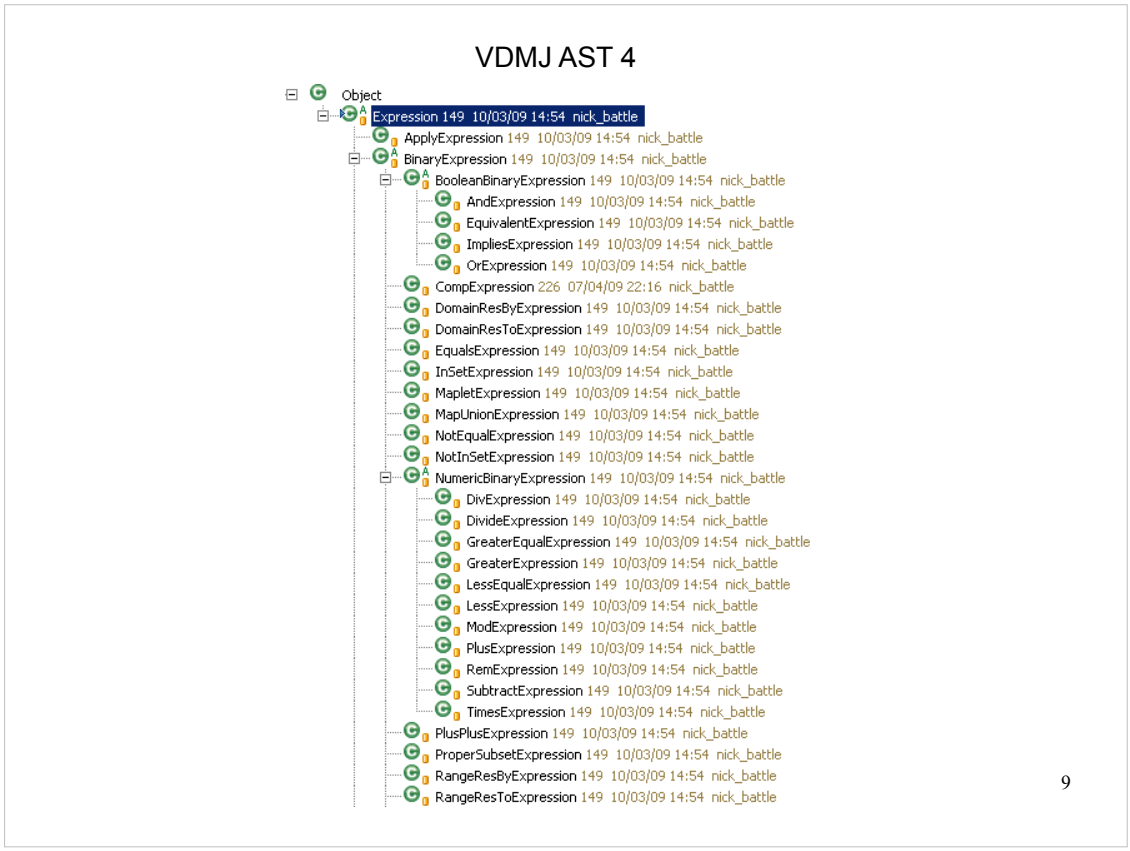
Types in AST are represented by subclasses of the Type class. Their names closely follow the names in the grammar.

Notice that the types which can have invariants (NamedTypes and RecordTypes) are distinguished in the Type hierarchy, whereas in the grammar the invariant is part of the TypeDefinition rather than the Type itself. This is for the VDMJ runtime, which needs access to the invariant, given only the type, when new values are being created.

VoidType means "no type" and is used to indicate that most statements don't return a value. The VoidReturnType is the type of the bare "return" statement – ie. this should return from the operation, but the return type is still void.

UnknownType is used during error handling. This type will pretend to be anything and tries to behave in a way that will not aggravate the type checker into producing a cascade of spurious errors due to a single cause.

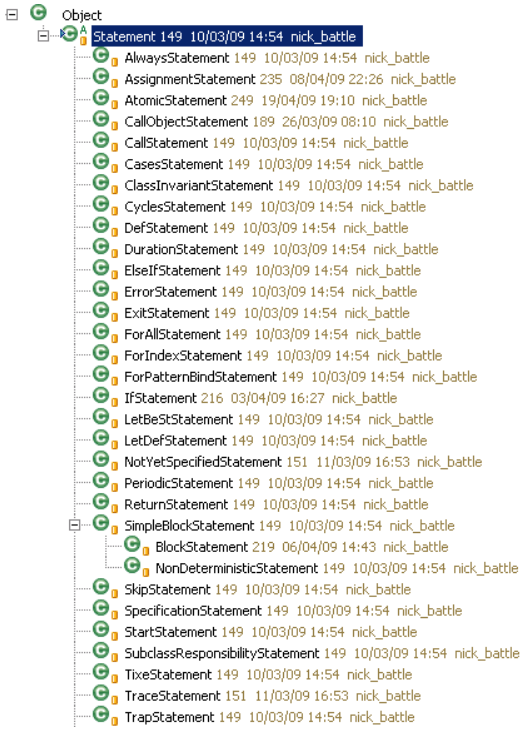
UnresolvedType is used to hold named types when then come from the syntax phase. They are replaced with real types in an early phase of type checking (called type resolution).



All AST expressions in VDMJ are subclasses of the Expression class.

Most Expressions relate directly to an expression in the grammar. A couple are manufactured in order to make the type checking and execution easier, such as PreOpExpression and PostOpExpression (which include a link to the module state definition, and are used exclusively for operations' pre and post expressions). Similarly StateInitExpression is used in the setup of module state (the body of the state's "init" clause, if any).

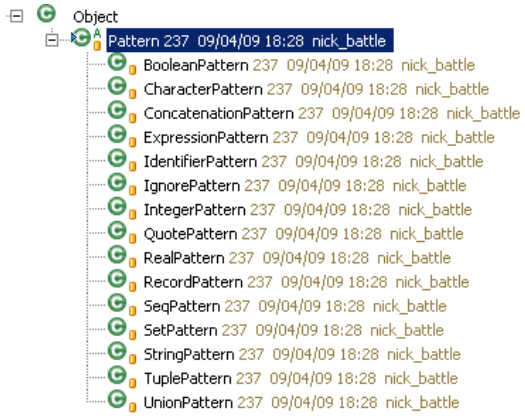
VDMJ AST 5



All AST statements in VDMJ are subclasses of Statement.

The class names closely follow the grammar. The hierarchy is less structured than the Expression hierarchy because generally statements are independent of each other. The only subclassing is with SimpleBlockStatements, which are just a sequence of statements, where the BlockStatement subclass can have additional DclStatements at the start, and a NonDeterministicStatement subclass is used to identify a block used in this way.

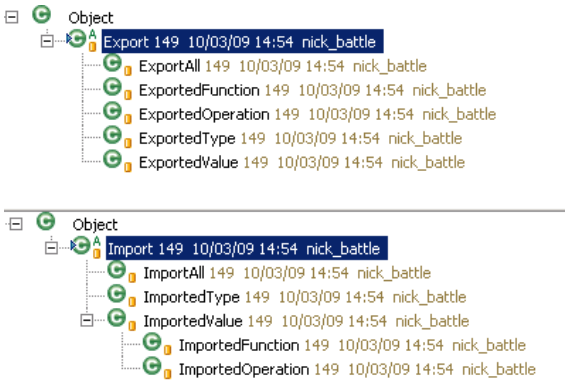
VDMJ AST 6



All AST patterns in VDMJ are subclasses of Pattern, and closely follow the grammar.

Note that a Pattern plus a Type is able to generate a set of typed Definitions for the identifiers it includes.

VDMJ AST 7

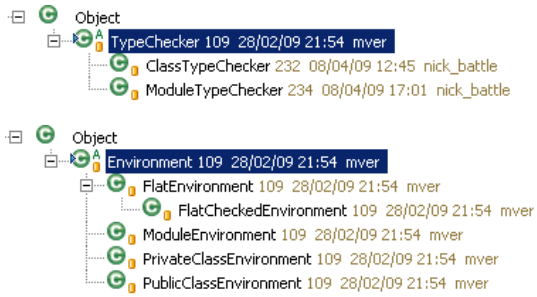


Modules differ from classes in AST in that they have import/export defintions in addition to a list of definitions.

VDMJ Type Checking 1
Principles

- VDM syntax allows "3 + true", but type checking does not
- Concerned with types, which are ultimately defined by the types of literals and named definitions, and their combination via expressions, statements and further function/operation definitions.
- Also concerned with the scope of the names of definitions, the duplication of names (error) or when local named definitions are unused or hide outer names (warning).
- Syntax does not link type names to their definitions, so "x:T = 123" has an UnresolvedType called "T", but this may not even exist.
- Note that functions cannot see "state" definitions, and operation post conditions can see "old" variable values
- Classes are types (hence ClassDefinition), but modules are not.

VDMJ Type Checking 2
Principles



- Type checking is coordinated by subclasses of TypeCheck. The actual business of checking is performed by the AST elements
- An Environment refers to a list of definitions and allows them to be searched by name
- Environments chain together to add definitions to the current "scope" temporarily, such as parameters inside a function or new definitions from a let expression

VDMJ Type Checking 3
ModuleTypeChecker outline

- Check for duplicate module names in the list passed
- For each module, generate its definitions' implicit definitions (like pre and post functions)
- For each module, check the export definitions exist and are of the declared type, and make a list of exported definitions for the module.
- For each module, go through the import definitions and resolve against the exports.
- Create a list of all definitions from all modules (including their imports), create an Environment that contains them all, and attempt to perform type resolution on them – ie. find the type definition for every named type.
- In the pass order: [types, values, definitions], for each module, create a ModuleEnvironment representing the visible definitions, and type check the definitions of the given pass.
- Report any discrepancies between the final checked types of the modules' definitions and their explicit imported types elsewhere.
- Any definition names that have not been referenced or exported produce "unused" warnings.

VDMJ Type Checking 4
ClassTypeChecker outline

- Make sure there are no duplicate class definitions.
- For all classes and their definitions, generate the implicit definitions. This includes the construction of the class type hierarchy and the implicit local names for access to inherited definitions.
- Create a PublicClassEnvironment that can see all public class definitions.
- For each class, chain a PrivateClassEnvironment to the public environment, and perform type resolution on the definitions in the class.
- For each class, check for overloading and overriding of its definitions.
- In the pass order: [types, values, definitions], for each class, chain a PrivateClassEnvironment to the public environment, and type check the definitions of the given pass.
- Check for any definition names that have not been referenced, and produce "unused" warnings.

VDMJ Type Checking 5
Key Methods

```
public abstract class Definition implements Serializable
{
      public void typeResolve(Environment env) ...
      public void implicitDefinitions(Environment base) ...
      abstract public void typeCheck(Environment base, NameScope scope);
      public void unusedCheck() ...

public abstract class Expression implements Serializable
{
      abstract public Type typeCheck(Environment env, TypeList qualifiers, NameScope scope);


public abstract class Statement implements Serializable
{
      abstract public Type typeCheck(Environment env, NameScope scope);
```

- Type checking methods on Definition are all void, whereas those on Expression and Statement return the type of their content
- Type resolution is only performed on Definitions, not Expressions and Statements. Resolution of their types is delayed until their type check
- Expression's typeCheck has a list of "qualifier" types, used to resolve overloading with function apply expressions
- NameScope indicates whether state etc. is in scope or not

VDMJ Type Checking 6
TypeComparator

```
/**
 * A class for static type checking comparisons.
 */

public class TypeComparator
{
      public synchronized static boolean compatible(Type to, Type from)
      public synchronized static boolean compatible(TypeList to, TypeList from)
      public synchronized static boolean isSubType(Type sub, Type sup)
```

- TypeComparator is used to check whether one type is assignment compatible with another, and whether one type is a subtype of another
- VDMJ always does type checking with "possible semantics"
- The "compatible" method provides possible semantics for type conversion, eg. a real is possibly an int
- The "isSubType" method provides definite semantics for type comparison, eg. a real is not a subtype of int. This is used in PO generation

VDMJ Type Checking 7
Checking Functions (1)

- If there are any polymorphic type parameters for this function, check that the overall function type does not reference any type parameters except those named type parameters.
- For each type parameter, create a LocalDefinition of a ParameterType and add this to a local Environment.
- Check that the parameter patterns match the overall Type's parameters, and iterate through curried sets of parameters, using the return value from the overall Type (and its return value and so on for subsequent sets of parameters). Remember the expected result.
- Extend the local Environment with definitions for all the variables of all the patterns from all of the curried parameter sets.
- Type check the definitions this produced in the base environment (this will just do type resolution, if necessary).
- Label the local Environment as static (VDM++) if the definition's access specifier is static.
- If we are in VDM++ and the function is not static, add a "self" definition to the local Environment.
- *continued...*

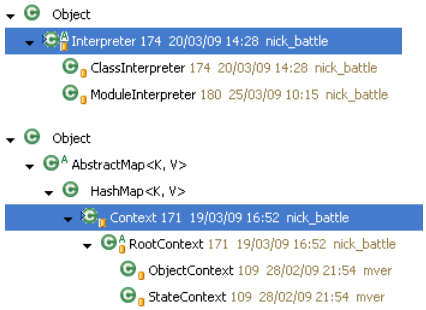
VDMJ Type Checking 8
Checking Functions (2)

- If there is a precondition expression, type check the definition for it.
- If there is a post condition expression, type check the definition for that too.
- Type check the body expression of the function, remembering the actual type returned.
- If the actual return type is not (possibly) assignable to the expected return type, raise an error.
- If the VDM++ accessibility of the expected return type is narrower than that of the definition itself, raise an error (eg. a public function cannot have a private or protected return type).
- If the function is recursive and does not define a "measure" function, raise a warning, else if there is a measure defined, check that it exists and has the correct type.
- Check that the parameter variables have been referenced in the local Environment, else raise an unused parameter warning. (This is suppressed for pre and post condition functions, which are permitted to not necessarily use their implicit parameters).

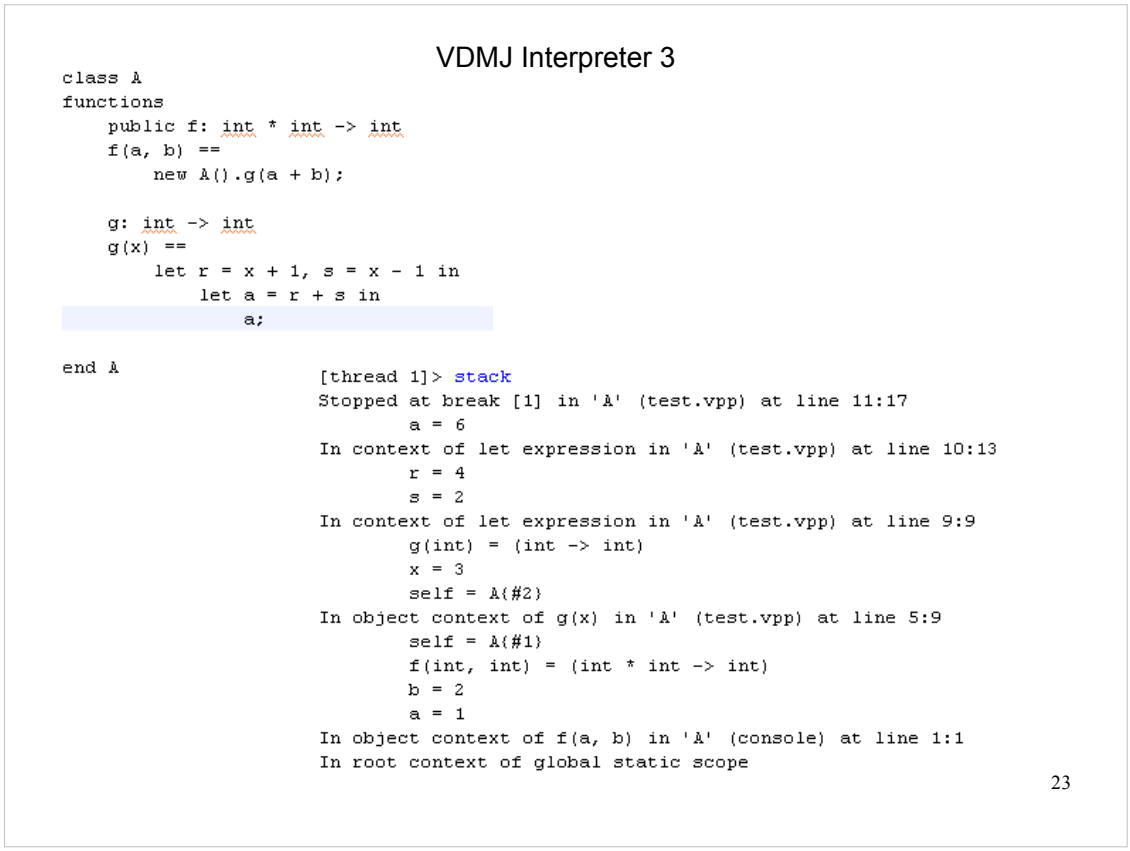
VDMJ Interpreter 1

- Interpreter evaluates expressions from their definitions in the AST
- Acts as the common “interface” for external user interaction classes (command line or IDE).
- Uses a default module or class. Used for simplified naming. Defaults to the first module/class on the AST list
- Contains a list of breakpoints, and methods to set/clear/list them, as well as to locate expressions/statements in AST by file/line number
- Contains a list of source file contents for source debugging
- Also acts as the initiator for PO generation, though POs are generated in the AST elements.
- Initialization method creates or restores an “initial context” representing the global static context for all subsequent evaluations
- Can evaluate expressions in the initial context, or in a local context when a breakpoint is reached

VDMJ Interpreter 2



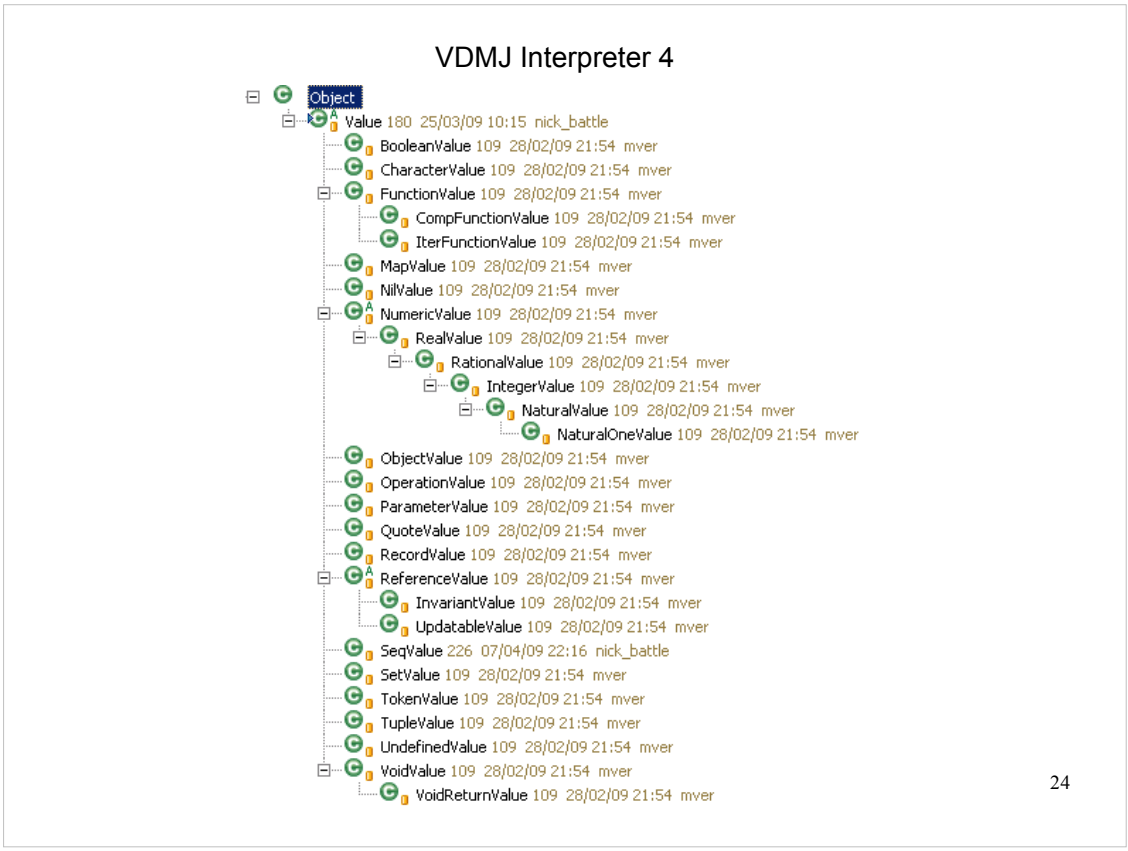
- Evaluation is coordinated by subclasses of Interpreter. The actual business of evaluation is performed by the AST elements
- Runtime name/value pairs held in Context subclasses which mirror the Environment subclasses used in type checking
- Contexts chain together to form the runtime stack. RootContext is the abstract base of a function/operation call. ObjectContext has a “self”, and StateContext points to any module state
- Global values (module state or class static fields) are held in ClassDefinition or Module definitions in AST



Contexts are used as "stack frames", but they encompass less than an entire frame for a function or operation.

In the example above, a breakpoint has been set on line 11 which returns the final value "a". The inset shows the stack trace when control reaches the breakpoint.

Note that the only name/value pair in the top Context is a=6, being the Context of the "let" expression that defines it. Then the outer "let" expression defines another Context that defines r and s. Then the object context for g(x) defines x, g itself, and the "self" value. Lastly the object context for f(a,b) defines a, b, f itself and its "self". Notice that the two self pointers have different object references (#1 and #2), due to the fact that "g" is applied via a new object reference.



The Value hierarchy represents all runtime VDM values in VDMJ.

It can be compared to the Type hierarchy, in that there are Values corresponding to most Types, but note that there is no "UnionValue" – a value cannot be a union of values, at runtime the value must be one of the alternative types discovered during type checking.

Note that NumericValues form a hierarchy: a nat1 is a nat, which is an int, what is a rat, which is a real.

Abstract ReferenceValues are values which refer to other values. The concrete InvariantValue is a value associated with a type with an invariant function, such that any value it refers to must conform to that invariant. An UpdatableValue adds a "set" method to allow changes to the value being referenced.

VoidValue is "no value", and is returned by statements which do not return a value. The VoidReturnValue is returned by the bare "return" statement, which does not return a value as such, but which causes the flow of control to return as though it did.

VDMJ Interpreter 5
Key Methods

```
public abstract class Expression implements Serializable
{
	abstract public Value eval(Context ctxt);

public abstract class Statement implements Serializable
{
	abstract public Value eval(Context ctxt);
```

- Expression and Statement's eval methods return their Value, given the passed Context
- Evaluation recurses over the AST, evaluating and combining the Values of sub-expressions or contained Statements
- The Context is extended by expressions or statements which add variables (eg. let expressions)
- Evaluations can throw ContextExceptions – a runtime exception which includes the Context (stack) as well as a number, text and location

VDMJ Interpreter 6
TailExpression.eval

```
@Override
public Value eval(Context ctxt)
{
    breakpoint.check(location, ctxt);

    ValueList seq = null;

    try
    {
        seq = new ValueList(exp.eval(ctxt).seqValue(ctxt));
    }
    catch (ValueException e)
    {
        return abort(e);
    }

    if (seq.isEmpty())
    {
        abort(4033, "Tail sequence is empty", ctxt);
    }

    seq.remove(0);
    return new SeqValue(seq);
}
```


This is the eval method of a TailExpression.

The AST element only contains one sub-expression, "exp", which is the sequence expression for which the tail is to be taken.

The sub-expression is evaluated using the same Context as passed – "exp.eval(ctxt)" – which yields a Value (presumably a SeqValue). The seqValue method will take any Value and return the inner List<Value> type that it contains, or throw a ValueException if the Value does not actually contain a sequence.

Note that the value returned by seqValue is duplicated by being passed to a new ValueList constructor. This is so that the copy can be modified – without doing this, the original list would be modified, which could change the value of a string literal, for example.

For a tail expression, the resulting list must not be empty, and a test is made for that case, aborting (throwing a ContextException) with error 4033 if that is the case.

Otherwise the list is manipulated to remove the head, and a new SeqValue is returned based on the remaining tail of the list.

All statements and expressions must include the breakpoint check line at the start of their eval methods if a breakpoint is permitted to stop before their evaluation.

VDMJ Interpreter 7
BlockStatement.eval

```
@Override
public Value eval(Context ctxt)
{
    breakpoint.check(location, ctxt);

    Context evalContext = new Context(location, "block statement", ctxt);

    for (Definition d: assignmentDefs)
    {
        evalContext.put(d.getNamedValues(evalContext));
    }

    for (Statement s: statements)
    {
        Value rv = s.eval(evalContext);

        if (!rv.isVoid())
        {
            return rv;
        }
    }

    return new VoidValue();
}
```


This is the eval method of a block statement – ie. a statement that can include dcl statements to define new variables.

This evaluation must execute the statements in the block within a Context that is based on the one passed in, but extended with the name/value pairs defined by the dcl statements at the start of the block. The AST element calls these definitions "assignmentDefs" – a List<Definition>.

A new Context is created, given a sensible location and title (for stack display) and chained onto the end of the one passed in. The new context is populated with name/value pairs generated from the dcl definitions by calling their "getNamedValues" method. All Definitions' implementations of this method return a List<NameValuePair> which can be added directly to a Context.

Having extended the context passed in, each statement in the block is evaluated and the return value of each in sequence is tested to see whether it is non-void. If so, the block evaluation terminates at that point and the value is returned as the value of the block. Otherwise the statements are all executed and a new VoidValue is returned.

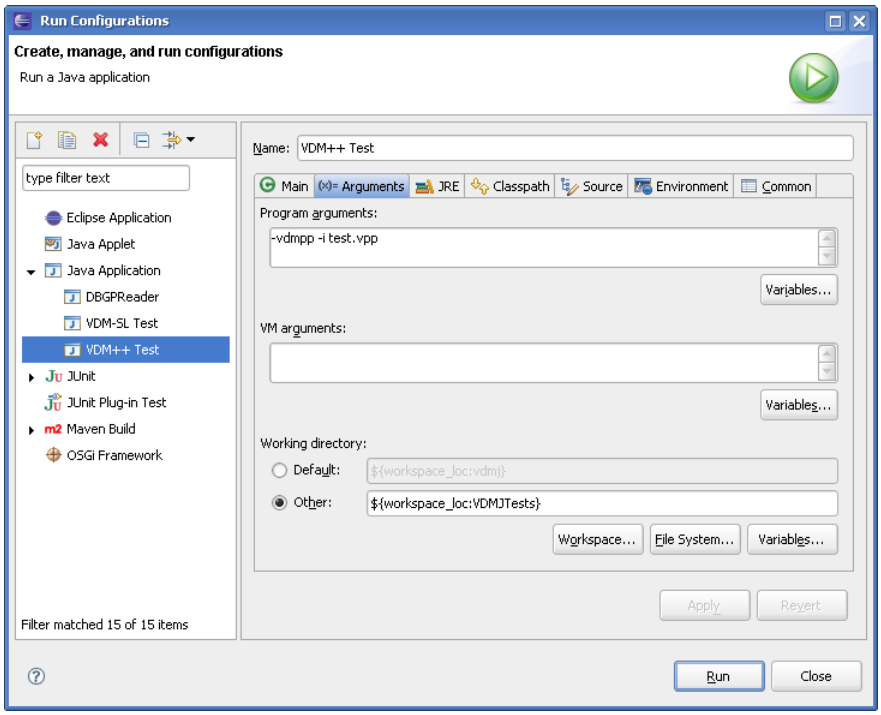
As with the Expression example, all Statements must call the breakpoint check method if a breakpoint is allowed to stop before their execution.

VDMJ Exercise 1

sequence for loop = ‘for’, pattern bind, ‘in’, [‘reverse’], expression, ‘do’, statement ;

- Currently, the "reverse" keyword is part of the sequence loop grammar
- It would be better to treat "reverse" as a new unary sequence operator, like hd, tl, len, elems, inds and conc.
- Grammar becomes: "for", pattern bind, "in", expression, "do", statement
- ReverseExpression.java is partly written in SVN. The VDMJ parser has been changed to build the AST already.
- Complete the eval method of ReverseExpression as an exercise – compare with TailExpression.java
- Test with "print rev [1,2,3]" – should give [3,2,1]. Check that the reverse for loop still works too. See whether you can set a breakpoint.
- The typeCheck method is also blank. Complete this as an exercise too – test with "rev 123". It should complain that the argument is not a sequence.
- What should the getProofObligations method look like?

VDMJ Exercise 2



VDMJ Exercise 3
Tests...

```
Interpreter started
> p rev 1234
Error 3292: Argument to 'rev' is not a sequence in 'A' (console) at line 1:1
> p rev [1,2,3,4]
= [4, 3, 2, 1]
Executed in 0.013 secs.
> p rev "hello"
= "olleh"
Executed in 0.0020 secs.
> p rev []
= []
Executed in 0.0010 secs.
>
```

VDMJ Exercise 4
Solutions...

```
@Override
public Type typeCheck(Environment env, TypeList qualifiers, NameScope scope)
{
    Type etype = exp.typeCheck(env, null, scope);

    if (!etype.isSeq())
    {
        report(3292, "Argument to 'rev' is not a sequence");
        return new SeqType(location, new UnknownType(location));
    }

    return etype;
}

@Override
public Value eval(Context ctxt)
{
    breakpoint.check(location, ctxt);

    ValueList seq = null;

    try
    {
        seq = new ValueList(exp.eval(ctxt).seqValue(ctxt));
        Collections.reverse(seq);
    }
    catch (ValueException e)
    {
        return abort(e);
    }

    return new SeqValue(seq);
}
```