CSC 868.01 – Spring 2013
Advanced Object Oriented Software Design and Development
Prof. Levine

DSS1 Interpreter
(4/1 Milestone)

Group 1
March 31, 2013

Group members

Robert (Bob) Bierman
Victor Woeltjen
Jason Lum
Steven Gimeno
Ying Kit Ng (Kent)
Bianca Uy
Kay Choi

Overview

The interpreter is implemented using the Visitor design pattern, traversing the Abstract Syntax Tree (AST) produced by the existing Compiler using an implementation of the provided ASTVisitor interface, performing computation as appropriate at every given node in the tree.

The Visitor design pattern leverages double dispatch to decouple a data structure from the operations which can be performed while traversing this data structure. The Visitor calls an “accept” method on a node within the data structure, which is itself overloaded to call a more specific method on the Visitor itself; “visitBlockTree”, for example. This permits the external object – the Visitor – to implement behavior using the data structure's type hierarchy, without adding that specific behavior to those types directly.

In the case of the Interpreter, the data structure is the AST, which describes a DSS1 program as a tree of elements – block (BlockTree), if statements (IfTree), et cetera. The Visitor is the IntepreterVisitor, which manages and performs the computation described by this program. This is done with the support of other underlying subsystems to describe variable state and perform type-specific evaluations, as described in the Architecture section.

Conventions

In some cases, the meaning of specific types of nodes is context-dependent. For instance, an IdTree may occur either as an expression, or as the left part of an assignment, or in function arguments. For consistency, the visitation behavior for this and similar nodes is implemented as the evaluation of an expression or similar – that is, the most computational, least structural interpretation as a node. As such, visitIdTree shall return the value currently stored in the variable described by the identifier; other uses of IdTree must be handled by their parents in the tree. For instance, the AssignTree inspects its left-hand child and, when this is an IdTree, performs storage to the variable it describes.

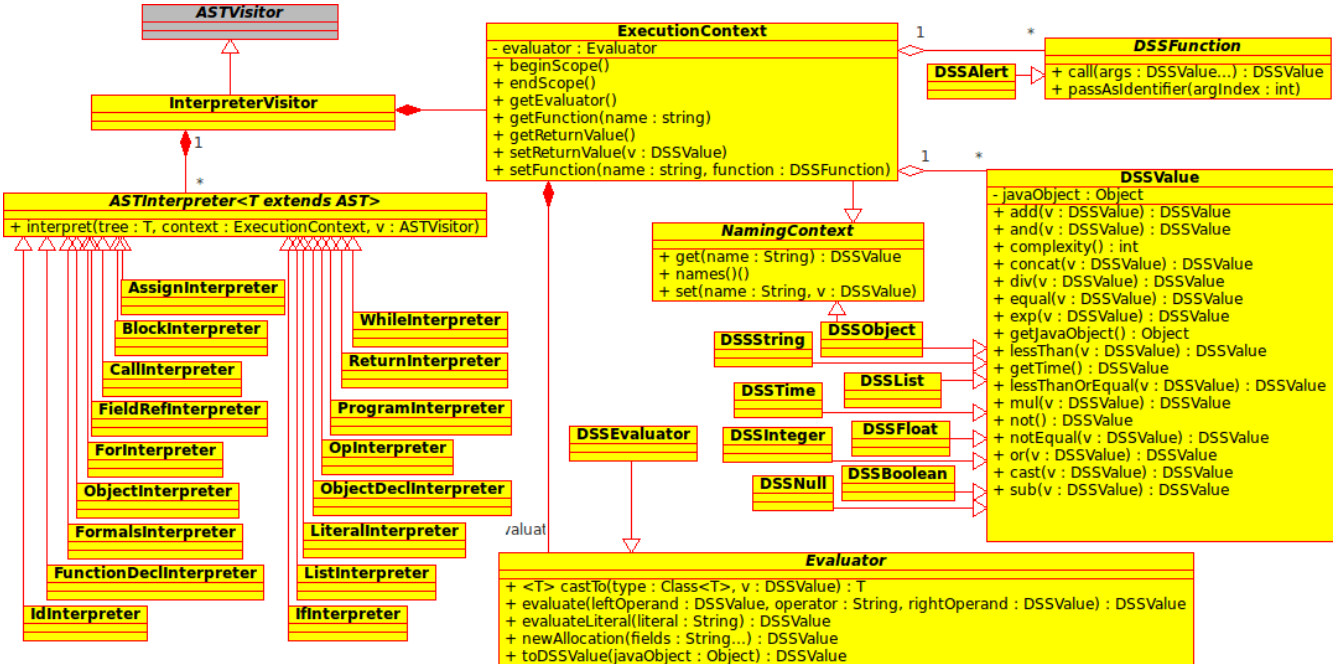
Architecture

The interpreter is implemented as three major subsystems:

- **Traversal subsystem:** InterpreterVisitor and other used classes. This subsystem is responsible for high-level interpretation of the program, including flow control, and coordinating complex expressions.
- **Execution context:** ExecutionContext. Used to store state relevant to the execution. This includes the current values of variables, and all defined functions (intrinsic or otherwise).
- **Evaluation subsystem:** Evaluator, its implementation, and several objects to represent types within DSS1. This subsystem is responsible for implementing the semantics of different types and of specific operations upon such values.

The current implementation effectively uses these subsystems in tiers. An Evaluator is provided to the ExecutionContext upon construction; this ExecutionContext is provided to the InterpreterVisitor on *its* construction. Note that these subsystems therefore require no knowledge of the systems "above" them (the execution context does not know about traversal; the evaluation subsystem is entirely self-contained).

Class Diagram



Subsystems

This section shall describe the roles of classes within the architecture. Bullets and indentation are used to describe conceptual relationships between objects from higher-level to fine-grained; this may include inheritance, composition, or aggregation. For clarity, please refer to the Class Diagram.

Traversal subsystem

- **InterpreterVisitor:** Provides an implementation of the ASTVisitor interface. Maintains several ASTInterpreter objects (one for each type), to which calls to the various visit methods are delegated. (This arrangement is primarily for purposes of code organization.)
  - ***ASTInterpreter:*** Interface used when interpreting specific node types. An ASTInterpreter knows how to interpret some specific type of node, as determined by concrete implementations. Note that, in addition to the relevant node, an ExecutionContext and an ASTVisitor are passed as arguments when this behavior is invoked. This allows implementations to consult variable state, retrieve the Evaluator, traverse the tree, et cetera. Examples follow:
    - **AssignInterpreter:** Handles variable assignment.
    - **BlockInterpreter:** Runs a block of code. Polls for return values in the ExecutionContext to exit early after a *return* has been encountered. (This state will eventually be cleared when control reaches the end of the user-defined function.)
    - **CallInterpreter:** Retrieves a named function from the ExecutionContext and invokes it, returning its return value.
    - …
    - **ReturnInterpreter:** Stores a return value to the ExecutionContext.
    - **WhileInterpreter:** Evaluates an expression, and continues running until that expression is false.

The implementation of WhileInterpreter is presented here as an example:

```
public class WhileInterpreter implements ASTInterpreter<WhileTree> {
    public Object interpret(WhileTree tree, ExecutionContext context, ASTVisitor visitor) {
        // Evaluate the condition; use the result as a Java while's condition
        while (evaluate(tree.getKid(1), context, visitor)) {
            tree.getKid(2).accept(visitor); // Traverse to block child
        }
        return null;
    }

    // Helper function to evaluate the condition tree to return a Java boolean
    private boolean evaluate(AST t, ExecutionContext context, ASTVisitor v) {
        Object result = t.accept(v);
        if (result instanceof DSSValue) {
            // Convert the result to a Java boolean
            return context.getEvaluator().castTo(Boolean.class, (DSSValue)result);
        }
        // Default to false
        return false;
    }
}
```

ExecutionContext

- **ExecutionContext:** Provides a means to store and retrieve: Return values; variable states; and named functions. Handles rules of scope to hide variables during function calls. Also exposes the Evaluator. Note that this may be populated with functions or even variables before being given to the InterpreterVisitor, allowing the definition of intrinsics.
    - ***DSSFunction:*** An interface used to describe any function called from DSS1 (either intrinsic or user-defined). This permits function calling to be implemented identically for both categories of function. Additionally includes a method for testing if a given argument should be passed as a raw identifier instead of evaluated directly (as used by some intrinsics.)
        - **DSSAlert:** An implementation of the *alert* intrinsic; when called, simply prints its arguments to the console.
        - *User-defined functions:* Defined by an inner class of FunctionDeclTree.

Evaluation subsystem

- ***Evaluator* / DSSEvaluator:** Exposes methods to perform operations upon DSS1 values, to interpret literals, allocate DSS objects, and perform conversions between DSS1 values and similar Java objects. (Note that Evaluator is an interface, whereas DSSEvaluator is the implementation; this distinction is used to permit and promote a decoupled implementation of the traversal subsystem by programming to an interface, as well as to facilitate stubbing and testing. The Evaluator interface is not strictly necessary and may be removed in a later milestone.)
    - ***DSSValue:*** An abstract class describing a value in a running DSS1 program. Primarily this exposes methods for the various operations available under DSS1. Additionally, there is the *cast* method, and a *complexity* method which returns an integer indicating the relative "complexity" of each type of value; these the DSSEvaluator to attempt type promotion when dealing with mismatched operand (for instance, DSSString will cast any DSSValue to a DSSString using its *toString* method). Types presented below are in order of increasing complexity in this promotion scheme. Finally, every DSSValue has a corresponding Java object that describes its internal state, which is also made accessible.
        - **DSSNull:** A null value in DSS1. Underlying Java object is an Object.
        - **DSSBoolean:** A boolean value in DSS1. Underlying Java object is a Boolean. (Exposes two constants, DSSBoolean.TRUE and DSSBoolean.FALSE, as a convenience.)
        - **DSSInteger:** An integer value in DSS1. Underlying Java object is a Long.
        - **DSSFloat:** A floating-point value in DSS1. Underlying Java object is a Double.
        - **DSSTime:** A date & time value in DSS1. Underlying Java object is a Date.
        - **DSSList:** A list value in DSS1. Underlying Java object is a List<DSSValue>
        - **DSSObject:** An object in DSS1. Underlying Java object is a Map<String, DSSValue>. Note that DSSObject also implements the interface *NamingContext*, allowing values to be stored to & retrieved from it in the same manner as ExecutionContext.
        - **DSSString:** A string of text in DSS1. Underlying Java object is String. Considered the top-level type, as all other types can readily support toString.

Sample Results

Input file *math.dss*	**Console output***

`program`
`function newInstance(obj) {`
`    return new obj`
`}`
 	
`// MAIN BLOCK`
`{`
`    alert(44.5)`
`    alert(true)`
`    alert(true & false)`
`    alert(true | false)`
 	
`    alert("hello")`
 	
`    alert(0)`
`    alert(55 + 22)`
`    alert(55 + 22.5)`
 	
`    alert("Temperature is " + 98.6)`
 	
`    o := Object(x,y)`
`    o.x := 10`
`    o.y := "Words"`
 	
`    alert(o)`
`    alert(newInstance(o))`
 	
`    alert (44 < 44.1)`
 	
`    alert (2  ** 2)`
`    alert (16 ** 0.5)`
`}`
	`Source file: chol.dss`
	`user.dir: *(omitted)*`
	
	`44.5`
	`true`
	`false`
	`true`
	
	`hello`
	
	`0`
	`77`
	`77.5`
	
	`Temperature is 98.6`
	
	`Object( y=Words , x=10 )`
	`Object( y=null , x=null )`
	
	`true`
	
	`4`
	`4.0`
	`BUILD SUCCESSFUL (total time: 0 seconds)`

* Note that new-lines have been added to console output to align results with corresponding *alert* calls.

<table>
<tr><th>Input file chol.dss</th><th>Console output*</th></tr>
<tr><td>

```
program

function check(n) {
    if n < 200  then {
      return "Desirable Cholesterol"}
    elsif n < 239 then {
      return
      "Borderline High Cholesterol" }
    else { return "High Cholesterol!" }
}

function cratio(chol){
    return chol/30
}

// MAIN BLOCK
{
    t := Object(name, count)
    t.name := "Mary"
    t.count := 456

    alert("Mary: " +
      t.count + " : " + check(t.count))
    if check(t.count) ==
      "High Cholesterol!" then {
        alert("  Cholesterol Ratio: "
          + cratio(t.count) + "-to-1")}
    t.count := 150
    alert(t.count + " : " +
          check(t.count))
}
```

</td><td>

```
Source file: chol.dss
user.dir: (omitted)
















Mary: 456 : High Cholesterol!



  Cholesterol Ratio: 15-to-1



150 : Desirable Cholesterol
```

</td></tr>
</table>

* Note that new-lines have been added to console output to align results with corresponding *alert* calls.