Bart's blog

A blog about software development.

TUESDAY, MARCH 08, 2011

7. Interpreting and evaluating *TL* I

*This is part 7/9 of **Creating your own programming language with ANTLR**.*

Index

Introduction

First, make sure your files are all up to date. If you're not sure they are, download all source files, ANTLR jar, build file etc. **here** so that you won't get any strange errors in this part of the tutorial.

This part of the tutorial will make some necessary preparations needed to actually evaluating the input source.

Preparation

Before actually evaluating the input source, we'll have to create a couple of supporting classes like a class that represents a function, a generic value and something that keeps track of our variables: a scope.

TLValue.java

Since *TL* is a dynamically type language, we'll create a generic value, called `TLValue` that gets checked and operated on at runtime. Such a class could look like this (place it in **src/main/tl/**):

view plain print ?

```
01. package tl;
02.
03. import java.util.List;
04.
05. public class TLValue implements Comparable<TLValue> {
06.
07.   public static final TLValue NULL = new TLValue();
08.   public static final TLValue VOID = new TLValue();
09.
10.   private Object value;
11.
12.   private TLValue() {
13.     // private constructor: only used for NULL and VOID
14.     value = new Object();
15.   }
16.
17.   public TLValue(Object v) {
18.     if(v == null) {
19.       throw new RuntimeException("v == null");
20.     }
21.     value = v;
22.     // only accept boolean, list, number or string types
23.     if(!(isBoolean() || isList() || isNumber() || isString())) {
24.       throw new RuntimeException("invalid type: " + v + " (" + v.getClass() + ")");
25.     }
26.   }
27.
28.   public Boolean asBoolean() {
29.     return (Boolean)value;
30.   }
31.
```

```
32.   public Double asDouble() {
33.     return ((Number)value).doubleValue();
34.   }
35.
36.   public Long asLong() {
37.     return ((Number)value).longValue();
38.   }
39.
40.   @SuppressWarnings("unchecked")
41.   public List<TLValue> asList() {
42.     return (List<TLValue>)value;
43.   }
44.
45.   public String asString() {
46.     return (String)value;
47.   }
48.
49.   @Override
50.   public int compareTo(TLValue that) {
51.     if(this.isNumber() && that.isNumber()) {
52.       if(this.equals(that)) {
53.         return 0;
54.       }
55.       else {
56.         return this.asDouble().compareTo(that.asDouble());
57.       }
58.     }
59.     else if(this.isString() && that.isString()) {
60.       return this.asString().compareTo(that.asString());
61.     }
62.     else {
63.       throw new RuntimeException("illegal expression: can't compare `" +
64.           this + "` to `" + that + "`");
65.     }
66.   }
67.
68.   @Override
69.   public boolean equals(Object o) {
70.     if(this == VOID || o == VOID) {
71.       throw new RuntimeException("can't use VOID: " + this + " ==/!= " + o);
72.     }
73.     if(this == o) {
74.       return true;
75.     }
76.     if(o == null || this.getClass() != o.getClass()) {
77.       return false;
78.     }
79.     TLValue that = (TLValue)o;
80.     if(this.isNumber() && that.isNumber()) {
81.       double diff = Math.abs(this.asDouble() - that.asDouble());
82.       return diff < 0.00000000001;
83.     }
84.     else {
85.       return this.value.equals(that.value);
86.     }
87.   }
88.
89.   @Override
90.   public int hashCode() {
91.     return value.hashCode();
92.   }
93.
94.   public boolean isBoolean() {
95.     return value instanceof Boolean;
96.   }
97.
98.   public boolean isNumber() {
99.     return value instanceof Number;
100.  }
101.
102.  public boolean isList() {
103.    return value instanceof List<?>;
104.  }
105.
106.  public boolean isNull() {
107.    return this == NULL;
108.  }
109.
110.  public boolean isVoid() {
111.    return this == VOID;
112.  }
113.
114.  public boolean isString() {
115.    return value instanceof String;
116.  }
117.
118.  @Override
119.  public String toString() {
120.    return isNull() ? "NULL" : isVoid() ? "VOID" : String.valueOf(value);
121.  }
122. }
```

Function.java

Remember that in **part 5. Building an AST**, we omitted the rule responsible for parsing a function declaration from our AST. Let's implement that now by creating a Function class. We'll put it in the same package as our

Main.java class: `package tl;`. But first, we'll need to adjust our combined grammar file **TL.g** a bit.

So instead of what we had:

```
view plain print ?
01. functionDecl
02.   :  Def Identifier '(' idList? ')' block End { /* implemented later */ }
03.   ;
```

we'll make a call to the method `defineFunction(...)`

```
view plain print ?
01. functionDecl
02.   :  Def Identifier '(' idList? ')' block End
03.      {defineFunction($Identifier.text, $idList.tree, $block.tree);}
04.   ;
```

That method does not exist in our `TLParser` class of course: we'll need to create it on our own. You can add custom class variables and methods to a parser by putting them in the **@parser::members** section, which should be placed *after* the **@header** sections.

```
view plain print ?
01. // tokens { ... }
02.
03. @parser::header {
04.   package tl.parser;
05.   import tl.*;
06.   import java.util.Map;
07.   import java.util.HashMap;
08. }
09.
10. @lexer::header {
11.   package tl.parser;
12. }
13.
14. @parser::members {
15.   public Map<String, Function> functions = new HashMap<String, Function>();
16.
17.   private void defineFunction(String id, Object idList, Object block) {
18.
19.     // `idList` is possibly null!  Create an empty tree in that case.
20.     CommonTree idListTree = idList == null ? new CommonTree() : (CommonTree)idList;
21.
22.     // `block` is never null
23.     CommonTree blockTree = (CommonTree)block;
24.
25.     // The function name with the number of parameters after it, is the unique key
26.     String key = id + idListTree.getChildCount();
27.     functions.put(key, new Function(id, idListTree, blockTree));
28.   }
29. }
30.
31. // rules ...
```

Notice that I also added a couple of extra imports in the **@parser::header** section. So, our parser now builds a `Map` of `Function`s for us. Let's create that class now:

```
view plain print ?
01. package tl;
02.
03. import org.antlr.runtime.RecognitionException;
04. import org.antlr.runtime.tree.CommonTree;
05. import org.antlr.runtime.tree.CommonTreeNodeStream;
06. import tl.parser.TLTreeWalker;
07. import tl.tree.TLNode;
08.
09. import java.util.ArrayList;
10. import java.util.List;
11. import java.util.Map;
12.
13. public class Function {
14.
15.   private String id;
16.   private List<String> identifiers;
17.   private CommonTree code;
18.   private Scope scope;
19.
20.   public Function(String i, CommonTree ids, CommonTree block) {
21.     id = i;
22.     identifiers = toList(ids);
23.     code = block;
24.     scope = new Scope();
25.   }
26.
27.   public Function(Function original) {
28.     // Used for recursively calling functions
```

```
    id = original.id;
    identifiers = original.identifiers;
    code = original.code;
    scope = original.scope.copy();
  }

  public TLValue invoke(List<TLNode> params, Map<String, Function> functions) {

    if(params.size() != identifiers.size()) {
      throw new RuntimeException("illegal function call: " + identifiers.size() +
          " parameters expected for function `" + id + "`");
    }

    // Assign all expression parameters to this function's identifiers
    for(int i = 0; i < identifiers.size(); i++) {
      scope.assign(identifiers.get(i), params.get(i).evaluate());
    }

    try {
      // Create a tree walker to evaluate this function's code block
      CommonTreeNodeStream nodes = new CommonTreeNodeStream(code);
      TLTreeWalker walker = new TLTreeWalker(nodes, scope, functions);
      return walker.walk().evaluate();
    } catch (RecognitionException e) {
      // do not recover from this
      throw new RuntimeException("something went wrong, terminating", e);
    }
  }

  private List<String> toList(CommonTree tree) {
    List<String> ids = new ArrayList<String>();

    // convert the tree to a List of Strings
    for(int i = 0; i < tree.getChildCount(); i++) {
      CommonTree child = (CommonTree)tree.getChild(i);
      ids.add(child.getText());
    }
    return ids;
  }
}
```

Notice that a `Function` creates its own `Scope` instance and the `invoke(...)` takes a list of `TLNode` instances. It also returns a `TLValue` by invoking `evaluate()` on what `walker.walk()` returns. Which doesn't return anything at the moment, but we'll change that later on.

Let's define `TLNode` and `Scope` next.

TLNode.java

Our tree now contains only **CommonTree** objects, but later on, we'll create custom node classes. All of our custom node classes implement the same interface: namely the `TLNode` interface:

```
view plain print ?
package tl.tree;

public interface TLNode {
  tl.TLValue evaluate();
}
```

which I've placed in the `tl.tree` package. So create a folder called **tree** in the **src/main/tl/** directory and save the interface in it.

Scope.java

Every code block, and functions, have their own local variable scope. Note that `for`-, `while`- and `if`-statements also have their own code block! So we'll create a class called `Scope` and place it in **src/main/tl/** as well. This class keeps track of it's parent scope to resolve variables itself has no notion of, and will store all declared variables (or ask a parent scope to re-assign a certain variable if it's not declared in its own scope).

```
view plain print ?
package tl;

import java.util.HashMap;
import java.util.Map;

public class Scope {

  private Scope parent;
  private Map<String, TLValue> variables;

  public Scope() {
    // only for the global scope, the parent is null
    this(null);
  }

  public Scope(Scope p) {
    parent = p;
```

```
    variables = new HashMap<String, TLValue>();
  }

  public void assign(String var, TLValue value) {
    if(resolve(var) != null) {
      // There is already such a variable, re-assign it
      this.reAssign(var, value);
    }
    else {
      // A newly declared variable
      variables.put(var, value);
    }
  }

  public Scope copy() {
    // Create a shallow copy of this scope. Used in case functions are
    // are recursively called. If we wouldn't create a copy in such cases,
    // changing the variables would result in changes to the Maps from
    // other "recursive scopes".
    Scope s = new Scope();
    s.variables = new HashMap<String, TLValue>(this.variables);
    return s;
  }

  public boolean isGlobalScope() {
    return parent == null;
  }

  public Scope parent() {
    return parent;
  }

  private void reAssign(String identifier, TLValue value) {
    if(variables.containsKey(identifier)) {
      // The variable is declared in this scope
      variables.put(identifier, value);
    }
    else if(parent != null) {
      // The variable was not declared in this scope, so let
      // the parent scope re-assign it
      parent.reAssign(identifier, value);
    }
  }

  public TLValue resolve(String var) {
    TLValue value = variables.get(var);
    if(value != null) {
      // The variable resides in this scope
      return value;
    }
    else if(!isGlobalScope()) {
      // Let the parent scope look for the variable
      return parent.resolve(var);
    }
    else {
      // Unknown variable
      return null;
    }
  }
}
```

TLTreeWalker.g

Inside our tree walker, we'll also have a reference to the `Map` of functions and we'll keep track of the current `Scope` our tree walker is in. Finally, I've let the entry point of our walker, the `walk` rule, return a `TLNode`:

```
view plain  print  ?

tree grammar TLTreeWalker;

options {
  tokenVocab=TL;
  ASTLabelType=CommonTree;
}

@header {
  package tl.parser;
  import tl.*;
  import tl.tree.*;
  import java.util.Map;
  import java.util.HashMap;
}

@members {
  public Map<String, Function> functions = null;
  Scope currentScope = null;

  public TLTreeWalker(CommonTreeNodeStream nodes, Map<String, Function> fns) {
    this(nodes, null, fns);
  }

  public TLTreeWalker(CommonTreeNodeStream nds, Scope sc, Map<String, Function> fns) {

    super(nds);
    currentScope = sc;
    functions = fns;
```

```
  }
}

walk returns [TLNode node]
  :  block {node = null;}
  ;

// other rules ...
```

Main.java

We can now test all changes with the new `Main` class:

```
view plain print ?
package tl;

import tl.parser.*;
import tl.tree.*;
import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;

public class Main {
  public static void main(String[] args) throws Exception {
    // create an instance of the lexer
    TLLexer lexer = new TLLexer(new ANTLRFileStream("test.tl"));

    // wrap a token-stream around the lexer
    CommonTokenStream tokens = new CommonTokenStream(lexer);

    // create the parser
    TLParser parser = new TLParser(tokens);

    // walk the tree
    CommonTree tree = (CommonTree)parser.parse().getTree();
    CommonTreeNodeStream nodes = new CommonTreeNodeStream(tree);

    // pass the reference to the Map of functions to the tree walker
    TLTreeWalker walker = new TLTreeWalker(nodes, parser.functions);

    // get the returned node
    TLNode returned = walker.walk();
    System.out.println(returned == null ? "null" : returned.evaluate());
  }
}
```

That should make our project compilable again. Test everything by changing the contents of **test.tl** with some different sources. Again, if nothing is printed, you're okay. Of course, **null** *is* printed since that's what our `walk` now still returns.

In case you haven't been able to get things running properly so far, download the correct sources of the project developed until now, **here**.

Evaluation I

As you've seen, our `walk` rule now returns a `TLNode`. We're going to let all rules in our tree grammar return such an object (except some rules that are only created to support a single other rule, like `idList` or `expList` or the child-rules of an if-statement).

Our tree grammar including the return types now looks like:

```
view plain print ?
tree grammar TLTreeWalker;

options {
  tokenVocab=TL;
  ASTLabelType=CommonTree;
}

@header {
  package tl.parser;
  import tl.*;
  import tl.tree.*;
  import java.util.Map;
  import java.util.HashMap;
}

@members {
  public Map<String, Function> functions = null;
  Scope currentScope = null;

  public TLTreeWalker(CommonTreeNodeStream nodes, Map<String, Function> fns) {
    this(nodes, new Scope(), fns);
  }

  public TLTreeWalker(CommonTreeNodeStream nds, Scope sc, Map<String, Function> fns) {
```

```
    super(nds);
    currentScope = sc;
    functions = fns;
  }
}

walk returns [TLNode node]
  :  block
  ;

block returns [TLNode node]
  :  ^(BLOCK ^(STATEMENTS statement*) ^(RETURN expression?))
  ;

statement returns [TLNode node]
  :  assignment
  |  functionCall
  |  ifStatement
  |  forStatement
  |  whileStatement
  ;

assignment returns [TLNode node]
  :  ^(ASSIGNMENT Identifier indexes? expression)
  ;

functionCall returns [TLNode node]
  :  ^(FUNC CALL Identifier exprList?)
  |  ^(FUNC CALL Println expression?)
  |  ^(FUNC CALL Print expression)
  |  ^(FUNC CALL Assert expression)
  |  ^(FUNC_CALL Size expression)
  ;

ifStatement returns [TLNode node]
  :  ^(IF ifStat elseIfStat* elseStat?)
  ;

ifStat
  :  ^(EXP expression block)
  ;

elseIfStat
  :  ^(EXP expression block)
  ;

elseStat
  :  ^(EXP block)
  ;

forStatement returns [TLNode node]
  :  ^(For Identifier expression expression block)
  ;

whileStatement returns [TLNode node]
  :  ^(While expression block)
  ;

idList returns [java.util.List<String> i]
  :  ^(ID_LIST Identifier+)
  ;

exprList returns [java.util.List<TLNode> e]
  :  ^(EXP_LIST expression+)
  ;

expression returns [TLNode node]
  :  ^(TERNARY expression expression expression)
  |  ^(In expression expression)
  |  ^('||' expression expression)
  |  ^('&&' expression expression)
  |  ^('==' expression expression)
  |  ^('!=' expression expression)
  |  ^('>=' expression expression)
  |  ^('<=' expression expression)
  |  ^('>' expression expression)
  |  ^('<' expression expression)
  |  ^('+' expression expression)
  |  ^('-' expression expression)
  |  ^('*' expression expression)
  |  ^('/' expression expression)
  |  ^('%' expression expression)
  |  ^('^' expression expression)
  |  ^(UNARY MIN expression)
  |  ^(NEGATE expression)
  |  Number
  |  Bool
  |  Null
  |  lookup
  ;

lookup returns [TLNode node]
  :  ^(LOOKUP functionCall indexes?)
  |  ^(LOOKUP list indexes?)
  |  ^(LOOKUP expression indexes?)
  |  ^(LOOKUP Identifier indexes?)
  |  ^(LOOKUP String indexes?)
  ;
```

```
123.
124. list returns [TLNode node]
125.   :  ^(LIST exprList?)
126.   ;
127.
128. indexes returns [java.util.List<TLNode> e]
129.   :  ^(INDEXES expression+)
130.   ;
```

The next step is to write custom `TLNode` classes and let these be returned by our tree grammar rules. That way, we simply invoke our top-most rule, `walk` that returns the root of the tree, and when we call its `evaluate()` method, it should recursively call all it's child `evaluate()` methods, and so on.

The second part of this evaluate-tutorial can be read here: **8. Interpreting and evaluating *TL* II**.

Download the most recent source files of this tutorial **here**.

Posted by Bart Kiers at **7:59 AM**
Labels: **antlr**, **dsl**, **java**, **parsing**

No comments:

Post a Comment

Newer Post **Home** **Older Post**

Subscribe to: **Post Comments (Atom)**