Bart's blog

A blog about software development.

WEDNESDAY, MARCH 09, 2011

8. Interpreting and evaluating *TL* II

*This is part 8/9 of **Creating your own programming language with ANTLR**.*

Index

Introduction

Okay, welcome back. In this part, we're going to create custom tree nodes that will perform the actual evaluation of our *TL* source.

Before continuing, make sure your source files are up to date. If they're not, or you're unsure of it, download the most recent source files of this tutorial **here**.

Evaluation II

I won't handle all different nodes types, but I'll explain a couple different ones instead. If I explain the `AddNode`, there is no need to handle `SubNode`, for example. In the final part of this tutorial, I'll put a link to a zip file containing the entire project including all custom nodes.

AddNode

Okay, the very first node we're going to handle is the `AddNode`. An `AddNode` always has two children: a **lhs** (left hand side) and a **rhs** (right hand side). This operation not only adds two numerical values together, but we'll also support string concatenation and add elements to a list using the `+` operator. Okay, this is how an implementation of such a node could look like:

view plain print ?

```
01. // place this file in: src/main/tl/tree/
02. package tl.tree;
03.
04. import tl.TLValue;
05. import java.util.List;
06.
07. public class AddNode implements TLNode {
08.
09.   private TLNode lhs;
10.   private TLNode rhs;
11.
12.   public AddNode(TLNode lhs, TLNode rhs) {
13.     this.lhs = lhs;
14.     this.rhs = rhs;
15.   }
16.
17.   @Override
18.   public TLValue evaluate() {
19.
20.     TLValue a = lhs.evaluate();
21.     TLValue b = rhs.evaluate();
22.
23.     // number + number
24.     if(a.isNumber() && b.isNumber()) {
25.       return new TLValue(a.asDouble() + b.asDouble());
26.     }
```

```
27.
28.     // list + any
29.     if(a.isList()) {
30.       List<TLValue> list = a.asList();
31.       list.add(b);
32.       return new TLValue(list);
33.     }
34.
35.     // string + any
36.     if(a.isString()) {
37.       return new TLValue(a.asString() + "" + b.toString());
38.     }
39.
40.     // any + string
41.     if(b.isString()) {
42.       return new TLValue(a.toString() + "" + b.asString());
43.     }
44.
45.     throw new RuntimeException("illegal expression: " + this);
46.   }
47.
48.   @Override
49.   public String toString() {
50.     return String.format("(%s + %s)", lhs, rhs);
51.   }
52. }
```

As you can see, inside the `evaluate` method, the left- and right children are first evaluated, and based on the type they return, the add returns the end result. And if we "fall" all the way through, it must mean it is an illegal expression, and throw an exception. Pretty straight forward (I hope!).

Okay, now we'll need to edit our tree grammar file, and make sure the `node` variable that is being returned, is assigned to an `AddNode`. This can be done by doing:

```
view plain print ?
01. expression returns [TLNode node]
02.   :  ^(TERNARY expression expression expression)
03.   |  ^(In expression expression)
04.   |  ^('||' expression expression)
05.   |  ^('&&' expression expression)
06.   |  ^('==' expression expression)
07.   |  ^('!=' expression expression)
08.   |  ^('>=' expression expression)
09.   |  ^('<=' expression expression)
10.   |  ^('>' expression expression)
11.   |  ^('<' expression expression)
12.   |  ^('+' a=expression b=expression) {node = new AddNode($a.node, $b.node);}
13.   |  ^('-' expression expression)
14.   |  ^('*' expression expression)
15.   |  ^('/' expression expression)
16.   |  ^('%' expression expression)
17.   |  ^('^' expression expression)
18.   |  ^(UNARY MIN expression)
19.   |  ^(NEGATE expression)
20.   |  Number
21.   |  Bool
22.   |  Null
23.   |  lookup
24.   ;
```

Since there is more than one `expression` inside `^('+' expression expression)`, we can't reference a particular rule using `$expression`. That's why I named the left `expression` variable `a` and the right `expression` variable `b`. After the sub-rule, in plain code, I then created an instance of a `AddNode` and assigned that to `node` which is the object the `expression` rule returns.

In the entire grammar, I named the variable that is being returned `node`, but you can use anything you want there. Let's say I would have named it `TREE`:

```
view plain print ?
01. expression returns [TLNode TREE]
02.   :  ...
03.   ;
```

I'd simply needed to make sure that inside the custom code I also used that variable name, like this:

```
view plain print ?
01. expression returns [TLNode TREE]
02.   :  ...
03.   |  ^('+' a=expression b=expression) {TREE = new AddNode($a.TREE, $b.TREE);}
04.   |  ...
05.   ;
```

AtomNode.java

Okay, to be able to perform a first small test, we'll also need to create a node that represents the "smallest" part of the tree (a leaf node) containing one of our datatypes: number, string, boolean or list (or null). Instead of creating four (or five) different nodes, we'll simply create a single node called `AtomNode` that wraps a `TLValue` around the object we pass to it.

view plain print ?

```
01. package tl.tree;
02.
03. import tl.TLValue;
04.
05. public class AtomNode implements TLNode {
06.
07.   private TLValue value;
08.
09.   public AtomNode(Object v) {
10.     value = (v == null) ? TLValue.NULL : new TLValue(v);
11.   }
12.
13.   @Override
14.   public TLValue evaluate() {
15.     return value;
16.   }
17.
18.   @Override
19.   public String toString() {
20.     return value.toString();
21.   }
22. }
```

Let's see how that works for a `Number` token:

view plain print ?

```
01. expression returns [TLNode node]
02.   :  ^(TERNARY expression expression expression)
03.   |  ^(In expression expression)
04.   |  ^('||' expression expression)
05.   |  ^('&&' expression expression)
06.   |  ^('==' expression expression)
07.   |  ^('!=' expression expression)
08.   |  ^('>=' expression expression)
09.   |  ^('<=' expression expression)
10.   |  ^('>' expression expression)
11.   |  ^('<' expression expression)
12.   |  ^('+' a=expression b=expression) {node = new AddNode($a.node, $b.node);}
13.   |  ^('-' expression expression)
14.   |  ^('*' expression expression)
15.   |  ^('/' expression expression)
16.   |  ^('%' expression expression)
17.   |  ^('^' expression expression)
18.   |  ^(UNARY MIN expression)
19.   |  ^(NEGATE expression)
20.   |  Number {node = new AtomNode(Double.parseDouble($Number.text));}
21.   |  Bool
22.   |  Null
23.   |  lookup
24.   ;
```

BlockNode.java

We can almost do a first test of a script that actually evaluates something: we only need to implement a node that represents a block of code:

view plain print ?

```
01. block returns [TLNode node]
02.   :  ^(BLOCK ^(STATEMENTS statement*) ^(RETURN expression?))
03.   ;
```

That's a bit different than our previous two nodes. This node is actually a collection of nodes. Here's how that could look like:

view plain print ?

```
01. package tl.tree;
02.
03. import tl.TLValue;
04.
05. import java.util.ArrayList;
06. import java.util.List;
07.
08. public class BlockNode implements TLNode {
09.
10.   private List<TLNode> statements;
11.   private TLNode returnStatement;
12.
13.   public BlockNode() {
14.     statements = new ArrayList<TLNode>();
```

```
    returnStatement = null;
  }

  public void addReturn(TLNode stat) {
    returnStatement = stat;
  }

  public void addStatement(TLNode stat) {
    statements.add(stat);
  }

  @Override
  public TLValue evaluate() {
    for(TLNode stat : statements) {
      TLValue value = stat.evaluate();
      if(value != TLValue.VOID) {
        // return early from this block if value is a return statement
        return value;
      }
    }

    // return VOID or returnStatement.evaluate() if it's not null
    return returnStatement == null ? TLValue.VOID : returnStatement.evaluate();
  }
}
```

As you can see, it starts as an empty code block and can have statements added to it through its `addStatement(...)` method, and a return statement can optionally be set by using its `addReturn(...)` method. Let's integrate that in our tree grammar:

view plain print ?

```
walk returns [TLNode node]
  :  block {node = $block.node;}
  ;

block returns [TLNode node]
@init {
  BlockNode bn = new BlockNode();
  node = bn;
  Scope scope = new Scope(currentScope);
  currentScope = scope;
}
@after {
  currentScope = currentScope.parent();
}
  :  ^(BLOCK
       ^( STATEMENTS (statement  {bn.addStatement($statement.node);})* )
       ^( RETURN     (expression {bn.addReturn($expression.node);  })? )
     )
  ;
```

Before anything from the `block` rule is iterated over, a `BlockNode` is instantiated in an **@init** section, and assigned to the node being returned. After that, the zero or more statements are added to this `BlockNode`, as well as the optional return statement (expression actually).

Note that I squeezed in the pushing and popping of scopes. Whenever we enter a new code block, we create a new current scope, and when we leave the code block, we pop back to the parent scope again.

Notice how I also assigned the return node of the `walk` rule to become the node of the `block` rule.

Okay, edit the **test.tl** file to contain the contents:

view plain print ?

```
return 5 + 2 + 3;
```

and execute the `run` Ant target. You will see the following being printed to the console:

```
$ ant run

...

run:
     [echo] Running the main class...
     [java] 10.0
```

Yay! It works! :)

IfNode.java

Let's implement a slightly more difficult node: the `IfNode`. Such a node might look like this:

```
view plain print ?
01. package tl.tree;
02.
03. import tl.TLValue;
04. import java.util.ArrayList;
05. import java.util.List;
06.
07. public class IfNode implements TLNode {
08.
09.   private List<Choice> choices;
10.
11.   public IfNode() {
12.     choices = new ArrayList<Choice>();
13.   }
14.
15.   public void addChoice(TLNode e, TLNode b) {
16.     choices.add(new Choice(e, b));
17.   }
18.
19.   @Override
20.   public TLValue evaluate() {
21.
22.     for(Choice ch : choices) {
23.       TLValue value = ch.expression.evaluate();
24.
25.       if(!value.isBoolean()) {
26.         throw new RuntimeException("illegal boolean expression " +
27.             "inside if-statement: " + ch.expression);
28.       }
29.
30.       if(value.asBoolean()) {
31.         return ch.block.evaluate();
32.       }
33.     }
34.
35.     return TLValue.VOID;
36.   }
37.
38.   private class Choice {
39.
40.     TLNode expression;
41.     TLNode block;
42.
43.     Choice(TLNode e, TLNode b) {
44.       expression = e;
45.       block = b;
46.     }
47.   }
48. }
```

As you can see, the contents of an `IfNode` is simply a list of choices, where a choice is a (boolean) expression with a block of code attached to it. When evaluating an if-statement, we simply iterate over these choices, and as soon as we encounter an expression that evaluates to true, the corresponding code block is executed and its return value is returned.

Let's simplify the tree grammar rules a bit. Currently, our `ifStatement` consists of 4 rules:

```
view plain print ?
01. ifStatement returns [TLNode node]
02.   :  ^(IF ifStat elseIfStat* elseStat?)
03.   ;
04.
05. ifStat
06.   :  ^(EXP expression block)
07.   ;
08.
09. elseIfStat
10.   :  ^(EXP expression block)
11.   ;
12.
13. elseStat
14.   :  ^(EXP block)
15.   ;
```

Although we can keep this into four rules, it's easier to integrate them all in just one rule. Notice that in the tree grammar, the rules `ifStat` and `elseIfStat` are the same, so `ifStat elseIfStat*` could also be written as simply: `ifStat+`. The four rules combined would now look like:

```
view plain print ?
01. ifStatement returns [TLNode node]
02.   :  ^(IF
03.        (^(EXP expression block))+
04.        (^(EXP block))?
05.      )
06.   ;
```

And with the code mixed in, it looks like:

```
view plain print ?
ifStatement returns [TLNode node]
@init  {
  IfNode ifNode = new IfNode();
  node = ifNode;
}
  :  ^(IF
       (^(EXP expression b1=block){ifNode.addChoice($expression.node,$b1.node);})+
       (^(EXP b2=block)           {ifNode.addChoice(new AtomNode(true),$b2.node);})?
     )
  ;
```

Notice that in the `else` sub-rule, a `true`-atom is inserted.

LTNode.java

To be able test the `if` statement in our language, we'll need to implement an expression node that evaluates to a boolean value. We'll implement the less-than comparison node, which looks like:

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

import tl.TLValue;

public class LTNode implements TLNode {

  private TLNode lhs;
  private TLNode rhs;

  public LTNode(TLNode lhs, TLNode rhs) {
    this.lhs = lhs;
    this.rhs = rhs;
  }

  @Override
  public TLValue evaluate() {

    TLValue a = lhs.evaluate();
    TLValue b = rhs.evaluate();

    if(a.isNumber() && b.isNumber()) {
      return new TLValue(a.asDouble() < b.asDouble());
    }

    if(a.isString() && b.isString()) {
      return new TLValue(a.asString().compareTo(b.asString()) < 0);
    }

    throw new RuntimeException("illegal expression: " + this);
  }

  @Override
  public String toString() {
    return String.format("(%s < %s)", lhs, rhs);
  }
}
```

As you can see, only 2 numerical values or 2 string values can be compared to each other. Edit the tree grammar file to include this new node:

```
view plain print ?
expression returns [TLNode node]
  :  ...
  |  ^('<' a=expression b=expression) {node = new LTNode($a.node, $b.node);}
  | ...
  ;
```

PrintlnNode.java

Let's also create one of the built-in functions: the `PrintlnNode`. We'll place all the functions in their own package: `tl.tree.functions`, so create a folder in **src/main/tl/tree/** called **functions** an put the following class in it:

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

import tl.TLValue;
import tl.tree.TLNode;
import java.io.PrintStream;

public class PrintlnNode implements TLNode {

  private TLNode expression;
  private PrintStream out;

  public PrintlnNode(TLNode e) {
```

```
    this(e, System.out);
  }

  public PrintlnNode(TLNode e, PrintStream o) {
    expression = e;
    out = o;
  }

  @Override
  public TLValue evaluate() {
    TLValue value = expression.evaluate();
    out.println(value);
    return TLValue.VOID;
  }
}
```

And adjust the tree grammar accordingly:

```
view plain print ?

functionCall returns [TLNode node]
  :  ...
  |  ^(FUNC_CALL Println expression?) {node = new PrintlnNode($expression.node);}
  |  ...
  ;
```

Also add the import statement: `import tl.tree.functions.*;` to the **@header { ... }** section of our tree grammar.

And edit the rule `statement` from the tree grammar to look like this:

```
view plain print ?

statement returns [TLNode node]
  :  assignment     {node = $assignment.node;}
  |  functionCall   {node = $functionCall.node;}
  |  ifStatement    {node = $ifStatement.node;}
  |  forStatement   {node = $forStatement.node;}
  |  whileStatement {node = $whileStatement.node;}
  ;
```

Now edit the **test.tl** file with the followinf source:

```
view plain print ?

if 4 < 3 do
  println(1);
else if 4 < 5 do
  println(2);
else do
  println(3);
end
```

And if you now execute the `run` target again:

```
$ ant run

...

run:
     [echo] Running the main class...
     [java] 2.0
     [java] VOID
```

You see that the block inside `else if` is executed and `2.0` is printed to the console. The `VOID` is printed since the entire script did not specifically return a value, so that is correct.

AssignmentNode.java

Now a trickier node: the assignment node. As you can see by looking at the tree grammar rule:

```
view plain print ?

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

If there are no indexes, it's easy: we'd just let the scope assign (or re-assign) the identifier to its new value, the `expression`.

But an identifier can have one or more indexes after it. For example, this is valid syntax:

```
view plain print ?

arr = [[1,2,3], [4,5,6], [7,8,9]];
arr[2][0] = arr[2][0] * 6;
assert(arr == [[1,2,3], [4,5,6], [42,8,9]]);
```

in which case, we'll first have to iterate over the indexes to get a hold of the value `[7,8,9]` (the before last index) and then reassign index `0` to become `arr[2][0] * 6`, which equals `7 * 6`.

The code could look like this:

```
view plain print ?

package tl.tree;

import tl.Scope;
import tl.TLValue;

import java.util.ArrayList;
import java.util.List;

public class AssignmentNode implements TLNode {

  protected String identifier;
  protected List<TLNode> indexNodes;
  protected TLNode rhs;
  protected Scope scope;

  public AssignmentNode(String i, List<TLNode> e, TLNode n, Scope s) {
    identifier = i;
    indexNodes = (e == null) ? new ArrayList<TLNode>() : e;
    rhs = n;
    scope = s;
  }

  @Override
  public TLValue evaluate() {

    TLValue value = rhs.evaluate();

    if (value == TLValue.VOID) {
      throw new RuntimeException("can't assign VOID to " + identifier);
    }

    if (indexNodes.isEmpty()) { // a simple assignment
      scope.assign(identifier, value);
    }
    else { // a possible list-lookup and reassignment

      TLValue list = scope.resolve(identifier);

      // iterate up to `foo[x][y]` in case of `foo[x][y][z] = 42;`
      for (int i = 0; i < indexNodes.size() - 1 && list != null; i++) {
        TLValue index = indexNodes.get(i).evaluate();

        if (!index.isNumber() || !list.isList()) { // sanity checks
          throw new RuntimeException("illegal statement: " + this);
        }

        int idx = index.asLong().intValue();
        list = list.asList().get(idx);
      }
      // list is now pointing to `foo[x][y]` in case of `foo[x][y][z] = 42;`

      // get the value `z`: the last index, in `foo[x][y][z] = 42;`
      TLValue lastIndex = indexNodes.get(indexNodes.size() - 1).evaluate();

      if (!lastIndex.isNumber() || !list.isList()) { // sanity checks
        throw new RuntimeException("illegal statement: " + this);
      }

      // re-assign `foo[x][y][z]`
      List<TLValue> existing = list.asList();
      existing.set(lastIndex.asLong().intValue(), value);
    }

    return TLValue.VOID;
  }

  @Override
  public String toString() {
    return String.format("(%s[%s] = %s)", identifier, indexNodes, rhs);
  }
}
```

And some adjustments inside the tree grammar:

```
view plain print ?

assignment returns [TLNode node]
  :  ^(ASSIGNMENT i=Identifier x=indexes? e=expression)
     {node = new AssignmentNode($i.text, $x.e, $e.node, currentScope);}
  ;

expression returns [TLNode node]
```

```
  :  ...
  |  lookup {node = $lookup.node;}
  ;

lookup returns [TLNode node]
  :  ^(LOOKUP functionCall indexes?)
  |  ^(LOOKUP list indexes?)
  |  ^(LOOKUP expression indexes?)
  |  ^(LOOKUP i=Identifier x=indexes?)
      {
        node = ($x.e != null)
          ? new LookupNode(new IdentifierNode($i.text, currentScope), $x.e)
          : new IdentifierNode($i.text, currentScope);
      }
  |  ^(LOOKUP String indexes?)
  ;

indexes returns [List<TLNode> e]
@init {e = new ArrayList<TLNode>();}
  :  ^(INDEXES (expression {e.add($expression.node);})+)
  ;
```

As you can see in the `lookup` rule, a variable lookup can be, just as with an assignment, as simple as fetching a variable from the scope, but it can also have a couple indexes after it. Here's an implementation for both the `Lookup`- and `IdentifierNode`s:

LookupNode.java

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

import tl.TLValue;
import java.util.ArrayList;
import java.util.List;

public class LookupNode implements TLNode {

  private TLNode expression;
  private List<TLNode> indexes;

  public LookupNode(TLNode e, List<TLNode> i) {
    expression = e;
    indexes = i;
  }

  @Override
  public TLValue evaluate() {

    TLValue value = expression.evaluate();

    List<TLValue> indexValues = new ArrayList<TLValue>();

    for (TLNode indexNode : indexes) {
      indexValues.add(indexNode.evaluate());
    }

    for(TLValue index : indexValues) {

      if(!index.isNumber() || !(value.isList() || value.isString())) {
        throw new RuntimeException("illegal expression: " +
            expression + "[" + index + "]");
      }

      int idx = index.asLong().intValue();

      if(value.isList()) {
        value = value.asList().get(idx);
      }
      else if(value.isString()) {
        value = new TLValue(String.valueOf(value.asString().charAt(idx)));
      }
    }

    return value;
  }
}
```

As you can see, I also added support for indexing a string literal: you can do `ch = ("abc")[1];` after which `ch` equals `"b"`.

IdentifierNode.java

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

import tl.Scope;
import tl.TLValue;

```

```
06. public class IdentifierNode implements TLNode {
07.
08.   private String identifier;
09.   private Scope scope;
10.
11.   public IdentifierNode(String id, Scope s) {
12.     identifier = id;
13.     scope = s;
14.   }
15.
16.   @Override
17.   public TLValue evaluate() {
18.     TLValue value = scope.resolve(identifier);
19.     if(value == null) {
20.       throw new RuntimeException("no such variable: " + this);
21.     }
22.     return value;
23.   }
24.
25.   @Override
26.   public String toString() {
27.     return identifier;
28.   }
29. }
```

Wrap up

If you now put the following source inside **test.tl**:

```
view plain print ?
01. a = 21;
02. b = a;
03. println(a + b);
```

or:

```
view plain print ?
01. a = 5;
02.
03. if a < 4 do
04.   a = 1;
05. else if a < 0 do
06.   a = 2;
07. else do
08.   a = 42;
09. end
10.
11. println(a);
```

You'll see `42.0` being printed to the console in both cases, after you executed the `run` Ant target.

That pretty much concludes the technical part of my tutorial about ANTLR and how to create a small, dynamically typed scripting language with it. The last part of this series includes a download link containing *all* sources (so including the missing node-classes) and some recommendations to continue from here on. **Continue reading**.

You can download the sources created up to this point by clicking **here**.

Posted by Bart Kiers at **7:18 AM**
Labels: **antlr**, **dsl**, **java**, **parsing**

2 comments:

Anonymous said...

node from line 0:0 mismatched tree node: IF expecting IF

what's the error here? thanks!

2:39 AM

Bart Kiers said...

No idea. You're probably not following the tutorial exactly. Are you using the same ANTLR version?

10:30 AM

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)