Bart's blog

A blog about software development.

THURSDAY, MARCH 10, 2011

9. Room for improvement

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

Index

Introduction

Welcome to the final part of the tutorial. If you want to try to finish (some of) the implementation of the *TL*-node classes on your own, make sure you're up to date with the latest source files developed so far: download them **here**. You'll want to test your implementation using the test script below.

Complete source

In order to test all functionality of TL, replace the contents of the **test.tl** file with the following source:

view plain print ?

```
/*
    A script for testing Tiny Language syntax.
*/

// boolean expressions
assert(true || false);
assert(!false);
assert(true && true);
assert(!true || !false);
assert(true && (true || false));

// relational expressions
assert(1 < 2);
assert(666 >= 666);
assert(-5 > -6);
assert(0 >= -1);
assert('a' < 's');
assert('sw' <= 'sw');

// add
assert(1 + 999 == 1000);
assert([1] + 1 == [1,1]);
assert(2 - -2 == 4);
assert(-1 + 1 == 0);
assert(1 - 50 == -49);
assert([1,2,3,4,5] - 4 == [1,2,3,5]);

// multiply
assert(3 * 50 == 150);
assert(4 / 2 == 2);
assert(1 / 4 == 0.25);
assert(999999 % 3 == 0);
assert(-5 * -5 == 25);
assert([1,2,3] * 2 == [1,2,3,1,2,3]);
assert('ab'*3 == "ababab");

// power
assert(2^10 == 1024);
assert(3^3 == 27);

// for- and while statements
a = 0;
```

```
for i=1 to 10 do
  a = a + i;
end
assert(a == (1+2+3+4+5+6+7+8+9+10));

b = -10;
c = 0;
while b < 0 do
  c = c + b;
  b = b + 1;
end
assert(c == -(1+2+3+4+5+6+7+8+9+10));

// if
a = 123;
if a > 200 do
  assert(false);
end

if a < 100 do
  assert(false);
else if a > 124 do
  assert(false);
else if a < 124 do
  assert(true);
else do
  assert(false);
end

if false do
  assert(false);
else do
  assert(true);
end

// functions
def twice(n)
  temp = n + n;
  return temp;
end

def squared(n)
  return n*n;
end

def squaredAndTwice(n)
  return twice(squared(n));
end

def list()
  return [7,8,9];
end

assert(squared(666) == 666^2);
assert(twice(squared(5)) == 50);
assert(squaredAndTwice(10) == 200);
assert(squared(squared(squared(2))) == 2^2^2^2);
assert(list() == [7,8,9]);
assert(size(list()) == 3);
assert(list()[1] == 8);

// naive bubble sort
def sort(list)
  while !sorted(list) do
  end
end
def sorted(list)
  n = size(list);
  for i=0 to n-2 do
    if list[i] > list[i+1] do
      temp = list[i+1];
      list[i+1] = list[i];
      list[i] = temp;
      return false;
    end
  end
  return true;
end
numbers = [3,5,1,4,2];
sort(numbers);
assert(numbers == [1,2,3,4,5]);

// resursive calls
def fib(n)
  if n < 2 do
    return n;
  else do
    return fib(n-2) + fib(n-1);
  end
end
sequence = [];
for i = 0 to 10 do
  sequence = sequence + fib(i);
end
assert(sequence == [0,1,1,2,3,5,8,13,21,34,55]);

// lists and lookups, `in` operator
n = [[1,0,0],[0,1,0],[0,0,1]];
p = [-1, 'abc', true];
```

```
142.
143.  assert('abc' in p);
144.  assert([0,1,0] in n);
145.  assert(n[0][2] == 0);
146.  assert(n[1][1] == n[2][2]);
147.  assert(p[2]);
148.  assert(p[1][2] == 'c');
```

The script above is just a number of `assert` calls that fail whenever the expression provided as parameter evaluates to false. So, after running the script above, you'll know that everything went okay if you didn't see anything printed to the console.

If you don't want to implement all node classes yourself, download the complete implementation of *TL* **here**. Execute the Ant task `run` to execute the **test.tl** file.

Improvements

First of all, realize that all source files posted here are for educational purposes only. In no way are they production worthy: there are no unit tests, I haven't re-factored anything and there are very little comments in the source.

That said, where to go from here?

Imagine the code *was* properly unit tested, there's proper exception handling etc. Then the next step would be to provide better/friendlier (error) messages to the user (programmer) if some illegal syntax is used. To find out more about that, please see: **ANTLR 3.0 Error Reporting and Recovery** on the official ANTLR Wiki.

After that, the language could be extended by some sort of structs/classes. To find out how to do that, get a hold of **Language Implementation Patterns** of the same author of the (almost) mandatory book **The Definitive ANTLR Reference**, Terence Parr, creator of ANTLR.

That's it for now. If you have questions about the source of this tutorial, or found an error, please post a message here below any of my blogs. If you are writing your own language using ANTLR and are stuck on that, please use the **ANTLR mailing list**, or post a question on **StackOverflow** where I frequently answer ANTLR related questions.

Bye!

PS. For clarity, download the complete sources **here**.

Posted by Bart Kiers at **8:03 AM**
Labels: **antlr**, **dsl**, **java**, **parsing**

8 comments:

Anonymous said...

Hi!

I really enjoyed reading your blog posts! They are well organized and a good starting point for working with antlr!

I would definitly enjoy reading some more (e.g. includes) but anyway:

Thanks for sharing!

5:45 AM

Mateo said...

Really great stuff Bart! It is much appreciated.

7:54 PM

Anonymous said...

Thanks much for the great blog post! It was a huge help in clarifying how to use the AST to execute the DSL language constructs.

5:22 PM

Anonymous said...

That was a really great tutorial! Following your instructions, wandering off into own syntax and language constructs I really learned a lot. I also like your style of programming and the overall architecture.

I have several questions:

-What is in your experience the better approach for implementing comparison nodes: by tedious code duplication or by some switching in the evaluate part of a more general class?

-Is the private scoping of your functions intended? The way you implemented it, they're no closures with lexical scoping, but rather have their own private scope without access to outer/global scopes.
What you do is:
1. Register function in function table, when it's found in the parser. Give it a new scope.
2. Pass the function table to the tree walker. When a function is found in the AST call it by making a shallow copy of it's params, assigning the actual parameters to it and reinvoke the tree walker with the altered function scope.
That way, functions can never see the global scope or be nested closure-style. Which is a reasonable approach! Did I get it right, that this is what you inteded?

-Do you think it would be a good idea to write the tree-package stuff in a more succinct way using Scala?

a minor thing:
-your "a" < "b" operator should use compareTo < / > 0 instead of 1 / -1

Thanks again!
Felix (funlix@funlix.de)

6:40 AM

Bart said...

> -What is in your experience the better approach for implementing comparison nodes ...

Personally, I like to have separate classes for each operation. You could reduce the duplication of code by creating a BinaryNode that holds references to two othe TLNode classes (left-hand- and right-hand-size nodes). Then AddNode, EqNode, ModNode etc. all would extend this BinaryNode and make use of the super's left and right child nodes.

> -Is the private scoping of your functions intended? The way you implemented it ...

Correct: that is what I intended. I did it like that to keep the implementation as simple as possible.

> -Do you think it would be a good idea to write the tree-package stuff in a more succinct way using Scala?

I'm not familiar with Scala unfortunately (it looks like a real neat language!). But I agree with your sentiment that a tree grammar is a bit of an odd way to walk the AST (assuming that was/is your sentiment!): a tree grammar is more or less a copy of the parser rules making it error prone when editing existing parser rules and forgetting to edit the corresponding tree grammar.
At the moment (with ANTLR v3) it's either a tree grammar, or walk the tree manually, in which case I prefer using a tree grammar. The next big release of ANTLR (v4), will not have any tree walkers: you will be able to iterate of the AST's in a far easier way. More info on v4: http://www.antlr.org/wiki/display/ANTLR4/Home

> -your "a" < "b" operator should use compareTo < / > 0 instead of 1 / -1

Good catch, thanks!
I fixed it in the final zip of part 9 and replaced it.

Regards,

Bart.

5:11 AM

jjuiddong said...

your article is so beautiful and elegance
and useful

4:25 AM

Unknown said...

Hello Bart,

Thanks for the series it has been really helpful to me in understanding how to use ANTLR. I have been searching the net but most of them use the out-dated version. These have really helped me in getting started toward writing my own DSL for my upcoming project.

However I don't understand the use of tree walker. I have been following
http://www.codeproject.com/Articles/18880/State-of-the-Art-Expression-Evaluation
and
http://ncalc.codeplex.com/
so far and these use the visitor pattern which I think are doing the same thing, in terms of evaluation Am I correct in my understanding?

Thanks once again for your posts.

Muffadal J.

9:51 AM

Bart Kiers said...

Hi Muffadal, yes, you are correct: a tree walker simply "visits" the AST produced by the parser similar to the way a manual visitor pattern would be implemented.

10:49 AM

Post a Comment

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

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