A Tale of Two Grammars

Posted on November 22, 2012

A Tale of Two Grammars

In days gone by, language creation was a highly sophisticated art practiced by wizards. The tools of the trade were flex, lex, yacc and bison. In the Java technology stack, powerful alternatives to these venerable technologies have emerged. In particular, Antlr and JavaCC are particularly capable.

While I am a fan of JavaCC, years ago having implemented a dialect of Perl which runs on the Java virtual machine, not much seems to have changed for JavaCC in the last 10 years. It seems stagnant and has passed through many hands in search of an enthusiastic steward. In practice, JavaCC has mostly disappeared from the lexer/parser generator scene.

The Antlr project on the other hand, has a vibrant community with an extremely dedicated champion; Terence Parr. Terence has spent 25 years perfecting and simplifying the task of language creation. 25 years folks. Antlr version 4 is a remarkable evolutionary step.

Creating your own languages is amazingly simple. So simple, we will do so in a single blog post.

ANTLR V4.0

Recently, I needed a foundation on which to create a new language. Initially I reached for the most recent and stable version of the ANTLR, however when I read about the new features in ANTLR v4.0, I was immediately attracted to it despite its beta status.

ANTLR 4 resonated with me and here's why:

Standards

Grammars are written more or less using EBNF or Extended Backus-Naur form. This isn't new in ANTLR 4, but bears mentioning. You want a grammar description to be as portable as possible. Sticking with standards is the surest way to achieve this.

Adaptive LL(*)

Terence Parr states:

"As luck would have it, I came up with a cool new version called adaptive LL(*) that pushes all of the grammar analysis effort to runtime."

It is amazing how luck always seems to follow large amounts of blood, sweat, tears and dedication.

This feature allows for a more natural expression of a grammar.

```
1 expression
2 : expression '*' expression
3 | expression '+' expression
4 | INT;
```

Adaptive LL(*) is kind of like the JIT compiler in Java can make aggressive and situational code optimizations which are impossible to do through static analysis. I think this is the right direction.

Automatic Parse Tree Generation

In ANTLR 4, manual creation of the parse tree is unnecessary and I hope that manual tree generation goes away. Personally, I always found that writing a Grammar feeding into a Tree grammar to be overly complex and unnecessary.

ANTLR 4 automatically builds these trees for you. Simply define the nodes within this tree which matter to your language implementation.

This makes life much simpler and allows a much cleaner break between the language specification and its implementation.

I say "kind of" here because you do have the ability to attach labels to an alternative within that rule.

```
expression
   : expression '*' expression      # Multiplication
   | expression '+' expression      # Addition
   | INT                            # Integer;
```

This will create a node for each alternative given that these names do not conflict with the name of a rule.

Clear Separation

I mentioned this earlier, but it deserves more discussion. Grammars are cleanly separated from implementation. You can walk the tree via the traditional visitor pattern or via a listener paradigm.

When implementing your ANTLR application you simply extend from the base listener or visitor (depending on whether you choose listener or visitor for tree traversal) and define those events or node visitations with which your application is concerned.

A Simple Calculator

The Calculator is a wonderful example for a number of reasons:

- **Simplicity** – Everyone understands the concept of a calculator.
- **Reference Examples** – Calculators are the "Hello World" of the lexer/parser generator world. As such, plenty of other examples exist for cross reference.

ANTLR Calculator V3

A really good ANTLR v3 example of a Calculator can be found here. I like this example because it's simple. It's tightly bound to CSharp, and complex enough to display the advantages that ANTLR v4 brings to the table.

The base grammar isn't too bad:

```
grammar SimpleCalc;

tokens {
PLUS   = '+' ;
MINUS  = '-' ;
```

```
MULT   = '*' ;
DIV    = '/' ;
RPAREN = ')' ;
LPAREN = '(' ;
ASSIGN = '=' ;
}

/*----------------
* PARSER RULES
*----------------*/

prog :       stat+;
stat :       expr NEWLINE
       |      ID ASSIGN expr NEWLINE
       |      NEWLINE;                   //Do nothing

expr   :      multExpr ((PLUS | MINUS )multExpr)*;

multExpr:    atom ((MULT | DIV) atom )*;

atom   :      INT
       |      ID
       |      LPAREN expr RPAREN;

/*----------------
* LEXER RULES
*----------------*/

ID     :      ('a'..'z'|'A'..'Z')+;
INT    :      '0'..'9'+;
NEWLINE :    '\r'?'\n';
WS     :      (' '|'\t'|'\n'|'\r')+;
```

However, when you inject the actions and target language of CSharp, its gets messier and pretty much has no reuse without a major overhaul.

```
grammar SimpleCalc3;

options
{
language=CSharp;
}
tokens {
PLUS   = '+' ;
MINUS  = '-' ;
MULT   = '*' ;
DIV    = '/' ;
RPAREN = ')' ;
LPAREN = '(' ;
ASSIGN = '=' ;
}

@members
{
public static System.Collections.Generic.Dictionary IDTable = new System.Collections.
public static void Main(string[] args)
{
  Console.WriteLine("type '@' to quit...");
  string line = "";
  while (true)
  {
      line = Console.ReadLine();
      if (line.Contains("@"))
        break;

      SimpleCalc3Lexer lex = new SimpleCalc3Lexer(new ANTLRStringStream(line+Environm
      CommonTokenStream tokens = new CommonTokenStream(lex);
      SimpleCalc3Parser parser = new SimpleCalc3Parser(tokens);

      try
```

```
      {
        parser.prog();
      }
      catch (RecognitionException e)
      {
        Console.Error.WriteLine(e.StackTrace);
      }
  }
}

}

/*----------------
* PARSER RULES
*----------------*/

prog : stat+;
stat : expr NEWLINE               {System.Console.WriteLine($expr.value);}
|      ID ASSIGN expr NEWLINE           {IDTable[$ID.Text] =$expr.value;}
|      NEWLINE;                   //Do nothing

expr returns[int value]
:      a=multExpr {$value = $a.value;} (
PLUS b=multExpr {$value+=$b.value;}
|
MINUS b=multExpr{$value-=$b.value;})*;

multExpr returns[int value]
:      a=atom {$value = $a.value;} (
MULT b=atom {$value*=$b.value;}
|
DIV b=atom{$value/=$b.value;})*;

atom returns[int value]
:      INT                        {$value = int.Parse($INT.text);}
|      ID                         {if (IDTable.ContainsKey($ID.Text)){$value = IDTabl
|      LPAREN expr RPAREN          {$value = $expr.value;};

/*----------------
* LEXER RULES
*----------------*/

ID :   ('a'..'z'|'A'..'Z')+;

INT :  '0'..'9'+;

NEWLINE :    '\r'?'\n';

WS :   (' '|'\t'|'\n'|'\r')+ {Skip();};
```

ANTLR Calculator V4

Using the previous grammar as a base, let's rewrite and extend this grammar for v4. I'm also taking the liberty of adding a new command "print" so that I can view results on the fly. First things first though, my environment of choice is Eclipse, so let's set up a workspace.

Eclipse Setup

One downside of using ANTLR 4 is that the graphical IDE tools are not quite there yet. The good news is that command line equivalents are easily ant scripted. We'll get into that in a minute.

In order to set up eclipse:

1. Create a new java workspace called AntlrPlayground.

2. Create a package called “calculator4”.
3. Create a folder called “test” to store my future test cases.
4. Create a folder called “gen” to store generated code.
5. Right click the “gen” folder and set it to derived so that Eclipse knows that the contents of this folder are derived from another asset (ANTLR).
6. Right click the “gen” directory and select “Build Path” -> “Use as Source Folder”.
7. Create a folder called “lib” to store our referenced antlr support jar.
8. Download the latest antlr4 jar from github.
9. Save it into the “lib” directory and rename it to “antlr4.jar” to remain consistent with the build script we’ll be using later.
10. Right click the antlr4.jar and select “Build Path” -> “Add to Build Path”

Create “build.xml” the workspace and insert the following code into it:

```
<project name="calculator4" default="generate" basedir=".">
    <property name="src" location="src" />
    <property name="gen" location="gen" />
    <property name="src" location="src" />
    <property name="package" value="calculator4" />

    <path id="classpath">
        <pathelement location="lib/antlr4.jar" />
        <pathelement location="bin" />
    </path>

    <target name="generate" depends="clean">
        <mkdir dir="${gen}/${package}" />
        <java classname="org.antlr.v4.Tool" classpathref="classpath" fork="true">
            <arg value="-o" />
            <arg path="${gen}/${package}" />
            <arg value="-lib" />
            <arg path="${src}/${package}" />
            <arg value="-listener" />

            <arg value="${src}/${package}/Calculator4.g4" />
        </java>
    </target>

    <target name="showtree">
        <input message="Enter Script To Test:" addproperty="test.script" defaultvalue
        <java classname="org.antlr.v4.runtime.misc.TestRig" classpathref="classpath"
            <arg value="${package}.Calculator4" />
            <arg value="program" />
            <arg value="-gui" />
            <arg value="${test.script}" />
        </java>
    </target>

    <target name="clean">
        <delete file="${gen}/*" includeemptydirs="true" />
    </target>
</project>
```

Our Grammar

```
grammar Calculator4;

@header
{
package calculator4;
}

// PARSER
```

```
program : ((assignment|expression) ';')+;

assignment : ID '=' expression;

expression
    : '(' expression ')'               # parenExpression
    | expression ('*'|'/') expression   # multOrDiv
    | expression ('+'|'-') expression   # addOrSubtract
    | 'print' arg (',' arg)*            # print
    | STRING                            # string
    | ID                                # identifier
    | INT                               # integer;

arg : ID|STRING;

// LEXER

STRING : '"' (' '..'~')* '"';
ID     : ('a'..'z'|'A'..'Z')+;
INT    : '0'..'9'+;
WS     : [ \t\n\r]+ -> skip ;
```

Is it just me or is this absurdly simple? Also, we have extended the capabilities to include print capabilities not found in the original v3 grammar. Let's slice and dice this example:

```
grammar Calculator4;
```

In the code snippet above, we are associating our grammar with the name "Calculator4"

```
@header
{
package calculator4;
}
```

This contains our only java specific piece of code. It might be possible to use the <<standin>> convention to avoid even this, however, I'm not sure how to do this yet.

```
// PARSER
program : ((assignment|expression) ';')+;
```

Our parser rules begin here. Our top level rule is a program. Programs consist of one or more assignments or expressions separated by a mandatory ';'. We could "fancy grammar" our way out of this draconian requirement, however, for our simple example, this is fine.

```
assignment : ID '=' expression;
```

Assignments consist of an identifier, the equals token and an expression. I separated this out from the base expression because:

```
A = 1 + 2 * 3
```

Was causing a grammar ambiguity where the expression was being interpreted as:

(A = 1) + (2 * 3)

Separation of the rules keep this unwanted side-effect from occurring.

```
expression
    : '(' expression ')'               # parenExpression
    | expression ('*'|'/') expression   # multOrDiv
    | expression ('+'|'-') expression   # addOrSubtract
    | 'print' arg (',' arg)*            # print
```

```
    | STRING                  # string
    | ID                      # identifier
    | INT                     # integer;
```

Here we can see the power of ANTLR v4. Precedence is achieved through stating the alternatives in the order of descending based off of precedence. Thus, parenthesis will be evaluated before multiplication/division, etc...

```
<span style="text-decoration: underline;">arg</span> : ID|STRING;
```

The print statement takes a list of one or more arguments separated by comma tokens. I separate this out because it creates an ArgContext object which preserves the order of these arguments and filters out the unwanted ',' tokens. From there I simply have to determine whether I am dealing with an identifier or a string literal.

```
// LEXER

STRING : '"' (' '..'~')* '"';
ID  :   ('a'..'z'|'A'..'Z')+;
INT :   '0'..'9'+;
WS  :   [ \t\n\r]+ -> skip ;
```

Here we see the standard lexer tokens. The CAPS indicate that they are tokens. In the ASCII table, the printable characters range from ' ' to '~'. This says that strings are identified by begin and end double quotes, with printable characters within. No fancy escape sequences for this parser.

IDs are one or more alphabetical sequences. INTs are sequences of one or more numbers and we skip all spaces, tabs, carriage returns and linefeeds thanks to our single action '->'.

More complex situations might require us to write this to an alternate channel, but that goes well beyond the scope of this discussion. Let's just skip it for now.

The Interpreter

It doesn't do much yet because no one is listening yet. Here it is:

calculator.java:

```
package calculator4;

import org.antlr.v4.runtime.ANTLRFileStream;
import org.antlr.v4.runtime.CommonTokenStream;
import org.antlr.v4.runtime.ParserRuleContext;
import org.antlr.v4.runtime.Token;

public class calculator
{
  public static void main(String[] args) throws Exception
  {
    Calculator4Lexer lexer = new Calculator4Lexer(new ANTLRFileStream(args[0]));
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    Calculator4Parser p = new Calculator4Parser(tokens);
    p.setBuildParseTree(true);
    p.addParseListener(new CalculatorListener());
    ParserRuleContext<Token> t = p.program();
  }
}
```

This code performs the following:

1. Create the lexer from the supplied filename in argument 0.
2. Parse the token stream with the lexer.

3. Create the parser with this token stream.
4. Add a listener of type CalculatorListener to the parse tree.
5. Call the program.

The Listener

Here is our listener.

CalculatorListener.java:

```
1  package calculator4;
2
3  import java.util.HashMap;
4  import java.util.Map;
5  import java.util.Stack;
6
7  import calculator4.Calculator4Parser.AddOrSubtractContext;
8  import calculator4.Calculator4Parser.ArgContext;
9  import calculator4.Calculator4Parser.AssignmentContext;
10 import calculator4.Calculator4Parser.IntegerContext;
11 import calculator4.Calculator4Parser.MultOrDivContext;
12 import calculator4.Calculator4Parser.PrintContext;
13
14 public class CalculatorListener extends Calculator4BaseListener
15 {
16   public Stack<Integer>       stack = new Stack<Integer>();
17   public Map<String, Integer> sym   = new HashMap<String, Integer>();
18
19   public void exitPrint(PrintContext ctx)
20   {
21     for (ArgContext argCtx : ctx.arg())
22     {
23       if (argCtx.ID() != null)
24       {
25         System.out.print(sym.get(argCtx.ID().getText()));
26       }
27       else
28       {
29         System.out.print(argCtx.STRING().getText().replaceAll("^\"|\"$", ""));
30       }
31     }
32     System.out.println();
33   }
34
35   public void exitAssignment(AssignmentContext ctx)
36   {
37     sym.put(ctx.ID().getText(), stack.pop());
38   }
39
40   public void exitInteger(IntegerContext ctx)
41   {
42     stack.push(new Integer(ctx.INT().getText()));
43   }
44
45   public void exitAddOrSubtract(AddOrSubtractContext ctx)
46   {
47     Integer op1 = stack.pop();
48     Integer op2 = stack.pop();
49     if (ctx.getChild(1).getText().equals("-"))
50     {
51       stack.push(op2 - op1);
52     }
53     else
54     {
55       stack.push(op1 + op2);
56     }
57   }
```

```
58
59    public void exitMultOrDiv(MultOrDivContext ctx)
60    {
61      Integer op1 = stack.pop();
62      Integer op2 = stack.pop();
63      if (ctx.getChild(1).getText().equals("/"))
64      {
65        stack.push(op2 / op1);
66      }
67      else
68      {
69        stack.push(op2 * op1);
70      }
71    }
72  }
```

Basically we keep an internal symbol table “sym” for identifier lookup and a stack of integers for the results of our calculations.

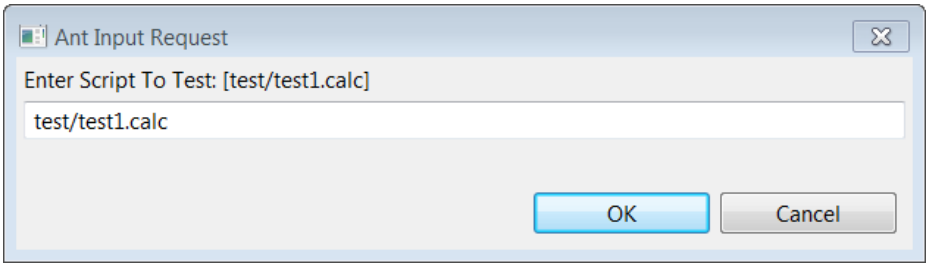
There are many nodes in our parse tree, but these are the only ones which matter to our application. Also notice how contexts are introspected.

Addition and subtraction are differentiated by introspecting the context’s child directly via ctx.getChild(1).getText() while the print statement can have a variable number of children of assorted types. So there we simply use the arg() list to get only those of type ArgContext. From there we simply determine if they are an ID or a STRING to get their value from the symbol table or by trimming the “ from the string literal.

Ant Build

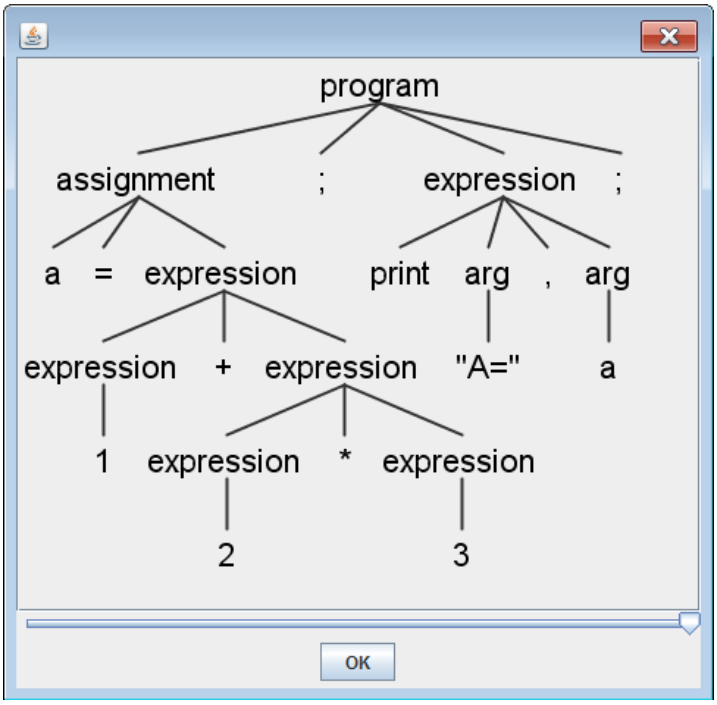
Our ant build contains a very useful target called “showtree”. Showtree will execute the ant utility in GUI mode which will display the tree of the user supplied grammar.
First, it will query the user for the test program.

Ant Dialog:



Then will display the tree.

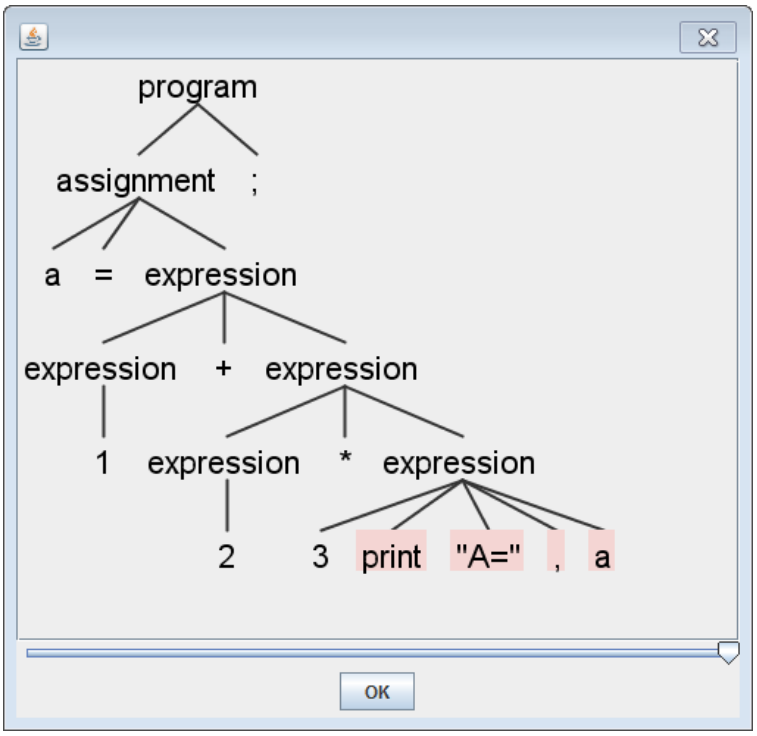
Sample Tree:



Any errors will be in red such as this case of the missing semi-colon in the code:

```
a=1+2*3
print "A=", a;
```

Sample Error Tree:



Running this program anyway yields:

line 2:0 mismatched input ‘print’ expecting {‘)’, ‘+’, ‘*’, ‘-‘, ‘/’, ‘;’}

which is a remarkably good error message.

Sample Programs

test1.calc:

```
a=1+2*3;
print "A=", a;
```

Outputs:

```
A=7
```

Notice the interpreter enforced precedence properly.

test2.calc:

```
a=1;
print "1=", a;
b=1+2;
print "1+2=", b;
c=2*3;
print "2*3=",c;
d=100/50;
print "100/50=",d;
e=10-5;
print "10-5=",e;
a=(1+2)*3;
print "(1+2)*3=", a;
```

Outputs:

```
1=1
1+2=3
2*3=6
100/50=2
10-5=5
(1+2)*3=9
```

Conclusion

I hope that this blog entry has proven that writing languages is easier than ever. Also, ANTLR v4 is very promising and looks to greatly simplify our lives in this space.

Rate this:

5 Votes

Share this:

Facebook Twitter Reddit Google Email More

Like

2 bloggers like this.

Related

The birth of a computer language
In "General"

More Fun With ANTLR v4 and Groovy
In "Dex"

ANTLR v4: Lexical Scoping
In "ANTLR"

About patmartin
I am a coder and Data Visualization/Machine Learning enthusiast.
View all posts by patmartin →

This entry was posted in General, Java and tagged antlr, dex, java, lexer, parser. Bookmark the permalink.

14 Responses to *A Tale of Two Grammars*

ted *says:*
March 17, 2013 at 2:25 pm

Dear Sir :
It is a goog example for ANTLR V4. I have tested the example, and I have a question the program need the \n to separate the assignment and expression. But I think that the \n is skip in grammar.
Thanks

Reply

patmartin *says:*
March 18, 2013 at 9:57 pm

Hi Ted. I am pretty sure that if you explicitly specify the characters to skip, the list must be complete. Otherwise it would be impossible to specify a grammar where newlines were significant wouldn't it?

I haven't tested this so take my response with a grain of salt.

Reply

Selamet Hariadi *says:*
April 11, 2013 at 10:48 pm

Hello, how to check a script java? example script to check true or false Array or script code print "helloe world".

Reply

patmartin *says:*
April 16, 2013 at 9:34 pm

Sorry Selamet, I do not quite understand the question. Can you be more specific about what you are trying to do?

Reply

ekim *says:*
April 25, 2013 at 12:15 am

hi
excellent work.
i am trying with antlr4
this is the grammar
but doesn't work fine, could you help me to figure out what is wrong with it.

```
grammar Calculator4;
expresion : expSuma
;
expSuma : expResta (OP_MAS expResta)* #OPSUMA
;
expResta : expProducto (OP_MENOS expProducto)* #OPRESTA
;
expProducto : expCociente (OP_PRODUCTO expCociente)* #OPPRODDUCTO
;
expCociente : expBase (OP_COCIENTE expBase)* #OPCOCIENTE
;
expBase : NUMERO #UNNUMERO
| PARENT_AB! expresion PARENT_CE! #PARENTESIS
;
BLANCO : (' '|'\t') -> skip ;
OP_MAS : '+';
OP_MENOS : '-';
OP_PRODUCTO : '*';
OP_COCIENTE : '/';
PARENT_AB : '(';
PARENT_CE : ')';
NUMERO : ('0'..'9')+('.' ('0'..'9')+)?;
```

Reply

ekim *says:*
April 25, 2013 at 12:44 am

p.d: i am doing this with a visitor

Reply

patmartin *says:*
June 11, 2013 at 4:52 pm

Thanks,

You would be better served with posting this to the ANTLR group on Google Groups or Stack Overflow. I honestly don't have enough time for deep troubleshooting.

Sorry,

– Pat

Reply

Ratul Saha *says:*
May 9, 2013 at 10:00 am

Wonderful description!

I am trying to make a model checker tool similar to PRISM (http://prismmodelchecker.org/). A few example cases has been written in python. There is a template of the backend as well. the grammar specification is ready and now I need ANTLR to recognize the grammar and give me a backend code to run.

My questions:

1. ANTLR4 seems much better than ANTLR3. However, I could not understand how to run this seamlessly with output=Python. Can you please comment on some possible problems to stick to python?

2. When I ship the product source code, it seems I need to ship antlr as well (the parser and lexer depends on antlr jar file, if I am not wrong). Can you please let me know how to do that?

Thanks,

Ratul Saha.
Doctoral Student.

Reply

patmartin *says:*
June 11, 2013 at 2:34 pm

Last I checked ANTLR4 does not support a Python target. If you want Python to be generated, you will unfortunately have to stick with ANTLR3.

You are correct about packaging. You will have to ship out the runtime dependencies with languages generated with ANTLR. In the Java world, this is just a jar and is not an uncommon practice.

– Pat

Reply

Maryam Ghorbani *says:*
July 11, 2014 at 1:31 pm

Hi,
I am also involved with parsing a text file in antlr .Python target is preferred language for my work. I sent an email to Mr.Trence Parr and he replyed, “use antlr 4. we’ll have python2,3 generation out within a week.”
this conversion was for yesterday
U can hope to use antlr 4 for python .

Reply

Pedro Nogueira *says:*
November 20, 2014 at 1:27 am

Hi I enjoyed your example but i cannot put it to work in antlr4 with command line in linux.
Although all my other grammars run when i try the calculator example I get:
Exception in thread “main” java.lang.NoClassDefFoundError: calculator (wrong name: calculator4/calculator)
at java.lang.ClassLoader.defineClass1(Native Method)
at java.lang.ClassLoader.defineClass(ClassLoader.java:800)
at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
at java.net.URLClassLoader.defineClass(URLClassLoader.java:449)
at java.net.URLClassLoader.access$100(URLClassLoader.java:71)

at java.net.URLClassLoader$1.run(URLClassLoader.java:361)
at java.net.URLClassLoader$1.run(URLClassLoader.java:355)
at java.security.AccessController.doPrivileged(Native Method)
at java.net.URLClassLoader.findClass(URLClassLoader.java:354)
at java.lang.ClassLoader.loadClass(ClassLoader.java:425)
at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:308)
at java.lang.ClassLoader.loadClass(ClassLoader.java:358)
at org.antlr.v4.runtime.misc.TestRig.process(TestRig.java:159)
at org.antlr.v4.runtime.misc.TestRig.main(TestRig.java:143)
ABRT problem creation: 'success'

Can you help me?
Thx

Reply

Faizan *says:*
March 8, 2015 at 11:22 pm

Hi. Its my first day with antlr 4 and I'm doing it on my own. I'm certainly missing few things here
1. Where to put the grammer code i.e. what file in which package?
2. There are a lot of things in Java code that are not being resolved e.g. PrintContext, ArgContext and calculator4 imports.

I think there is something which in your thinking is obvious but I certainly don't know anything about it.

Any help would be highly appreciated.

Reply

Joel *says:*
January 14, 2016 at 4:27 am

Great example; well done. I am have been playing with antler4 for a while and really enjoy it compare to my previous use of lexx/yacc. My current work is mostly Java but my personal preference is C# so I have a question about the example. Why did you switch from generating C# in antlers to Java in antler4?

Reply

patmartin *says:*
January 15, 2016 at 2:17 am

All I could find was a reference ANTLR3 grammar with a C# target. I didn't think that the language shift would be too distracting so chalk up the initial reference point to laziness I mostly stick to Java for this kind of work myself.

Reply