Using ANTLR 4 with Python

POSTS

March 15, 2016 - 4 minutes read - 761 words

ANTLR (Another Tool for Language Recognition) is an established tool for writing parsers. It is written in Java, but generates code in a variety of languages, including Python.

There seemed to be a dearth of examples out there using ANTLR 4 with Python, and having used ANTLR only with Java, I was interested to explore how difficult it would be to use. You can see the results in a GitHub repository.

To get started we need to download the ANTLR complete JAR file. We also need to install the necessary Python module. The easiest way is using pip.

```
pip install antlr4-python2-runtime
```

ANTLR Grammar

ANTLR generates code from a grammar, which describes what is valid in the language and how the language is structured. For this example, we are using the "getting started" example (with one small change to make the Python a little clearer. Here is the grammar:

```
// Define a grammar called Hello
grammar Hello;
hi : 'hello' ID ;          // match keyword hello followed by an identifier
ID : [a-z]+ ;              // match lower-case identifiers
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

The comments already explain the pieces of the grammar, so I'll move on to focus on the tool.

Compiling

ANTLR turns this grammar file into two parts: a lexer, which reads the input stream and turns it into tokens; and a parser, which associates the tokens with the elements of the grammar we named above.

To compile the grammar we use the downloaded JAR:

```
java -Xmx500M -cp <path to ANTLR complete JAR> org.antlr.v4.Tool -Dlanguage=Python2 Hello.
```

This command generates `HelloLexer.py`, `HelloParser.py`, and `HelloListener.py`. The listener is a new design element of ANTLR 4 and is designed to make it easier to write code that handles events from the parser, without being impacted if the grammar is modified and re-compiled.

Using

To use the generated Python, we need our own listener class that extends from the generated listener. (Note that even though the generated listener doesn't do anything in its methods, we still inherit from it so our methods will be called correctly).

Here is the hand-written code for this example:

```
from antlr4 import *
from HelloLexer import HelloLexer
from HelloListener import HelloListener
from HelloParser import HelloParser
import sys

class HelloPrintListener(HelloListener):
    def enterHi(self, ctx):
        print("Hello: %s" % ctx.ID())

def main():
    lexer = HelloLexer(StdinStream())
    stream = CommonTokenStream(lexer)
    parser = HelloParser(stream)
    tree = parser.hi()
    printer = HelloPrintListener()
    walker = ParseTreeWalker()
    walker.walk(printer, tree)

if __name__ == '__main__':
    main()
```

The `HelloPrintListener` defines a method that will be called when a "hi" rule is entered while walking the tree. The variable `ctx` holds the current context; we know that it has a method called `ID()` that holds the identifier because that's what that item is called in the grammar.

The `main()` method is mostly boilerplate. First we create a lexer from standard in; we could also use a `FileStream` or `InputStream`. The lexer is used to make a stream of tokens (basically breaking up the input stream on whitespace). This stream of tokens is used to create the parser.

At this point, we call the “hi” method on the parser; by calling this method we tell the parser what rule to start matching with the incoming tokens. At this point, the parser will build a tree from the input; we can then walk this tree to invoke the listener method when the rule is found.

Running

To run the example:

```
$ python Hello.py
hello sir
^D
```

The output is:

```
Hello: sir
```

Extending

Note that as written, the grammar does not handle multiple lines of input. To handle this, we would need to modify the grammar slightly:

```
// Define a grammar called Hello
grammar Hello;
prog : hi* EOF ;          // match one or more instances of hi
hi : 'hello' ID ;         // match keyword hello followed by an identifier
ID : [a-z]+ ;             // match lower-case identifiers
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

This grammar allows for zero or more instances of our hello command. Each one will fire the `enterHi()` rule. We just need to change the `main()` method so it uses `prog` as the start rule.

Conclusion

ANTLR is incredibly powerful and this just touches the surface. Documentation for Python is a little sparse, but the method names are similar to the Java implementation so there’s generally a one-to-one mapping in terms of what needs to be done. In any case, with a working example, the key changes to make real working code would be mostly improvements to the grammar.