ANTLR FAQ

The answers common and interesting questions about ANTLR

STRUMENTA

ANTLR FAQ

We extensively use ANTLR in our work and we frequent the community. So, we have seen and received many questions about this wonderful technology. This document represents a summary with the common issues people encounter when using ANTLR.

Some questions are about basic patterns, some are common misconceptions or issues that people have. There are also a few advanced questions that have stumbled even fellow ANTLR experts.

The questions all come from the ANTLR community; these are real questions made by real people. The answers are sometimes our own or sometimes the best answers from the community itself.
The questions are divided in three sections:

- The first section explains common issues or misconceptions
- The second section contains questions about basic patterns
- The third section talks about advanced issues and limitations of ANTLR

Table of Contents

Common Issues

In this section we see questions about common issues or misconceptions. Read this section to avoid common mistakes and understand the basics of ANTLR.

What does “fragment” mean in ANTLR?

QUESTION

I’ve seen both rules:

```
fragment DIGIT     : ‘0’..’9’;
```

and

```
DIGIT     : ‘0’..’9’;
```

What is the difference?

ANSWER

The difference between a rule and a fragment is that the fragment is not a rule. The lexer will never try to match a fragment on its own. A fragment is essentially syntactic sugar, a shorthand. It is used to make the grammar more readable, so you can reuse the fragment in different rules. Typically, a fragment DIGIT will be used in defining the rules for numbers, such as INTEGER, FLOAT, etc.

This means that a fragment is only visible in the lexer, you cannot use it in parser rules. That is because the parser acts on the results of the lexer, so it will never see any fragment.

Original question from StackOverflow (but not the answer)

When it is EOF needed in ANTLR?

QUESTION

The Hello grammar in the ANTLR4 Getting Started Guide doesn't use EOF anywhere, so I inferred that it's better to avoid explicit EOF if possible. What is the best practice for using EOF? When do you actually need it?

ANSWER

You should include an explicit EOF at the end of your entry rule any time you are trying to parse an entire input file. If you do not include the EOF, it means you are not trying to parse the entire input, and it's acceptable to parse only a portion of the input if it means avoiding a syntax error.

For example, consider the following rule:

```
file     : item* ;
```

This rule means "Parse as many item elements as possible, and then stop." In other words, this rule will never attempt to recover from a syntax error because it will always assume that the syntax error is part of some syntactic construct that's beyond the scope of the file rule. Syntax errors will not even be reported, because the parser will simply stop.

If instead I had the following rule:

```
file     : item* EOF ;
```

In means "A file consists exactly of a sequence of zero-or-more item elements." If a syntax error is reached while parsing an item element, this rule will attempt to recover from (and report) the syntax error and continue because the EOF is required and has not yet been reached.

For rules where you are only trying to parse a portion of the input, ANTLR 4 often works, but not always. The following issue describes a technical problem where ANTLR 4 does not always make the correct decision if the EOF is omitted.

https://github.com/antlr/antlr4/issues/118

Unfortunately the performance impact of this change is substantial, so until that is resolved there will be edge cases that do not behave as you expect.

Original question from StackOverflow

Why I got mismatched Input ‘x’ expecting ‘x’?

QUESTION

I have been starting to use ANTLR and have noticed that it is pretty fickle with its lexer rules. An extremely frustrating example is the following:

```
grammar output;
test: FILEPATH NEWLINE TITLE ;
FILEPATH: (‘A’..’Z’|’a’..’z’|’0’..’9’|’:’|’\\’|’/’|’ ‘|’-’|’_’|’.’)+ ;
NEWLINE: ‘\r’? ‘\n’ ;
TITLE: (‘A’..’Z’|’a’..’z’|’ ‘)+ ;
```

This grammar will not match something like:

```
c:\test.txt
x
```

Oddly if I change `TITLE` to be `TITLE: ‘x’ ;` it still fails this time giving an error message saying “mismatched input ‘x’ expecting ‘x’” which is highly confusing. Even more oddly if I replace the usage of `TITLE` in `test` with `FILEPATH` the whole thing works (although FILEPATH will match more than I am looking to match so in general it isn’t a valid solution for me).

I am highly confused as to why ANTLR is giving such extremely strange errors and then suddenly working for no apparent reason when shuffling things around.

ANSWER

This seems to be a common misunderstanding of ANTLR:

Language Processing in ANTLR:

The Language Processing is done in two strictly separated phases:

- Lexing, i.e. partitioning the text into tokens
- Parsing, i.e. building a parse tree from the tokens

Since lexing must preceed parsing there is a consequence: The lexer is independent of the parser, **the parser cannot influence lexing.**

Lexing

Lexing in ANTLR works as following:

- all rules with uppercase first character are lexer rules
- the lexer starts at the beginning and tries to find a rule that matches best to the current input
- a best match is a match that has maximum length, i.e. the token that results from appending the next input character to the maximum length match is not matched by any lexer rule
- tokens are generated from matches:
    - if one rule matches the maximum length match the corresponding token is pushed into the token stream
    - if multiple rules match the maximum length match the first defined token in the grammar is pushed to the token stream

Example: What is wrong with your grammar

Your grammar has two rules that are critical:

```
FILEPATH: ('A'..'Z'|'a'..'z'|'0'..'9'|':'|'\\'|'/'|' '|'-'|'_'|'.')+ ;
TITLE: ('A'..'Z'|'a'..'z'|' ')+ ;
```

Each match, that is matched by TITLE will also be matched by FILEPATH. And FILEPATH is defined before TITLE: **So each token that you expect to be a title would be a FILEPATH.**

There are two hints for that:

- keep your lexer rules disjunct (no token should match a superset of another).
- if your tokens intentionally match the same strings, then put them into the right order (in your case this will be sufficient).
- if you need a parser driven lexer you have to change to another parser generator: PEG-Parsers or GLR-Parsers will do that (but of course this can produce other problems).

Original question from StackOverflow

How do I get the original text that an antlr4 rule matched?

QUESTION

Using a Java 7 grammar. I want to find methods with a specific name and then just print out that method. I see that I can use the `methodDeclaration` rule when I match. So I subclass Java7BaseListener and override this listener method:

```
@Override public void enterMethodDeclaration(Java7Parser MethodDeclarationContext ctx) { }
```

How do I get the original text out? `ctx.getText()` gives me a string with all the whitespace stripped out. I want the comments and original formatting.

ANSWER

ANTLR's `CharStream` class has a method `getText(Interval interval)` which will return the original source in the give range. The Context object has methods to get the beginning and end. Assuming you have a field in your listener called input which has the `CharStream` being parsed, you can do this:

```
int a = ctx.start.getStartIndex();
int b = ctx.stop.getStopIndex();
Interval interval = new Interval(a,b);
input.getText(interval);
```

If you don't have access to or don't want to keep track of the `CharStream`, use `ctx.start.getInputStream()`

Original question from StackOverflow

Negating inside lexer- and parser rules

QUESTION

How can the negation meta-character, ~, be used in ANTLR's lexer- and parser rules?

ANSWER

Negating can occur inside lexer and parser rules.

Inside lexer rules you can negate characters, and inside parser rules you can negate tokens (lexer rules). But both lexer- and parser rules can only negate either single characters, or single tokens, respectively.

A couple of examples:

lexer rules

To match one or more characters except lowercase ascii letters, you can do:

```
NO_LOWERCASE : ~('a'..'z')+ ;
```

(the negation-meta-char, ~, has a higher precedence than the +, so the rule above equals (~('a'..'z'))+)

Note that 'a'..'z' matches a single character (and can therefore be negated), but the following rule is invalid:

```
ANY_EXCEPT_AB : ~('ab') ;
```

Because 'ab' (obviously) matches 2 characters, it cannot be negated. To match a token that consists of 2 characters, but not 'ab', you'd have to do the following:

```
ANY_EXCEPT_AB
 : 'a' ~'b' // any two chars starting with 'a' followed by any other than 'b'
 | ~'a' .   // other than 'a' followed by any char
 ;
```

parser rules

Inside parser rules, ~ negates a certain token, or more than one token. For example, you have the following tokens defined:

```
A : 'A';
B : 'B';
C : 'C';
D : 'D';
E : 'E';
```

If you now want to match any token except the A, you do:

```
p : ~A ;
```

And if you want to match any token except B and D, you can do:

```
p : ~(B | D) ;
```

However, if you want to match any two tokens other than A followed by B, you cannot do:

```
p : ~(A B) ;
```

Just as with lexer rules, you cannot negate more than a single token. To accomplish the above, you need to do:

```
P
 : A ~B
 | ~A .
 ;
```

Note that the . (DOT) char in a parser rules does ***not*** match any character as it does inside lexer rules. Inside parser rules, it matches any token (A, B, C, D or E, in this case).

Note that you cannot negate parser rules. The following is illegal:

```
p : ~a ;
a : A  ;
```

Original question from StackOverflow

Catching (and keeping) all comments with ANTLR

QUESTION

I'm writing a grammar in ANTLR that parses Java source files into ASTs for later analysis. Unlike other parsers (like JavaDoc) I'm trying to keep all of the comments. This is difficult comments can be used literally anywhere in the code. If a comment is somewhere in the source code that doesn't match the grammar, ANTLR can't finish parsing the file.

Is there a way to make ANTLR automatically add any comments it finds to the AST? I know the lexer can simply ignore all of the comments using either `{skip();}` or by sending the text to the hidden channel. With either of those options set, ANTLR parses the file without any problems at all.

Any ideas are welcome.

ANSWER

No, you'll have to sprinkle your entire grammar with extra comments rules to account for all the valid places comments can occur:

```
...
if_stat
 : 'if' comments '(' comments expr comments ')' comments ...
 ;
...
comments
 : (SingleLineComment | MultiLineComment)* ;
SingleLineComment
 : '//' ~('\r' | '\n')*  ;
MultiLineComment
 : '/*' .* '*/'  ;
```

Original question from StackOverflow

ANTLR4: Using non-ASCII characters in token rules

QUESTION

On page 74 of the ANTRL4 book it says that any Unicode character can be used in a grammar simply by specifying its codepoint in this manner:

‘\uxxxx’

where xxxx is the hexadecimal value for the Unicode codepoint.

So I used that technique in a token rule for an ID token:

```
grammar ID;
id : ID EOF ;
ID : (‘a’ .. ‘z’ | ‘A’ .. ‘Z’ | ‘\u0100’ .. ‘\u017E’)+ ;
WS : [ \t\r\n]+ -> skip ;
```

When I tried to parse this input:

```
Gŭnter
```

ANTLR throws an error, saying that it does not recognize ŭ. (The ŭ character is hex 016D, so it is within the range specified)

What am I doing wrong please?

ANSWER

ANTLR is ready to accept 16-bit characters but, by default, many locales will read in characters as bytes (8 bits). You need to specify the appropriate encoding when you read from the file using the Java libraries. If you are using the `TestRig`, perhaps through alias/script `grun`, then use argument `-encoding utf-8` or whatever. If you look at the source code of that class, you will see the following mechanism:

```
InputStream is = new FileInputStream(inputFile);
Reader r = new InputStreamReader(is, encoding); // e.g., euc-jp or utf-8
ANTLRInputStream input = new ANTLRInputStream(r);
XLexer lexer = new XLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
...
```

Original question from StackOverflow

Basic grammar question

QUESTION

I have this basic grammar.

```
grammar Basic ;
programStatement : 'PROGRAM' progName NEWLINE;
progName : LETTER (LETTER)* ;
LETTER : [a-zA-Z] ;
NEWLINE : '\n' ;
WS : [ \t\r]+ -> skip ;
```

The following one-line file parses the way I expected.

```
PROGRAM MYPROG
```

However, this file does not parse correctly

```
PROGRAM MYPROGRAM
```

with the following output

```
"line 1:11 extraneous input 'PROGRAM' expecting '
'
```

I am probably misunderstanding how to construct the grammar file.

ANSWER

The issue with your grammar is that you are effectively mixing up the lexer and the parser. Basically, you are making the lexer doing only half of its job. The problem is that lexer rule LETTER recognizes just one letter, but the implicit rule PROGRAM recognizes more. The ANTLR lexer prefers to match the longest possible string. So ANTLR always creates the token PROGRAM whenever it founds the sequence of characters PROGRAM.

So you should rewrite the grammar to be like the following.

```
grammar Basic ;
programStatement : PROGRAM progName NEWLINE;
progName : NAME;
PROGRAM : 'PROGRAM';
NAME : LETTER LETTER*;
```

```
fragment LETTER : [a-zA-Z] ;
NEWLINE : '\n' ;
WS : [ \t\r]+ -> skip ;
```

In this grammar a single LETTER is just a fragment. So, it does not create a token, it is just syntactic sugar to avoid repeating `[a-zA-Z]`. We use this fragment to create the rule NAME which matches a whole sequence of letters. Notice that for this to work and to make sure PROGRAM is still matched you have to put the rule NAME after the rule PROGRAM. That is because the sequence PROGRAM can match both NAME and PROGRAM, so ANTLR will choose the one that is defined first.

We also explicitly create the rule PROGRAM. This is not strictly necessary to solve your immediate problem, but it is a good practice since it makes everything clearer.

How does the ANTLR lexer disambiguate its rules (or why does my parser produce “mismatched input” errors)?

QUESTION

When I test this very simple grammar:

```
grammar KeyValues;

keyValueList: keyValue*;
keyValue: key=IDENTIFIER '=' value=INTEGER ';';

IDENTIFIER: [A-Za-z0-9]+;
INTEGER: [0-9]+;

WS: [ \t\r\n]+ -> skip;
```

With the following input:

```
foo = 42;
```

I end up with the following run-time error:

```
line 1:6 mismatched input '42' expecting INTEGER
line 1:8 mismatched input ';' expecting '='
```

Why doesn’t ANTLR recognize 42 as an INTEGER in this case?
It should match the pattern [0-9]+ just fine.

If I invert the order in which INTEGER and IDENTIFIER are defined it seems to work, but why does the order matter in the first place?

ANSWER

In ANTLR, the lexer is isolated from the parser, which means it will split the text into typed tokens according to the lexer grammar rules, and the parser has no influence on this process (it cannot say “*give me an* INTEGER *now*” for instance). It produces a token stream by itself. Furthermore, the parser doesn’t care about the token text, it only cares about the token types to match its rules.

This may easily become a problem when several lexer rules can match the same input text. In that case, the token type will be chosen according to these **precedence rules:**

- First, select the lexer rules which match the longest input substring
- If the longest matched substring is equal to an implicitly defined token (like '='), use the implicit rule as the token type
- If several lexer rules match the same input, choose the first one, based on definition order

These rules are very important to keep in mind in order to use ANTLR effectively.

In the example from the question, the parser expects to see the following token stream to match the keyValue parser rule: IDENTIFIER '=' INTEGER ';' where '=' and ';' are implicit token types.

Since 42 can match both INTEGER and IDENTIFIER, and IDENTIFIER is defined first, the parser will receive the following input: IDENTIFIER '=' IDENTIFIER ';' which it won't be able to match to the keyValue rule. Remember, the parser *cannot* communicate *to* the lexer, it can only receive data from it, therefore it cannot say "*try to match* INTEGER *next*".

It's advisable to minimize the lexer rules overlap to limit the impact of this effect. In the above example, we have several options:

- Redefine IDENTIFIER as [A-Za-z] [A-Za-z0-9]* (require it to start with a letter). This avoids the problem entirely but prevents identifier names starting with a number from being defined, so it changes the intent of the grammar.
- Reorder INTEGER and IDENTIFIER. This solves the problem for most cases, but prevents fully numeric identifiers from being defined, therefore it also changes the intent of the grammar in a subtle, not so obvious way.
- Make the parser accept both token types when lexer rules overlap: First, swap INTEGER and IDENTIFIER in order to give priority to INTEGER. Then, define a parser rule id: IDENTIFIER | INTEGER; then use that rule instead of IDENTIFIER in other parser rules, which would change keyValue to key=id '=' value=INTEGER ';'.

Here's a second lexer behavior example to sum up:

The following combined grammar:

```
grammar LexerPriorityRulesExample;

// Parser rules

randomParserRule: 'foo'; // Implicitly declared token type

// Lexer rules

BAR: 'bar';
IDENTIFIER: [A-Za-z]+;
BAZ: 'baz';

WS: [ \t\r\n]+ -> skip;
```

Given the following input:

```
aaa foo bar baz barz
```

Will produce the following token sequence from the lexer:

```
IDENTIFIER 'foo' BAR IDENTIFIER IDENTIFIER EOF
```

- `aaa` is of type `IDENTIFIER`

Only the `IDENTIFIER` rule can match this token, there is no ambiguity.

- `foo` is of type `'foo'`

The parser rule `randomParserRule` introduces the implicit `'foo'` token type, which is prioritary over the `IDENTIFIER` rule.

- `bar` is of type `BAR`

This text matches the BAR rule, which is defined *before* the `IDENTIFIER` rule, and therefore has precedence.

- `baz` is of type `IDENTIFIER`

This text matches the `BAZ` rule, but it also matches the `IDENTIFIER` rule. The latter is chosen as it is defined *before* `BAR`.

Given the grammar, BAZ will never be able to match, as the IDENTIFIER rule already covers everything BAZ can match.

- barz is of type IDENTIFIER

The BAR rule can match the first 3 characters of this string (bar), but the IDENTIFIER rule will match 4 characters. As IDENTIFIER matches a longer substring, it is chosen over BAR.

- EOF (*end of file*) is an implicitly defined token type which always occurs at the end of the input.

As a rule of thumb, specific rules should be defined before more generic rules. If a rule can only match an input which is already covered by a previously defined rule, it will never be used.

Implicitly defined rules such as 'foo' act as if they were defined before all other lexer rules. As they add complexity, it's advisable to avoid them altogether and declare explicit lexer rules instead. Just having a list of tokens in one place instead of having them scattered across the grammar is a compelling advantage of this approach.

Original question from StackOverflow

Why whitespace creates problems even if I ignore it?

QUESTION

I have this simple grammar for arithmetic expressions. Now I am running in a curious issue.

(1+2)^3 is NOT recognized as an expression

but

(1 + 2)^3 is recognized as an expression

Even though in this grammar I have this whitespace rule

In the grammar I do have this lexer rule:

```
WS
  : [ \r\n\t] + -> skip
  ;
```

So why does it behave differently on the 2 inputs?

I have no idea why... The only difference between them are the whitespaces.

Here is some additional info about the error raised when working on this input

```
(1+2)^3
C:\MathSystem>node main.js
line 1:2 extraneous input '+2' expecting ')'
====> Found number:  1
CONSTRUCTOR:  NumberContext
VALUE:  1

====> Found atom:  1
CONSTRUCTOR:  AtomContext
VALUE:  1

====> Found expression:
Found child [ 0 ] -> CONSTRUCTOR :  AtomContext TEXT:  1
VALUE:  1

====> Found expression:
```

```
Found child [ 0 ] -> CONSTRUCTOR :  TerminalNodeImpl TEXT:  (
Found child [ 1 ] -> CONSTRUCTOR :  ExpressionContext TEXT:  1
Found child [ 2 ] -> CONSTRUCTOR :  ErrorNodeImpl TEXT:  +2
Found child [ 3 ] -> CONSTRUCTOR :  TerminalNodeImpl TEXT:  )
```

This is the partial grammar, with the relevant rules.

```
grammar arithmetic;

file : expression* EOF;

expression
  :  expression (TIMES | DIV | PLUS | MINUS | POW) expression
  |  LPAREN expression RPAREN
  |  atom
  ;

atom
  : number
  | variable
  ;

number
  : SCIENTIFIC_NUMBER
  ;

SCIENTIFIC_NUMBER
  : SIGN? NUMBER (E SIGN? UNSIGNED_INTEGER)?
  ;

PLUS
  : '+'
  ;

MINUS
  : '-'
  ;

fragment NUMBER
  : ('0' .. '9') + ('.' ('0' .. '9') +)?
  ;

fragment UNSIGNED_INTEGER
  : ('0' .. '9')+
  ;
```

```
fragment SIGN
  : ('+' | '-')
  ;

[..]

WS
  : [ \r\n\t] + -> skip
  ;
```

ANSWER

The issue is that + in the +2 is matched as the SIGN for the SCIENTIFIC_NUMBER. However, when there is a space between them the + get matched correctly as a PLUS token.

You confusion may rise from misunderstanding how the skip command works. The ANTLR lexer DOES NOT make a first pass in which discard the text matched by skip(ed) rules, before looking at the rest of the rules. It cannot do that. For example, whitespace in isolation should be skipped, but what if you had STRING rule that matched all text between " and "?

The lexer matches the rules according to the grammar. Then, if it realizes that it has matched a rule with the skip command, it discards the token. Actually, it does not really discard the token, it sends it to the HIDDEN channel, but that is not relevant here.

Finding which rule matched between two alternatives

QUESTION

I am sure there is a simple solution, but I cannot find it.

I have these rules:

```
and_check :
      expr  andness  expr  #end_incl1
;

andness : and | not_and ;

not_and : NOT AND;
and : AND ;
```

When I'm in `EnterEnd_incl1` (as generated from #end_incl1), which subrule of `andness` was matched; `and` or `not_and`?

I can get it working using `context.andness.GetText();` which will return the text, and that's workable for now, but feels hacky and wrong.
Also if I have more complicated examples then just looking at the raw text returned will become increasingly messy, so I really need to know how do I do that in C#?

Silly question I know, but I am stuck on it.

ANSWER

The solution is to check which rule does not return null on the context object inside `EnterEnd_incl1`.

```
if(context.andness.and() != null) {
// and matched
}
else if(context.andness.and_not() != null) {
// not_and matched
}
```

Is “Implicit token definition in parser rule” something to worry about?

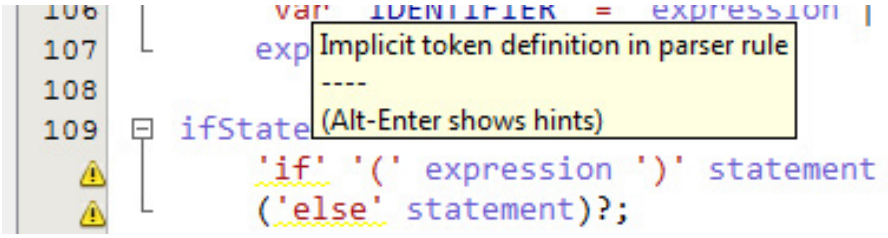
QUESTION

I’m creating my first grammar with ANTLR and ANTLRWorks 2. I have mostly finished the grammar itself (it recognizes the code written in the described language and builds correct parse trees), but I haven’t started anything beyond that.

What worries me is that every first occurrence of a token in a parser rule is underlined with a yellow squiggle saying “Implicit token definition in parser rule”.

For example, in this rule, the ‘var’ has that squiggle:

variableDeclaration: ‘var’ IDENTIFIER (‘=’ expression)?;
How it looks exactly:



The odd thing is that ANTLR itself doesn’t seem to mind these rules (when doing test rig test, I can’t see any of these warning in the parser generator output, just something about incorrect Java version being installed on my machine), so it’s just ANTLRWorks complaining.

Is it something to worry about or should I ignore these warnings? Should I declare all the tokens explicitly in lexer rules? Most exaples in the official bible The Defintive ANTLR Reference seem to be done exactly the way I write the code.

ANSWER

I highly recommend correcting all instances of this warning in code of any importance.

This warning was created (by me actually) to alert you to situations like the following:

shiftExpr : ID ((‘<<’ | ‘>>’) ID)?;

Since ANTLR 4 encourages action code be written in separate files in the target language instead of embedding them directly in the grammar, it's important to be able to distinguish between << and >>. If tokens were not explicitly created for these operators, they will be assigned arbitrary types and no named constants will be available for referencing them.

This warning also helps avoid the following problematic situations:

- A parser rule contains a misspelled token reference. Without the warning, this could lead to silent creation of an additional token that may never be matched.
- A parser rule contains an unintentional token reference, such as the following:

```
number : zero | INTEGER;
zero   : '0'; // <-- this implicit definition causes 0 to get its own token
```

Original question from StackOverflow

How do I escape an escape character with ANTLR 4?

QUESTION

Many languages bound a string with some sort of quote, like this:

```
“Rob Malda is smart.”
```

ANTLR 4 can match such a string with a lexer rule like this:

```
QuotedString : ‘”’ .*? ‘”’;
```

To use certain characters within the string, they must be escaped, perhaps like this:

```
“Rob \”Commander Taco\” Malda is smart.”
```

ANTLR 4 can match this string as well;

```
EscapedString : ‘”’ (‘\\”|.)*? ‘”’;
```

(taken from p96 of The Definitive ANTLR 4 Reference)

Here’s my problem: Suppose that the character for escaping is the same character as the string delimiter. For example:

```
“Rob “”Commander Taco”” Malda is smart.”
```

(This is perfectly legal in Powershell.)

What lexer rule would match this? I would think this would work:

```
EscapedString : ‘”’ (‘””’|.)*? ‘”’;
```

But it doesn’t. The lexer tokenizes the escape character “ as the end of string delimiter.

ANSWER

Negate certain characters with the ~ operator:

```
EscapedString : ‘”’ ( ‘””’ | ~[“] )* ‘”’;
```

or, if there can’t be line breaks in your string, do:

```
EscapedString : ‘”’ ( ‘””’ | ~[“\r\n] )* ‘”’;
```

You don’t want to use the non-greedy operator, otherwise “” would never be consumed and “a””b” would be tokenized as “a” and “b” instead of a single token.

Original question from StackOverflow

In ANTLR, can I look-ahead for specific tokens without actually matching them?

QUESTION

Basically, I need to lookahead to know if a certain token exists, but without matching it (i.e. so that the another parser rule can still match it).

ANSWER

You can test for the preceding or next token with a semantic predicate.

If you want to test for the preceding token, you can use the _input.LT(-1), but you can only do that for parser rules. For example, if you want to enable a mention rule only if preceded by a WHITESPACE token.

```
// C#
mention: {_input.Lt(-1).Type == WHITESPACE}? '@' WORD ;
// Java
mention: {_input.LT(-1).getType() == WHITESPACE}? '@' WORD ;
// Python
mention: {self._input.LT(-1).text == ' '}? '@' WORD ;
// JavaScript
mention: {this._input.LT(-1).text == ' '}? '@' WORD ;
```

You can do something similar with the following token, you can use the _ input.LT(2). For example, if you want to enable a mention rule only if followed by a WHITESPACE token.

```
// C#
mention: {_input.Lt(2).Type == WHITESPACE}? '@' WORD ;
// Java
mention: {_input.LT(2).getType() == WHITESPACE}? '@' WORD ;
// Python
mention: {self._input.LT(2).text == ' '}? '@' WORD ;
// JavaScript
mention: {this._input.LT(2).text == ' '}? '@' WORD ;
```

Original question from StackOverflow (but not the answer)

ANTLR 4: Avoid error printing to console

QUESTION

With ANTLR I parse a grammar. The ANTLR errors are indicated in a custom editor for the grammar. However I would like to disable the printing of error messages to a Java console.

I've implemented my own BaseErrorListener and removed the default as described in the ANTLR book:

```
MyErrorListener errList=new MyErrorListener ();
lexer.removeErrorListeners();
lexer.addErrorListener(errList);
parser.removeErrorListeners();
parser.addErrorListener(errList);
```

Still I get printed outputs to my Java console (connected to the Java output and error stream).

How can I disable the printing to the console in ANTLR?

ANSWER

By default, `ConsoleListener` is activated [1].

You can remove it in your code:

```
lexer.removeErrorListener(ConsoleErrorListener.INSTANCE);
```

[1]
https://github.com/antlr/antlr4/blob/master/runtime/Java/src/org/antlr/v4/runtime/Recognizer.java#L56 (see add(ConsoleErrorListener.INSTANCE);)

Original question from StackOverflow

Getting the line number in the ParserVisitor?

QUESTION

I'm trying to get line numbers for more specific error messages in my ParserVisitor (visits the parse tree generated by antlr). However, all I have in this class is the context ctx, and I can do things like `ctx.getText()` but not `getLine()`. Is there a way to do this?

Can `ctx.getPayload()` be used here? If so, how?

Edit: I'm using ANTLR 4 to create java files.

Trying to access the line number in a visitor in a method such as this:

```
@Override
public Type visitStatAssign(@NotNull BasicParser.StatAssignContext ctx) {
...
// some semantic error detected
int lineNo = ...
System.err.("Semantic error at line " + lineNo);
}
```

Edit 2: My lexer and parser rules are fairly standard, for example in the lexer:

```
INT : 'int' ;
CHAR : 'char' ;
BOOL : 'bool' ;
STRING : 'string' ;
```

...is in the parser rule baseType:

```
baseType : INT | CHAR | BOOL | STRING ;
```

ANSWER

You can get the first token in the rule with `ctx.start` or `ctx.getStart()`. Then use `getLine()` on the token to get the line number (and `getCharPositionInLine()` to get the column).

Original question from StackOverflow

Useful Patterns

In this section we see questions that are solved by learning useful patterns. This section will help discover the techniques and strategies to create effective ANTLR grammars.

Nested Boolean Expression Parser using ANTLR

QUESTION

I'm trying to parse a Nested Boolean Expression and get the individual conditions within the expression separately. For e.g., if the input string is:

(A = a OR B = b OR C = c AND ((D = d AND E = e) OR (F = f AND G = g)))

I would like to get the conditions with the correct order. i.e.,

D =d AND E = e OR F = f AND G = g AND A = a OR B = b OR C = c

I'm using ANTLR 4 to parse the input text and here's my grammar:

```
grammar SimpleBoolean;

rule_set : nestedCondition* EOF;

AND : 'AND' ;
OR  : 'OR' ;
NOT : 'NOT';

TRUE  : 'TRUE' ;
FALSE : 'FALSE' ;

GT : '>' ;
GE : '>=' ;
LT : '<' ;
LE : '<=' ;
EQ : '=' ;

LPAREN : '(' ;
RPAREN : ')' ;

DECIMAL : '-'?[0-9]+('.'[0-9]+)? ;

IDENTIFIER : [a-zA-Z_][a-zA-Z_0-9]* ;

WS : [ \r\t\u000C\n]+ -> skip;

nestedCondition : LPAREN condition+ RPAREN (binary nestedCondition)*;
condition: predicate (binary predicate)*
         | predicate (binary component)*;
component: predicate | multiAttrComp;
multiAttrComp : LPAREN predicate (and predicate)+ RPAREN;
predicate : IDENTIFIER comparator IDENTIFIER;
comparator : GT | GE | LT | LE | EQ ;
binary: AND | OR ;
unary: NOT;
and: AND;
```

And here's the Java Code that I'm using to parse it:

```
ANTLRInputStream inputStr = new ANTLRInputStream(input);
SimpleBooleanLexer lexer = new SimpleBooleanLexer(inputStr);
TokenStream tokens = new CommonTokenStream(lexer);
SimpleBooleanParser parser = new SimpleBooleanParser(tokens);
parser.getBuildParseTree();
ParseTree tree = parser.rule_set();
System.out.println(tree.toStringTree(parser));
```

The output is:

```
(rule_set (nestedCondition ( (condition (predicate A (comparator =) a) (binary OR) (component (predicate B (comparator =) b)) (binary OR) (component (predicate C (comparator =) c)) (binary AND) (component (multiAttrComp ( (predicate ( D (comparator =) d) (and AND) (predicate E (comparator =) e) ))) (binary OR) (component (multiAttrComp ( (predicate F (comparator =) f) (and AND) (predicate G (comparator =) g) ))))) ) )) <EOF>)
```

I'm looking for help on how to parse this tree to get the conditions in the correct order? In ANTLR 3, we could specify ^ and ! to decide how the tree is built (refer this thread), but I learnt that this is not supported in ANTLR 4. Can someone suggest how I can parse the String in the correct order in Java using the ParseTree created by ANTLR?

ANSWER

I'd just wrap all the expressions into a single `expression` rule. Be sure to define the `comparator` expressions alternative *before* your `binary` expression alternative to make sure `comparator` operators bind more tightly than `OR` and `AND`:

```
grammar SimpleBoolean;

parse
 : expression EOF
 ;

expression
 : LPAREN expression RPAREN                  #parenExpression
 | NOT expression                            #notExpression
 | left=expression op=comparator right=expression #comparatorExpression
 | left=expression op=binary right=expression     #binaryExpression
 | bool                                      #boolExpression
 | IDENTIFIER                                #identifierExpression
 | DECIMAL                                   #decimalExpression
 ;

comparator
 : GT | GE | LT | LE | EQ
```

```
;

binary
 : AND | OR
 ;

bool
 : TRUE | FALSE
 ;

AND       : 'AND' ;
OR        : 'OR' ;
NOT       : 'NOT';
TRUE      : 'TRUE' ;
FALSE     : 'FALSE' ;
GT        : '>' ;
GE        : '>=' ;
LT        : '<' ;
LE        : '<=' ;
EQ        : '=' ;
LPAREN    : '(' ;
RPAREN    : ')' ;
DECIMAL   : '-'? [0-9]+ ( '.' [0-9]+ )? ;
IDENTIFIER : [a-zA-Z_] [a-zA-Z_0-9]* ;
WS        : [ \r\t\u000C\n]+ -> skip;
```

To test the grammar above, use the following quick-and-dirty test classes:

```
public class Main {
  public static void main(String[] args) throws Exception {
    Map<String, Object> variables = new HashMap<String, Object>() {{
      put("A", true);
      put("a", true);
      put("B", false);
      put("b", false);
      put("C", 42.0);
      put("c", 42.0);
      put("D", -999.0);
      put("d", -1999.0);
      put("E", 42.001);
      put("e", 142.001);
      put("F", 42.001);
      put("f", 42.001);
      put("G", -1.0);
      put("g", -1.0);
    }};

    String[] expressions = {
        "1 > 2",
```

```
      "1 >= 1.0",
      "TRUE = FALSE",
      "FALSE = FALSE",
      "A OR B",
      "B",
      "A = B",
      "c = C",
      "E > D",
      "B OR (c = B OR (A = A AND c = C AND E > D))",
      "(A = a OR B = b OR C = c AND ((D = d AND E = e) OR (F = f AND G = g)))"
    };

    for (String expression : expressions) {
 SimpleBooleanLexerlexer=newSimpleBooleanLexer(newANTLRInputStream(expression));
                    SimpleBooleanParser parser = new SimpleBooleanParser(new
CommonTokenStream(lexer));
      Object result = new EvalVisitor(variables).visit(parser.parse());
      System.out.printf("%-70s -> %s\n", expression, result);
    }
  }
}

class EvalVisitor extends SimpleBooleanBaseVisitor<Object> {
  private final Map<String, Object> variables;
  public EvalVisitor(Map<String, Object> variables) {
    this.variables = variables;
  }

  @Override
  public Object visitParse(SimpleBooleanParser.ParseContext ctx) {
    return super.visit(ctx.expression());
  }

  @Override
   public Object visitDecimalExpression(SimpleBooleanParser.DecimalExpressionContext
ctx) {
    return Double.valueOf(ctx.DECIMAL().getText());
  }

  @Override
  public Object visitIdentifierExpression(SimpleBooleanParser.IdentifierExpressionContext
ctx) {
    return variables.get(ctx.IDENTIFIER().getText());
  }

  @Override
  public Object visitNotExpression(SimpleBooleanParser.NotExpressionContext ctx) {
    return !((Boolean)this.visit(ctx.expression()));
```

```
  }

  @Override
  public Object visitParenExpression(SimpleBooleanParser.ParenExpressionContext ctx) {
    return super.visit(ctx.expression());
  }

  @Override
  public Object visitComparatorExpression(SimpleBooleanParser.
ComparatorExpressionContext ctx) {
    if (ctx.op.EQ() != null) {
      return this.visit(ctx.left).equals(this.visit(ctx.right));
    }
    else if (ctx.op.LE() != null) {
      return asDouble(ctx.left) <= asDouble(ctx.right);
    }
    else if (ctx.op.GE() != null) {
      return asDouble(ctx.left) >= asDouble(ctx.right);
    }
    else if (ctx.op.LT() != null) {
      return asDouble(ctx.left) < asDouble(ctx.right);
    }
    else if (ctx.op.GT() != null) {
      return asDouble(ctx.left) > asDouble(ctx.right);
    }
        throw new RuntimeException("not implemented: comparator operator " + ctx.
op.getText());
  }

  @Override
  public Object visitBinaryExpression(SimpleBooleanParser.BinaryExpressionContext ctx) {
    if (ctx.op.AND() != null) {
      return asBoolean(ctx.left) && asBoolean(ctx.right);
    }
    else if (ctx.op.OR() != null) {
      return asBoolean(ctx.left) || asBoolean(ctx.right);
    }
    throw new RuntimeException("not implemented: binary operator " + ctx.op.getText());
  }

  @Override
  public Object visitBoolExpression(SimpleBooleanParser.BoolExpressionContext ctx) {
    return Boolean.valueOf(ctx.getText());
  }

  private boolean asBoolean(SimpleBooleanParser.ExpressionContext ctx) {
    return (boolean)visit(ctx);
  }
```

```
  private double asDouble(SimpleBooleanParser.ExpressionContext ctx) {
    return (double)visit(ctx);
  }
}
```

Running the Main class will result in the following output:

```
1 > 2                                   -> false
1 >= 1.0                                -> true
TRUE = FALSE                            -> false
FALSE = FALSE                           -> true
A OR B                                  -> true
B                                       -> false
A = B                                   -> false
c = C                                   -> true
E > D                                   -> true
B OR (c = B OR (A = A AND c = C AND E > D))             -> true
(A = a OR B = b OR C = c AND ((D = d AND E = e) OR (F = f AND G = g)))     -> true
```

Original question from StackOverflow

Allow Whitespace sections ANTLR4

QUESTION

I have an antlr4 grammar designed for an a domain specific language that is embedded into a text template.
There are two modes:

- Text (whitespace should be preserved)
- Code (whitespace should be ignored)

Sample grammar part:

```
template
 :  '{' templateBody '}'
 ;
templateBody
 : templateChunk*
 ;
templateChunk
 : code # codeChunk // dsl code, ignore whitespace
 | text # textChunk // any text, preserve whitespace
 ;
```

The rule for `code` may contain a nested reference to the `template` rule. So the parser must support nesting whitespace/non-whitespace sections.

Maybe **lexer modes** can help - with some drawbacks:

- the code sections must be parsed in another compiler pass
- I doubt that nested sections could be mapped correctly

Yet the most promising approach seems to be the **manipulation of hidden channels.**

My question: Is there a best practice to fill these requirements? Is there an example grammar, that has already solved similar problems?

ANSWER

This is how I solved the problem at the end:

The idea is to enable/disable whitespace in a parser rule:

```
templateBody : {enableWs();} templateChunk* {disableWs();};
```

So we will have to define enableWs and disableWs in our parser base class:

```
public void enableWs() {
   if (_input instanceof MultiChannelTokenStream) {
      ((MultiChannelTokenStream) _input).enable(HIDDEN);
   }
}
public void disableWs() {
   if (_input instanceof MultiChannelTokenStream) {
      ((MultiChannelTokenStream) _input).disable(HIDDEN);
   }
}
```

Now what is this MultiChannelTokenStream?

- Antlr4 defines a CommonTokenStream which is a token stream reading only from **one channel**.
- MultiChannelTokenStream is a token stream reading from the **enabled channels**. For implementation I took the source code of CommonTokenStream and replaced each reference to the channel by channels (equality comparison gets contains comparison)

An example implementation with the grammar above could be found at antlr4multichannel

Original question from StackOverflow

Can I add Antlr tokens at runtime?

QUESTION

I have a situation where my language contains some words that aren't known at build time but will be known at run time causing the need to constantly rebuild / redeploy the program to take into account new words. I was wondering if it was possible in Antlr to generate some of the tokens from a config file?

e.g In a simplified example if I have a rule

```
rule : WORDS+;
WORDS : 'abc';
```

And my language comes across 'bcd' at runtime, I would like to be able to modify a config file to define bcd as a word rather than having to rebuild then redeploy.

ANSWER

You can dynamically add tokens to ANTLR by taking advantage of semantic predicates. You can add them dynamically after you have created the parser or even during the parser run. An example for your case. The example code is in Python but it is easy to adapt to any language supported by ANTLR.

The grammar file.

```
grammar DynamicTokens;
@lexer::members {
# a custom constructore
def __init__(self, input, keywords):
    super().__init__(input, sys.stdout)
    self.dynamicKeywords = keywords
                    self._interp     =     LexerATNSimulator(self,     self.atn,     self.decisionsToDFA,
PredictionContextCache())
    self._actions = None
    self._predicates = None

# the method that checks against a list of dynamic keywords
def isDynamicKeyword(self, name):
    if name in self.dynamicKeywords:
        return True
    else:
        return False
}
baseRule          : WORDS+;
// the rule that checks whether the text is one of the dynamic keywords
// the final action that print the keyword is not required
WORDS          : [a-zA-Z0-9]+ {self.isDynamicKeyword(self.text)}? {print("Dynamic keyword:",
self.text)}
                | 'abc'
                ;

WHITESPACE         : (' ' | '\t') -> skip;
NEWLINE          : ('\r'? '\n' | '\r') ;
```

The Python file that runs the parser.

```
import sys
from antlr4 import *
from DynamicTokensLexer import DynamicTokensLexer
from DynamicTokensParser import DynamicTokensParser

def main(argv):
    keywords = ['one', 'two']
    input = InputStream('one abc two')
    lexer = DynamicTokensLexer(input, keywords)
    stream = CommonTokenStream(lexer)
    parser = DynamicTokensParser(stream)
    tree = parser.baseRule()

if __name__ == '__main__':
    main(sys.argv)
```

Original question from StackOverflow (but not the answer)

Getting ANTLR to generate a script interpreter?

QUESTION

Say I have the following Java API that all packages up as blocks.jar:

```
public class Block {
   private Sting name;
   private int xCoord;
   private int yCoord;

   // Getters, setters, ctors, etc.

   public void setCoords(int x, int y) {
      setXCoord(x);
      setYCoord(y);
   }
}

public BlockController {
   public static moveBlock(Block block, int newXCoord, int newYCoord) {
      block.setCooords(newXCoord, newYCoord);
   }

   public static stackBlocks(Block under, Block onTop) {
      // Stack “onTop” on top of “under”.
      // Don’t worry about the math here, this is just for an example.
      onTop.setCoords(under.getXCoord() + onTop.getXCoord(), under.getYCoord());
   }
}
```

Again, don’t worry about the math and the fact that (x,y) coordinates don’t accurately represent blocks in 3D space. The point is that we have Java code, compiled as a JAR, that performs operations on blocks. I now want to build a lightweight scripting language that allows a non-programmer to invoke the various block API methods and manipulate blocks, and I want to implement its interpreter with ANTLR (latest version is 4.3).

The scripting language, we’ll call it *BlockSpeak*, might look like this:

```
block A at (0, 10)   # Create block “A” at coordinates (0, 10)
block B at (0, 20)   # Create block “B” at coordinates (0, 20)
stack A on B         # Stack block A on top of block B
```

This might be equivalent to the following Java code:

```
Block A, B;
A = new Block(0, 10);
B = new Block(0, 20);
BlockController.stackBlocks(B, A);
```

So the idea is that the ANTLR-generated interpreter would take a *.blockspeak script as input, and use the commands in this script to invoke blocks.jar API operations. I read the excellent Simple Example which creates a simple calculator using ANTLR. However in that link, there is an **ExpParser** class with an eval() method:

```
ExpParser parser = new ExpParser(tokens);
parser.eval();
```

The problem here is that, in the case of the calculator, the tokens represent a mathematical expression to evaluate, and eval() returns the evaluation of the expression. In the case of an interpreter, the tokens would represent my BlockSpeak script, but calling eval() shouldn't evaluate anything, it should know how to map the various BlockSpeak commands to Java code:

```
BlockSpeak Command:          Java code:
=========================================
block A at (0, 10)     ==>   Block A = new Block(0, 10);
block B at (0, 20)     ==>   Block B = new Block(0, 20);
stack A on B           ==>   BlockController.stackBlocks(B, A);
```

So my question is, where do I perform this "mapping"? In other words, how do I instruct ANTLR to call various pieces of code (packaged inside blocks.jar) when it encounters particular grammars in the BlockSpeak script? **More importantly**, can someone give me a pseudo-code example?

ANSWER

I would simply evaluate the script on the fly, not generate Java source files which need to be compiled themselves again.

With ANTLR 4 it is highly recommended to keep the grammar and target specific code separate from each other and put any target specific code inside a tree-listener or -visitor.

I will give a quick demo how to use a listener.

A grammar for your example input could look like this:

File: blockspeak/BlockSpeak.g4

```
grammar BlockSpeak;

parse
 : instruction* EOF
 ;

instruction
```

```
 : create_block
 | stack_block
 ;

create_block
 : 'block' NAME 'at' position
 ;

stack_block
 : 'stack' top=NAME 'on' bottom=NAME
 ;

position
 : '(' x=INT ',' y=INT ')'
 ;

COMMENT
 : '#' ~[\r\n]* -> skip
 ;

INT
 : [0-9]+
 ;

NAME
 : [a-zA-Z]+
 ;

SPACES
 : [ \t\r\n] -> skip
 ;
```

Some supporting Java classes:

File: blockspeak/Main.java

```
package blockspeak;

import org.antlr.v4.runtime.ANTLRInputStream;
import org.antlr.v4.runtime.CommonTokenStream;
import org.antlr.v4.runtime.tree.ParseTreeWalker;

import java.util.Scanner;

public class Main {

    public static void main(String[] args) throws Exception {
```

```
        Scanner keyboard = new Scanner(System.in);

        // Some initial input to let the parser have a go at.
        String input = "block A at (0, 10)   # Create block \"A\" at coordinates (0, 10)\n" +
                "block B at (0, 20)   # Create block \"B\" at coordinates (0, 20)\n" +
                "stack A on B         # Stack block A on top of block B";

        EvalBlockSpeakListener listener = new EvalBlockSpeakListener();

        // Keep asking for input until the user presses 'q'.
        while(!input.equals("q")) {

            // Create a lexer and parser for `input`.
            BlockSpeakLexer lexer = new BlockSpeakLexer(new ANTLRInputStream(input));
        BlockSpeakParser parser = new BlockSpeakParser(new CommonTokenStream(lexer));

            // Now parse the `input` and attach our listener to it. We want to reuse
            // the same listener because it will hold out Blocks-map.
            ParseTreeWalker.DEFAULT.walk(listener, parser.parse());

            // Let's see if the user wants to continue.
            System.out.print("Type a command and press return (q to quit) $ ");
            input = keyboard.nextLine();
        }

        System.out.println("Bye!");
    }
}

// You can place this Block class inside Main.java as well.
class Block {

    final String name;
    int x;
    int y;

    Block(String name, int x, int y) {
        this.name = name;
        this.x = x;
        this.y = y;
    }

    void onTopOf(Block that) {
        // TODO
    }
}
```

This main class is pretty self explanatory with the inline comments. The tricky part is what the listener is supposed to look like. Well, here it is:

File: blockspeak/EvalBlockSpeakListener.java

```
package blockspeak;

import org.antlr.v4.runtime.misc.NotNull;

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

/**
 * A class extending the `BlockSpeakBaseListener` (which will be generated
 * by ANTLR) in which we override the methods in which to create blocks, and
 * in which to stack blocks.
 */
public class EvalBlockSpeakListener extends BlockSpeakBaseListener {

    // A map that keeps track of our Blocks.
    private final Map<String, Block> blocks = new HashMap<String, Block>();

    @Override
    public void enterCreate_block(@NotNull BlockSpeakParser.Create_blockContext ctx) {

        String name = ctx.NAME().getText();
        Integer x = Integer.valueOf(ctx.position().x.getText());
        Integer y = Integer.valueOf(ctx.position().y.getText());

        Block block = new Block(name, x, y);

        System.out.printf("creating block: %s\n", name);

        blocks.put(block.name, block);
    }

    @Override
    public void enterStack_block(@NotNull BlockSpeakParser.Stack_blockContext ctx) {

        Block bottom = this.blocks.get(ctx.bottom.getText());
        Block top = this.blocks.get(ctx.top.getText());

        if (bottom == null) {
            System.out.printf("no such block: %s\n", ctx.bottom.getText());
        }
         else if (top == null) {
            System.out.printf("no such block: %s\n", ctx.top.getText());
        }
```

```
      else {
        System.out.printf("putting %s on top of %s\n", top.name, bottom.name);
        top.onTopOf(bottom);
      }
    }
}
```

The listener above has 2 methods defined that map to the following parser rules:

```
create_block
 : 'block' NAME 'at' position
 ;

stack_block
 : 'stack' top=NAME 'on' bottom=NAME
 ;
```

Whenever the parser "enters" such a parser rule, the corresponding method inside the listener will be called. So, whenever `enterCreate_block` (the parser enters the `create_block` rule) is called, we create (and save) a block, and when `enterStack_block` is called, we retrieve the 2 block involved in the operation, and stack one on top of the other.

To see the 3 classes above in action, download ANTLR 4.4 inside the directory that holds the `blockspeak/` directory with the `.g4` and `.java` files.

Open a console and perform the following 3 steps:

1. generate the ANTLR files:

```
java -cp antlr-4.4-complete.jar org.antlr.v4.Tool blockspeak/BlockSpeak.g4 -package blockspeak
```

2. compile all Java sources files:

```
javac -cp ./antlr-4.4-complete.jar blockspeak/*.java
```

3. Run the main class:

3.1. Linux/Mac

```
java -cp .:antlr-4.4-complete.jar blockspeak.Main
```

3.2. Windows

```
java -cp .;antlr-4.4-complete.jar blockspeak.Main
```

Here is an example session of running the Main class:

```
bart@hades:~/Temp/demo$ java -cp .:antlr-4.4-complete.jar blockspeak.Main
creating block: A
creating block: B
putting A on top of B
Type a command and press return (q to quit) $ block X at (0,0)
creating block: X
Type a command and press return (q to quit) $ stack Y on X
no such block: Y
Type a command and press return (q to quit) $ stack A on X
putting A on top of X
Type a command and press return (q to quit) $ q
Bye!
bart@hades:~/Temp/demo$
```

More info on tree listeners:

https://theantlrguy.atlassian.net/wiki/display/ANTLR4/Parse+Tree+Listeners

Original question from StackOverflow

What is minimal sample Gradle project for ANTLR4 (with antlr plugin)?

QUESTION

Assuming I created src/main/antlr/test.g4 file with the following content

```
grammar test;
r   : 'hello' ID;
ID  : [a-z]+ ;
WS  : [ \t\r\n]+ -> skip ;
```

What my build.gradle file should look like?

ANSWER

This is a basic build.gradle file for an ANTLR Java project.

```
plugins {
   id 'java'
   id 'antlr'
}

repositories {
   mavenCentral()
}

ext.antlr_version='4.9.2'

dependencies {
   antlr "org.antlr:antlr4:$antlr_version"
   implementation "org.antlr:antlr4-runtime:$antlr_version"
}

generateGrammarSource {
   maxHeapSize = "128m"
       arguments  +=  ['-package',  'com.strumenta.examples.TestParser',  '-no-visitor',  '-no-
listener']
}
compileJava.dependsOn generateGrammarSource

sourceSets {
   generated {
      java.srcDir 'generated-src/antlr/main/'
   }
}
compileJava.source sourceSets.generated.java, sourceSets.main.java
```

```
clean{
    delete “generated-src”
}

// this is not stricly necessary but it is useful if you are not using an IDE
// it will create a JAR with all dependencies included
task fatJar(type: Jar) {
    manifest {
        attributes ‘Main-Class’: ‘com.strumenta.examples.TestParser.App’
    }
    baseName = project.name + ‘-all’
    from { configurations.compile.collect { it.isDirectory() ? it : zipTree(it) } }
    with jar
}
```

If you are using IntelliJ IDEA and Junit, you want something like this one. This file will set up IntelliJ IDEA so it can automatically recognize the generated ANTLR library.

```
plugins {
    id ‘java’
    id ‘antlr’
    // adding the idea plugin
    id ‘idea’
}

repositories {
    mavenCentral()
}

ext.antlr_version=’4.9.2’

dependencies {
    antlr “org.antlr:antlr4:$antlr_version”
    implementation “org.antlr:antlr4-runtime:$antlr_version”
    testImplementation(platform(‘org.junit:junit-bom:5.7.0’))
    testImplementation(‘org.junit.jupiter:junit-jupiter’)
}

generateGrammarSource {
    maxHeapSize = “128m”
        arguments += [‘-package’, ‘com.strumenta.examples.TestParser’, ‘-no-visitor’, ‘-no-
listener’]
}
compileJava.dependsOn generateGrammarSource

sourceSets {
```

```
    generated {
        java.srcDir 'generated-src/antlr/main/'
    }
}
compileJava.source sourceSets.generated.java, sourceSets.main.java

clean{
    delete "generated-src"
}

idea {
    module {
        sourceDirs += file("generated-src/antlr/main/")
        generatedSourceDirs += file("generated-src/antlr/main")
    }
}

test {
    useJUnitPlatform()
    testLogging {
        events "passed", "skipped", "failed"
    }
}

// this is not stricly necessary but it is useful if you are not using an IDE
// it will create a JAR with all dependencies included
task fatJar(type: Jar) {
    manifest {
        attributes 'Main-Class': 'com.strumenta.examples.TestParser.App'
    }
    baseName = project.name + '-all'
    from { configurations.compile.collect { it.isDirectory() ? it : zipTree(it) } }
    with jar
}
```

Finally, if you are using Kotlin, you may want to start from a similar gradle file.

```
buildscript {
    ext.kotlin_version = '1.5.21'

    repositories {
        mavenCentral()
    }

    dependencies {
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    }
```

```
}

plugins {
    id 'java'
    id 'org.jetbrains.kotlin.jvm' version "$kotlin_version"
    id 'antlr'
    id 'idea'
}

repositories {
    mavenCentral()
}

ext.antlr_version='4.9.2'

dependencies {
    antlr "org.antlr:antlr4:$antlr_version"
    implementation "org.antlr:antlr4-runtime:$antlr_version"
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8:$kotlin_version"
    implementation "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
    implementation "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"
    testImplementation "org.jetbrains.kotlin:kotlin-test:$kotlin_version"
    testImplementation "org.jetbrains.kotlin:kotlin-test-junit:$kotlin_version"
    testImplementation(platform('org.junit:junit-bom:5.7.0'))
    testImplementation('org.junit.jupiter:junit-jupiter')
}

compileKotlin {
    kotlinOptions {
        jvmTarget = "1.8"
    }
}
compileTestKotlin {
    kotlinOptions {
        jvmTarget = "1.8"
    }
}

generateGrammarSource {
    maxHeapSize = "128m"
        arguments += ['-package', 'com.strumenta.examples.TestParser', '-no-visitor', '-no-
listener']
}
compileJava.dependsOn generateGrammarSource
compileKotlin.dependsOn generateGrammarSource

idea {
    module {
```

```
        sourceDirs += file("generated-src/antlr/main")
        generatedSourceDirs += file("generated-src/antlr/main")
        sourceDirs += file('src/main/kotlin')
        testSourceDirs += file('src/test/kotlin')
    }
}

sourceSets {
    generated {
        java.srcDir 'generated-src/antlr/main/'
    }
}
compileJava.source sourceSets.generated.java, sourceSets.main.java
compileKotlin.source  sourceSets.generated.java,  sourceSets.main.java,  sourceSets.main.
kotlin

clean{
    delete "generated-src"
}

idea {
    module {
        sourceDirs += file("generated-src/antlr/main/")
        generatedSourceDirs += file("generated-src/antlr/main")
        sourceDirs += file('src/main/kotlin')
        testSourceDirs += file('src/test/kotlin')
    }
}

test {
    useJUnitPlatform()
    testLogging {
        events "passed", "skipped", "failed"
    }
}

// this is not stricly necessary but it is useful if you are not using an IDE
// it will create a JAR with all dependencies included
task fatJar(type: Jar) {
    manifest {
        attributes 'Main-Class': 'com.strumenta.examples.TestParser.App'
    }
    baseName = project.name + '-all'
    from { configurations.compile.collect { it.isDirectory() ? it : zipTree(it) } }
    with jar
}
```

Original question from StackOverflow (but we changed question details and answer)

Removing Left Recursion in ANTLR

QUESTION

As is explained in Removing left recursion , there are two ways to remove the left recursion.

- Modify the original grammar to remove the left recursion using some procedure
- Write the grammar originally not to have the left recursion

What people normally use for removing (not having) the left recursion with ANTLR? I've used flex/bison for parser, but I need to use ANTLR. The only thing I'm concerned about using ANTLR (or LL parser in genearal) is left recursion removal.

- In practical sense, how serious of removing left recursion in ANTLR? Is this a showstopper in using ANTLR? Or, nobody cares about it in ANTLR community?
- I like the idea of AST generation of ANTLR. In terms of getting AST quick and easy way, which method (out of the 2 removing left recursion methods) is preferable?

ANSWER

You can actually have left recursion in ANTLR 4. However, this is the general pattern to remove left recursion.

Consider something like a typical parameter list:

```
parameter_list: parameter
        | parameter_list ',' parameter
        ;
```

Since you don't care about anything like precedence or associativity with parameters, this is fairly easy to convert to right recursion, at the expense of adding an extra production:

```
parameter_list: parameter more_params
        ;

more_params:
      | ',' parameter more_params
      ;
```

For the most serious cases, you might want to spend some time in the Dragon Book. Doing a quick check, this is covered primarily in chapter 4.

Original question from StackOverflow (answer modified)

ANTLR4 visitor pattern on simple arithmetic example

QUESTION

I am a complete ANTLR4 newbie, so please forgive my ignorance. I ran into this presentation where a very simple arithmetic expression grammar is defined. It looks like:

```
grammar Expressions;

start : expr ;

expr  : left=expr op=('*'|'/') right=expr #opExpr
      | left=expr op=('+'|'-') right=expr #opExpr
      | atom=INT #atomExpr
      ;

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

WS    : [ \t\r\n]+ -> skip ;
```

Which is great because it will generate a very simple binary tree that can be traversed using the visitor pattern as explained in the slides, e.g., here's the function that visits the expr:

```
public Integer visitOpExpr(OpExprContext ctx) {
  int left = visit(ctx.left);
  int right = visit(ctx.right);
  String op = ctx.op.getText();
  switch (op.charAt(0)) {
    case '*': return left * right;
    case '/': return left / right;
    case '+': return left + right;
    case '-': return left - right;
    default: throw new IllegalArgumentException("Unkown opeator " + op);
  }
}
```

The next thing I would like to add is support for parentheses. So I modified the expr as follows:

```
expr  : '(' expr ')'                  #opExpr
      | left=expr op=('*'|'/') right=expr #opExpr
      | left=expr op=('+'|'-') right=expr #opExpr
      | atom=INT #atomExpr
      ;
```

Unfortunately, the code above fails because when encountering parentheses the three attributes op,left and right are null (fails with NPE).

I think I could work around that by defining a new attribute, e.g., parenthesized='(' expr ')', and then deal with that in the visitor code. However, it seems overkill to me to have a whole extra node type to represent an expression in parentheses. A simpler but uglier solution is to add the following line of code at the beginning of the visitOpExpr method:

```
if (ctx.op == null) return visit(ctx.getChild(1)); // 0 and 2 are the parentheses!
```

I don't like the above at all because it's very fragile and highly dependent on the grammar structure.

I am wondering if there is a way to tell ANTLR to just "eat" the parentheses and treat the expression like a child. Is there? Is there a better way to do this?

Note: My end goal is to extend the example to include boolean expressions that can themselves contain arithmetic expressions, e.g., (2+4*3)/10 >= 11, that is, a relation (<,>,==,~=,etc.) between arithmetic expressions can define an atomic boolean expression. This is straight forward and I already have the grammar sketched out but I have the same problem with parenthesis, i.e., I need to be able to write stuff like (I will also add support for variables):

```
((2+4*x)/10 >= 11) | ( x>1 & x<3 )
```

ANSWER

Sure, just label it differently. After all, the alternative '(' expr ')' isn't a #opExpr:

```
expr  : '(' expr ')'                  #parenExpr
    | left=expr op=('+'|'-') right=expr #opExpr
    | left=expr op=('*'|'/') right=expr #opExpr
    | atom=INT                    #atomExpr
    ;
```

And in your visitor, you'd do something like this:

```
public class EvalVisitor extends ExpressionsBaseVisitor<Integer> {

   @Override
   public Integer visitOpExpr(@NotNull ExpressionsParser.OpExprContext ctx) {
      int left = visit(ctx.left);
      int right = visit(ctx.right);
      String op = ctx.op.getText();
      switch (op.charAt(0)) {
         case '*': return left * right;
```

```
            case '/': return left / right;
            case '+': return left + right;
            case '-': return left - right;
            default: throw new IllegalArgumentException("Unknown operator " + op);
        }
    }

    @Override
    public Integer visitStart(@NotNull ExpressionsParser.StartContext ctx) {
        return this.visit(ctx.expr());
    }

    @Override
    public Integer visitAtomExpr(@NotNull ExpressionsParser.AtomExprContext ctx) {
        return Integer.valueOf(ctx.getText());
    }

    @Override
    public Integer visitParenExpr(@NotNull ExpressionsParser.ParenExprContext ctx) {
        return this.visit(ctx.expr());
    }

    public static void main(String[] args) {
        String expression = "2 * (3 + 4)";
        ExpressionsLexer lexer = new ExpressionsLexer(CharStreams.fromString(expression));
        ExpressionsParser parser = new ExpressionsParser(new CommonTokenStream(lexer));
        ParseTree tree = parser.start();
        Integer answer = new EvalVisitor().visit(tree);
        System.out.printf("%s = %s\n", expression, answer);
    }
}
```

If you run the class above, you'd see the following output:
2 * (3 + 4) = 14

Original question from StackOverflow

Antlr4 how to build a grammar allowed keywords as identifier

QUESTION

This is a demo code

```
label:
var id
let id = 10
goto label
If allowed keyword as identifier will be
let:
var var
let var = 10
goto let
```

This is totally legal code. But it seems very hard to do this in antlr.

AFAIK, If antlr match a token let, will never fallback to id token. so for antlr it will see

```
LET_TOKEN :
VAR_TOKEN <missing ID_TOKEN>VAR_TOKEN
LET_TOKEN <missing ID_TOKEN>VAR_TOKEN = 10
```

although antlr allowed predicate, I have to control ever token match and problematic. grammar become this

```
grammar Demo;
options {
  language = Go;
}
@parser::members{
   var _need = map[string]bool{}
   func skip(name string,v bool){
      _need[name] = !v
      fmt.Println("SKIP",name,v)
   }
   func need(name string)bool{
      fmt.Println("NEED",name,_need[name])
      return _need[name]
   }
}

proj@init{skip("inst",false)}: (line? NL)* EOF;
line
   : VAR ID
   | LET ID EQ? Integer
```

```
  ;

NL: '\n';
VAR: {need("inst")}? 'var' {skip("inst",true)};
LET: {need("inst")}? 'let' {skip("inst",true)};
EQ: '=';

ID: ([a-zA-Z] [a-zA-Z0-9]*);
Integer: [0-9]+;

WS: [ \t] -> skip;
Looks so terrible.
But this is easy in peg, test this in pegjs
Expression = (Line? _ '\n')* ;

Line
  = 'var' _ ID
  / 'let' _ ID _ "=" _ Integer

Integer "integer"
  = [0-9]+ { return parseInt(text(), 10); }

ID = [a-zA-Z] [a-zA-Z0-9]*

_ "whitespace"
  = [ \t]*
```

I actually done this in peggo and javacc.

My question is how to handle these grammars in antlr4.6, I was so excited about the antlr4.6 go target, but seems I choose the wrong tool for my grammar ?

ANSWER

The simplest way is to define a parser rule for identifiers:

```
id: ID | VAR | LET;

VAR: 'var';
LET: 'let';
ID: [a-zA-Z] [a-zA-Z0-9]*;
```

And then use ID (the parser rule) instead of ID (the lexer rule) in your parser rules.

A different way is to use ID for identifiers *and* keywords, and use predicates for disambiguation. But it's less readable, so I'd use the first way instead.

Original question from StackOverflow

How to create AST with ANTLR4?

QUESTION

I've been searching A LOT about this and I couldn't find anything useful that REALLY helps me build an AST. I already know that ANTLR4 doesn't build AST like ANTLR3 used to do. Everyone say: "Hey, use visitors!", but I couldn't find any example or more detailed explanation on HOW can I do this...

I have a grammar must like C, but with every commands written in Portuguese (portuga programming language). I can easily generate the parse tree using ANTLR4. My question is: What I need to do now to create an AST?

BTW, I'm using Java and IntelliJ...

EDIT1: The closest I could get was using the answer of this topic: Is there a simple example of using antlr4 to create an AST from java source code and extract methods, variables and comments? But it only prints the name of the visited methods..

Since the first attempt didn't work for me as I expected, I tried to use a tutorial from ANTLR3, but I couldn't figure out how to use StringTamplate instead of ST...

Reading the book The Definitive ANTLR 4 Reference I also couldn't find anything related to ASTs.

EDIT2: Now I have one class to create the DOT file, I just need figure out on how to use visitors properl

ANSWER

Ok, let's build a simple math example. Building an AST is totally overkill for such a task but it's a nice way to show the principle.

I'll do it in C# but the Java version would be very similar.

The grammar

First, let's write a very basic math grammar to work with:

```
grammar Math;

compileUnit
   :  expr EOF
```

```
    ;

expr
    :  '(' expr ')'                    # parensExpr
    |  op=('+'|'-') expr               # unaryExpr
    |  left=expr op=('*'|'/') right=expr   # infixExpr
    |  left=expr op=('+'|'-') right=expr   # infixExpr
    |  func=ID '(' expr ')'            # funcExpr
    |  value=NUM                       # numberExpr
    ;

OP_ADD: '+';
OP_SUB: '-';
OP_MUL: '*';
OP_DIV: '/';

NUM :  [0-9]+ ('.' [0-9]+)? ([eE] [+-]? [0-9]+)?;
ID  :  [a-zA-Z]+;
WS  :  [ \t\r\n] -> channel(HIDDEN);
```

Pretty basic stuff, we have a single expr rule that handles everything (precedence rules etc).

The AST nodes

Then, let's define some AST nodes we'll use. These are totally custom and you can define them in the way you want to.

Here are the nodes we'll be using for this example:

```
internal abstract class ExpressionNode
{
}

internal abstract class InfixExpressionNode : ExpressionNode
{
    public ExpressionNode Left { get; set; }
    public ExpressionNode Right { get; set; }
}

internal class AdditionNode : InfixExpressionNode
{
}

internal class SubtractionNode : InfixExpressionNode
{
}

internal class MultiplicationNode : InfixExpressionNode
```

```
{
}

internal class DivisionNode : InfixExpressionNode
{
}

internal class NegateNode : ExpressionNode
{
   public ExpressionNode InnerNode { get; set; }
}

internal class FunctionNode : ExpressionNode
{
   public Func<double, double> Function { get; set; }
   public ExpressionNode Argument { get; set; }
}

internal class NumberNode : ExpressionNode
{
   public double Value { get; set; }
}
```

Converting a CST to an AST

ANTLR generated the CST nodes for us (the `MathParser.*Context` classes). We now have to convert these to AST nodes.

This is easily done with a visitor, and ANTLR provides us with a `MathBaseVisitor<T>` class, so let's work with that.

```
internal class BuildAstVisitor : MathBaseVisitor<ExpressionNode>
{
      public override ExpressionNode VisitCompileUnit(MathParser.CompileUnitContext
context)
   {
      return Visit(context.expr());
   }

      public override ExpressionNode VisitNumberExpr(MathParser.NumberExprContext
context)
   {
      return new NumberNode
      {
            Value = double.Parse(context.value.Text, NumberStyles.AllowDecimalPoint |
NumberStyles.AllowExponent)
      };
   }
```

```
public override ExpressionNode VisitParensExpr(MathParser.ParensExprContext context)
 {
    return Visit(context.expr());
 }

 public override ExpressionNode VisitInfixExpr(MathParser.InfixExprContext context)
 {
    InfixExpressionNode node;

    switch (context.op.Type)
    {
       case MathLexer.OP_ADD:
          node = new AdditionNode();
          break;

       case MathLexer.OP_SUB:
          node = new SubtractionNode();
          break;

       case MathLexer.OP_MUL:
          node = new MultiplicationNode();
          break;

       case MathLexer.OP_DIV:
          node = new DivisionNode();
          break;

       default:
          throw new NotSupportedException();
    }

    node.Left = Visit(context.left);
    node.Right = Visit(context.right);

    return node;
 }

 public override ExpressionNode VisitUnaryExpr(MathParser.UnaryExprContext context)
 {
    switch (context.op.Type)
    {
       case MathLexer.OP_ADD:
          return Visit(context.expr());

         case MathLexer.OP_SUB:
          return new NegateNode
          {
```

```
                InnerNode = Visit(context.expr())
            };

        default:
            throw new NotSupportedException();
    }
  }

  public override ExpressionNode VisitFuncExpr(MathParser.FuncExprContext context)
  {
    var functionName = context.func.Text;

    var func = typeof(Math)
      .GetMethods(BindingFlags.Public | BindingFlags.Static)
      .Where(m => m.ReturnType == typeof(double))
     .Where(m => m.GetParameters().Select(p => p.ParameterType).SequenceEqual(new[]
{ typeof(double) }))
                  .FirstOrDefault(m  =>  m.Name.Equals(functionName,  StringComparison.
OrdinalIgnoreCase));

    if (func == null)
       throw new NotSupportedException(string.Format("Function {0} is not supported",
functionName));

    return new FunctionNode
    {
            Function  =  (Func<double,  double>)func.CreateDelegate(typeof(Func<double,
double>)),
      Argument = Visit(context.expr())
    };
  }
}
```

As you can see, it's just a matter of creating an AST node out of a CST node by using a visitor. The code should be pretty self-explanatory (well, maybe except for the VisitFuncExpr stuff, but it's just a quick way to wire up a delegate to a suitable method of the System.Math class).

And here you have the AST building stuff. That's all that's needed. Just extract the relevant information from the CST and keep it in the AST.

The AST visitor

Now, let's play a bit with the AST. We'll have to build an AST visitor base class to traverse it. Let's just do something similar to the AbstractParseTreeVisitor<T> provided by ANTLR.

```
internal abstract class AstVisitor<T>
{
   public abstract T Visit(AdditionNode node);
   public abstract T Visit(SubtractionNode node);
   public abstract T Visit(MultiplicationNode node);
   public abstract T Visit(DivisionNode node);
   public abstract T Visit(NegateNode node);
   public abstract T Visit(FunctionNode node);
   public abstract T Visit(NumberNode node);

   public T Visit(ExpressionNode node)
   {
      return Visit((dynamic)node);
   }
}
```

Here, I took advantage of C#'s dynamic keyword to perform a double-dispatch in one line of code. In Java, you'll have to do the wiring yourself with a sequence of if statements like these:

```
if (node is AdditionNode) {
   return Visit((AdditionNode)node);
} else if (node is SubtractionNode) {
   return Visit((SubtractionNode)node);
} else if ...
```

But I just went for the shortcut for this example.

Work with the AST

So, what can we do with a math expression tree? Evaluate it, of course! Let's implement an expression evaluator:

```
internal class EvaluateExpressionVisitor : AstVisitor<double>
{
   public override double Visit(AdditionNode node)
   {
      return Visit(node.Left) + Visit(node.Right);
   }

   public override double Visit(SubtractionNode node)
   {
```

```
        return Visit(node.Left) - Visit(node.Right);
    }

    public override double Visit(MultiplicationNode node)
    {
        return Visit(node.Left) * Visit(node.Right);
    }

    public override double Visit(DivisionNode node)
    {
        return Visit(node.Left) / Visit(node.Right);
    }

    public override double Visit(NegateNode node)
    {
        return -Visit(node.InnerNode);
    }

    public override double Visit(FunctionNode node)
    {
        return node.Function(Visit(node.Argument));
    }

    public override double Visit(NumberNode node)
    {
        return node.Value;
    }
}
```

Pretty simple once we have an AST, isn't it?

Putting it all together

Last but not least, we have to actually write the main program:

```
internal class Program

{
    private static void Main()
    {
        while (true)
        {
            Console.Write("> ");
            var exprText = Console.ReadLine();

            if (string.IsNullOrWhiteSpace(exprText))
                break;

            var inputStream = new AntlrInputStream(new StringReader(exprText));
```

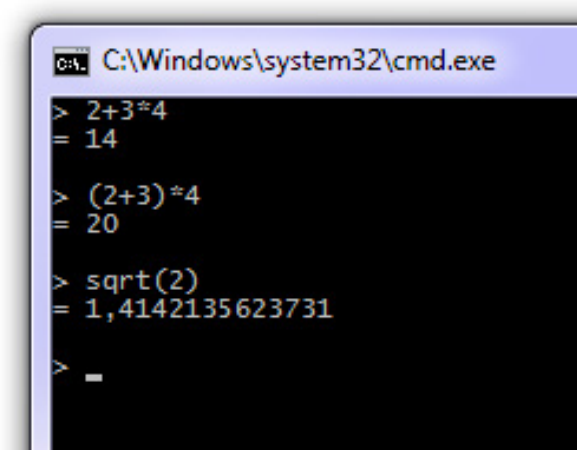
```
            var lexer = new MathLexer(inputStream);
            var tokenStream = new CommonTokenStream(lexer);
            var parser = new MathParser(tokenStream);

            try
            {
               var cst = parser.compileUnit();
               var ast = new BuildAstVisitor().VisitCompileUnit(cst);
               var value = new EvaluateExpressionVisitor().Visit(ast);

               Console.WriteLine("= {0}", value);
            }
            catch (Exception ex)
            {
               Console.WriteLine(ex.Message);
            }

            Console.WriteLine();
         }
      }
   }
```

And now we can finally play with it:



Original question from StackOverflow

If/else statements in ANTLR using listeners

QUESTION

I'm creating a simple programming language for a school project. I'm using ANTLR 4 to generate a lexer and a parser from my grammar. Until now, I have been using ANTLRs listener pattern to apply the actual functionality of the programming language.

Now I would like to implement if/else statements but I'm not sure that these can actually be implemented when using the listener pattern as ANTLR decides in which order to traverse the parse tree when using listeners and I imagine that the implementation of if/else statements will require jumping around the parse tree depending on which condition in the statement is satisfied.

Can anyone tell me if it will be possible to implement if/else statements using ANTLR or if I will have to implement the visitor pattern myself? Also, can anyone give an extremely simple example of the implementation the statements?

ANSWER

By default, ANTLR 4 generates listeners. But if you give org.antlr.v4.Tool the command line parameter -visitor, ANTLR generates visitor classes for you. These work much like listeners, but give you more control over which (sub) trees are walked/visited. This is particularly useful if you want to exclude certain (sub) trees (like else/if blocks, as in your case). While this can be done using listeners, it's much cleaner to do this with a visitor. Using listeners, you'll need to introduce global variables that keep track if a (sub) tree needs to be evaluated, and which do not.

As it happens to be, I'm working on a small ANTLR 4 tutorial. It's not done yet, but I'll post a small working example that demonstrates the use of these visitor classes and an if statement construct.

---

1. Grammar

Here's a simple grammar supporting basic expressions, if-, while- and log-statements:

Mu.g4

```
grammar Mu;

parse
```

```
 : block EOF
 ;

block
 : stat*
 ;

stat
 : assignment
 | if_stat
 | while_stat
 | log
 | OTHER {System.err.println("unknown char: " + $OTHER.text);}
 ;

assignment
 : ID ASSIGN expr SCOL
 ;

if_stat
 : IF condition_block (ELSE IF condition_block)* (ELSE stat_block)?
 ;

condition_block
 : expr stat_block
 ;

stat_block
 : OBRACE block CBRACE
 | stat
 ;

while_stat
 : WHILE expr stat_block
 ;

log
 : LOG expr SCOL
 ;

expr
 : <assoc=right>expr POW expr #powExpr
 | MINUS expr                 #unaryMinusExpr
 | NOT expr                   #notExpr
 | expr op=(MULT | DIV | MOD) expr     #multiplicationExpr
 | expr op=(PLUS | MINUS) expr         #additiveExpr
 | expr op=(LTEQ | GTEQ | LT | GT) expr #relationalExpr
 | expr op=(EQ | NEQ) expr             #equalityExpr
```

```
 | expr AND expr                  #andExpr
 | expr OR expr                   #orExpr
 | atom                           #atomExpr
 ;

atom
 : OPAR expr CPAR #parExpr
 | (INT | FLOAT)  #numberAtom
 | (TRUE | FALSE) #booleanAtom
 | ID             #idAtom
 | STRING         #stringAtom
 | NIL            #nilAtom
 ;

OR : '||';
AND : '&&';
EQ : '==';
NEQ : '!=';
GT : '>';
LT : '<';
GTEQ : '>=';
LTEQ : '<=';
PLUS : '+';
MINUS : '-';
MULT : '*';
DIV : '/';
MOD : '%';
POW : '^';
NOT : '!';

SCOL : ';';
ASSIGN : '=';
OPAR : '(';
CPAR : ')';
OBRACE : '{';
CBRACE : '}';

TRUE : 'true';
FALSE : 'false';
NIL : 'nil';
IF : 'if';
ELSE : 'else';
WHILE : 'while';
LOG : 'log';

ID
 : [a-zA-Z_] [a-zA-Z_0-9]*
 ;
```

```
INT
 : [0-9]+
 ;

FLOAT
 : [0-9]+ '.' [0-9]*
 | '.' [0-9]+
 ;

STRING
 : '"' (~["\r\n] | '""')* '"'
 ;

COMMENT
 : '#' ~[\r\n]* -> skip
 ;

SPACE
 : [ \t\r\n] -> skip
 ;

OTHER
 : .
 ;
```

Now let's say you would like to parse, and evaluate, input like this:

test.mu

```
a = true;
b = false;

if a && b {
  log "1 :: a=" + a +", b=" + b;
}
else if a || b {
  log "2 :: a=" + a +", b=" + b;
}
else {
  log "3 :: a=" + a +", b=" + b;
}

log "Done!";
```

2. Visitor I

Start by generating the parser and visitor classes:

```
java -cp antlr-4.0-complete.jar org.antlr.v4.Tool Mu.g4 -visitor
```

The command above would have generated, among others the file `MuBaseVisitor<T>`. This is the class we're going to extend with out own logic:

EvalVisitor.java

```
public class EvalVisitor extends MuBaseVisitor<Value> {
   // ...
}
```

where `Value` is just a wrapper for any of our language's types (`String`, `Boolean`, `Double`):

Value.java

```
public class Value {

   public static Value VOID = new Value(new Object());

   final Object value;

   public Value(Object value) {
      this.value = value;
   }

   public Boolean asBoolean() {
      return (Boolean)value;
   }

   public Double asDouble() {
      return (Double)value;
   }

   public String asString() {
      return String.valueOf(value);
   }

   public boolean isDouble() {
      return value instanceof Double;
   }

   @Override
   public int hashCode() {
```

```
        if(value == null) {
            return 0;
        }

        return this.value.hashCode();
    }

    @Override
    public boolean equals(Object o) {

        if(value == o) {
            return true;
        }

        if(value == null || o == null || o.getClass() != this.getClass()) {
            return false;
        }

        Value that = (Value)o;

        return this.value.equals(that.value);
    }

    @Override
    public String toString() {
        return String.valueOf(value);
    }
}
```

---

3. Test I

To test the classes, use the following Main class:

Main.java

```
import org.antlr.v4.runtime.ANTLRFileStream;
import org.antlr.v4.runtime.CommonTokenStream;
import org.antlr.v4.runtime.tree.ParseTree;

public class Main {
    public static void main(String[] args) throws Exception {
        MuLexer lexer = new MuLexer(new ANTLRFileStream("test.mu"));
        MuParser parser = new MuParser(new CommonTokenStream(lexer));
        ParseTree tree = parser.parse();
        EvalVisitor visitor = new EvalVisitor();
        visitor.visit(tree);
    }
}
```

and compile and run the source files:

```
javac -cp antlr-4.0-complete.jar *.java
java -cp .:antlr-4.0-complete.jar Main
```

(on Windows, the last command would be: `java -cp .;antlr-4.0-complete.jar Main`*)*

After running `Main`, nothing happens (of course?). This is because we didn't implement any of the rules in our `EvalVisitor` class. To be able to evaluate the file test.mu properly, we need to provide a proper implementation for the following rules:

- `if_stat`
- `andExpr`
- `orExpr`
- `plusExpr`
- `assignment`
- `idAtom`
- `booleanAtom`
- `stringAtom`
- `log`

---

4. Visitor II & Test II

Here's a implementation of these rules:

```
import org.antlr.v4.runtime.misc.NotNull;

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

public class EvalVisitor extends MuBaseVisitor<Value> {

   // used to compare floating point numbers
   public static final double SMALL_VALUE = 0.00000000001;

   // store variables (there's only one global scope!)
   private Map<String, Value> memory = new HashMap<String, Value>();
```

```
// assignment/id overrides
@Override
public Value visitAssignment(MuParser.AssignmentContext ctx) {
   String id = ctx.ID().getText();
   Value value = this.visit(ctx.expr());
   return memory.put(id, value);
}

@Override
public Value visitIdAtom(MuParser.IdAtomContext ctx) {
   String id = ctx.getText();
   Value value = memory.get(id);
   if(value == null) {
      throw new RuntimeException("no such variable: " + id);
   }
   return value;
}

// atom overrides
@Override
public Value visitStringAtom(MuParser.StringAtomContext ctx) {
   String str = ctx.getText();
   // strip quotes
   str = str.substring(1, str.length() - 1).replace("\"\"", "\"");
   return new Value(str);
}

@Override
public Value visitNumberAtom(MuParser.NumberAtomContext ctx) {
   return new Value(Double.valueOf(ctx.getText()));
}

@Override
public Value visitBooleanAtom(MuParser.BooleanAtomContext ctx) {
   return new Value(Boolean.valueOf(ctx.getText()));
}

@Override
public Value visitNilAtom(MuParser.NilAtomContext ctx) {
   return new Value(null);
}

// expr overrides
@Override
public Value visitParExpr(MuParser.ParExprContext ctx) {
   return this.visit(ctx.expr());
}
```

```
  @Override
  public Value visitPowExpr(MuParser.PowExprContext ctx) {
    Value left = this.visit(ctx.expr(0));
    Value right = this.visit(ctx.expr(1));
    return new Value(Math.pow(left.asDouble(), right.asDouble()));
  }

  @Override
  public Value visitUnaryMinusExpr(MuParser.UnaryMinusExprContext ctx) {
    Value value = this.visit(ctx.expr());
    return new Value(-value.asDouble());
  }

  @Override
  public Value visitNotExpr(MuParser.NotExprContext ctx) {
    Value value = this.visit(ctx.expr());
    return new Value(!value.asBoolean());
  }

  @Override
  public Value visitMultiplicationExpr(@NotNull MuParser.MultiplicationExprContext ctx) {

    Value left = this.visit(ctx.expr(0));
    Value right = this.visit(ctx.expr(1));

    switch (ctx.op.getType()) {
      case MuParser.MULT:
        return new Value(left.asDouble() * right.asDouble());
      case MuParser.DIV:
        return new Value(left.asDouble() / right.asDouble());
      case MuParser.MOD:
        return new Value(left.asDouble() % right.asDouble());
      default:
        throw new RuntimeException("unknown operator: " + MuParser.tokenNames[ctx.
op.getType()]);
    }
  }

  @Override
  public Value visitAdditiveExpr(@NotNull MuParser.AdditiveExprContext ctx) {

    Value left = this.visit(ctx.expr(0));
    Value right = this.visit(ctx.expr(1));

    switch (ctx.op.getType()) {
      case MuParser.PLUS:
        return left.isDouble() && right.isDouble() ?
```

```
                new Value(left.asDouble() + right.asDouble()) :
                new Value(left.asString() + right.asString());
        case MuParser.MINUS:
          return new Value(left.asDouble() - right.asDouble());
        default:
          throw new RuntimeException("unknown operator: " + MuParser.tokenNames[ctx.
op.getType()]);
      }
    }

    @Override
    public Value visitRelationalExpr(@NotNull MuParser.RelationalExprContext ctx) {

      Value left = this.visit(ctx.expr(0));
      Value right = this.visit(ctx.expr(1));

      switch (ctx.op.getType()) {
        case MuParser.LT:
          return new Value(left.asDouble() < right.asDouble());
        case MuParser.LTEQ:
          return new Value(left.asDouble() <= right.asDouble());
        case MuParser.GT:
          return new Value(left.asDouble() > right.asDouble());
        case MuParser.GTEQ:
          return new Value(left.asDouble() >= right.asDouble());
        default:
          throw new RuntimeException("unknown operator: " + MuParser.tokenNames[ctx.
op.getType()]);
      }
    }

    @Override
    public Value visitEqualityExpr(@NotNull MuParser.EqualityExprContext ctx) {

      Value left = this.visit(ctx.expr(0));
      Value right = this.visit(ctx.expr(1));

      switch (ctx.op.getType()) {
        case MuParser.EQ:
          return left.isDouble() && right.isDouble() ?
                new Value(Math.abs(left.asDouble() - right.asDouble()) < SMALL_VALUE) :
                new Value(left.equals(right));
        case MuParser.NEQ:
          return left.isDouble() && right.isDouble() ?
                new Value(Math.abs(left.asDouble() - right.asDouble()) >= SMALL_VALUE) :
                new Value(!left.equals(right));
        default:
          throw new RuntimeException("unknown operator: " + MuParser.tokenNames[ctx.
```

```
op.getType()]);
        }
    }

    @Override
    public Value visitAndExpr(MuParser.AndExprContext ctx) {
        Value left = this.visit(ctx.expr(0));
        Value right = this.visit(ctx.expr(1));
        return new Value(left.asBoolean() && right.asBoolean());
    }

    @Override
    public Value visitOrExpr(MuParser.OrExprContext ctx) {
        Value left = this.visit(ctx.expr(0));
        Value right = this.visit(ctx.expr(1));
        return new Value(left.asBoolean() || right.asBoolean());
    }

    // log override
    @Override
    public Value visitLog(MuParser.LogContext ctx) {
        Value value = this.visit(ctx.expr());
        System.out.println(value);
        return value;
    }

    // if override
    @Override
    public Value visitIf_stat(MuParser.If_statContext ctx) {

        List<MuParser.Condition_blockContext> conditions =  ctx.condition_block();

        boolean evaluatedBlock = false;

        for(MuParser.Condition_blockContext condition : conditions) {

            Value evaluated = this.visit(condition.expr());

            if(evaluated.asBoolean()) {
                evaluatedBlock = true;
                  // evaluate this block whose expr==true
                this.visit(condition.stat_block());
                break;
            }
        }

        if(!evaluatedBlock && ctx.stat_block() != null) {
            // evaluate the else-stat_block (if present == not null)
```

```
            this.visit(ctx.stat_block());
        }

        return Value.VOID;
    }

    // while override
    @Override
    public Value visitWhile_stat(MuParser.While_statContext ctx) {

        Value value = this.visit(ctx.expr());

        while(value.asBoolean()) {

            // evaluate the code block
            this.visit(ctx.stat_block());

            // evaluate the expression
            value = this.visit(ctx.expr());
        }

        return Value.VOID;
    }
}
```

When you re-compile and run Main, the following would be printed to your console:

```
2 :: a=true, b=false
```

Done!

For an implementation of all other rules, see: https://github.com/bkiers/Mu

Original question from StackOverflow

ANTLR maximum recursion depth exceeded error when parsing a file

QUESTION

I am trying to parse a C file that contains a large array. My problem is that I get a max recursion depth error when testing an array with a large initialization list.

A typical array that cause this problem looks like this.

```
const unsigned int foo[] = {
    1,0,500,21,3,5,2206,113,7,
    ...
    // A total of 320 lines of data, nothing else in the file.
};
```

This is the (partial) grammar with rules that are involved in parsing the code.
declaration

```
  : declarationSpecifiers initDeclaratorList? ';'
  | staticAssertDeclaration
  ;

[..]

initDeclaratorList
  : initDeclarator (',' initDeclarator)*
  ;

initDeclarator
  : declarator ('=' initializer)?
  ;
initializer
  : assignmentExpression
  | '{' initializerList '}'
  | '{' initializerList ',' '}'
  ;

initializerList
  : designation? initializer
  | initializerList ',' designation? initializer
  ;
```

I think that ANTLR should be able to parse a simple initialization. Do you have any advice?

ANSWER

The problem is in the rule initializerList, which use a recursive rather than an iterative stile.

This is the current rule.

```
initializerList
  :  designation? initializer
  |  initializerList ‘,’ designation? initializer
  ;
```

This is problematic because it adds one level for each value that is used to initialize the array. So, you can see how this can quickly get out of control when having a large initialization.

You should change the rule into this one:

```
initializerList
  :  designation? initializer (‘,’ designation? initializer)*
  ;
```

You see grammars use the first (very inefficient) style usually when they are directly translated from a grammar reference or an old grammar. That is because old grammars were written using the BNF (Backus–Naur form) format, which does not allow quantifiers like *.

How should I handle same rules in different lexical modes?

QUESTION

When using lexical modes sometimes you have to use the define the same rule multiple times, one for each lexical mode. Because two rules cannot have the same name, I prefix the name of the rule with the lexical mode.

For example, something like this.

```
lexer grammar ExpressionLexer;

PLUS: '+';

PUSH_MODE: '[' -> pushMode(ISLAND);

[..]

mode ISLAND;

[..]

POP_MODE: ']' -> popMode;

ISLAND_PLUS: '+';
```

So far so good. The problem is that now I have to use two tokens every time I need to reference the PLUS token in the parser.

```
parser grammar ExpressionParser;

[..]

expr: NUMBER (PLUS|I_PLUS) NUMBER;
```

Am I doing this correctly? Should I do something different?

ANSWER

What you are doing is correct. However, I think that you are missing something. You should add the command type to change the type of the token created by rules in secondary modes.

```
lexer grammar ExpressionLexer;

PLUS: '+';
```

```
PUSH_MODE: '[' -> pushMode(ISLAND);

[..]

mode ISLAND;

[..]

POP_MODE: ']' -> popMode;

ISLAND_PLUS: '+' -> type(PLUS);
```

By adding the command type you effectively rename the rules, as least as far the parser is concerned. So, the parser will only see PLUS tokens regardless of the lexical mode.

Then you can just use the PLUS tokens in your parser grammar.

```
parser grammar ExpressionParser;

[..]

expr: NUMBER PLUS NUMBER;
```

Identify and handle reserved keywords

QUESTION

I am using antlr4 to parse mathematical formulae. I would like to disallow the use of java specific keywords (like ‘new’, ‘while’, ..., etc). Is there a good way to go about this? Ideally, I would define them as a lexer rule in the grammar and then throw a syntax error if rule is matched and handle with a custom error listener though I do not know how to do this.

This is what an example output may look like:

```

line 3:3 token recognition error at: ‘for ‘

( for.x && while.new )
```

ANSWER

The simplest way to achieve this is with actions and a combination of lexer and parser rules.

First, you define the forbidden keywords, like this, before any identifier rule.

```
R_NEW: ‘new’;
ID: [a-zA-Z]+;
```

Then, you create a parser rule that accepts all bad/reserved keywords.
reserved_keyword:

```
  R_NEW
  ;
```

Finally, you replace any use of the lexer rule ID in the parser with a parser rule id that can accept both allowed identifiers and the reserved keywords.

```
id
: ID
| reserved_keyword {notifyErrorListeners(“Error reserved keyword”);}
;
```

The trick is that if a bad/reserved keyword is matched, then you raise an error, by notifying the error listeners. Notice that you cannot use the method `notifyErrorListeners` directly in lexer rules, because this is a method available only in the parser.

This does not require you to implement a custom error listener. You will get the custom error message directly.

Another way to handle this is not use an error listener or custom lexer rules at all, instead you catch an improper use of keywords by looking at the text of the identifier when you are working on the parser results. So, for instance you can notify the user while you are traversing the parse tree with a visitor or listener.

This would require more work, but has the chance of doing more. For instance, you could rename the keywords into something else, like new into expr_new.

Advanced Issues

In this section we see questions about advanced issues and limitations of ANTLR. You will not always find the one-trick to solve them, but you will have a better idea of how to deal with the advanced issues you will encounter.

Can ANTLR Parse All Context Sensitive Languages?

QUESTION

I am reading the Definitive ANTLR reference by Terence Parr, where he says:

Semantic predicates are a powerful means of recognizing context-sensitive language structures by allowing runtime information to drive recognition

But the examples in the book are very simple. What i need to know is: can ANTLR parse context-sensitive rules like:

xAy --> xBy

If ANTLR can't parse these rules, is there is another tool that deals with context-sensitive grammars?

ANSWER

ANTLR parses only grammars which are LL(*). It can't parse using grammars for full context-sensitive languages such as the example you provided. I think what Parr meant was that ANTLR can parse *some* languages that require *some* (left) context constraints.

In particular, one can use semantic predicates on "reduction actions" (we do this for GLR parsers used by our DMS Software Reengineering Toolkit but the idea is similar for ANTLR, I think) to inspect any data collected by the parser so far, either as ad hoc side effects of other semantic actions, or in a partially-built parse tree.

For our DMS-based DMS-based Fortran front end, there's a context-sensitive check to ensure that DO-loops are properly lined up. Consider:

```
 DO  20, I= ...
   DO 10, J = ...
      ...
20  CONTINUE
10  CONTINUE
From the point of view of the parser, the lexical stream looks like this:
DO  <number> , <variable> =  ...
    DO <number> , <variable> = ...
       ...
<number> CONTINUE
<number> CONTINUE
```

How can the parser then know which DO statement goes with which CONTINUE statement? (saying that each DO matches its closest CONTINUE

won't work, because FORTRAN can share a CONTINUE statement with multiple DO-heads).

We use a semantic predicate "CheckMatchingNumbers" on the reduction for the following rule:

```
block = 'DO' <number> rest_of_do_head newline
      block_of_statements
      <number> 'CONTINUE' newline ; CheckMatchingNumbers
```

to check that the number following the DO keyword, and the number following the CONTINUE keyword match. If the semantic predicate says they match, then a reduction for this rule succeeds and we've aligned the DO head with correct CONTINUE. If the predicate fails, then no reduction is proposed (and this rule is removed from candidates for parsing the local context); some other set of rules has to parse the text.

The actual rules and semantic predicates to handle FORTRAN nesting with shared continues is more complex than this but I think this makes the point.

What you want is full context-sensitive parsing engine. I know people have built them, but I don't know of any full implementations, and don't expect them to be fast.

I did follow Quinn Taylor Jackson's MetaS grammar system for awhile; it sounded like a practical attempt to come close.

Original question from StackOverflow

ANTLR4 grammar token recognition error after import

QUESTION

I am using a parser grammar and a lexer grammar for antlr4 from GitHub to parse PHP in Python3.

When I use these grammars directly my PoC code works:

antlr-test.py

```
from antlr4 import *
# from PHPParentLexer import PHPParentLexer
# from PHPParentParser import PHPParentParser
# from PHPParentParser import PHPParentListener

from PHPLexer import PHPLexer as PHPParentLexer
from PHPParser import PHPParser as PHPParentParser
from PHPParser import PHPParserListener as PHPParentListener

class PhpGrammarListener(PHPParentListener):
    def enterFunctionInvocation(self, ctx):
        print("enterFunctionInvocation " + ctx.getText())

if __name__ == "__main__":
    scanner_input = FileStream('test.php')
    lexer = PHPParentLexer(scanner_input)
    stream = CommonTokenStream(lexer)
    parser = PHPParentParser(stream)
    tree = parser.htmlDocument()
    walker = ParseTreeWalker()
    printer = PhpGrammarListener()
    walker.walk(printer, tree)
```

which gives the output

```
/opt/local/bin/python3.4 /Users/d/PycharmProjects/name/antlr-test.py
enterFunctionInvocation echo("hi")
enterFunctionInvocation another_method("String")
enterFunctionInvocation print("print statement")

Process finished with exit code 0
```

When I use the following PHPParent.g4 grammar, I get a lot of errors:

```
grammar PHPParent;
options { tokenVocab=PHPLexer; }
import PHPParser;
```

After swapping comments on pythons imports, I get this error

```
/opt/local/bin/python3.4 /Users/d/PycharmProjects/name/antlr-test.py
line 1:1 token recognition error at: '?'
line 1:2 token recognition error at: 'p'
line 1:3 token recognition error at: 'h'
line 1:4 token recognition error at: 'p'
line 1:5 token recognition error at: '\n'
...
line 2:8 no viable alternative at input '<('
line 2:14 mismatched input ';' expecting {<EOF>, '<', '{', '}', ')', '?>', 'list', 'global', 'continue', 'return', 'class', 'do', 'switch', 'function', 'break', 'if', 'for', 'foreach', 'while', 'new', 'clone', '&', '!', '-', '~', '@', '$', <INVALID>, 'Interface', 'abstract', 'static', Array, RequireOperator, DecimalNumber, HexNumber, OctalNumber, Float, Boolean, SingleQuotedString, DoubleQuotedString_Start, Identifier, IncrementOperator}
line 3:28 mismatched input ';' expecting {<EOF>, '<', '{', '}', ')', '?>', 'list', 'global', 'continue', 'return', 'class', 'do', 'switch', 'function', 'break', 'if', 'for', 'foreach', 'while', 'new', 'clone', '&', '!', '-', '~', '@', '$', <INVALID>, 'Interface', 'abstract', 'static', Array, RequireOperator, DecimalNumber, HexNumber, OctalNumber, Float, Boolean, SingleQuotedString, DoubleQuotedString_Start, Identifier, IncrementOperator}
line 4:28 mismatched input ';' expecting {<EOF>, '<', '{', '}', ')', '?>', 'list', 'global', 'continue', 'return', 'class', 'do', 'switch', 'function', 'break', 'if', 'for', 'foreach', 'while', 'new', 'clone', '&', '!', '-', '~', '@', '$', <INVALID>, 'Interface', 'abstract', 'static', Array, RequireOperator, DecimalNumber, HexNumber, OctalNumber, Float, Boolean, SingleQuotedString, DoubleQuotedString_Start, Identifier, IncrementOperator}
```

However I get no errors when running the antlr4 tool over the grammars. I'm stumped here - what could be causing this issue?

```
$ a4p PHPLexer.g4
warning(146): PHPLexer.g4:363:0: non-fragment lexer rule DoubleQuotedStringBody can match the empty string
$ a4p PHPParser.g4
warning(154): PHPParser.g4:523:0: rule doubleQuotedString contains an optional block with at least one alternative that can match an empty string
$ a4p PHPParent.g4
warning(154): PHPParent.g4:523:0: rule doubleQuotedString contains an optional block with at least one alternative that can match an empty string
```

ANSWER

Import is ANTLR4 is kind of messy.

First, tokenVocab can not generate the lexer you need. It just means that this grammar is using the tokens of PHPLexer. If you delete PHPLexer.tokens, it won't even compile!

Take a look at PHPParser.g4 where we also use options { tokenVocab=PHPLexer; }. Yet in the python script we still need to use lexer from PHPLexer to make it work. Well, this PHPParentLexer is not useable at all. That's why you got all the error.

To generate a new lexer out of combined grammar, you need to import it like this:

```
grammar PHPParent;
import PHPLexer;
```

However, mode is not supported when importing. PHPLexer itself uses mode a lot. So it's also not an option.

Can we simply replace PHPParentLexer with PHPLexer? Sadly, no. Because PHPParentParser is generated with PHPParentLexer, they are tightly coupled and can not be used seperatly. If you use PHPLexer, PHPParentParser also won't work. As for this grammar, thanks to the error recovery, it actually works, but gives some error.

There seems to be no better way but to rewrite some of the grammar. There are definitely some design issues in this import part of ANTLR4.

Original question from StackOverflow

Abort on parse error with useful message

QUESTION

I've got an ANTLR 4 grammar and built a lexer and parser from that. Now I'm trying to instantiate that parser in such a way that it will parse until it encounters an error. If it encounters an error, it should not continue parsing, but it should provide useful information about the problem; ideally a machine-readable location and a human-readable message.

Here is what I have at the moment:

```
grammar Toy;

@parser::members {

  public static void main(String[] args) {
    for (String arg: args)
      System.out.println(arg + " => " + parse(arg));
  }

  public static String parse(String code) {
    ErrorListener errorListener = new ErrorListener();
    CharStream cstream = new ANTLRInputStream(code);
    ToyLexer lexer = new ToyLexer(cstream);
    lexer.removeErrorListeners();
    lexer.addErrorListener(errorListener);
    TokenStream tstream = new CommonTokenStream(lexer);
    ToyParser parser = new ToyParser(tstream);
    parser.removeErrorListeners();
    parser.addErrorListener(errorListener);
    parser.setErrorHandler(new BailErrorStrategy());
    try {
      String res = parser.top().str;
      if (errorListener.message != null)
        return "Parsed, but " + errorListener.toString();
      return res;
    } catch (ParseCancellationException e) {
      if (errorListener.message != null)
        return "Failed, because " + errorListener.toString();
      throw e;
    }
  }

  static class ErrorListener extends BaseErrorListener {

    String message = null;
```

```
        int start = -2, stop = -2, line = -2;

        @Override
        public void syntaxError(Recognizer<?, ?> recognizer,
                                Object offendingSymbol,
                                int line,
                                int charPositionInLine,
                                String msg,
                                RecognitionException e) {
          if (message != null) return;
          if (offendingSymbol instanceof Token) {
            Token t = (Token) offendingSymbol;
            start = t.getStartIndex();
            stop = t.getStopIndex();
          } else if (recognizer instanceof ToyLexer) {
            ToyLexer lexer = (ToyLexer)recognizer;
            start = lexer._tokenStartCharIndex;
            stop = lexer._input.index();
          }
          this.line = line;
          message = msg;
        }

        @Override public String toString() {
          return start + "-" + stop + " l." + line + ": " + message;
        }
      }

}

top returns [String str]: e* EOF {$str = "All went well.";};
e: 'a' 'b' | 'a' 'c' e;
Save this to Toy.g, then try these commands:
> java -jar antlr-4.5.2-complete.jar Toy.g
> javac -cp antlr-4.5.2-complete.jar Toy*.java
> java -cp .:tools/antlr-4.5.2-complete.jar ToyParser ab acab acc axb abc
ab => All went well.
acab => All went well.
acc => Failed, because 2-2 l.1: no viable alternative at input 'c'
axb => Parsed, but 1-1 l.1: token recognition error at: 'x'
Exception in thread "main" org.antlr.v4.runtime.misc.ParseCancellationException
    at org.antlr.v4.runtime.BailErrorStrategy.recoverInline(BailErrorStrategy.java:90)
    at org.antlr.v4.runtime.Parser.match(Parser.java:229)
    at ToyParser.top(ToyParser.java:187)
    at ToyParser.parse(ToyParser.java:95)
    at ToyParser.main(ToyParser.java:80)
Caused by: org.antlr.v4.runtime.InputMismatchException
    at org.antlr.v4.runtime.BailErrorStrategy.recoverInline(BailErrorStrategy.java:85)
    ... 4 more
```

On the one hand, I feel that I'm already doing too much. Looking at how much code I wrote for what should be a simple and common task, I can't help but wonder whether I'm missing some simpler solution. On the other hand, even that doesn't seem enough, for two reasons. Firstly, while I managed to get lexer error reported, they still don't prevent the parser from continuing on the remaining stream. This is evidences by the Parsed, but string for input axb. And secondly, I'm still left with errors which don't get reported to the error listener, as evidenced by the stack trace.

If I don't install the BailErrorStrategy, I get more useful output:

```
acc => Parsed, but 2-2 l.1: mismatched input 'c' expecting 'a'
axb => Parsed, but 1-1 l.1: token recognition error at: 'x'
abc => Parsed, but 2-2 l.1: extraneous input 'c' expecting {<EOF>, 'a'}
```

Is there any way to get this kind of error messages but still bail on error?
I can see from the sources that the extraneous input message is indeed generated by the DefaultErrorStrategy, apparently after it has worked out how it would go about fixing the issue. Should I let it do that and *then* bail out, i.e. write my own variant of BailErrorStrategy which calls to super before throwing?

ANSWER

In the same situation I ended up with extending DefaultErrorStrategy and overriding report* methods. It's pretty straightforward *(you can use ANTLRErrorStrategy as well)*.

Here you can find an example of fail-fast strategy. I think in your case you can collect all errors in the same way and build detailed report.

Original question from StackOverflow

Grammar with embedded SQL/different keywords inside and outside

QUESTION

I'm building a transpiler that reads a source program in a toy language and produces C++ code and an Sqlite database. I want to be able to write embedded SQL statements (surrounded by some kind of quotation syntax) within a program in a different outer language.

Although I could treat the embedded SQL like "special strings" (i.e. not parsing inside them) I want to parse them too so that I can do tricks like resolving SQLite parameter `@foo` to lexically scoped outer language variable `foo` or automatically adding "`IF NOT EXISTS`" to "`CREATE TABLE`" statements.

I read about island grammars and lexical modes, but I'm somewhat confused as nothing in the ANTLR4 book talks about parsing two languages in one. I.e. the "islands and sea" metaphor confuses me if neither language is "sea" but rather they are "bordering countries".

One difficulty I perceive is the plethora of keywords in SQL causing lexical problems for my outer language (which I do not want to have as many keywords). I guess modes can help here, but that leads to another problem I experienced: when I experimented with lexical modes, an error message from antlr indicated they're only supported in use with lexer grammars (?) and not regular parser grammars, so does that mean I can only have a lexer? Is it just the lexer grammar then has to be in one file and the parser grammar another (i.e. I can no longer have them all in one g4 file?)

I'm also asking from a practicality standpoint ("is this is a good idea") to try to craft one grammar that parses the whole thing? Would I be better off treating the embedded SQL as special strings, then programmatically run a stand-alone SQLite parser on the extracted SQL strings as a second pass?

ANSWER

Let's start with the easy part, your doubt about lexical modes. Yes, lexical modes can help. Usually `.g4` files contain a combined grammar, so they combine the lexer and parse rule. This handy simplification is not possible when using lexical modes. As you guessed the solution is easy:

Is it just the lexer grammar then has to be in one file and the parser grammar another (i.e. I can no longer have them all in one g4 file)

You just need to put them in two separate files. For example:

```
-> LanguageLexer.g4
lexer grammar LanguageLexer;
// lexer rules

-> LanguageParser.g4
parser grammar LanguageParser;
// parser rules
```

An important point is that when using separate lexer and parser grammars, you cannot use implicitly defined tokens. So, you need to explicitly define all tokens with a lexer rule inside the lexer grammar.

Lexical modes govern how the lexer works, however they do not change the parser. The parser does not care about which lexical mode originate the tokens it sees. It only sees the tokens. Therefore, once you take care of defining the proper lexer rules to support SQL and the toy language, you can create one parser.

Coming to the more complicated question:

> *I'm also asking from a practicality standpoint ("is this is a good idea") to try to craft one grammar that parses the whole thing? Would I be better off treating the embedded SQL as special strings, then programmatically run a stand-alone SQLite parser on the extracted SQL strings as a second pass?*

Lexical modes are great for markup languages and to handle things like string interpolation. In both cases there is really one language, as far the parser is concerned Markup languages contain a lot of text, that surrounds the proper language, with a meaningful structure. String interpolation is a feature of some language that allows to simplify writing dynamic/smart strings. In most cases the syntactic rules for the smart parts of the string are the same as the outer language. So, effectively there is only one language, you just need the lexer to handle the differences between inside and outside the string.

In your case, instead there are two distinct languages, the toy language and SQL. A parser is meant to handle only one language. That is both because it can get messy to have to handle two different languages and also because it usually complicates what happens after parsing. What I mean is that a parser is just a small part of a complete program, in your case a transpiler. I am guessing that a transpiler for C++ and one for SQL would require two different structure and organization. For example, you may need to keep a list of variables to handle the declaration and definition of variables in separate C++ files (header and source).

In addition to that SQL is a complex language and one that is widely used. So it would be easier to find a ready-to-use SQL grammar and then just create a separate transpiler for SQL.

So, is this a good idea? There is not a definitive answer. If your embedded SQL statements can contain all the SQL language, then you will be better off using a separate SQL parser and transpiler to handle that language. You do not want to write a SQL parser from scratch if you do not need to. Otherwise, if your embedded SQL just contain a very limited subset of SQL and the SQL code and the surrounding toy language code are deeply linked then it may make sense to write one parser. By deeply linked I mean, for example, that both refer to the table structure of some data. In such case it would be a good idea to have one transpiler because the support code will be very similar.