Improving the performance of an ANTLR parser

Written by
Gabriele Tomassetti
in ANTLR

We had many people asking us how to improve the performance of their parser. Sometimes the parser is done with some old library, sometimes the parser is created with ANTLR. In this article we offer some suggestions on improving the performance of your ANTLR parser.

Let us start by saying this: **if you want the absolute best performance you may want to opt for a custom parser**. You will spend ten times as much on maintenance and you will be less productive, but you will get the best performance possible. This is how the official parsers for languages like C# or Java are built. A custom parser makes sense in those cases, but in the vast majority of cases, it does not. You will be much more productive using ANTLR, or another parser generator, than writing a custom parser by hand. Besides, unless you have experience in building custom parsers, you will probably get worse performance, too.

You should also check that the problem really is the parser and not what you are doing after parsing. In most applications parsing is a small part of what the program does. After all, you do not want just to parse some code, you want to do something with the information obtained with parsing.

Now that this is out of the way, let us see how to improve your ANTLR parser.

The ANTLR runtimes are different

There is just one ANTLR tool that generates the parser for all supported target languages. However, each supported language then requires a different runtime.

Every runtime will have different performance and potential issues. Runtimes will usually follow the performance characteristics of their respective language. For instance, the Python runtime will be generally slower than the C# runtime. There are some exceptions depending on the maturity of the runtime. For example, a new runtime might be less performing than what it is possible because it is still not optimized for performance.

You can see an updated list of supported targets in the official documentation: Runtime Libraries and Code Generation Targets.

This means that if you have performance problems, it might be useful to change the target language of your runtime. For example, you can generate your parser in C++ instead of Python. This does not mean that you have to rewrite your whole program in a different language. Parsers are mostly used in a pipeline, that transforms code into something else. A parser itself will produce a parse tree. This can then be transformed in an Abstract Syntax Tree and consumed by your application. A common tactic that we use for our clients is this:

- we create the parser in Java, we transform the parse tree into an AST and then emit a JSON or XML file
- the client application will then consume this file in whatever language they prefer

This allows us to be able to guarantee a certain level of performance that we may not be able to get in Python. And it also allows the client to keep using Python in their own code, which consumes the AST for some goal.

Some example numbers

It is obviously not possible to give a definitive and general answer about the performance of different runtimes. That is to say, **we cannot tell you how using a different runtime will impact your parser specifically**. However, we can give you a general idea of the difference in performance. We tested the performance of two different grammars: Kotlin and Json. The first one is a complex grammar for a programming language, the second is a simple grammar for a data format.

We tested them in Python and Java. We choose these two languages to give you an idea of the jump in performance that you get passing from a interpreted language, with dynamic typing, to a language that run on a virtual machine, with static typing. We did not test all supported languages because we cannot reliably tell if the same level of performance will hold

true for your particular grammar. We could get some nice numbers, but they will not have much practical purpose. Generally speaking, the C++ runtime is the fastest one, the Java and C# runtime have also reliably good performance.

The purpose of this test is just to show you that you can get significant improvements by choosing a different runtime. You can get the biggest jump in performance going from a “slow” language to a “fast” language. In our experience going from “fast” to “faster” is usually not worth the loss in productivity. If you are already using a language that is fast enough, you are better off working on the grammar itself.

Kotlin

These are the results for parsing a Kotlin test file.

Language	Startup	Time elapsed (seconds)
Java	Cold	0.8
Java	Warm-up	0.6
Python	Cold	13.1
Python	Warm-up	4.3

As you can see, Java is generally faster than Python. However, you might be surprised by some results. There is a big difference in parsing a Kotlin file without warm-up and with warm-up. In this case warmup means that we parse the file 10 times and

then we average the results. This difference is because the combination of two facts:

- the Kotlin test file contains example of many different Kotlin constructs
- ANTLR uses a global cache to store the results of partial parsing

This means that parsing time can decrease after you have parsed a few files.

JSON

Let's see the results for parsing a JSON file.

Language	Startup	Time elapsed (seconds)
Java	Cold	0.10
Java	Warmup	0.10
Python	Cold	1.63
Python	Warmup	1.67

The results are both similar and different. They are similar because we again see a large difference in performance between Java and Python. They are different because this time there is no difference between a cold a warmup start. This happens because in this case the grammar is simple and the test file is large. This means that the ANTLR cache has little impact. There are few parsing rules and they are already used multiple times while parsing the first

file. Basically, in such cases you get the best result the first time. This is also the reason because something like a SQL file with thousands of INSERT statements can be parsed very quickly. It takes less time parsing a file like that than a smaller but more complicated file.

Let me repeat: comparing performance of different languages is a complex subject (and touchy for some). Different languages might be optimized in different ways. Also, depending on your personal proficiency with them, you can get better or worse results. If you are interested into this specific topic you can read the discussion on the official repository: Perfomance across targets. You can also look at the ANTLR Benchmarks project. **These tests were just to show that there is a significant difference between runtimes and you might be interested in seeing for yourself how it applies to your use case**.

Changing the grammar

Changing the language of your parser might be too extreme for your case. So, the obvious alternative is to change the grammar to improve performance. **The basic rule is that to improve performance you need to reduce ambiguity and complexity in your grammar**. Note that the term ambiguous has a specific meaning in parsing, it refers to an input that can be parsed in multiple ways. Here, we mean it in the more general sense of having many similar rules.

You can surely improve performance if you can make a change that make simpler for the parser to choose one path rather than another as soon as possible. In this section we are going to see a couple of rules that are valid for all grammars. In the more advanced section we are going to see how you can study specifically your grammar to improve it.

Are semantic predicates good or bad?

Generally speaking semantic predicates usually slow down your parser. However, if you use them wisely they can improve performance. If your option is either to use only one over-complicated rule or a few with a semantic predicate, then the ones with a semantic predicate can be the better option.

The issue is that changes to semantic predicates can be tricky. For instance, as a general rule, semantic predicates should be put at the beginning for parser rules, but at the end for lexer rules.

```
1.  parser: {code}? rule ;
2.
3.  LEXER: rule {code}? ;
```

In parser rules a predicate must appear on the left hand side to have any impact in the parsing process. But that has only as much performance impact as time is needed to evaluate the predicate. However,

alternatives containing a left-hand predicate will not be stored in the ANTLR cache, because they have to be re-evaluated on each parse run. That could have an additional impact if a rule with many semantic predicates in different alternatives.

In lexer rules semantic predicates are usually positioned at the end. This is because a lexer chooses a token after it had seen the entire text captured by the rule. However, you could put a semantic predicate anywhere inside a lexer rule. About the placement the documentation says:

> *Some positions might be more or less efficient than others; ANTLR makes no guarantees about the optimal spot.*

Which basically means that is impossible to know beforehand how moving the semantic predicate inside the lexer rule will change performance. You have to experiment. It may worsen it because the semantic predicate will have to be evaluated each time the parser is looking at that particular position of the lexer rule. It can also improve it because it can quickly kill the exploration of a rule that is not valid in most cases.

While the documentation states that you could put a semantic predicate anywhere inside a lexer rule, our reader (and one of the main contributors to the ANTLR C++ runtime) Mike Lischke, pointed out is that you should not put one at the beginning of a lexer rule. That is because rules with semantic predicates

at the beginning are not cached. And in lexer rules the effect of no caching is really heavy. **All lexer rules are handled as if they were alternatives in a single big outer rule**. So, even if only one lexer rule contains a predicate on the left hand side, no lexer rule is cached at all! When you consider that all your keywords also are lexer rules, you can easily imagine how thousands of lexer rules behave, performance-wise, if they are not cached. If you are interested in an in-depth explanation specifically about the ANTLR C++ runtime you can go read it on Mike's website.

So, are semantic predicates good or bad for performance? The simple answer that you get when you are beginning to learn parsing is that they are bad. The reason is that they can have an unpredictable result on performance. They are evaluated each time the parser encounters them, but they can speed up correct parsing. However, once you experiment and test them, you find that they can help for your specific use case. For instance, we helped one client to significantly improve performance on the lexer just by adding a semantic predicate at the end of a rule.

Handling expressions

When changing the grammar you have to consider performance together with all your other needs. In particular, does it make sense to improve performance at the expense of usability and ease-of-use? A common example of this dilemma is the way

you parse expressions. The traditional way to parse expressions is with a cascade of ever more precise expressions. So, you start with the rule with the higher precedence and then you get deeper and deeper into the parse tree to determine the exact expression. A (partial) example of this approach is given by the official Python 3 reference grammar:

```
1.  expression:
2.      | disjunction 'if' disjunction 'else' expression
3.      | disjunction
4.      | lambdef
5.  disjunction:
6.      | conjunction ('or' conjunction )+
7.      | conjunction
8.  conjunction:
9.      | inversion ('and' inversion )+
10.     | inversion
```

This approach is cumbersome to handle by the user of the parser and confusing to design. It is cumbersome to handle because the parse tree will have a large depth and essentially a lot of useless nested nodes. It is confusing to design because there are so many levels that you might get confused on where to put a particular expression. However, it has better performance than the simpler alternative:

```
1.  expression:
2.      | expression ('or' expression )+
3.      | expression ('and' expression )+
4.      | atom
```

The more complicated design has a better performance because the rule in this form can be ambiguous. For example, take the input:

```
5 and 3 or 1
```

It could be considered as the combination of an `and` expression between `5` and `3 or 1` or as the combination of the or expression between `5 and 3` and `1`. The ambiguity is solved automatically by applying an order of precedence. That is the rule that comes first is chosen first. In fact, notice that technically this rule will not be used as it is, instead ANTLR itself will internally rewrite such rules in a way that can solve the ambiguity,

For instance, let's see an example straight from the official documentation.

```
1.  // this rule
2.  expr: expr '*' expr
3.      | expr '+' expr
4.      | expr '(' expr ')' // f(x)
5.      | id
6.      ;
7.
8.  // effectively becomes this one
9.  expr[int pr] : id
10.                 ( {4 >= $pr}? '*' expr[5]
11.                 | {3 >= $pr}? '+' expr[4]
12.                 | {2 >= $pr}? '(' expr[0]
    ')'
13.                 )*
14.               ;
```

This transformation is effective, but a bit less efficient than if you wrote the rules by hand.

Which approach is better depends on your use case. When we design a grammar most of the times we prefer to go for the simpler design. The gains in productivity are usually well worth the cost in performance.

However, you have to keep in mind that the difference grow larger the more complicated the one expression rule becomes. And there are also some limitations that can force you to pick the more complicated design. To explain this, we quote again Mike Lischke:

> *ANTLR4 based parsers enter a new stack frame for each evaluation of a rule. If you have a large set of expression rules, which call each other recursively, you might quickly reach the maximum stack depth of your environment. The worst result of that effect is with large enough expressions your entire app can crash unpredictably! Depending on the target language you have not much control over how to deal with such a situation. Increasing the stack depth is possible in some languages (e.g. C++), but not on all platforms. Threads (except of the main thread) usually have a pretty small stack size (IIRC only 256KB or 512KB).*

> *Because of that effect it might sometimes not be possible to parse in a separate thread.*

In short, writing nested expression rules in one direct expression might not always be possible. If your project absolutely need to meet certain performance requirements, you should opt for the more complicated design. In case you are improving an existing grammar you might also need to rewrite expressions in this manner. However, in our experience we always found other means to reach the performance our clients need it.

A few optimizations

This section has been contributed by Mike Lischke.

This is a collection of ideas on how to improve your grammar: mistakes to avoid and optimal patterns. To understand this section it is helpful to know how to read an ATN (Augmented Transition Network) graph. There is no better person to explain this than Mike Lischke himself:

> *The ATN graph in the image represents a single rule, consisting of ATN states produced by the parser generator. These are mostly interesting for developers who want to write code that uses the ATN in some way (e.g. for code completion). Usually you don't need this information for your*

grammar work. It might also come in handy to see how the ATN graph changes when you change something in the grammar (for the tuning of your grammar).

What you see in the image are circles for ATN states, with their unique ID (no 2 states share the same state number) along with a label indicating the state's type (rule start/end state, basic state etc.). You can get a bit more info by hovering with the mouse over a state until you get the tooltip. The rounded rectangle describes a rule called by this rule.

Most states are connected via transitions which describe the direction a parser has to walk when executing this state machine. A transition can be executed without consuming input (which is then called an epsilon transition, marked by that little epsilon symbol) or requires certain input to match (which is denoted as `label` *in the ATN and also attached to the transition arrow in the graph image).*

> From StackOverflow

Consolidating sub rules

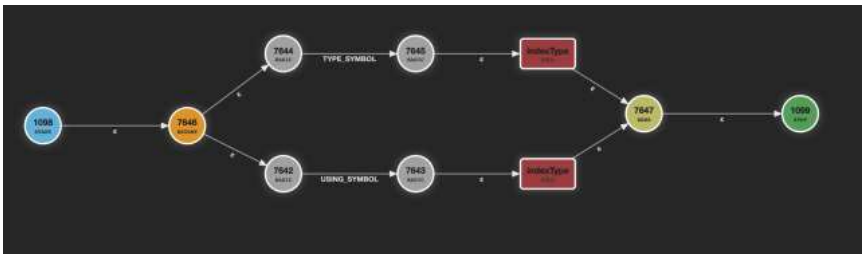
It is not so obvious when you only look at your grammar, but with the right consolidation of rule sub parts you can greatly improve speed. That idea helped me to remove a bottleneck in my grammar where many keywords where mentioned as alternatives for an identifier (the well known keyword-as-identifier problem).

Look at the following rules:

```
1.  indexTypeClause:
2.      USING_SYMBOL indexType
3.      | TYPE_SYMBOL indexType
4.  ;
5.
6.  indexTypeClauseOpt:
7.      (USING_SYMBOL | TYPE_SYMBOL)
    indexType
8.  ;
```

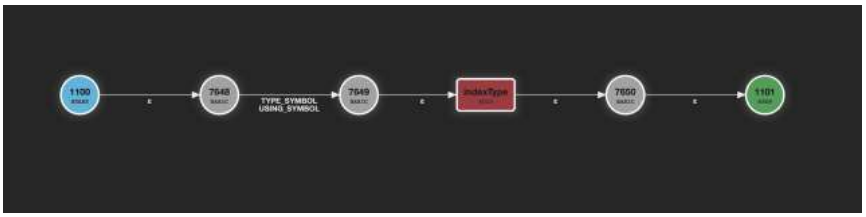
The difference is that the second one has the lexer tokens enclosed in parentheses.

The first shows a typical case where different keywords define alternatives with a common tail. For the first rule you get this ATN:



ATN for rule indexTypeClause

This is a pretty normal network. The parser has to visit two branches to find the one that matches. Now look at the picture from the second rule:



ATN for rule indexTypeClauseOpt

There is only a single branch to take and the check for the keyword to match is a simple find-in-a-set operation. Now imagine a rule with 100 such keywords (lexer tokens), each in an own alternative, which would result in 100 branches to check. Instead with a simple set of parentheses around the lexer tokens you can get a pretty remarkable speed increase, in particular if that rule is often used (like an *identifier*).

Here's a real-world example from my MySQL grammar:

```
1.  interval:
2.      intervalTimeStamp
3.      | (
4.          SECOND_MICROSECOND_SYMBOL
5.          | MINUTE_MICROSECOND_SYMBOL
6.          | MINUTE_SECOND_SYMBOL
7.          | HOUR_MICROSECOND_SYMBOL
8.          | HOUR_SECOND_SYMBOL
9.          | HOUR_MINUTE_SYMBOL
10.         | DAY_MICROSECOND_SYMBOL
11.         | DAY_SECOND_SYMBOL
12.         | DAY_MINUTE_SYMBOL
13.         | DAY_HOUR_SYMBOL
14.         | YEAR_MONTH_SYMBOL
```

```
15.       )
16.   ;
```


Note: this only works with single lexer tokens in each alternative. As soon as there is more than that single token, it will be moved out into an own branch in the ATN.

You can take this idea further, by collecting all common leading or trailing rule parts into a single entity to avoid lengthy alt checks. Here is a nice example from the MySQL grammar (with a twist):

```
1.   bitExpr:
2.       simpleExpr
3.       | bitExpr op = BITWISE_XOR_OPERATOR
    bitExpr
4.       | bitExpr op = (
5.           MULT_OPERATOR
6.           | DIV_OPERATOR
7.           | MOD_OPERATOR
8.           | DIV_SYMBOL
9.           | MOD_SYMBOL
10.      ) bitExpr
11.      | bitExpr op = (PLUS_OPERATOR |
    MINUS_OPERATOR) bitExpr
12.      | bitExpr op = (PLUS_OPERATOR |
    MINUS_OPERATOR) INTERVAL_SYMBOL expr
    interval
13.      | bitExpr op = (SHIFT_LEFT_OPERATOR |
    SHIFT_RIGHT_OPERATOR) bitExpr
14.      | bitExpr op = BITWISE_AND_OPERATOR
    bitExpr
15.      | bitExpr op = BITWISE_OR_OPERATOR
    bitExpr
```

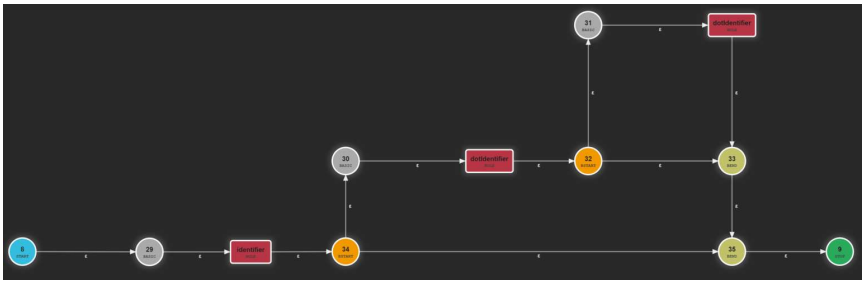
Several alts use this approach here, but you need to be careful in left-recursive rules, because if you combine operators with different precedence levels you might change the precedence completely. That's why the alt for MULT/DIV etc. is not combined with PLUS/MINUS, even though they share a common prefix.

Avoid leading optional values

What this means is that you should not start a rule or alt with an optional value. Check this rule:

```
1.  simpleIdentifier:
2.      identifier (dotIdentifier dotIdentifier?)?
3.  ;
```

That results in this ATN:

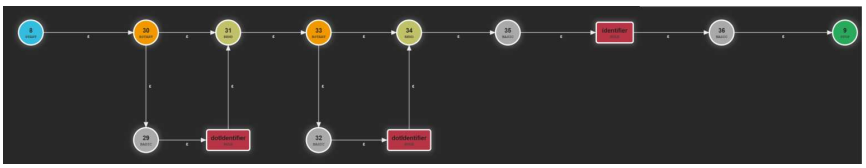


ATN for simpleIdentifier without leading optional value

You can see there is a check for both optional `dotIdentifiers`. If there's no dot then no further check for the second optional part is done.
You could also write that as:

```
simpleIdentifier:
    (dotIdentifier)? (dotIdentifier)? identifier
;
```

However, the second form requires 2 checks instead of one, with some optional parts.



ATN for simpleIdentifier with leading optional value

This is not an huge performance hog, but the first version can also save you from ambiguity issues, especially when this rule is used in another rule where also an identifier could be matched.

Ambiguities

Speaking of ambiguities: there is a case that can get you into trouble and be hard to solve, rules that can match nothing.

```
test:
    identifier*
;
```

You should never use a rule in this format, because this rule can match an empty input. You should either change the reference to the rule `identifier` to

require at least one match (e.g., `identifier+`) or embed directly the rule `test` where it is used.

You should never combine both: a rule with potential empty match and an optional call. However, ANTLR4 will warn about that.

Ensuring you are parsing the whole input

In case you are having trouble with the performance there are a couple of things you should have in your grammar. **These changes will not improve performance *per se*, but they will help you ensure that your grammar is parsing the whole input correctly**. This is important because any unexpected input can slow down parsing, especially in the lexer. You parser might not throw errors, but might parse the code in a different way than what you expect. Once you cover these basic steps, you can look further in your analysis.

The first rule is adding an EOF token to the end of your main parser rule. The EOF token is automatically defined by ANTLR to capture the end of file.

```
1.  file: statements EOF;
```

This means that your parser can handle the whole input. In other words, it ensures that you are warned if you are not parsing the whole input.

The second suggestion is adding an ANY rule at the end of the lexe

```
1.  ANY: . ;
```

It is vital that this it the last rule, otherwise it will capture any content. This lexer rule matches any character. So, if you find it somewhere in the output of the parser this means that your grammar is not handling some parts of the input. The parsing might still succeed because ANTLR can overcome typos and small errors, but these errors will impact performance.

Deep into the bowels of ANTLR

ANTLR is a complex and capable tool. If you are willing to look in its internals there are some things that you can tweak to improve performance.

Changing the Prediction Mode

A simple improvement regards the prediction mode. ANTLR can use two different *prediction modes*: SLL and LL*. They are also known as parsing strategies. Basically these dictate how much effort the parser

will put to parse the input. The first one is simpler and quicker, the second one is more powerful but a bit slower. By more powerful we mean that it can parse more complicated inputs that would result as ambiguous using just the first strategy. You can read a more in-depth explanation about the differences in the paper published by the author of ANTLR: Adaptive LL(*) Parsing: The Power of Dynamic Analysis

.

In the documentation, you will also find a third prediction mode: LL with better ambiguity detection. This is very slow and it is only used to debug your parser. So we are not interested in it when improving performance.

It is not complicated to switch between predictions mode: all you have to change is a setting. To take advantage of these two prediction modes you can use a two-steps parsing strategy:

- you configure the parser to use the SLL prediction mode
- you setup the `BailErrorStrategy`, an `ANTLRErrorStrategy` that cancel parsing as soon as it find an error. An error strategy dictates the behavior to take when finding a parsing error
- then:
    - if the parsing succeeds, all is well
    - if the parsing fails, you re-try parsing the input using the more powerful LL*

- prediction mode
  - if the parsing fails a second time, then you have an error in the input

Here it is some example code in Java to do just that.

```
1.  // this code comes from The Definitive ANTLR 4 Reference by Terence Parr, main author of ANTLR
2.  // try with simpler/faster SLL(*)
3.  parser.getInterpreter().setPredictionMode(PredictionMode.SLL);
4.  // we don't want error messages or recovery during first try
5.  parser.removeErrorListeners();
6.  parser.setErrorHandler(new BailErrorStrategy());
7.  try {
8.    parser.startRule();
9.    // if we get here, there was no syntax error and SLL(*) was enough;
10.   // there is no need to try full LL(*)
11. }
12. catch (ParseCancellationException ex) {
13.   // thrown by BailErrorStrategy
14.   tokens.reset();
15.   // rewind input stream
16.   parser.reset();
17.   // back to standard listeners/handlers
18.   parser.addErrorListener(ConsoleErrorListener.INSTANCE);
19.   parser.setErrorHandler(new DefaultErrorStrategy());
20.   // full now with full LL(*)
21.   parser.getInterpreter().setPredictionMode(PredictionMode.LL);
22.   parser.startRule();
23. }
```

The bail-out strategy is quicker because it throws an exception on an error that is not catched in the ANTLR runtime (all other parser exceptions are catched and forwarded as error callbacks via the error listeners). That way the parser will not do any recovery attempt and directly stop the parsing process.

There are some caveats in using this strategy. Obviously you are actually reducing your performance if most of your input need the LL* prediction mode to be parsed. That is because you add the time needed for your first failed attempt, in addition to the normal parsing time. In that case you need to use another approach.

Again Mike Lischke has another important observation: this strategy works well if your input is mostly correct. In case your input has many errors you will get worse results. That is because you will have to pass through two parsing attempts to determine that you actually have an error.

This strategy generally works best if you have a lot of small and simple files to parse. That is simply because there is an higher chance that a small file can be parsed using the SLL prediction mode. Of course that is not always true, it actually depends on the complexity of the file. However, as a rule of thumb it is usually true. In that case most files will be parsed using the quicker strategy and the slower one will be used rarely.

Profiling the parser

Another powerful method to improve the performance of the parser is with profiling. By profiling the parser you can see which parts of the grammar require more effort for the parser. Basically profiling records the behavior of the parser when parsing a specific input. The objective is to use this information to change the problematic parts of the grammar. Therefore you use it with an input that takes much time to parse in order to see where the parser get stuck.

Using it is quite simple.

```
1.  // standard ANTLR code
2.  // in this example we are parsing an ANTLR grammar file
3.  var lexer = ANTLRv4Lexer(CharStreams.fromFileName("Example.g4"))
4.  var commonTokenStream = CommonTokenStream(lexer)
5.  var parser = ANTLRv4Parser(commonTokenStream)
6.
7.  // activating profiling
8.  parser.setProfile(true)
9.
10.  // using the parse as usual
11.  var s = parser.grammarSpec()
12.  // looking at the result of profiling
13.  profileParser(parser)
```

You start by activating profiling on the parser. This is a standard setting included in ANTLR. Then you parse

your input. Finally you analyze the result of the profiling. A sample `profileParser` method follows:


```
fun profileParser(parser: ANTLRv4Parser) {
    print(String.format("%-" + 35 + "s", "rule"))
    print(String.format("%-" + 15 + "s", "time"))
    print(String.format("%-" + 15 + "s", "invocations"))
    print(String.format("%-" + 15 + "s", "lookahead"))
    print(String.format("%-" + 15 + "s", "lookahead(max)"))
    print(String.format("%-" + 15 + "s", "ambiguities"))
    println(String.format("%-" + 15 + "s", "errors"))
    for (decisionInfo in parser.getParseInfo().getDecisionInfo()) {
        val ds: DecisionState = parser.getATN().getDecisionState(decisionInfo.decision)
        val rule: String = parser.getRuleNames().get(ds.ruleIndex)
        if (decisionInfo.timeInPrediction > 0) {
            print(String.format("%-" + 35 + "s", rule))
            print(String.format("%-" + 15 + "s", decisionInfo.timeInPrediction))
            print(String.format("%-" + 15 + "s", decisionInfo.invocations))
            print(String.format("%-" + 15 + "s", decisionInfo.SLL_TotalLook))
            print(String.format("%-" + 15 + "s", decisionInfo.SLL_MaxLook))
            print(String.format("%-" + 15 + "s", decisionInfo.ambiguities))
            println(String.format("%-" + 15 + "s", decisionInfo.errors))
        }
    }
}
```

As you can see, in this method we simply look at results of the profiling, which is done all by ANTLR. The only thing to remember is to deactivate profiling in production, because it will slow down parsing.

What do you get exactly with profiling? I am glad you asked.

```
1.  rule                                    time
    invocations     lookahead
    lookahead(max)  ambiguities     errors
2.  module                                  233135
    1               1               1
    0               0
3.  module                                  197097
    1               1               1
    0               0
4.  module                                  159070
    1               1               1
    0               0
5.  module                                  321058
    1               3               3
    0               0
6.  module                                  173378
    1               1               1
    0               0
7.  moduleBody                              101065
    2               2               1
    0               0
```

You will get something like this: a long list of the rules of your grammar linked to the steps taken to parse this rule. It is not strictly necessary to understand exactly what `SLL_TotalLook` or `SLL_MaxLook` are. There are also no hard *good* or *bad* numbers (i.e., you cannot say “anything under 1000 is good”). You will obviously benefit from understanding the actual meaning and implication of each property, but you

can just start to look for abnormal numbers and then inspect the corresponding rule.

For instance, in one grammar that we worked on we saw a lot of values in the low thousands and one rule with over 65000 thousands for `SLL_TotalLook`. We changed that particular rule and the performance improved dramatically. I should mention also that this is just an example, there are others properties of DecisionInfo that you can see in the documentation.

A common problem that you can catch with profiling is whether the parser get confused in choosing among two similar rules. In this case there will be a lot of back-and-forth exploration between two rules which leads to high numbers in both of them. The typical solution is to eliminate the ambiguity with either semantic predicates, removing one of the rule or incorporating one in the other.

Profiling the Lexer

In some cases can be even useful to profile separately the lexer and the parser. This is especially useful in positional languages. Generally speaking when the lexer handles some complex structural logic, such languages in which whitespace is meaningful or when lines must have a certain length. In such cases the lexer can be a bottleneck because you usually must implement some custom logic in the lexer.

There is no simple way to profile exclusively the lexer, but you can take advantage of the fact that the whole

process of parsing is the sum of lexing plus proper parsing. That is to say when you parse something with ANTLR, technically the lexer is called first and then its output is fed to the proper parser. You can read

our guide to parsing, if this process seems unclear to you

. So, all you have to do is to call the lexer directly and then activate the whole process of parsing. The difference between the time needed to lex and the one to complete the whole process of parsing gives you the time spent in the lexing and the parsing phase.

What follows is an example of how you can call directly the lexer until it reaches the EOF token.

```
1.  CharStream inputStream = CharStreams.fromFileName("test.json");
2.  JSONLexer jsonLexer = new JSONLexer(inputStream);
3.  while(jsonLexer.nextToken().getType() != JSONLexer.EOF) {
4.        // wait
5.  }
6.  // if you reuse the lexer, remember to reset it first
7.  jsonLexer.reset();
```

Understanding the ANTLR cache

We have previously seen how warming-up the parser can improve performance. Now let us see why it

works and why it is necessary to run the same input at least 2 times when measuring the performance. This is not the place for a complete description of the ANTLR algorithm, but an explanation of how the cache works can be useful to improve performance further. If you want to know more on ANTLR, you can read the official paper

Adaptive LL(*) Parsing: The Power of Dynamic Analysis

.

This section has been written by Mike Lischke.

The ANTLR cache is more properly known as *DFA (Deterministic Finite Automaton) cache*. DFA simply refers to the technology used by ANTLR. It is filled with ATN configurations whenever the prediction process (the ALL(*) algorithm used by ANTLR) finds a new path through the ATN.

During the first parse run all the configs must be generated and cached, which is much slower (sometimes around 10 times) than all following parse runs with the same input, where the ATN simulator can just use the cached values. Downside: this cache grows unlimited and can consume quite a lot of RAM (though that is cheap these days). There is a method to clear this cache, but that is really only a temporary solution to lower memory load and will actually slow down parsing, because that essentially starts a new warm-up phase.

On the other hand this cache could serve as a (very special) performance improvement, if you would be

able to make it persistent and load it next time the application starts. The parser would then not have to do the warm-up all the time. However, firstly, this helps mostly if you restart your application frequently (e.g. a web app or an on-demand parsing service) and, secondly, there is no code in the ANTLR4 runtime currently to achieve that. You would have to extend the runtime yourself to add necessary get and set methods in the ATN simulator.

The DFA cache is shared between all instances of the same parser. For that it is the only structure in the entire runtime which is thread safe (which also adds to the warm up, because of access serialization). That also means that a parser performance can be impacted if another parser inserts new values into that cache in parallel. But that is a rather rare scenario.

Caching

Caching is one of the most useful techniques used in software development. We have seen how ANTLR adopt it internally, but you can apply it to the output of ANTLR parsers, too. You can cache the parse tree obtained by the parser or even the AST. An AST is the transformation of the parse tree in a tree that can be used by the rest of your code. With simple parsers you might use directly the parse tree. In most complex one it is most useful to create an AST. You write the parser to be efficient and performing, but this creates a parse tree that can be hard to work with in your

code. So, you transform the parse tree in something cleaner and that makes more sense for the rest of your program.

If parsing takes a lot of your time, you can cache either the parse tree or the AST and then recall it when needed. This is particularly useful in developer tools, that deals with parsing code almost in real time. For instance, if you are building a compiler you can cache the parsing results of each file dynamically. This would mean that effectively you will have only to parse the files that are changed from the last compilation, instead of parsing everything from scratch. In most cases where it makes sense to cache the parse tree created by ANTLR, it also make sense to cache the AST created by your transformations.

Caching can also be used in conjunction with segmenting your grammar. For example, imagine that you need to parse a language that supports string interpolation. You need to know whether you code contain a string, but you might not need to completely parse the interpolated string. You just need to know where the string ends. So, you can build a simple, more performing parser, that parses the general structure of your code and caches the results. Then, when your user needs to get to access the content of the interpolated string, you call a smaller parser designed just to handle interpolated strings.

Another useful trick can be used when you need to build a parser for an editor. You can save the state of the parser at the beginning of each line. This way, when there are changes to the text, you start parsing

from the line that has changed. Then, if you see that a specific line is in a known state, you can use the old results from that point onward.

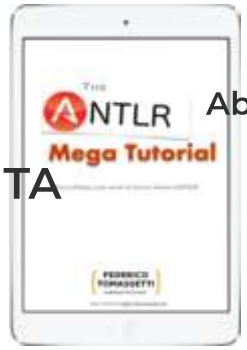
Conclusions

Many clients asks us how to improve the performance of their ANTLR parsers. In this article we have seen some of the techniques that we use to achieve just that. ANTLR is widely used and it has generally good performance, so the problem is often in the design of the grammar or even the language itself. This is where you should start, if you can. Otherwise you can use the knowledge in this article to improve the performance of your parser.

Thanks to Mike Lischke for its great feedback that helped improve this article. We add his contributions directly in some parts, but he also had feedback on the whole article.

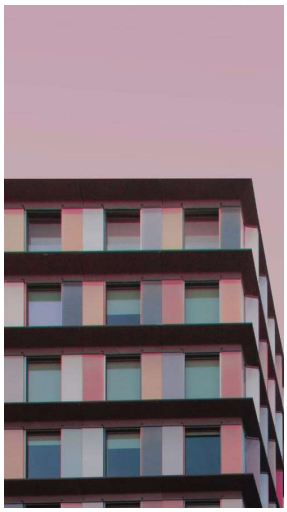
Read more:

If you want to know how to use ANTLR you can read our article The ANTLR Mega Tutorial.



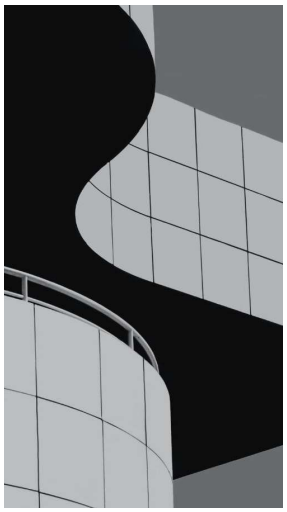
More on ANTLR

ANTLR
Application modernization
Code processing
Consulting
Domain specific languages
Editors
Jetbrains MPS
Language design
Language Engineering
Language Workbenches
Miscellany
Model driven development
Open-source
Parsing
Software Development



We better Go with ANTLR 4.11

14 September 2022



Interview with Kevin Mackey

22 June 2022