NioGram

Grammar Analysis Toolkit

User Manual

Version 1.0.0

1. Introduction

NioGram is a tool for LL(k) syntax analysis of context free grammars. Such analysis can be beneficial in the process of language and grammar design and for the process of hand-coded parser implementation.

At present the only directly supported by NioGram grammar specification language is the language of the parser generator ANTLR 4. The grammar model and analysis methods of NioGram however are not dependent on a specific grammar specification language. If appropriate parsers to NioGram AST are implemented then NioGram will be able to process grammars specified in other languages such as e.g. JavaCC, YACC, Bison etc.

This manual assumes that the readers are familiar with the theory of grammar directed syntax analysis. Readers who need to refresh or acquire knowledge on the subject will need to first look into any of the many available compiler design books. The one most favored by the NioGram author is:

R.Wilhelm, H. Seidl, S.Hack : Compiler Design – Syntactic and Semantic Analysis, Springer 2013

Familiarity of the readers with the ANTLR 4 parser generator is also beneficial since NioGram can process grammars defined in the ANTLR 4 grammar specification language. A starting point for access to relevant information is the site at http://www.antlr.org/. As a matter of convenience further in this manual ANTLR 4 is referred as simply “ANTLR”.

2. Product Overview

2.1. Use Cases

2.1.1. ANTLR IDEs

In ANTLR IDEs the data from NioGram analysis can be used to signal possible grammar inefficiencies and bugs. In the author's opinion this can be extremely helpful for ANTLR grammar developers.

2.1.2. Command Line Tool

NioGram provides a command line tool which servers the same purpose as the potential ANTLR IDE enhancements but in a less convenient way.

2.1.3. Code Generation

NioGram generates analysis data artifacts to be used in hand-coded parsers. This data is often from hard to practically impossible to collect by hand.

2.2. ANTLR Grammar Syntax Analysis

ANTLR implements an extremely powerful parsing strategy. It can deal with almost any grammar which lacks indirect left recursion. For developers this power is both a blessing and a curse. A blessing – because ANTLR will almost always do the job. A curse - because during grammar development ANTLR provides no diagnostics of possible grammar inefficiencies and errors (other than purely syntactic). NioGram mitigates this problem by providing tools for traditional **LL(k)** syntax analysis of ANTLR grammars. The information computed in the process of NioGram analysis is as follows:

- Nonterminal productivity
- Nonterminal reachability
- Nonterminal use
- Nonterminal nullability
- Grammar dependency graph
- Simple cycles in the grammar dependency graph
- Strongly connected components of the grammar depencency graph
- Left-recursive cycles in the grammar
- First/Follow sets
- FirstK/FollowK sets
- Linearized FirstK/FollowK sets
- Conflicts

Perhaps most important for ANTLR grammar development is the information about LL(k) conflicts. Even though ANTLR will typically deal with those automatically, the grammar author will be prompted to look into the corresponded rules for inefficiencies, ambiguities and bugs.

2.3. Hand Coded Parser Development

Despite the existence of excellent parser generator tools, hand-coded parsers are still being developed even for A-list languages such as Java. More often than not though the analysis data needed for hand coding is difficult to collect by hand. Unfortunately there appear to be no publicly available tools for automatic computation of the needed data. NioGram fills this gap by providing the analysis information described above. Furthermore, since NioGram supports the ANTLR grammar specification language, hand-coded parsers can be validated against parsers generated by ANTLR. NioGram also facilitates integration of hand-coded parsers with ANTLR lexers.

In a bit more detail :

First order of business in grammar development is to clean up the grammar of non-productive nonterminals and left recursion. Doing this by hand is usually feasible but with NioGram analysis the task is easier to accomplish and verify. Then the typical situation will be :

1. Most rules are **LL(1)**
2. Some rules are **LL(k)** with small k > 1.
3. A few rules may be not **LL(k)** for any k.

If the situation is worse than this then the language is either old, influential and bloated or poorly designed (or both). Apart from the subject of language implementation LL(k) properties of the grammar are important from the standpoint of ease of comprehension of the language by its "speakers". Heavily recursive features which are not LL(k) – decidable are often also difficult to comprehend, so language designers should be using those sparingly.

Finding out which rules belong to which of the above categories is a crucially important task in parser development. Doing this by hand though is far from trivial. NioGram on the other hand fully automates the task and thus makes it reliable, cheap and error free.

Parsing of the **LL(k)** rules can be easily implemented by hand in a recursive descent parser **if** the FirstK/FollowK sets are known. This is often a really big "**IF**" since the FirstK/FollowK sets for higher level nonterminals tend to be quite sizable. Collecting the information by hand is tedious and error prone. It is questionable whether the task is even practically feasible for "serious" language grammars. NioGram fully automates the process. Thus the feasibility is always guaranteed and a lot of time for development and even more time for testing and debugging is saved.

The non-**LL(k)** rules (if any) have to be resolved by one or more of the following:

1. Context dependency
2. Temporary switch to a different parsing strategy
3. Temporary switch to a different grammar
4. Backtracking
5. Ad hoc solutions

NioGram strives to facilitate the above solutions with analysis data artifacts.

3. Project Overview

The NioGram project is hosted at GitHub. The sites of the project are:

- **Home** - `https://niogram.github.io/niogram/`
- **API** - `https://niogram.github.io/niogram/apidocs/`
- **User Manual** - `https://niogram.github.io/niogram/NioGram_User_Manual.pdf`
- **Git Repository** - `https://github.com/niogram/niogram`

3.1. Project Structure

The NioGram project is Maven – based. It is a 2-tier project with 3 modules as follows:

- `niogram-core`
  The core grammar analysis library.
- `niogram-tool`
  The parser for ANTLR grammars and the command line tool.
- `niogram-complete`
  A packaging module which builds a distributive of the command line tool with all dependencies included.

3.2. Project Build

It is a standard Maven build:

```
mvn install
```

This generates in the target folders of the subprojects the project artifacts. The most directly usable artifact is `niogram-complete-x.y.z-SNAPSHOT.jar` in the folder `niogram-complete/target/`.

Where "`x.y.z`" stays for the current version.
This is the NioGram command line tool packaged with all dependencies. The tool can be invoked by:

```
java -jar niogram-complete-x.y.z-SNAPSHOT.jar
```

When invoked without arguments it prints out a self-explanatory usage message as follows:

```
Usage : niogram [options] <grammar-file>
        -q     quiet mode - do not print error messages
        -nm    parse the grammar in NioGram mode
        -sg    store the grammar serialized object
        -pd    print the parsing diagnostic information
        -pb    print the grammar basic information
        -psx   print the grammar AST in XML
        -psd   print the grammar AST in DOT
        -psr   print the grammar railroad diagrams in DOT
        -pfd   print the grammar full    dependency graph in DOT
        -prd   print the grammar reduced dependency graph in DOT
        -ppx   print the grammar parse tree in XML
        -ppd   print the grammar parse tree in DOT
        -pff   print the firstX/followX sets
        -pffc  print the LL(k) conflict information
        -ff    calculate the first   / follow   sets
        -ffk   calculate the firstK  / followK  sets
        -ffkl  calculate the firstKL / followKL sets
        -ffall calculate all firstX  / followX  sets
```

```
-k=n   set the k parameter for the LL(k) analysis
```

The options of the tool are not mutually exclusive but cumulative – multiple options can be specified simultaneously including multiple analysis options.

The project artifacts are published in the Maven central repository, so users who are not interested in building the tool themselves can download it from there.

3.3. Embedding in 3d Party Products

The project artifacts are published in the Maven central repository. The group id is "`net.ognyanov.niogram`". The aretefact IDs are:

- `niogram-core`
- `niogram-tool`
- `niogram-complete`

The analysis facilities which would be the primary motivation for embedding of NioGram are contained in the package `net.ognyanov.niogram.analysis`. The ANTLR grammar parser is in the class `net.ognyanov.niogram.parser.antlr4.Antlr4AstParser.` Please refer to the API documentation for the details of the invocation and function of these facilities.

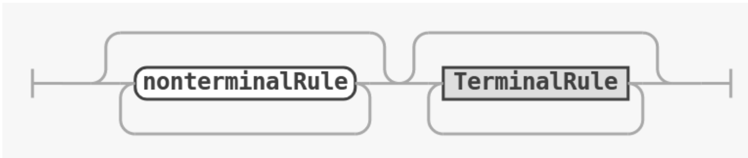
Embedding of NioGram in ANTLR IDEs deserves some special attention. Normally an ANTLR IDE maintains its own internal object model of the grammar. Typically a parse tree or a model close to it. Startup IDEs can use the NioGram parse tree as (a starting point for) their grammar object model. Preexisting IDEs with already established grammar models will need to develop a translator from the IDE grammar object model to the NioGram AST grammar model. An example of such parser is the class `net.ognyanov.NioGram.parser.GrammarParser`. It translates an ANLTR – generated parse tree into NioGram AST tree. The source code of this class should be studied by eventual AST parser implementators. One more class which is beneficial to get acquainted with is the main class of the command line tool – `net.ognanov.niogram.tool.Tool`. Finally : NioGram provides a small facility meant to facilitate the interlinking of the two object models – a publicly accessible (through a getter and a setter) field `sourceContext` which is available for use by third party AST parsers.

3.4. AST Model

The primary structure of interest for NioGram users is the AST (Abstract Syntax Tree) generated by the AST parser. It holds all substantial (for the analysis) information about the grammar in a structured form. The model of the AST is described by the following grammar :

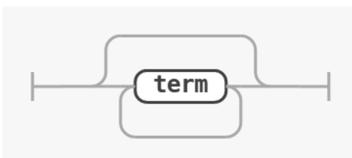
```
tokens {Nonterminal, Terminal, TerminalRule}

grammar         : nonterminalRule* TerminalRule*;
nonterminalRule : alternative+;
alternative     : term*;
term            : Terminal | Nonterminal | block;
block           : alternative*;
```

grammar_

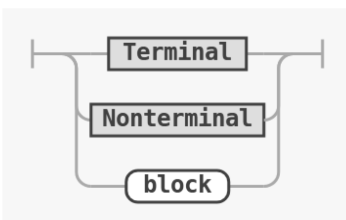


nonterminalRule

alternative

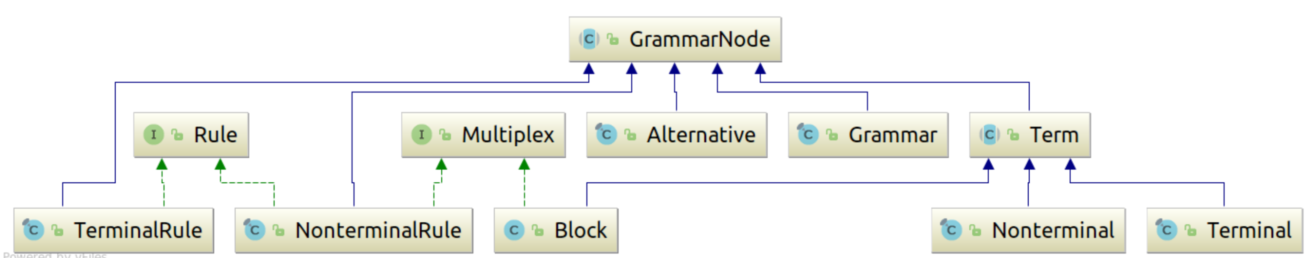


term



block

Here the names match up to first letter capitalization the names of the implementing Java classes. The type hierarchy of the implementing Java classes is as follows:



Grammar – related tools tend to have different conventions about the representation of the empty strings in grammar rules. Some use explicit epsilon symbol for that and others do not. NioGram adheres to the second convention and defines no explicit epsilon terminal in its AST model. Empty alternatives are literally empty rather than containing the epsilon terminal.

EBNF occurrence indicators have a representation in the AST only for AST blocks. Therefore if other type of term (i.e. - terminal or nonterminal) has an EBNF occurrence indicator in the source text of the grammar then it (the term) is embedded in a dedicated AST block of its own. If an occurrence indicator in the source is one of '?', '??', '*' or '*?' then an empty alternative is inserted into the block.

For the sake of precision let us finally note that even though the AST is viewed mostly as a tree, technically it is a general directed graph. Rules contain references to all their instances on the right hand side of productions and instances contain references to their correspondent rules.

4. Grammar Specification Topics

4.1. Grammar Types

NioGram focuses primarily on pure parser grammars and. Combined grammars are supported for grammar analysis too. Processing of pure lexer grammars is technically doable but pointless.

4.2. Grammar Imports

ANTLR supports certain type of grammar import into other grammars. This facility is relatively rarely used at all and even more rarely used to import parser rules. NioGram on the other hand is solely focused on parser rules and at this time does not support grammar imports.

4.3. Grammar Options

When parsing ANTLR grammars source text, NioGram supports the token vocabulary grammar option and can import token vocabularies in ANTLR format. In addition to this NioGram recognized the "k" option and if it is present, the parser sets its numeric value as both the K and KL analysis parameters in the created Grammar object. Other options are accepted and passed to the client but otherwise ignored.

4.4. Dot Expressions

ANRLR supports a feature called dot expressions which is not traditional to parser (as opposed to lexer) grammars. NioGram does not support this feature and converts all such sets to instances of a dedicated built-in terminal. Diagnostic information is provided whenever this happens. With such a substitution further grammar analysis will not be necessarily entirely correct but it may still provide useful suggestions about inefficient or/and incorrect rules.

4.5. Not Sets

ANRLR supports a feature called not sets which is not traditional to parser (as opposed to lexer) grammars. NioGram does not support this feature and converts all such sets to instances of a dedicated built-in terminal. Diagnostic information is provided whenever this happens. With such substitution further grammar analysis will not be necessarily entirely correct but it may still provide useful suggestions about inefficient or/and incorrect rules. Analysis in the presence of not sets could possibly yield somewhat better results if each individual not set was treated as an unique terminal but this enhancement does not appear to have a good enough price/performance ratio.

4.6. String Constants in Nonterminal Rules

The use of string literals in parser (as opposed to lexer) rules is a bad idea. It can be sort of convenience but can cause unexpected interference with the lexer rules. ANTLR though allows this feature, so NioGram has to address it. NioGram first ensures that all occurrences of a literal are treated as instances of one single token. Then in combined grammars and in parser grammars with token imports it tries to identify a declared lexer token which matches exactly the string. If such token is found then the literals are treated as its instances. If not then a new token is created and the literals are treated as instances of that token. In the second case correctness of the grammar analysis can not be guaranteed since it is not really known how the lexer will treat these literals.

4.7. Non-greedy occurrence indicators

ANTLR allows for non-greedy occurrence indicators in parser grammars (**??**, ***?**, **+?**). NioGram on the other hand treats all occurrence indicators as greedy. The impact on grammar analysis is difficult to assess exhaustively but is definitely not positive. NioGram provides diagnostic information in such cases.

5. Grammar Analysis Topics

5.1. Start Rule

NioGram always treats the first parser rule as the start rule of the grammar.

5.2. Nonterminal Productivity

NioGram uses the standard definition of nonterminal productivity : a nonterminal is productive if a string consisting of terminals only can be derived from it. Nonproductive nonterminals are an error which has to be eliminated from the grammar before proceeding with further analysis and implementation. For more details on the implementation of this and the following three topics please see the API documentation for the class `FlagsCalculator` in the package `net.ognyanov.niogram.analysis.`

5.3. Nonterminal Reachability

NioGram calculates reachability of nonterminal from the start rule in the dependency graph of the grammar. Normally all nonterminals with the possible exception of the start rule should be reachable. It may happen though that the developer wants to have multiple independent entry points to the parser. The presence of non-reachable nonterminals does not affect the **LL(k)** analysis.

5.4. Nonterminal Use

NioGram defines use of nonterminals as occurrence on the right side of a parser rule. Normally all nonterminals with the possible exception of the start one should be used this way. It may happen though as noted above that the developer wants to have multiple independent entry points to the parser and those are not otherwise used. The presence of unused nonterminals does not affect the **LL(k)** analysis.

5.5. Nonterminal Nullability

NioGram uses the standard definition of nullability : a string of terminal and nonterminal symbols is nullable if the empty string can be derived from it. Nullability of nonterminals is important for making parsing decisions. For nullable nonterminals the parser has to consider matches of the input with both the FirstX and the FollowX sets. By FirstX and FollowX here we refer to any type of first/follow set (see below).

5.6. Left-recursive Cycles

The presence of left-recursive cycles inevitably leads to unresolvable **LL(k)** conflicts in the grammar and prevents straightforward recursive descent implementation. It does not however invalidate the **LL(k)** analysis of the grammar. NioGram supports discovery of all left-recursive cycles in the grammar. For more details please see the API documentation of the class `GraphAnalysis` in the package `net.ognyanov.niogram.analysis.`

5.7. FirstK/FollowK Sets

If **a** is a sequence of nonterminals and terminals then:

- **FirstK(a)** is the set of strings of length not bigger than K which can occur as a prefix in some terminal-only string derived from a.
- **FollowK(a)** is the set of strings of length not bigger than K which can occur as a suffix to some string derived from **a** in any correct sentence of the language.

These sets are used in top-down parsing to choose the rule alternatives to be explored by the parser. NioGram supports computation of the FirstK/FollowK sets for any specified K. The computation however is rather expensive and in most real-life cases not really usable because of that. More practical alternatives are described below.

For more details please see the API documentation of the class `FirstKFollowKCalculator` in the package `net.ognyanov.niogram.analysis.`

5.8. Linearized FirstK/FollowK "Sets"

Calculation of FirstK and FollowK sets is a computationally expensive procedure. It is usually more practical to use instead the so-called linearized "sets" which NioGram denotes as FirstKL and FollowKL. These "sets" are actually strings of length k of sets of terminal symbols. The set at position i contains all terminals which can occur at position i in a string belonging to the correspondent FirstK/FollowK set. The FirstKL/FollowKL "sets" have less prediction power than the FirstK/FollowK sets but are much cheaper to compute.

For more details please see the API documentation of the class `FirstKLFollowKLCalculator` in the package `net.ognyanov.niogram.analysis.`

5.9. First/Follow Sets

The calculation of (an approximation to) FirstK/FollowK sets can be made even faster than FirstKL/FollowKL if we restrict ourselves to the case k=1. This approximation is denoted as First/Follow and is supported by NioGram. Its prediction/decision power is the least in the FirstX/FollowX family of sets but its computational cost is the lowest.

For more details please see the API documentation of the class `FirstFollowCalculator` in the package `net.ognyanov.niogram.analysis.`

5.10. Conflicts

LL(k) conflicts occur when 2 or more nonterminal rule alternatives have intersecting FirstX sets and the parser is not able to choose between these alternatives based on lookahead into the input. Unless the conflicts stem from the presence of a left-recursive cycle, they can always be solved in a recursive descent parser by backtracking. Backtracking however is computationally expensive and often it is more appropriate to resolve the conflicts by choosing the first matching alternative or an alternative enforced by a semantic predicate. Both options however can mask ambiguities in the grammar which are generally speaking undesirable.

For nullable nonterminals there can also be FirstX/FollowX conflicts where the FirstX set of one or more alternatives intersects with the FollowX set of the nonterminal. Such conflicts are resolved similarly to the FirstX/FirstX conflicts.

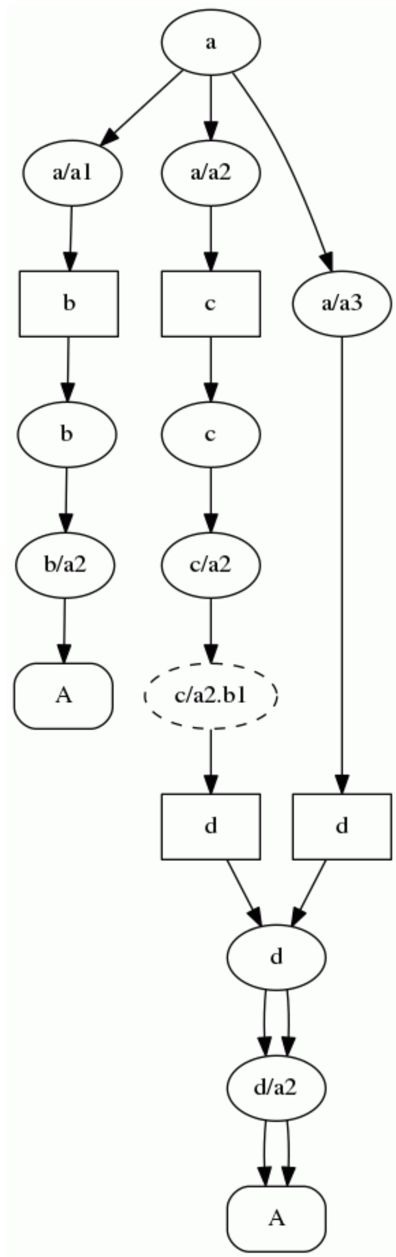
NioGram calculates conflict information as part of the FirstX/FollowX calculation. Conflicts are calculated for each alternative in rules and blocks. In addition to this, NioGram calculates FirstX/FollowX conflicts for the rules and blocks in the grammar. In case that a conflict of certain type does not exist, NioGram computes the minimum amount of lookahead for which the conflict still does not exits so that parsing decision can always be done at the lowest possible cost. For more details on the format of presentation of the conflict information please see the API documentation for the interface `net.ognyanov.niogram.ast.Multiplex`.

5.11. Terminal Occurrence Traces

When a terminal occurs in a conflict set it is good to know where the terminal “came from”. The answer to the question where do the terminals in FirstX/FollowX sets “come from” can be of interest in other cases too. Therefore NioGram supports computation of terminal occurrence traces. We will skip here the formal definition based on sequences of productions (which is rather straightforward) and will present instead an example. Let us consider the following grammar :

```
grammar traces;
tokens {A, B, C, D, E}
a : b | c | d;
b : B | A b;
c : E | C? d;
d : D | A d;
```

The terminal ‘A’ occurs at position 0 in the FirstX set of the nonterminal ‘a’. It also participates in a conflict between all 3 alternatives for ‘a’. There are 3 paths through the AST which lead from the occurrence of ‘A’ in the FirstX set of ‘a’ to the possible sources of that occurrence. The endpoints of the paths are the occurrences of ‘A’ in the rules for ‘b’ and ‘c’. Two of the paths overlap to some extent since they both pass through the rule for ‘d’. Note also that one of the paths passes through the occurrence of ‘C’ in the rule for ‘c’. That ‘C’ does not contain ‘A’ in its FirstX set but is nullable and therefore – “transparent” for ‘A’. And the complete trace for ‘A’ at position 0 in the FirstX set of ‘a’ looks as follows:



Generally a terminal occurrence trace is a directed acyclic graph which is a subgraph of the AST. Here we need to remind that in addition to the primary tree structure the AST also contains references from rules to their instances on the rhs of productions and from instances to their correspondent rules. Occurrence traces traverse the links from occurrences to their correspondent rules. All nodes in the trace either contain the specified terminal at the specified position in their FirstX/FollowX set or are nullable. Nullability is signaled in the image above by dashed edges of the correspondent node. NioGram supports computation of terminal occurrence traces for the following set types : First, FirstK, FirstKL, Follow, FollowK, FollowKL. For more details please see the API documentation for the classes `FirstTrace`, `FirstKTrace`, `FirstKLTrace`, `FollowTrace`, `FollowKTrace`, `FollowKLTrace` in the packate `net.ognyanov.niogram.analysis.`

Note that whenever NioGram computes FirstX/FollowX sets, it computes those sets not only for the rules but also for the root Grammar node of the AST. Therefore it is possible to compute grammar-wide traces of terminal occurrence.

6. Hand-Coded Parser Support

NioGram does not generate code. Even so though the results of its analysis can be used in hand-coded parsers. The command line tool has an option to record a serialized version of the analyzed grammar. This serialized object can later be deserialized and used in a hand-coded parser. Deserialized grammar objects can also be used in tools which generate more convenient and efficient (than the Grammar object itself) analysis data artifacts for use in hand-coded parsers.

The process of serialization/deserialization has two caveats as follows:

The “unique” IDs of the grammar nodes are only unique within a single running Java virtual machine. This may cause problems if nodes from different JVMs are mixed e.g. in the same container because the equality and hashing of grammar nodes are based for the sake of efficiency on the “unique” IDs only. It is (so far) the belief of the NioGram author that resolving these issues by more rigorous equality/hashing or by truly globally unique IDs would be an overkill.

The content of the source context and payload fields of the grammar nodes is not retained in the serialization/deserialization process. This is a design decision rather than a limitation of the technology.

7. Grammar Visualization

NioGram provides DOT format printouts of the grammar parse tree, the grammar AST, the railroad diagrams of all rules and the terminal occurrence traces. The parse tree and the AST can also be printed out with more details in XML format. For more details please see the API documentation for the following classes :

- `net.ognyanov.niogram.ast.GrammarNode`
- `net.ognyanov.niogram.ast.Grammar`
- `net.ognyanov.niogram.analysis.GraphAnalysis`
- `net.ognyanov.niogram.parser.antlr4.Antlr4AstParser`
- the trace classes listed above