NioGram

Grammar Analysis Toolkit

User Manual

Version 1.0.0

1. Introduction

NioGram is a Java 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 (Abstract Syntax Tree) 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. Hand-Coded Parser Development

NioGram grammar analysis generates data which is necessary for the implementation of hand-coded recursive descent parsers. This data is often from hard to practically impossible to collect by hand. With NioGram it is always readily available.

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
- FirstK/FollowK sets
- Linearized FirstK/FollowK sets
- First/Follow sets
- Conflicts
- Terminal occurrence traces

Perhaps most important for ANTLR grammar development is the information about **LL(k)** conflicts. Even though ANTLR will normally deal with those automatically, the grammar author will be prompted to look into the correspondent rules for inefficiencies, ambiguities and errors. It is often not trivial to identify the root causes of a conflict because they are “hidden behind” deep chains of rules. Terminal occurrence traces (see 5.11. below) are instrumental in solving such problems.

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 or at least the grammar is either bloated or poorly designed or both. Besides the subject of language implementation, **LL(k)** properties of the grammar are also important from the standpoint of ease of comprehension of the language by its "speakers". Apart from the trivial cases of left recursion, features which are not **LL(k)** – decidable are also difficult to comprehend and should be avoided.

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

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

At this time NioGram does not generate parsers or code of any other type. Its analysis data is available to parsers and code generation tools either by embedded use or in the form of serialized AST Java objects with embedded analysis data.

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 the 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
-pct   print the conflict traces in DOT
-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.

When the tool prints out various FirstX/FollowX sets it uses the following notation : lists are delimited by brackets ([…]) and sets are delimited by braces ({ … }); list items are separated by comas and set items are separated by dots. Let us note here (see in more detail below) that FirstK/FollowK sets are sets of lists of terminals, FirsKL/FollowKL "sets" are lists of sets of terminals and First/Follow sets are sets of terminals. Here are some examples:

- a list : `[A,B,C]`
- a set : `{A.B.C}`
- a list of sets : `[{A.B},{C.D}]`
- a set of lists : `{[A,B].[C,D]}`

The "print conflict traces" option means in more detail to print out for each conflict a dot representation of the occurrence traces (see below) for one terminal participating in the conflict. These traces are typically representative enough of the conflicts and printing out traces for all terminals can result in excessive amount of data. So the tool is restricted to one trace per conflict. Custom client code can compute traces for any/all terminals.

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 artifact IDs are:

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

The analysis tools 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.Antlr4ToAstParser.` 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 has its own internal object model of the grammar. Typically a parse tree or a model close to it. Startup IDEs can use the NioGram ANTLR - generated parse tree as (a starting point for) their grammar object model. Preexisting IDEs with already established grammar models will need to

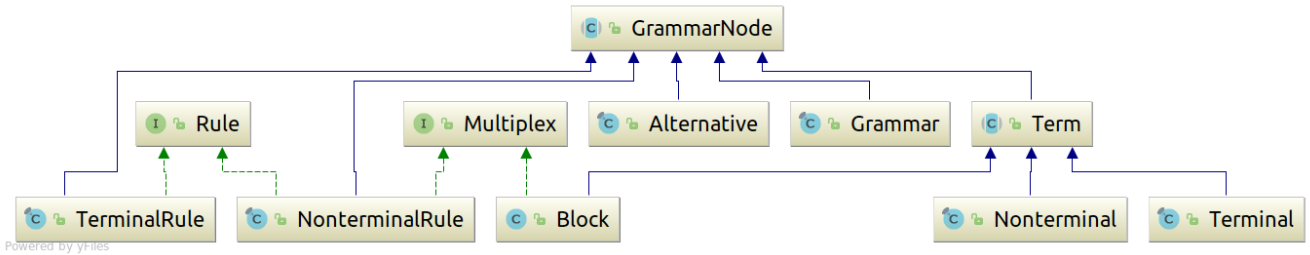
develop a translator from the IDE grammar object model to the NioGram AST grammar model. An example of such parser is the class `net.ognyanov.NioGram.parser.Antlr4ToAstParser`. It translates an ANLTR – generated parse tree into a NioGram AST tree. The source code of this class should be studied by eventual 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 support the interlinking of the two object models – a publicly accessible (through a getter and a setter) field `sourceContext` in the grammar nodes 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 and processed by the analysis tookit. 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*;
```

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 string 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 also 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.

For the sake of convenience alternatives and blocks in AST instances are assigned display names by the following scheme:

- alternative number `n` of a rule or block is assigned a display name `<rule or block name>/an`
- block number `n` in an enclosing alternative is assigned a display name`<alternative name>.bn`

For example `expression/a1.b2/a3.b4` could be a name for the fourth block in the third alternative of the second block of the first alternative of the rule `expression`.

When grammar nodes are printed out in DOT format the names of the alternatives are shortened to the last "`an`" in order to improve readability of the diagram. In other diagrams where the alternatives are present their names are printed in full.

NioGram does not invest much code and performance into making the AST model fool-proof. Client code has the technical capability to corrupt the structure and the data content of the AST. Analysis though is only guaranteed to work correctly (or to work at all) if the following rules are observed:

1. Client code never changes the structure of the AST.
2. Client code invokes directly only the following data mutation methods:
    1. `GrammarNode.setSourceContext()`
    2. `GrammarNode.setPayload()`
    3. `Grammar.setK()`
    4. `Grammar.setKL()`
    5. `Grammar.clearFlags()`
    6. `Grammar.clearFF()`
    7. `Grammar.clearFFK()`
    8. `Grammar.clearFFKL()`

4. Grammar Specification Topics

4.1. Grammar Types

NioGram focuses primarily on pure parser grammars. 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 inclusion into other grammars. This facility is relatively rarely used at all and is almost never used to import parser (as opposed to lexer) rules. NioGram is solely focused on parser rules and the price/performance ratio of grammar imports for it appears to be very low. Therefore at this time NioGram 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 of NioGram 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.

4.6. String Literals in Nonterminal Rules

The use of string literals in parser (as opposed to lexer) rules is not a good idea. It is sort of convenient but can be a source of 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 terminal. Then in combined grammars and in parser grammars with token imports it tries to identify a declared terminal which matches exactly the string. If such terminal is found then the literals are treated as its instances. If not then a new terminal is created and the literals are treated as instances of that terminal. 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. It is a good practice in grammar design to always suffix the start rule productions with EOF but NioGram does not enforce this or amend the rule on its own.

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. Syntax analysis in the presence of nonproductive nonterminals is technically feasible but these nonterminals are an error which has to be eliminated from the grammar. 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 terms 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.`

A grammar with left recursion can always be transformed into an equivalent grammar (generating the same language) without left recursion. The transformed grammar however may and is even likely to not reflect well the semantics of the language and for this reason may not be acceptable for practical purposes.

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 terminals 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. There is a subtle technical difference between FirstK/FollowK and FirstKL/FollowKL with k=1 on one hand and First/Follow on the other hand. The latter never contain the empty string while the first may contain it.

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 limited amount of lookahead into the input. 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.

NioGram calculates conflict information as part of the FirstX/FollowX calculation. Conflicts are calculated for each alternative in rules and blocks. In case that a conflict of certain type does not exist, NioGram computes the minimum amount of lookahead for which a conflict still does not exits so that parsing decisions can always be done at the lowest possible cost. That can even be 0 if the rule has a single alternative. 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`.

Conflicts can sometimes be eliminated by the so-called “left factoring” which is the technique of extracting common prefixes from two or more alternatives of a rule. Consider for example the following grammar fragment:

```
a : x y | x z;
```

The common prefix ‘`x`’ of the two alternatives results in an **LL(k)** conflict. The conflict may not be resolvable for any k if ‘`x`’ can generate terminal strings of unlimited length. This grammar fragment however can be transformed into the following equivalent (from the standpoint of language generation) :

```
a : x aRest;

aRest : y | z;
```

and the conflict disappears. The price paid for this kind of conflict elimination is that the transformed grammar reflects less closely the semantics of the language. Problems of this kind can escalate if the the grammar transformation has to be accomplished across several levels of nonterminal definitions. Consider for example this grammar segment:

```
a  : x T1 | y T2  | z;

y  : y1 s1 | z1 ;

y1 : y2 s2 | z2;

y2 : x s3 | z3;
```

There is a conflict between the first two alternatives for ‘a’ caused by the fact that ‘y’ can generate a string starting with ‘x’. To eliminate the conflict we can incrementally eliminate from the grammar ‘y2’, ‘y1’ and ‘y’ by replacing all their occurrences with equivalent blocks. The result will be :

```
a  : x T1  | (((x  s3 | z3) s2 | z2) s1 | z1) T2   | z;
```

If we now eliminate the parenthesises, we will end up with several new alternatives for ‘a’ and one of them will start with ‘x’. This new alternative and the old alternative starting with ‘x’ can then be subjected to left-factoring. It is however questionable whether such transformation is acceptable from the standpoint of the language semantics implementation.

Finally let us note that left factoring is not always feasible even in the presence of a common prefix. Here is an example:

```
s : a | b;

a : L a R | X;

b : L b R | Y;
```

Here we can not factor `L*` out of `a` and `b` in order to apply left factoring because then the grammar would not guarantee that the number `R`s is equal to the number of `L`s. This is a simplified model of various bracketing constructs in programming languages where opening brackets can not be factored out because then the grammar would not ensure proper bracket matching.

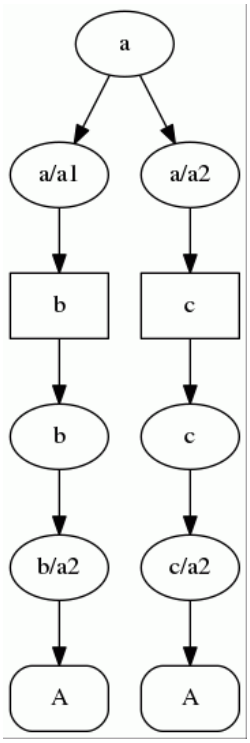
A more detailed discussion of the other methods for conflict resolution mentioned in 2.3. is out of the scope of this document.

5.11. Terminal Occurrence Traces

When a terminal occurs in a conflict set then in order to eliminate or otherwise resolve the conflict 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}
a : b | c ;
b : B | A b;
c : C | A c;
```

The terminal ‘A’ occurs at position 0 in the FirstX sets of the nonterminal ‘a’. It also participates in a **LL(1)** conflict between the two alternatives for ‘a’. There are two 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’. And this is how the trace diagram looks:



Traces are always trees (possibly degenerated to lists) and the leaf nodes are always instances of the traced terminal. All nodes in the trace either contain the traced terminal in their FirstX set or are nullable (or both).

In order to demonstrate a meaningful trace based on FollowX sets, let us consider an example of the classic “dangling ELSE” problem:

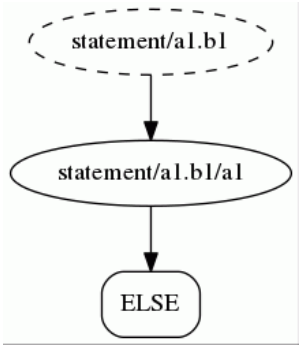
```
grammar ffconflict;
statement:
      IF LPAREN expression RPAREN
      THEN statement
      (ELSE statement | )
      | expression SEMI
      ;
expression:
      expression PLUS expression
      | LPAREN expression RPAREN
      | INTEGER
      ;

IF : 'if';
THEN : 'then';
ELSE : 'else';
PLUS : '+';
```

```
LPAREN : '(';
RPAREN : ')';
SEMI : ';';
INTEGER : '0' | [1-9] [0-9]*;
```

The problem with this grammar is its ambiguity. In statements like e.g. :

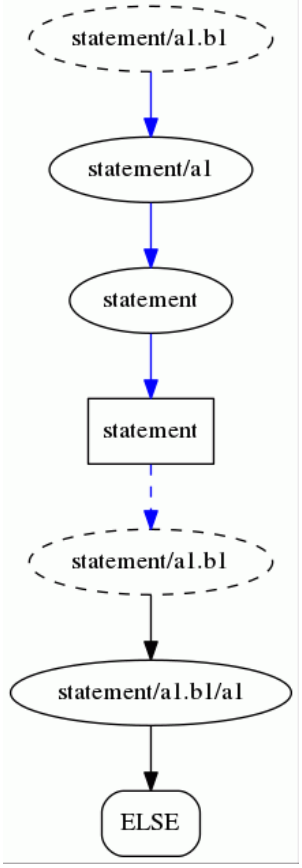
```
if(1) then if(2) then 3; else 4;
```

the `else` clause can be associated with either of the `if` clauses. In the LL(k) analysis this results in a FitstX/FollowX conflict on `ELSE` for the block `statement/a1.b1`. The FirstX trace of the conflict is:



Here the block has a dashed border in order to signal that it is nullable.

The followX trace for the conflict is:



Here the blue edges signify transitions based on the presence of ELSE in the FollowX set of the node and black edges signify transitions based on the presence of ELSE in the FirstX set of the node. The dashed blue edge signals the transition between the two. In FollowX traces the final segments at the bottom of the tree branches (and only these segments) are always FirstX – based. The sketchy explanation of this is that terminals occur in the follow set of a term in a production when they are in the first set of the suffix to the term in that production. But if the suffix is nullable then we also have to look into the follow set of the enclosing block or rule and this is how the blue edges come into being.

Terminals can in principle be traced for occurrence at any position in the a FirstX/FollowX set. Traces for positions beyond 0 though tend to be difficult to comprehend. For example - because of "interactions" between repeatable and nullable terms. Therefore NioGram restricts trace computation to position 0 in the conflict sets.

Note that whenever NioGram computes FirstX/FollowX sets for grammar rules, it also stores in the root Grammar node the unions of these sets. Therefore it is possible to compute grammar-wide terminal occurrence traces.

For more details please see the API documentation of the classes `TerminalTrace` and `TerminalTraceFactory` in the package `net.ognyanov.niogram.analysis.`

6. Hand-Coded Parser Support

NioGram is available for embedding in code generation tools related to language recognition and implementation. At the same time even though NioGram itself does not generate code, the results of its analysis can be used directly. 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 by tools which generate more convenient and efficient (than the Grammar object itself) analysis data artifacts for use in hand-coded parsers.

The command line tool records the serialized Grammar object in the folder where the grammar source file resides. The name is the same as the name of the grammar file except for the extension which is ".ser" instead of ".g4".

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

The IDs of the grammar nodes are only unique within a single running Java virtual machine. This may cause problems e.g. if nodes from different JVMs are mixed in the same container because the equality and hashing of nodes are based for the sake of efficiency on the 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.

By design decision the content of the source context and payload fields of the grammar nodes is not retained in the serialization/deserialization process.

7. Grammar Visualization

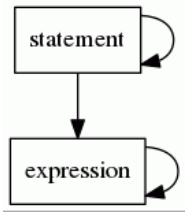
NioGram provides DOT format printouts of the grammar parse tree, the grammar AST, the grammar dependency graphs, the railroad diagrams of all rules and of the terminal occurrence traces. The parse tree and the AST can also be printed out with more details in XML format. For more details please see the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

- `net.ognyanov.niogram.ast.GrammarNode`
- `net.ognyanov.niogram.ast.Grammar`
- `net.ognyanov.niogram.analysis.GraphAnalysis`
- `net.ognyanov.niogram.analysis.TerminalTrace`
- `net.ognyanov.niogram.parser.antlr4.Antlr4ToAstParser`

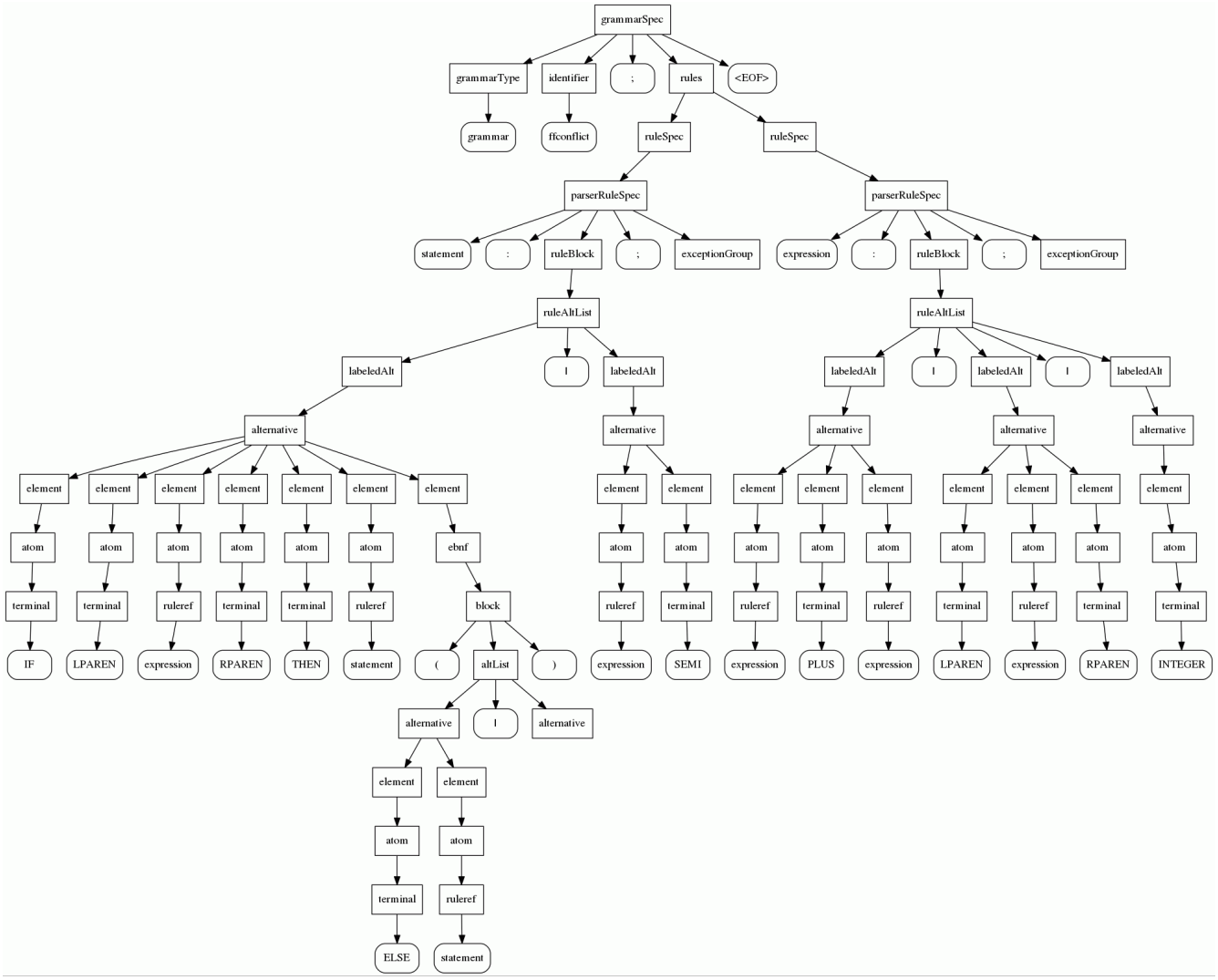
For illustration we present below a grammar dependency graph, a parse tree diagram, an AST diagram and railroad diagrams for the second grammar from 5.11. Some of the diagrams are not very readable when scaled down to fit in this document but nevertheless they do give an idea of the format.

```
grammar ffconflict;
statement:
      IF LPAREN expression RPAREN
      THEN statement
      (ELSE statement | )
      | expression SEMI
      ;
expression:
      expression PLUS expression
      | LPAREN expression RPAREN
      | INTEGER
      ;
```

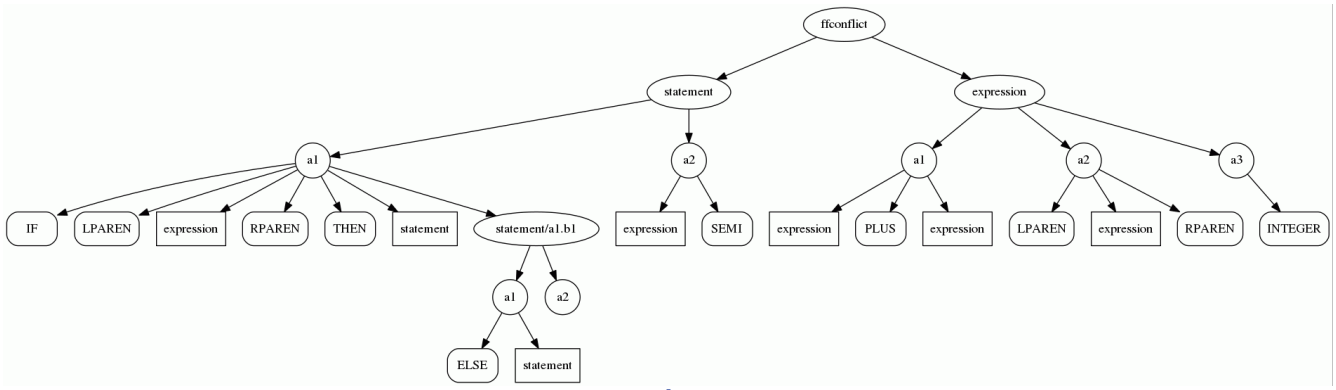
```
IF : 'if';
THEN : 'then';
ELSE : 'else';
PLUS : '+';
LPAREN : '(';
RPAREN : ')';
SEMI : ';';
INTEGER : '0' | [1-9] [0-9]*;
```



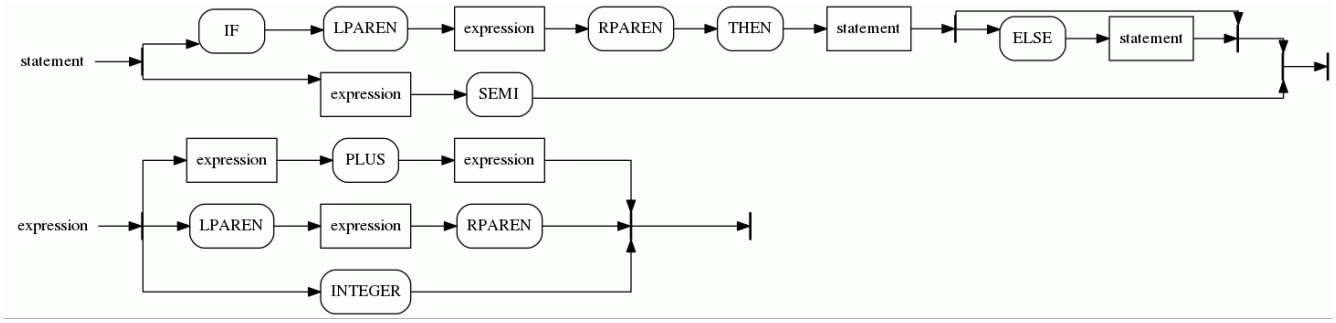
Dependency Diagram



Parse Tree Diagram



AST Diagram



Railroad Diagrams