Conflicts in the classic LR grammars

KIRILL KOBELEV

One of the reasons why the syntax of widely used programming languages, like C++, cannot be fully described with LR(n) grammar are grammar conflicts. The first part of this paper presents method that helps to understand the nature of the grammar conflicts. This method allows enumerating contexts where the grammar conflicts can appear and finding symbol sequences that can precede and follow the conflict location.

Conflicts in the formal grammars are typically described as shift/reduce and reduce/reduce. This paper shows that all grammar conflicts have shift/shift nature. Proposed method of conflict analysis reveals the hidden conflicting rules. Pairs of these rules conflict with each other. The conflict, which happens in these rules, is hidden under several non terminal substitutions. The conflicting positions in these rules always stay inside the rules. This makes it possible to say that these rules conflict in a shift/shift manner.

C++ Standard contains human language wording about the grammar ambiguities (conflicts) and some guidelines on resolving them. There is no guarantee that this list of ambiguities is complete and that these guidelines describe all possible cases. This paper proposes annotating grammar with conflict location markers and adding special "expected conflict" statements. This allows having formal list of all ambiguities in the grammar. In practice conflicts are resolved either by a manually written code or by using one of the flavors of the GLR/Backtracking algorithms. This paper proposes alternative approach that is based on using a hierarchy of derived nested grammars. Nested grammars are automatically derived from the main grammar when the grammar definition parser processes the conflict resolution statements inside the definition of the expected conflict.

Author's address: kobelev@cdsan.com

Information about the author: Kobelev graduated from the Moscow Institute of Physics and Technology (State University), worked in the Keldysh Institute of Applied Mathematics of the Russian Academy of Sciences and at the Microsoft Corporation.

Categories and Subject Descriptors: D.3.1 [Programming Languages]: Formal Definitions and Theory---*Syntax*; F.4.2 [Mathematical Logic and Formal Languages]: Grammars and Other Rewriting Systems---*Parsing*;

General Terms: Algorithms, Languages, Standardization, Theory.

Additional Key Words and Phrases: LR(1) grammar, grammar conflict, grammar conflict context, hidden conflicting rule, expected conflict, conflict resolution, nested grammar, derived grammar..

1. FOREWORD

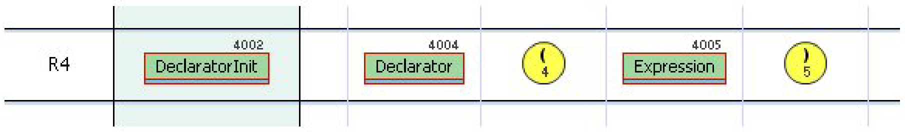
This paper describes results that were obtained using the research parser that is able to scan C/C++ style source code and grammar definitions. The code of this parser is not based on any existing compiler building tool. This parser was developed from scratch. Grammar definition language is similar to the one used in YACC, although it is not identical to it. The core part of the grammar definition language is the grammar rule. This parser is using the simplest form of the rule. Rules cannot have optional parts marked with "[ ]" or repetition groups marked with "…" as some other grammar definition languages allow. The rules are always direct and explicit. After scanning the grammar source files the grammar definition parser builds the analysis matrix using the full LR(1) algorithm. This algorithm is well known and it is described in details [Aho et al. 1986]. This algorithm is used without any changes. This means that all results presented in this paper can be verified using any other compiler building tool that implements the full LR(1) algorithm.

The research parser is accompanied with the grammar conflicts analysis engine and grammar structure viewer that can display sets of rules where the certain parsing state can appear, the trees of parsing state transitions for any given grammar rule, and other reports. The screen shots from this viewer are widely used as illustrations in this paper.

Like other grammar definition parsers, this parser uses numeric values to identify symbols, rules and other objects. The symbol value 0 is reserved for EOF. The terminal symbols are numbered starting from 1. The non terminals are numbered starting from 4000. Parser assigns indexes to the grammar rules starting from 0 in the order of rules discovery. Conflicts are marked as C0, C1, C2… in the order of discovery. In several cases grammar structure viewer shows rules in a table like form. Terminal symbols are presented in these tables as circles and non terminal symbols as rectangles. Viewer displays the name of the symbol in the grammar and its numeric symbol value. Displaying the numeric value makes sense because sometimes the name of the symbol is obscured or truncated. For example the source rule

```
DeclaratorInit:          Declarator  '('  Expression  ')'
```

is displayed in the form of the table as



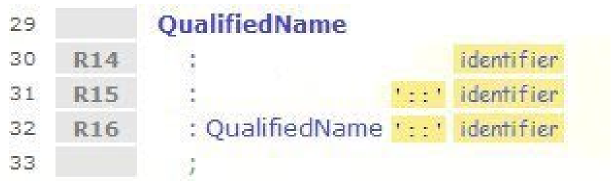
This table shows the rule number 4. The non terminal of the rule is highlighted with the bright vertical delimiters and with the colored background.

All examples in this paper, except for small ones, are based on the sample grammar that is a small part of the C++ language. Full C++ grammar consists of about 500 rules. The sample grammar has only 18 rules. Its analysis table has 57 states and it has 9 grammar conflicts. This tiny piece of the C++ language allows statements like:

```
int     f(int x);            // Function prototype.
C1      v1(20);              // Variable of the application defined type.
int     (*px)(C2 *b);        // Pointer to function with one parameter.
```

From the grammar stand point this small 18 rules grammar still allows conflicts between the declarator in parenthesis, variable with parameters of the constructor and the list of formal parameters in the prototype of the function. Conflicts appear because all these 3 constructs start from the same symbol **‘(’** and it turns out that they can be present in the same context.

One more type of conflict was introduced in this grammar intentionally. Non terminal QualifiedName is defined with the rules, presented on the picture 1.



```
      QualifiedName
R14     :                           identifier
R15     :                    '::'   identifier
R16     : QualifiedName      '::'   identifier
        ;
```

Picture 1.

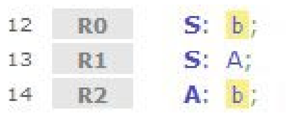
This is a well known method of defining names that can belong to any namespace. Identifier is the terminal symbol in this grammar. The non terminal QualifiedName is used everywhere where the identifier can be present. This allows statements like:

```
N1::C3      ::v2;                   // Variable definition.
```

This example illustrates the conflict that happens when the QualifiedName is built. Parser has problem deciding whether it should consider this sample as 2 separate QualifiedNames or as only one. This conflict was introduced to illustrate specific type of steps in the derivation paths that are discussed below. Full list of rules of the sample grammar is present in Appendix A. The complete grammar with the demo application can be downloaded from the web site http://www.cdsan.com.

2. SIMPLE GRAMMAR CONFLICT

In fact, not a lot is needed to introduce a conflict into the grammar. Look at the simple 3 rules grammar on the picture 2.



```
R0    S:  b;
R1    S:  A;
R2    A:  b;
```

Picture 2.

S is the axiom of this grammar and b is the terminal symbol. This grammar has only one valid statement: “b”. But this statement has 2 possible analyses. One analysis promotes ‘b’ directly to the axiom. Other analysis promotes ‘b’ to the non terminal A first and then promotes A to the axiom. Analysis table of this grammar confirms this. It shows that it contains one grammar conflict. This conflict is easy to understand.

But from the LR(1) standpoint this is **Grammar Conflict** in a bold font with capitalized starting letters in its full beauty and complexity. LR(1) parser does not care if particular conflict is easy to understand or not.

This is the reduce/reduce conflict because the parser is not sure what rule to apply: R0 or R2. Other funny point is that the input symbol for this conflict is EOF. This is a rare case, but this still sometimes happens. It is important to understand that taking any conflicting action is correct from the grammar standpoint. Parsing process comes to a sort of fork. In a typical case selecting the wrong conflicting action results in a syntax error is a few steps, while taking the right conflicting action does not lead to the syntax error. It also may happen that both actions will not come over syntax error, like in the example above. This means that input statement has 2 or sometimes even more valid Abstract Syntax Tree (AST) representations. If parsing process comes over several conflicts, each conflict gives a fork and the whole process will result in a "tree of ATS's". From the grammar standpoint all these trees have equal value. What tree should be taken and what tree should be discarded stays outside of the grammar theory.

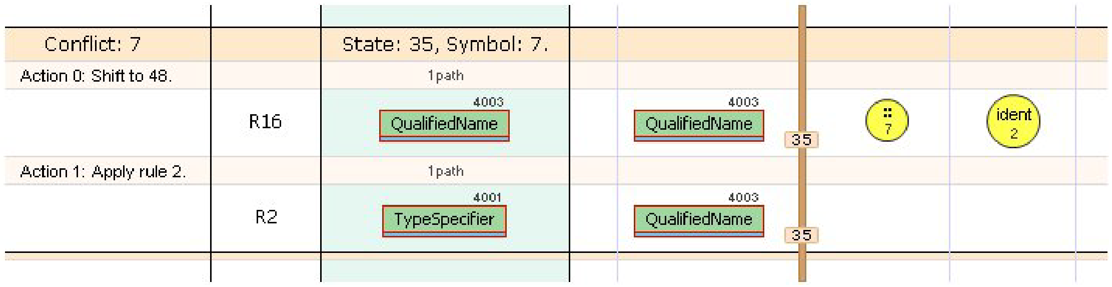
From the practical stand point every conflict always has single resolution because every programming language has conflict resolution guidelines that are typically expressed in the human language. In other words every program always has only one interpretation that corresponds to single AST.

3. DIRECTLY CONFLICTING RULES

The grammar theory defines conflict as a pair of the parsing state and the input symbol. An example may look like «state 26, character '**(**'». State 26 is the parsing state of one of the conflicts in the sample grammar. Character '**(**' is the conflicting symbol of this conflict. This description is not telling a lot when it is necessary to understand what is conflicting with what. Other practical questions are:

- Is it possible to avoid this conflict by rewriting the grammar?
- How to resolve this conflict when the grammar based parser comes over it?

The first step to understand the conflict is to look for rules that are involved. Besides the state and the symbol, information about the conflict contains the list of conflicting actions. Shift-reduce parsers have only 2 types of possible parsing actions. These actions are shift and reduce. When the conflicting action is shift, it is necessary to look for rules that contain the conflicting symbol plus the set of possible states in front of this symbol should contain the conflicting state. When the conflicting action is reduce, then only one rule is involved and the position stays at the end of this rule. The grammar structure viewer can search the grammar for such rules and show them in a table like form, aligning rules to the "conflict position".



Picture 3.

The sample grammar has 9 conflicts. Picture 3 shows the results of simple analysis for one of them. It describes the conflict C7. The parsing state of this conflict is 35. Grammar definition parser has assigned symbol value 7 to the symbol '**::**'. This information is written in brief in the header of the table. When the parser is in state 35 and it shifts in the symbol '**::**', according to the parsing table the parser changes its state to the state 48. Second header of the table describes the first conflicting action, which is shift to the state 48. Parsing state 35 is present before the symbol '**::**' only in one rule. In this case it is rule 16. The conflicting shift is typically found only in one rule, although sometimes it is present in 5-10 rules. The second conflicting action is reduce using the rule 2. Picture 3 shows the rule 2 below its conflicting action header.

Looking at the directly conflicting rules this conflict can be described as: "A parser cannot decide whether it should accept the symbol '**::**', i.e. continue constructing a more complex QualifiedName, or it should apply R2, i.e. promote the fragment of the already scanned input to the TypeSpecifier". Seeing the involved rules is much more than «state 35, character '**::**'», but this is still not enough to answer the practical questions.

The next step can be manual writing source code samples that involve resolution of the grammar conflict. These code samples will be examples of the context where this conflict appears. Validity of the samples can be verified with the help of the grammar based parser. Such samples can be useful. But it is not clear how to write them having information that was described above. And how to be sure that existing set of samples is complete and there are no other significantly different situations? Besides that the set of these samples is infinite. At the same time it seems that this set can be organized into groups of "similar samples". It would be great if infinite set of samples could be split into a finite number of groups where all samples inside the group are similar while the samples from other groups are "significantly different".

What should be similar inside a group of samples? The reasonable answer is the type of context where the conflict appears. This paper addresses the problem of building a complete list of all possible contexts for any conflict.

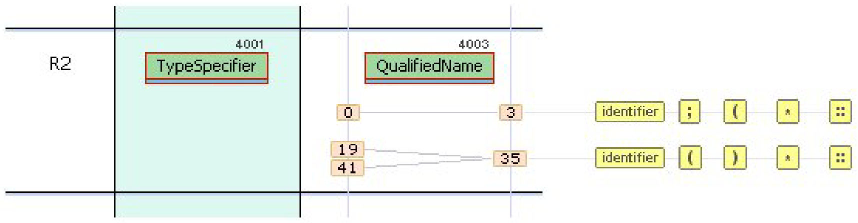
4. DERIVATION PATHS

This paper defines a context of the grammar conflict as a fragment of the AST tree. Classic AST contains an axiom in its root, non terminals in its intermediate nodes and terminal symbols in its leaves. An AST fragment can have any non terminal in its root, and leaves can contain non terminals. Like in the full AST, all non leaf nodes of the AST fragment should conform to the grammar rules. Having an AST fragment it is relatively easy to build the full AST that contain this fragment as its part. This can be done either manually or by using a technique that is similar to the method described below.

This paper uses a concept of derivation path. By definition the derivation path is a sequence of derivation steps. These steps show how this path was discovered. Having these steps, it is possible to reconstruct the AST fragment and/or convert these steps into the sequence of symbols. All possible types of derivation steps and conversion procedures are described later, after it will become clear how derivation paths are related to the grammar conflicts.

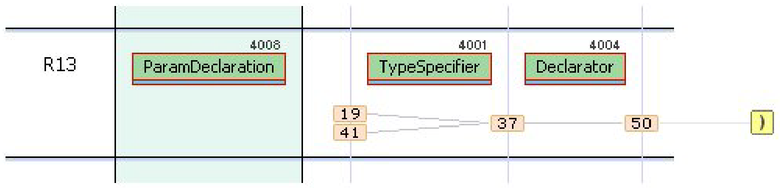
One step back. Applying any grammar rule can be split into 2 phases. During the first phase the grammar based parser discards symbols from its stack and switches back to parsing states that are stored in the stack. Parser stores symbols and states in its stack in pairs. The number of discarded pairs is equal to the number of symbols in the right hand side of the rule that is being applied. If the rule is empty, nothing is discarded from the stack. During the second phase the parser shifts in the newly constructed non terminal and the current state. Then the current state is replaced with another state according the parsing table using the new non terminal as an input symbol. This procedure of executing the rule can be used to see what may happen when the directly conflicting rule is executed.

Let’s consider the conflict C7 from the picture 3 once again. The idea is to follow the procedure of executing the grammar rule and look for situations where the rule 2 can be applied. When the parser is in state 35 and it applies the R2, what state can it retrieve from its stack? Can this be any state? The answer is no. This can be only one of the starting states of R2 that derives to the state 35. Look at the picture 4 that shows all possible states that the rule 2 can have. This picture is built using the list of rule positions that constitute each parsing state. This list of states is available inside the grammar definition parser when it builds the LR(1) parsing table.



Picture 4.

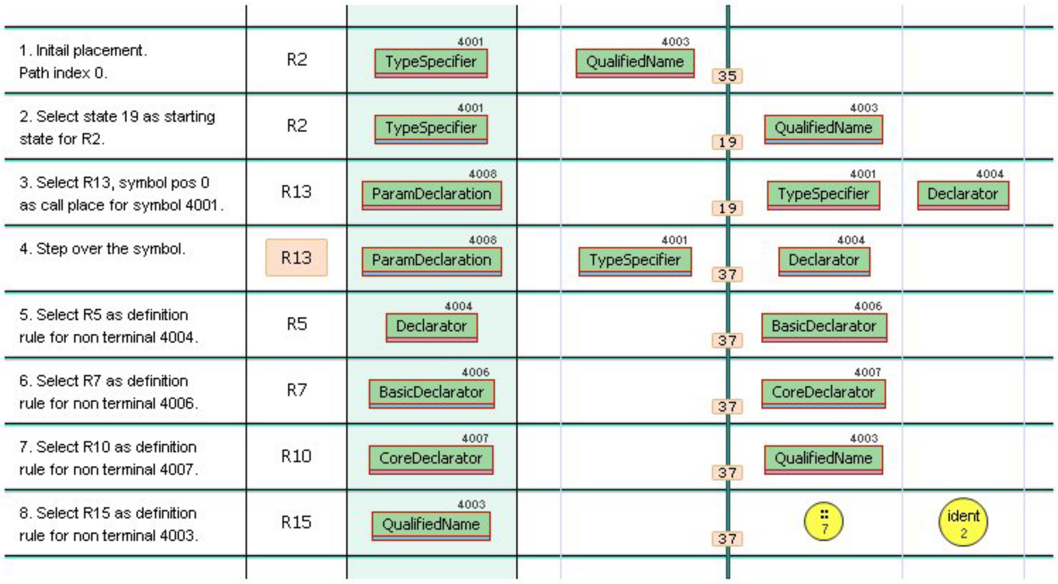
The picture 4 shows that initial states that derive to the final state 35 are 19 and 41. Both states should be considered separately. For now let's consider the state 19. So, at the end of the first phase of applying R2, the parser is in state 19 and it stays in front of the symbol TypeSpecifier. In what rules is this possible? Analysis shows that in the sample grammar this is possible only in the R13.



Picture 5.

The picture 5 shows all possible parsing states for R13. In general there can be several different possible rules and all these rules should be considered separately. In the current case there is only one rule. When the second phase of the rule execution is carried out, the parser shifts to the state 37. If the input string that stays after the current place is syntactically correct, this string should derive to Declarator. Now it is necessary to find rules that start from '**::**' and derive to Declarator. This can be done by iterating all rules that define non terminal Declarator. Those rules may not have '**::**' in the beginning, but they may start from other non terminals, and those non terminals may have symbol '**::**' in the beginning of their rules. Doing this procedure recursively, such rules may be found.

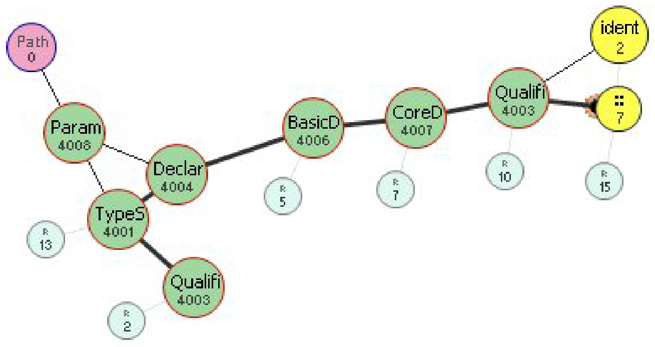
The sample grammar contains such sequence of rules and it is necessary to make 4 steps of recursion. The full sequence of steps in the current derivation path is displayed on the picture 6.



1. Initail placement. Path index 0.	R2	4001 TypeSpecifier	4003 QualifiedName	35		
2. Select state 19 as starting state for R2.	R2	4001 TypeSpecifier		19	4003 QualifiedName	
3. Select R13, symbol pos 0 as call place for symbol 4001.	R13	4008 ParamDeclaration		19	4001 TypeSpecifier	4004 Declarator
4. Step over the symbol.	R13	4008 ParamDeclaration	4001 TypeSpecifier	37	4004 Declarator	
5. Select R5 as definition rule for non terminal 4004.	R5	4004 Declarator		37	4006 BasicDeclarator	
6. Select R7 as definition rule for non terminal 4006.	R7	4006 BasicDeclarator		37	4007 CoreDeclarator	
7. Select R10 as definition rule for non terminal 4007.	R10	4007 CoreDeclarator		37	4003 QualifiedName	
8. Select R15 as definition rule for non terminal 4003.	R15	4003 QualifiedName		37	:: 7	ident 2

Picture 6.

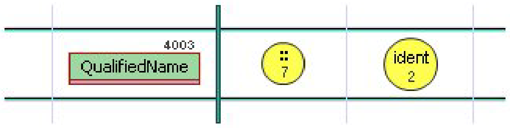
The picture 6 describes one of the valid derivation paths where the symbol '**::**' (on the last line) stays after the symbol QualifiedName (on the first line). This derivation path corresponds to one of the possible conflict contexts. This sequence of steps can be also presented as an Abstract Syntax Tree (AST) fragment. To create the AST fragment it is necessary to do literally what the steps of the derivation path are saying. These steps are very explicit. This makes is possible to check that the discovered path really conforms to the grammar.



Picture 7.

The picture 7 shows the AST fragment. The root circle tells that this three displays a derivation path and tells the index of this path. Nodes of the tree are terminals and non terminals. Small circles show indexes of the rules that were used. Thick line connects the symbol that stays at the end of R2 (directly conflicting rule) and input symbol of the conflict. This picture shows that QualifiedName (4003) from the rule 2 was promoted to TypeSpecifier (4001). R13 was selected as one of the possible contexts for the TypeSpecifier. On the descending phase of the derivation path Declarator (4004) was replaced with BasicDeclarator (4006) using R5, then with CoreDeclarator (4007) using R7, then with QualifiedName (4003) using R10. Finally QualifiedName was defined as '**::**' and an identifier. The picture 7 does not show why this or that decision was made and what parsing states were taking place. This information can be picked up from the more detailed description of the derivation path on the picture 6.

This derivation path can be also presented as a sequence of symbols. This sequence is the list of leaves of the AST fragment when they are iterated from left to right:



Picture 8.

The sequence of symbols on the picture 8 is plain simple. In other cases it can be more complex. It is important to note that this is not a complete sample that can be fed into the grammar based parser. This is only a fragment. When the parser is in a certain state and this fragment stays in the input stream, the required parsing conflict will occur. Other important point is that this fragment contains a non terminal. This means that instead of the QualifiedName any sequence that derives to QualifiedName can be used. This derivation path represents infinite set of sample fragments.

This is informal description of the derivation path and the process of finding such paths. There are several points worth mentioning here. First, simply looking at the picture 3, it is not easy to draw the picture 7. Second, all iterations like enumerating possible parsing states, looking for places in the rules, etc can be automated. And each type of iteration includes a finite number of steps. Third, there are only two ways of extending the AST fragment. It can be extended either up by looking for places where a non terminal is used, or down by looking for rules that define a non terminal.

This means that if there is a certain condition that limits extension of the AST fragment, all possible AST fragments that conform to the grammar and this condition can be iterated. This iteration will result in a set of derivation paths. Having full list of all possible derivation paths for any conflict in the grammar is a good step in the direction of understanding the conflict and giving guidelines on resolving it. It is true, that this iteration may take substantial time. It is not clear for now what should be the condition that should limit extension of the AST fragment. These questions will be addressed in the next section.

5. GRAMMAR CONFLICT ANALYSIS ALGORITHM

The previous section described the process of finding derivation paths informally. Now it is time to describe this process in more details. A general idea is to iterate the tree of possibilities. A classic approach to iterating trees is organizing stack of iterators. Each slot of this stack will iterate possibilities that are present on the layer. The overall state of the stack represents the path in the tree that is currently considered. This paper is using this classic approach.

An algorithm that finds derivation paths has 3 different iterators and several non iterateable slot types. Non iterateable slots can be considered as iterators with only one possible alternative. Here is the list:

- Rule start states iterator. This iterator iterates parsing states that can be present at the beginning of the given rule and derive to the given parsing state in the given position in this rule.
- Rule call places iterator. This iterator iterates rule positions where a given parsing state stays in front of a given symbol.
- Non terminal definition rules iterator. This iterator iterates rules that define a given non terminal.
- Initial placement slot. This slot is used at the start of processing the reduce action.
- Step over the symbol slot. This slot shifts the current position in the rule to the right. The purpose of this slot is mimicking the second phase of the rule execution.
- Step up the tree. This slot is added when the non terminal definition rules iterator processes a rule with an empty right hand side.
- An axiom action symbol. This slot cannot be present in the beginning or in the middle of the path. This slot is used when the conflicting symbol is EOF. This is a rare case. This slot is more often used in the nested grammars because these grammars allow other symbols to follow the axiom, not only the EOF. Nested grammars are discussed several sections below.

The top level code of the algorithm looks like:

```
DoInitialPlacement();
ProcessCurrentPosition();

while (len_stack > 0)
{
    If (GetNetxPositionOnTheCurrentLayer() == TRUE)
       ProcessCurrentPosition();
    else
       len_stack--;
}
```

In the beginning, before entering the main loop, DoInitialPlacement() function fills the stack with initial iterators according to the following rules:

The conflicting action is shift:

1. Add the rule call places iterator. The state and the symbol are taken from the conflict. Note that in this case the iterator will look for places where the conflicting state stands in front of the conflicting symbol. Since the conflicting symbol is a terminal symbol, this is not really a "rule call place". But the same code works fine, so there were no reasons for introducing a new type of iterator.

The conflicting action is reduce and the conflicting rule is not empty:

1. Add the initial placement slot.
2. Add the rule start states iterator. The rule in the iterator is rule from the conflict, the position is at the end of the rule and the state is taken from the conflict.

The conflicting action is reduce and the right hand side of the rule is empty:

1. Add the initial placement slot;
2. Add the rule call places iterator. The symbol is a non terminal of the rule from the conflict. The state is taken from the conflict.

Initial processing of reduce action has one exception when non terminal of the conflicting rule is an axiom. In this case the AST fragment cannot grow up because this parser does not allow using the axiom in the right hand side of the rules. This is not a limitation because any grammar can be converted into this form. Such conversion is discussed in [Aho et al. 1986]. Once the conflict is at the end of the rule that defines the axiom, this means that conflicting symbol should be one of the action symbols of this rule. Code verifies this, adds axiom action symbol slot and stores the derivation path. The iteration is not started in this case.

The function GetNetxPositionOnTheCurrentLayer() shifts iterator that sits on the top of the stack. If the iterator has more alternatives, a new position is stored in the slot and the function returns TRUE. If there are no more positions or the slot is not a real iterator, the function returns FALSE.

The function ProcessCurrentPosition() significantly differs for the action shift and the action reduce. It is easier to describe this function as two separate functions. For the action shift this function is simple. Every position that the rule call places iterator finds is already a derivation path. This path should be simply stored. Paths for conflicting action shift always consist of one step.

The code of ProcessCurrentPosition() for conflicting action reduce looks like:

```
if (LookForIdenticalSlot() == FALSE)
{
    switch (slot_type)
    {
    case RuleStartState:
        LoadIterator(RuleCallPlace);
        break;

    case RuleCallPlace:
        AddSlot(StepOverSymbolSlot);
        ProcessDerivationStep();
        break;

    case NonTermDefinitionRule:
        ProcessDerivationStep();
        break;
    }
}
```

The function LookForIdenticalSlot() checks the topmost slot of the stack of iterators. If the slot type and the current rule position are already present in the stack, the position is ignored. This is very important. Without this check the algorithm will never end and it will infinitely enlarge the length of the stack. This happens because grammars have recursive nature and this allows describing infinite number of input statements. The purpose of this algorithm is finding only non recursive sequences. For example, if a sequence like

```
int *x;
```

is the target sequence, then adding any number of similar sequences like

```
int **x;
int ***x;
int ****x;
...
```

will not add any value. This means that while looking at the discovered conflict context it is necessary to understand where the recursion was omitted. Any single discovered sequence should be understood as infinite set of similar sequences and omitted recursion is a one more way of extending derivation path into the input sequence besides extending the non terminals that stay in the leaves of the AST fragment. In practice this is not difficult. It is necessary to look at rules that are involved into the derivation path. The real code is using a bit more complex criteria for skipping the current position, but this complexity does not contradict to the statement above.

The procedure of adding new slots to the stack is based on the properties of the top slot of the stack only. This procedure is not using other slots inside the stack. Once this is so, this means that if the process of building the derivation path is entering the recursion loop, then it will infinitely repeat the same loop in the stack. This recursion will bring new paths that are similar to paths that are already discovered. And this recursion will be infinite. The coding was started without understanding this point. This became clear after looking at the contents of the infinitely growing stack. Once again, this consideration is very important and it allows making the process of finding all possible different conflict contexts finite.

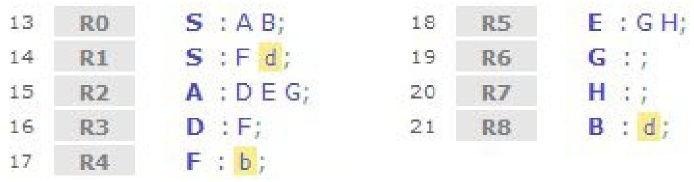
The code inside the switch processes the current slot according to the type of the slot iterator. When the rule start states iterator discovers some parsing state, algorithm starts looking where this non terminal is called with this state. If the rule call place iterator finds a call of the non terminal, this non terminal is stepped over and the position after the non terminal is processed. When the definition rules iterator finds a rule, the position in the beginning of this rule is processed.

The function `ProcessDerivationStep()` checks position that the current iterator has found or non iterateable slot contains. The following situations are possible:

- The position stays at the end of the rule;
- The position stays in front of the terminal symbol that is the same as the input symbol of the conflict;
- The position stays in front of the terminal symbol that differs from the conflicting symbol.
- The position stays in front of the non terminal.

Position can stay at the end of the rule for 2 different reasons: either the step over the symbol slot has stepped over the last symbol of the rule, or the non terminal definition iterator has found an empty rule. In the last case the position is starting and final at the same time.

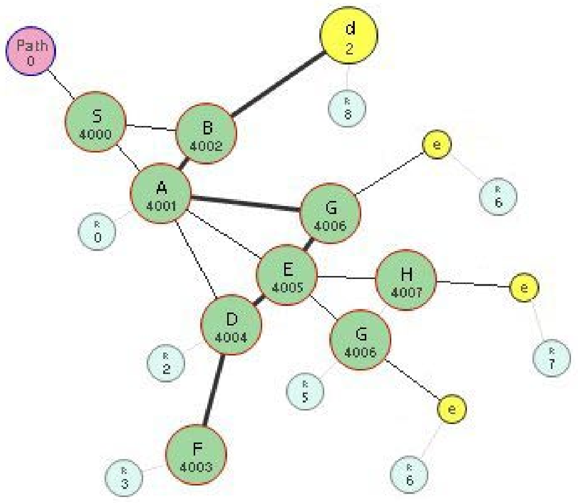
If the step over symbol came to the end of the rule the situation is simple. Code starts iterating starting states of the current rule. In other words it continues building the AST fragment up. The second case with an empty rule is more complex. The code adds the step up the tree slot that moves the current position into the rule that started the non terminal definitions iteration. Inside the rule the position is placed behind the non terminal. After that the same procedure is applied once again. Picture 9 shows an example of what happens in this case.



```
R0    S : A B;        R5    E : G H;
R1    S : F d;        R6    G : ;
R2    A : D E G;      R7    H : ;
R3    D : F;          R8    B : d;
R4    F : b;
```

Picture 9.

The grammar from the picture 9 is similar to the 3 rules grammar presented on the picture 2. Only it has several "useless" non terminal conversions and several non terminals that can be only empty. S is the axiom of the grammar. Small letters 'd' and 'b' are terminal symbols. Nevertheless the conflict analysis algorithm easily finds derivation paths for both conflicting actions. AST fragment for the second conflicting action is shown on the picture 10.



Picture 10.

The letter '**e**' in the small circle represents empty symbol of the empty rule. The picture 10 shows that the algorithm understood that it is necessary to promote non terminal F to non terminal D. Then it understood that non terminal E has empty indirect expansion and proceeded to the non terminal G. This non terminal has empty expansion either. Position came to the end of the rule. The algorithm started to look for places where the non terminal of R2 is called. Finally it turned out that non terminal B has an expansion that starts from the conflicting symbol 'd'. This path is rather complex. Till now the author has never seen such paths in practical grammars. This sample was constructed on purpose. This test run also shows that the analysis engine can handle complex cases. In reality derivation paths look more like the path presented on the picture 7.

Now let's return back to the classification of possible positions. When the position stays in front of the conflicting symbol, this is exactly what the whole process was looking for. The current state of the stack should be saved as a derivation path. When the position stays in front of the terminal that is not a conflicting symbol, this means that algorithm came to some random place that has nothing to do with the current conflict. This position should be ignored and the iteration continued.

When the position stays in front of the non terminal, then it is necessary to start iteration of rules for this non terminal. For the performance reasons it makes sense to check if this non terminal can start from the conflicting symbol or not. This is easy because of the function FIRSTs that was used to build the LR(1) analysis table [Aho et al. 1986]. If the non terminal cannot start from the conflicting symbol, it does not make sense to iterate its rules. Other performance motivated check is whether current non terminal has an empty representation or not. This can be done using the same FIRSTs function. These checks reduce the computation cost dramatically but from the theoretical stand point they do not bring anything new.

To make the description of the analysis algorithm simpler this paper omits description of iteration of the rule action symbols and passing current rule action symbols from one layer of iteration to another. This aspect of the study is not changing the actions that were presented above.

Once a sequence of steps is found, it is stored. At this point one more check is done. The code looks if it already has a derivation path with the same rule positions in its steps. Paths with identical rule positions can still differ in parsing states and rule action symbols. Paths with such difference result in the same AST fragments and the same sequences of symbols. This is why it makes sense to store only those paths that differ in rule positions. Nevertheless the storage contains parsing states as they were when the path was discovered for the first time and the count of subsequent rediscoveries.

There are 2 principal related questions:

- How many steps are it required to iterate this potentially huge set of possible derivation paths?
- How many different derivation paths this algorithm will discover?

Experiments show that for a relatively small grammar that consist of 20-50 rules the iteration time is in the range of minutes or hours. The computation was done on a regular modern days PC. The processing time is sensitive to even small changes in the grammar. Removing a non terminal may increase the computation time instead of decreasing. Grammars of approx 150 rules may require several days of processing or even more. It is worth mentioning that the existing code can be optimized, several CPUs/boxes can be used at the same time, etc. These possibilities were not seriously explored.

The positive fact is that once the code discovers a derivation path, it immediately prints it. There is no need to wait for the end of the whole process to see if anything was discovered or not. For now the author has never seen situations when the code was not able to find at least one derivation path during the first several hours of analysis. This means that if at least some of the conflict contexts are discovered, they can be viewed immediately. From the practical stand point this is better than nothing. Other possible shortcut is limiting the max length of the iteration stack. In this case the algorithm will find all contexts that are not longer than the certain limit. Changes in the grammar like removing rules that have no relation to the conflict typically reduce the computation time while the set of derivation paths tends to remain the same.

The number different of paths that the algorithm was discovering was varying from 1 to 60. This is good news because when the number of paths is not huge it is possible to check all cases manually and understand all situations where the conflict can happen. This applies only to cases when the iteration has managed to end. When the iteration is too big, only some of the conflict contexts can be checked.

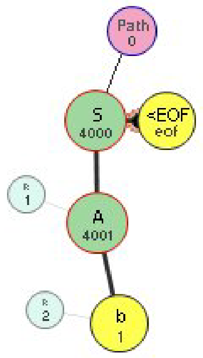
6. HIDDEN CONFLICTING RULES

A derivation path can be split into an ascending phase and a descending phase. All derivation paths have this turning point. This point can be defined as a point on the last "step over the symbol" step or the last "step up the tree" step whichever occurs to be the last. Equivalent definition for the turning point is the step before the first step of the last sequence of "non terminal definitions" if such sequence is present, otherwise it is the last step of the derivation path.

Derivation paths for a shift action consist of only one step that point to the directly conflicting rule. For this type of paths the turning point is in this step. Paths that end with an "axiom action symbol" step have their turning point at their end. They do not have a descending phase. This is an exception because in this case the turning point is not inside the rule. This case can be considered as a pseudo rule:

```
S  <eof>
```

A grammar based parser requires that each complete statement should be followed by the EOF symbol. Reaching the EOF symbol with the axiom in the stack of the parser is the condition when a grammar based parser terminates the parsing with success.

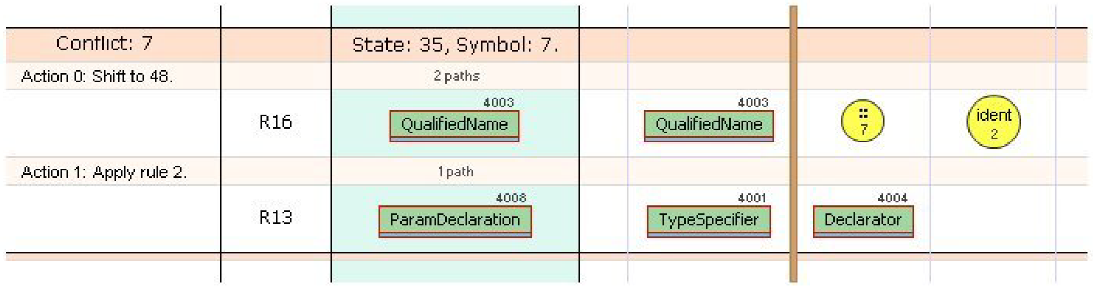


Picture 12.

The picture 12 shows an AST fragment for the second conflicting action of the conflict in the 3 rules grammar shown on the picture 2. Note that the upper non terminal is not accompanied with the small label with the rule number.

The turning point of the derivation path points to the hidden conflicting rule (or pseudo rule).

Let’s return back to the conflict C7 that is shown on the picture 3. Both conflicting actions of this conflict have only one derivation path each. The first conflicting action is shift. The hidden conflicting rule of its path is R16. It is the same to the directly involved rule. The derivation path for the second conflicting action is described on the picture 6. Its turning point is in the step 4 that points to R13. This can be illustrated by displaying the hidden conflicting rules in a table.



Conflict: 7		State: 35, Symbol: 7.				
Action 0: Shift to 48.		2 paths				
	R16	4003 QualifiedName	4003 QualifiedName		:: 7	ident 2
Action 1: Apply rule 2.		1 path				
	R13	4008 ParamDeclaration	4001 TypeSpecifier		4004 Declarator	

Picture 13.

The picture 13 shows hidden conflicting rules for C7. Compare this picture with the picture 3. Look at R13 on this picture and think that instead of the TypeSpecifier there is a QualifiedName and instead of the Declarator there is a ‘**::**’ and an identifier. This is possible if appropriate non terminal conversions will be applied. And rules of the grammar allow such substitutions.

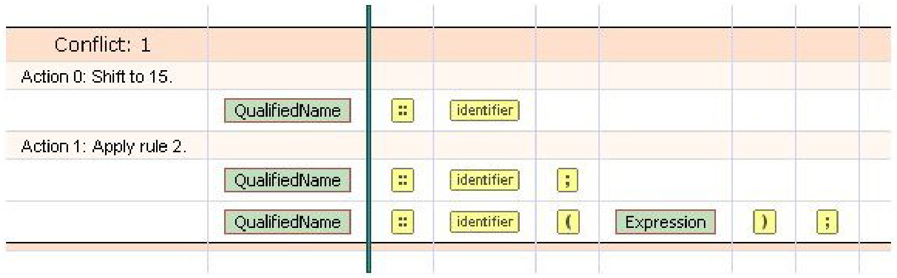
The most important conclusion of this paper is that hidden conflicting rules reveal the shift/shift nature of the grammar conflict. For example, in the conflict C7, when the LR(1) algorithm is stating that it has doubt applying R2, in reality this means that there is R13 with a suspicious context. The grammar based parser cannot go on because it cannot make decision between the context of R16 and the context of R13. Both contexts are in the middle of the rules. This is a shift/shift conflict! Or, being more precise, this is a shift/shift nature of the well known shift/reduce conflict. Unfortunately LR(1) algorithm is not explaining its aspect clearly. I hope that my paper brings more light.

In the case of reduce/reduce conflicts both hidden conflicting rules will differ from the directly involved rules. This makes the result of conflict analysis even more surprising. But again this reveals the real nature of the conflict. This reveals pairs of rules, where the real ambiguity sits.

It is worth mentioning that in a typical case conflict analysis results in several derivation paths. Different derivation paths tend to have their turning points in the same rule. This is typical situation. Sometimes there are cases when there are 2-3 different hidden rules for a single conflicting action. A general expectation is that for 50 derivation paths there should be 1-2 hidden conflicting rules.

7. EXTENDED CONFLICT CONTEXT

It is well known that it is necessary to look several symbols forward to resolve the conflict. How many symbols are actually needed for the resolution depends on the situation. There is no general rule here. As it is shown on the picture 8, symbols after the conflict point are ‘**::**’ and identifier. There is no surprise in this case. The symbol ‘**::**’ should be followed by an identifier in any way. In other cases the situation can be different. For example the conflict C1 of the sample grammar has 2 derivation paths for its second conflicting action.

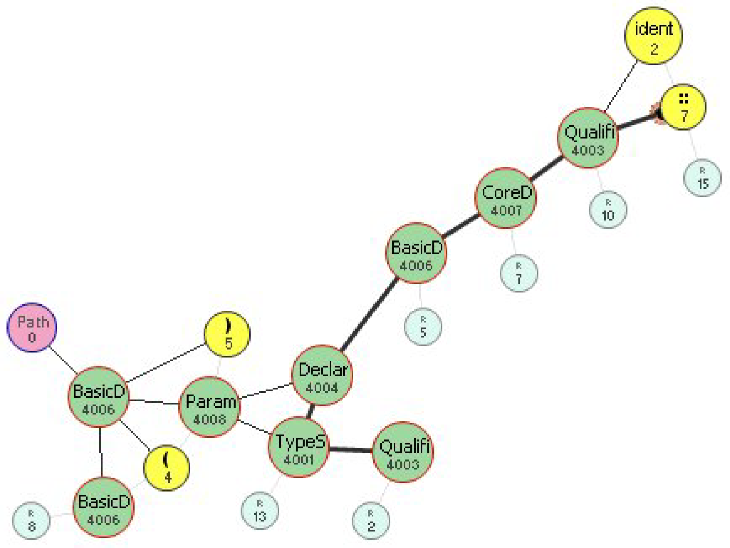


Conflict: 1							
Action 0: Shift to 15.							
	QualifiedName	::	identifier				
Action 1: Apply rule 2.							
	QualifiedName	::	identifier	;			
	QualifiedName	::	identifier	(	Expression	)	;

Picture 14.

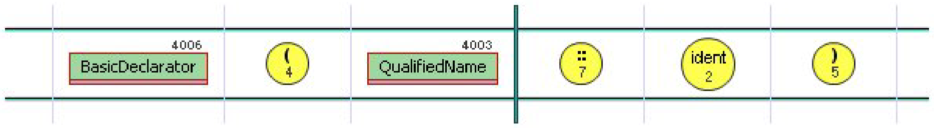
The picture 14 shows what symbols can be present after the identifier in the context of C1. Since the full analysis for this grammar is available, it is possible to say that other symbols cannot be present if the input string is syntactically correct and the R2 should be applied. This is important information for writing the conflict resolution code. Knowing what can be present after the conflicting symbol and what cannot is valuable information. Proposed analysis allows finding such symbols. Unfortunately some derivation paths contain symbols after the conflicting symbol like the paths on the picture 14, some paths not.

To solve this problem, it is possible to modify the analysis algorithm, described above and force it to look for more possible contexts. Modification is the following: when the derivation path is found, it is necessary to check, if the current path contains something after the conflicting symbol or not. If there is nothing there, it is necessary to continue extending the AST fragment up in pretty much the same way as it was done during the ascending phase of the path building. After adding each layer on top of existing AST fragment it is necessary to check if the current sequence of symbols became longer or not. By carrying out this procedure so it is possible to find all contests that are 1 symbol long, 2 symbols long, etc. Note that this procedure adds symbols both to the beginning and to the end of the sequence. Knowing what can precede the conflicting symbol might be interesting also. The conflicts analysis engine contains such code. As it can be expected, this procedure takes more time to complete. But on the other hand it gives back more derivation paths and better understanding. Example of an extended derivation path is displayed on the picture 15.



Picture 15.

This path is more complex than the path from the picture 7. It shows that non terminal ParamDeclaration (4008) can be used in R8. Since the analysis resulted in only one extended path, it is possible to say that this conflicting action does not have any other extended context. For the sample grammar of this article this is obvious, but for a more complex grammar not. Using the described method confirms this.

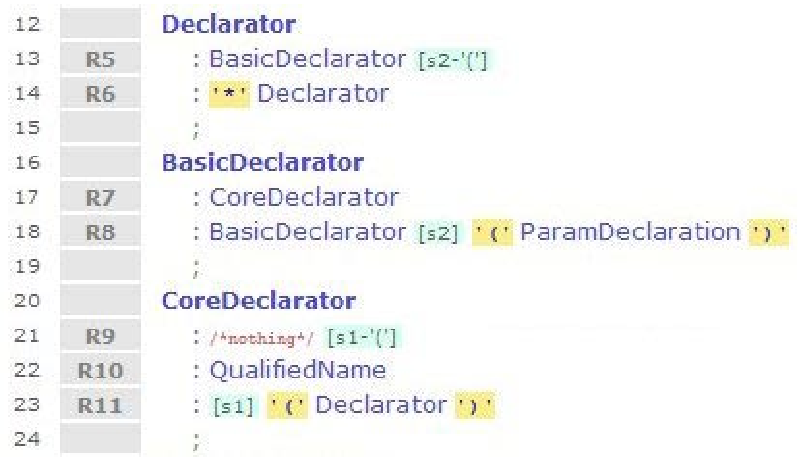


Picture 16.

The picture 16 shows derivation path from the picture 15 as a sequence of symbols. Thick vertical line divides symbols that stay before the conflict position from symbols that stay after the conflict.

8. IDENTIFYING POSSIBLE CONFLICTS IN THE SOURCE GRAMMAR

One of the problems of writing the conflict resolution code is that when a conflict happens, the grammar based parser tells only the parsing state and the input symbol. The parsing state number tells almost nothing to the callback handler. While the callback handler needs to understand what actually happened. Parsing states are assigned as 0, 1, 2, 3…. Even minor change in the grammar like a change in the order of the rules may completely change the numbering of the parsing states. It is not a good idea to hard code a state numbers or a conflict number in the callback handler. This paper proposes using the "expected conflicts" for this purpose.



```
     Declarator
R5     : BasicDeclarator [s2-'(']
R6     : '*' Declarator
       ;
     BasicDeclarator
R7     : CoreDeclarator
R8     : BasicDeclarator [s2] '(' ParamDeclaration ')'
       ;
     CoreDeclarator
R9     : /*nothing*/ [s1-'(']
R10    : QualifiedName
R11    : [s1] '(' Declarator ')'
       ;
```

Picture 17.

The picture 17 shows grammar rules that are annotated with conflict location markers. Location markers are enclosed in the square brackets. Location markers have names. In the example above location markers are named s1 and s2. If a location marker stays in the middle of the rule, it should contain only the name of the marker. If a marker stays at the end of the rule, it should contain an action symbol after the dash.

Location markers should satisfy the following conditions:

- Each marker name should have at least one marker at the end of the rule.
- All action symbols that are associated with the marker name should be the same.
- Intermediate markers should stay only in front of the action symbol that is associated with the marker name.
- Action symbol in the marker should be one of the action symbols of the rule.

If these conditions are not met, the grammar definition parser rejects the location marker object because it is bogus.

The meaning of the location marker object is the set of parsing states. This set is an intersection of possible states that can be present in each rule position where the location markers are placed. This set can be empty. In this case the location marker is bogus. The purpose of this definition might be not obvious. Let's look at the parsing states that are assigned to location markers in the sample grammar.

Maker	State	Maker	State
s1	1	s2	8
	9		
	12		
	23		
	26		
	37		

Picture 18.

The picture 18 shows lists of parsing states that are associated with location markers. The picture 19 describes conflicts of the sample grammar.

Conflict	State	Input		Conflict	State	Input
C0	1	**'('**				
C3	9	**'('**		C1	3	**'::'**
C4	12	**'('**		C7	35	**'::'**
C5	23	**'('**				
C6	26	**'('**		C2	8	**'('**
C8	37	**'('**				

Picture 19.

The similarity between these lists is striking. It is just necessary to add one more location marker to the grammar to describe conflicts C1 and C7. This marker has name q2 in the sample grammar. Location markers split the set of grammar conflicts into subsets. There are several good questions here. Will this work for any grammar? May it happen that sets of parsing stated from different location markers will overlap? Can it be that some parsing state from the location marker set will not correspond to any conflict?

The grammar definition parser checks that all states from location marker sets have corresponding conflicts. It also checks that these sets do not overlap, etc. This guarantees that the situation is fine for any grammar that has passed the grammar definition parser. A good aspect of using the location markers is that changes in the grammar affect the total number of parsing states and assignment of individual parsing state numbers. At the same time different parsing state values continue to land into the same subsets. Work with this mechanism has never showed problems.

After processing of all location markers it may happen that some conflicts are not associated with any marker. This means that more markers should be added to the grammar. These conflicts are called “unexpected”. The grammar definition parser prints them.

9. EXPECTED CONFLICTS AND CONFLICTS RESOLUTION

Location markers can be used in the conflict resolution handlers to understand what conflict happened. Several location markers can be grouped into “expected conflict”. For defining expected conflicts this paper uses the following syntax:

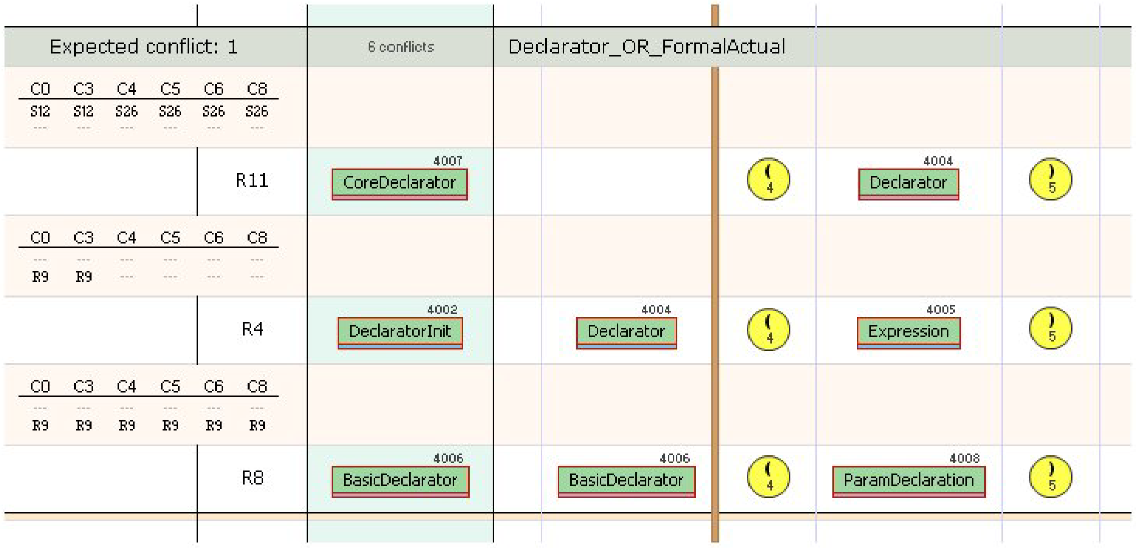
```
Qualified_OR_MoreQualified
{
   <locations>   q2;
}

Declarator_OR_FormalActual
{
   <locations>   s1;
}

Formal_OR_Actual
{
   <locations>   s2;
}
```

This notation defines 3 expected conflicts and gives names to them. The name of the expected conflict can be any identifier. Definition of the expected conflict contains locations where this conflict can happen.

Once the conflicts analysis engine has these definitions, it can do additional analysis. This procedure is carried out separately for each expected conflict and consists of the following steps:

1. Collect derivation paths for all involved conflicting actions.
2. Create a list of hidden conflicting rules from these derivation paths.
3. For each hidden conflicting rule create a list of conflicting actions where this rule shows up.
4. Show results of this analysis in a table like form.

The idea of this procedure is displaying possible conflict contexts for several grammar conflicts at the same time. Not as it was done before in picture 13 just for one conflict. The picture 20 shows the result of this analysis for expected conflict Declarator_OR_FormalActual (X1).



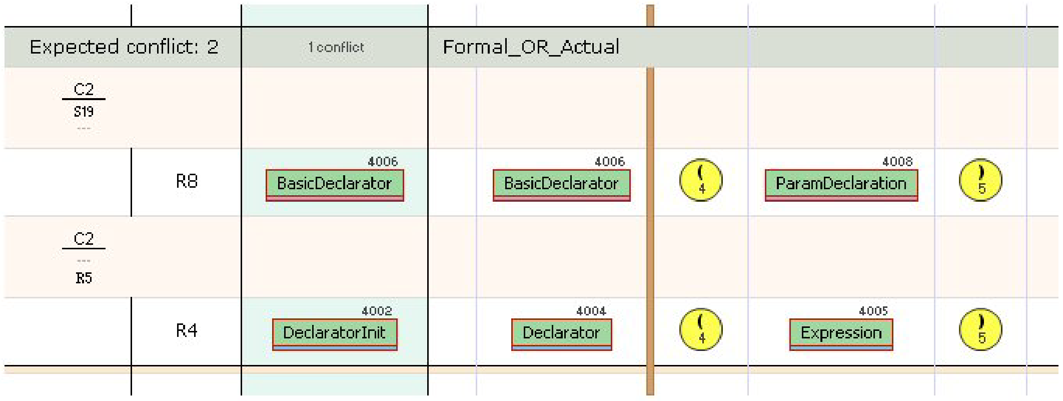
Expected conflict: 1							6 conflicts	Declarator_OR_FormalActual			
C0	C3	C4	C5	C6	C8						
S12	S12	S26	S26	S26	S26						
---	---	---	---	---	---						
						R11	4007 CoreDeclarator		(4	4004 Declarator	) 5
C0	C3	C4	C5	C6	C8						
---	---	---	---	---	---						
R9	R9	---	---	---	---						
						R4	4002 DeclaratorInit	4004 Declarator	(4	4005 Expression	) 5
C0	C3	C4	C5	C6	C8						
---	---	---	---	---	---						
R9	R9	R9	R9	R9	R9						
						R8	4006 BasicDeclarator	4006 BasicDeclarator	(4	4008 ParamDeclaration	) 5

Picture 20.

The upper header on the picture 20 tells what expected conflict is described. The hidden conflicting rules are preceded with headers that describe conflicting actions where these rules surface. Each header contains a matrix. The first row of the matrix contains list all grammar conflicts that belong to the current expected conflict. The columns under the conflict name contain either the short name of conflicting action (S12 means shift to state 12, R9 means apply rule 9) or triple dots. When a name of the action is present, this means that the hidden conflicting rule is present in at least one derivation path of this conflicting action. Dots mean that this hidden conflicting rule is not applicable.

From the application level stand point this picture brings good news. First it turns out that each hidden conflicting rule does not surface in different conflicting actions. This means that if the input looks like the symbols in the hidden rule, an associated conflicting action should be taken. Second, hidden rules show all possible symbols that can be present after the conflicting symbol in the possible contexts of all participating grammar conflicts. This means that identical resolution procedure can be used for the group of conflicts, not just for one. In other words, if the input symbols looks like Declarator, the action S12 should be taken. If the input symbols look like either Expression or ParamDeclaration, then the R9 should be applied. If input does not look like any of these alternatives, it is possible to say that this is a syntax error because conflict analysis has enumerated all possible contexts. The grammar does not allow other contexts. This is a direct hint on how to resolve this conflict! And note that this example is not something artificial. Sample grammar is part of the C++ language and this problem is present in C++.

Analysis for expected conflict Formal_OR_Actual (X2) is displayed in the picture 21.



Picture 21.

The expected conflict X2 consists of only one grammar conflict. In this case it is necessary to decide whether input is Expression or ParamDeclaration. The picture 21 shows what action should be taken in each case.

In general it may happen that hidden conflicting rules will not provide enough info. In this case similar pictures should be used, only instead of hidden conflicting rules they should contain symbol sequences created out of the extended derivation paths. Such pictures are bigger and sometimes more difficult to understand. Nevertheless they always give a good hint on how to resolve the conflict.

It is the responsibility of the programmer to make the final decision on how to resolve a particular conflict. The proposed method is not giving the solution automatically. It only helps to understand the conflict and shows what can be present in the context of the conflict and what can't.

The concept of combining several grammar conflicts into a single expected conflict turned out to be powerful. This concept itself is even more important than the analysis of contexts that was described above. For example, the grammar of the C++ language, as it is presented in the standard, has about 5000 grammar conflicts. After rearranging the rules for some of the non terminals it became possible to reduce the number of grammar conflicts to about 900 without changing the input language. About a thousand conflicts are still too much even to look threw. But after adding expected conflict locations and grouping them into expected conflicts, it turned out that the C++ grammar has only 30 expected conflicts. This is very reasonable number. Yes, C++ 2003 has only 30 significantly different types of grammar conflicts. It might make sense to list them explicitly in the standard. Note that this is a complete list. There are no other types of grammar ambiguities in the language.

10. DERIVED NESTED GRAMMARS

The example on the picture 21 shows that it is necessary to distinguish ParamDeclaration from Expression. What is the logical way of doing this? The answer is simple: use grammar. Nested parsing should be started right after the conflicting symbol. An appropriate conflicting action should be taken depending on its result. The number of symbols that will be involved in this nested parsing will depend on the situation. This is exactly what is needed. It is known from the literature that there is no finite number of look ahead symbols that is enough to solve the conflict in any context. Look at the example:

```
int       ((((… a …))));
```

The language allows any number of parentheses. Nested parsing will store these parentheses in the stack of the parser, figure out what is going on behind them and return this result back to the main grammar.

Symbols that were scanned during the nested parsing should be stored somewhere. When the nested grammar will give its answer, these symbols should be reprocessed again by the main grammar.

The nested grammar should differ from the main grammar because its purpose is slightly different. It should not recognize the whole statement of the main grammar. It should only give an answer in the form: “case1”, “case2”, “none of the above”. Proposed syntax of the expected conflict definition that contains the nested grammar is shown on the picture 22.

```
Formal_OR_Actual
{
    <locations>   s2;

    <resolution>
        Shift:                    ParamDeclaration;
        rule_BasicToDecl:         Expression;

    <conflicts>
        Qual_OR_MoreQual_2   { <locations> q2; }
}
```

Picture 22.

The resolution section of this definition contains resolution statements. Each statement contains in its left side the name of the action that should be taken and in its right side the symbol that should trigger this action. This notation resembles the rules of the grammar. Only the conflicting actions are used instead of the rule non terminals. The left side of the resolution rule may contain either one action or a list of several actions. The triggering symbol in the rule can be empty. When the conflicting action is shift, then the reserved keyword `Shift` is used. The situation with the action reduce is a little bit more complex. The expected conflict can involve several directly conflicting rules, like in the reduce/reduce case. This paper proposes assigning names to the grammar rules. Examples of such assigning are given in the Appendix A. This allows stating what rule should be applied. The exact syntax of all these definitions seems to be less important than the concept that stays behind this syntax.

After processing the resolution statements of the expected conflict the grammar definition parser will generate prototype of the nested grammar that will consist of 2 rules:

```
TranslationUnit:          ParamDeclaration;
TranslationUnit:          Expression;
```

TranslationUnit is an axiom of the main grammar. It is used here because some symbol is needed in any case and using axiom of the main grammar does not contradict to anything. After that the grammar definition parser will add rules that define ParamDeclaration and Expression plus rules for all non terminals that are used in these rules and so on recursively. This is why these nested grammars are called “derived”. There is no need to redefine the ParamDeclaration. It has the same meaning in the nested grammar.

The set of terminal symbols of the nested grammar is the subset of terminal symbols of the main grammar. This set consists of only those symbols that are present in the rules of the nested grammar. All other terminal symbols are mapped to EOF when the nested parsing is scanning the input stream.

The root grammar should recognize the full statement that should be followed by the EOF symbol. On the contrary, the nested grammar should recognize the prefix. If the triggering symbol is found, parsing should be stopped and the result should be reported to the upper parsing process. This should be done regardless of whether the triggering symbol is followed by EOF or by any other terminal symbol. This difference is reflected in the way the analysis table of the nested grammar is built.

There is one important point here. Can a nested grammar have conflicts? The answer is yes. This is a bad news but this is part of the reality around. Good news is that exactly the same technique can be used to resolve these conflicts. This creates a hierarchy of nested grammars. A positive point is that when a parser enters the nested parsing, it always steps 1 symbol forward. This means that the parser cannot enter infinite recursion when it starts the nested parsing.

As you can see on the picture 22, the expected grammar conflict contains the definition of the nested expected conflict. This conflict does not have resolution statements. This means that it should be resolved by the callback handler. The callback handler should decide if it is necessary to continue building the QualifiedName or stop. This decision is based on the knowledge about the identifier. The nested grammar is not needed here. In a more complex case several levels of nesting might be needed.

11. GLR / BACKTRACKING

When a classic LR(1)/LALR based parser meets a grammar conflict, it simply does not know what to do. At the same time the number of choices is not big. Typically there are just 2-3 alternatives. The main idea of GLR approach is the following: try all possibilities and see what will come out of this. When the processing of each alternative is started, parser switches into the trial mode. While being in trial mode, it can face a conflict again. This means that it is necessary to deal with the tree of possibilities. Once it becomes clear that the alternative failed, the parser rolls back its actions and starts new alternative. If the alternative succeeds, the parser commits its actions. There are several different flavors of the GLR/Backtracking that primarily differ in the way how parser decides if this or that alternative has succeeded or failed. Literature states that the order of trying alternatives is important for performance reasons.

In contrast with the GLR approach that checks alternatives one by one, the proposed approach processes all alternatives at the same time by using the nested grammar based parsing. The proposed approach has clear guidelines on deciding when it is necessary to stop processing symbols that stay after the conflicting symbol.

12. CONCLUSION

This paper described a formal algorithm that allows finding all possible non recursive derivation paths for a grammar conflict. These derivation paths describe all possible contexts where a grammar conflict can appear and reveal hidden conflicting rules. These hidden rules give a clear explanation of what is conflicting with what. The hidden conflicting rules always conflict in a shift/shift manner. The contexts of these rules are the real source of the grammar conflicts that the LR(1) algorithm discovers.

Annotating grammar with expected conflict locations and grouping these locations into expected conflicts turned out to be a convenient way of describing the conflicts in the grammar. This method is highly immune to the changes in the grammar that do not affect the rules that are involved in the conflict. All grammar conflicts of the C++ 2003 grammar can be described using about 30 expected conflicts.

Analysis of possible conflict contexts shows that in some cases the contexts of different conflicting actions always contain different non terminal symbols. This means that this type of conflicts can be effectively resolved by using the nested grammar based parsing.

REFERENCES

ISO. 2003. International Organization for Standardization: ISO/IEC 14882:2003(E): Programming languages – C++. ISO, Geneva, Switzerland.

Aho, A., Sethi, R., Ullman, J. 1986. Compilers: Principles, Techniques and Tools. Addison-Wesley Publishing Company. Reading, MA.

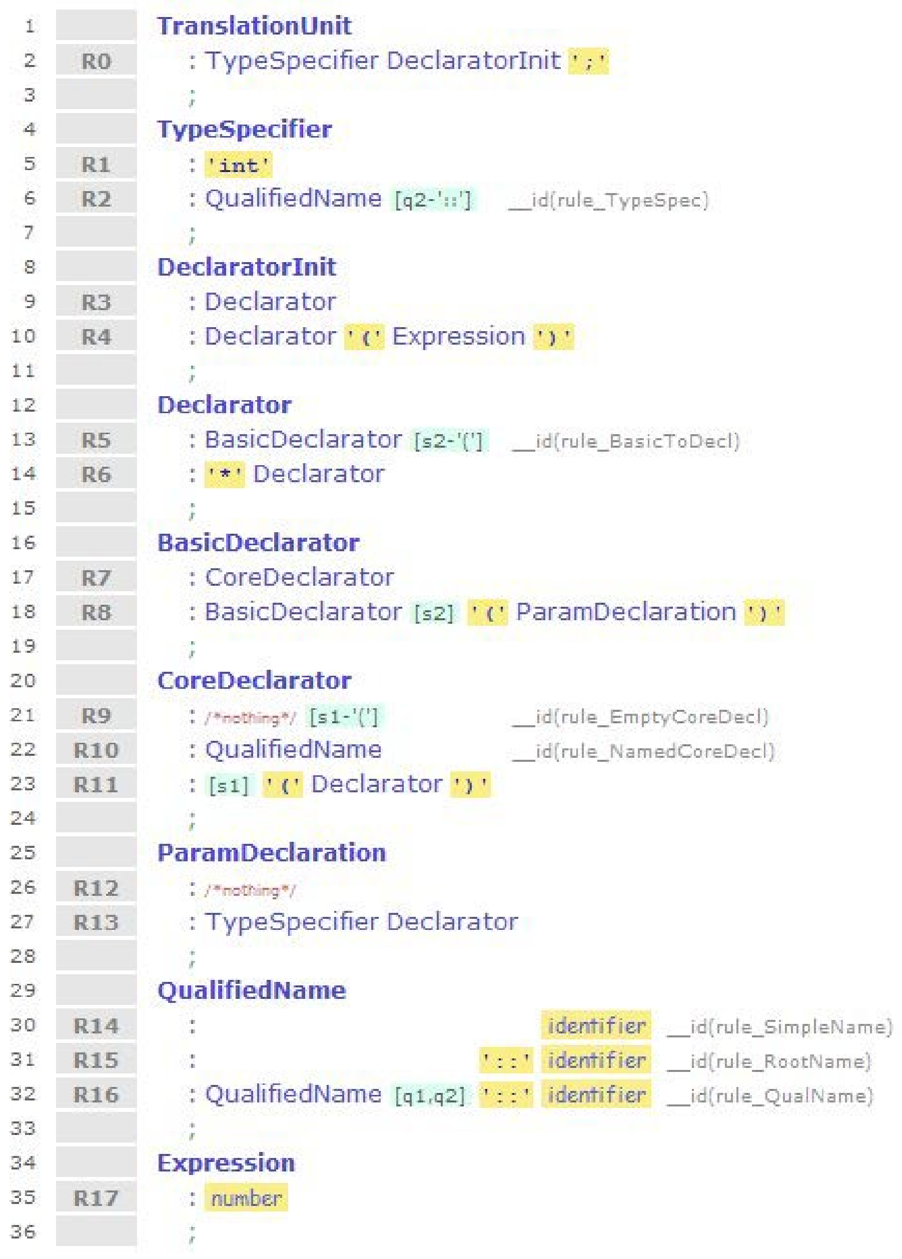
Levine, J., Mason, T., Brown, D. 1990. Lex & Yacc. Unix Programming Tools. O'Reilly & Associates. Sebastopol, CA.

Masaru, T., See-Kiong, N. 1991. The Generalized LR Parsing Algorithm. Kluwer Academic Publishers. Norwell, MA.

Thurston, A., Cordy, J. 2006. A Backtracking LR Algorithm for Parsing Ambiguous Context-Dependent Languages. In 2006 Conference of the Center for Advanced Studies on Collaborative Research (CASCON 2006), 39-53.

McPeak, S., Necula, G. 2004. Elkhound: A fast, practical GLR parser generator. In Compiler Construction: 13th International Conference (CC'04), volume 2985 of Lecture Notes in Computer Science, April 2004, 73-88.

Appendix A. RULES OF THE SAMPLE GRAMMAR



```
     TranslationUnit
R0     : TypeSpecifier DeclaratorInit ';'
       ;
     TypeSpecifier
R1     : 'int'
R2     : QualifiedName [q2-'::']      __id(rule_TypeSpec)
       ;
     DeclaratorInit
R3     : Declarator
R4     : Declarator '(' Expression ')'
       ;
     Declarator
R5     : BasicDeclarator [s2-'(']     __id(rule_BasicToDecl)
R6     : '*' Declarator
       ;
     BasicDeclarator
R7     : CoreDeclarator
R8     : BasicDeclarator [s2] '(' ParamDeclaration ')'
       ;
     CoreDeclarator
R9     : /*nothing*/ [s1-'(']         __id(rule_EmptyCoreDecl)
R10    : QualifiedName                __id(rule_NamedCoreDecl)
R11    : [s1] '(' Declarator ')'
       ;
     ParamDeclaration
R12    : /*nothing*/
R13    : TypeSpecifier Declarator
       ;
     QualifiedName
R14    :                                    identifier   __id(rule_SimpleName)
R15    :                              '::'  identifier   __id(rule_RootName)
R16    : QualifiedName [q1,q2]        '::'  identifier   __id(rule_QualName)
       ;
     Expression
R17    : number
       ;
```

All symbols in apostrophes are terminal symbols. Identifier and number are terminal symbols of the grammar. The notation

__id(name)

assigns a name to the rule. This name can be used in the definitions of the expected conflicts and in the callback handler.

TABLE OF CONTENTS