CONFLICT RESOLUTION AND OPERATOR PRIORITIES IN EXTENDED BNF

RESEARCH THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE IN COMPUTER SCIENCE

ANNA BEKKERMAN

SUBMITTED TO THE SENATE OF THE TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY

TAMUZ, 5764 HAIFA JULY, 2004

THE RESEARCH THESIS WAS DONE UNDER THE SUPERVISION OF DR. JOSEPH GIL
IN THE FACULTY OF COMPUTER SCIENCE

ACKNOWLEDGMENTS

I am grateful to my advisor Dr. Joseph Gil. He patiently guided me throughout the work, and taught me much about research.
I would like to thank my husband for his help. I thank my parents for their everlasting love, encouragement and support. Also, I am grateful to Felix Laventman for technical support.

THE GENEROUS FINANCIAL HELP OF THE TECHNION IS GRATEFULLY ACKNOWLEDGED

Contents

Notation **2**

1 Introduction and Related Works **4**
1.1 LL parsers . . . 6
1.2 LR parsers . . . 10
1.3 Employing EBNF for parsing . . . 13
1.4 Contributions . . . 19

2 Preliminaries and Definitions **21**
2.1 Grammars . . . 21
2.2 Derivation . . . 22
2.3 Parse Trees . . . 23
2.4 LR Parsing . . . 24
2.4.1 Actions . . . 26
2.4.2 Stack . . . 26
2.4.3 Parsing Table . . . 26
2.5 The Algorithm for LR-parsing Extended Context Free Grammars . . . 27

3 JAMOOS **29**
3.1 Extended BNF in JAMOOS . . . 29
3.2 Production Types . . . 31

4 Types of Conflicts **33**
4.1 Reduce/Reduce Conflicts . . . 34
4.2 Shift/Reduce Conflicts . . . 35
4.3 Pop/Reduce Conflicts . . . 35
4.4 Pop/Pop Conflicts . . . 36

5 JAMOOS Extension for Priorities and Associativity **37**
5.1 Priorities Section . . . 37
5.2 Local Priority Assignment . . . 39
5.3 Problems with Priorities in JAMOOS . . . 40
5.3.1 Assigning Priority and Associativity to a Production . . . 40
5.3.2 Assigning Priority and Associativity to a Token . . . 41
5.3.3 Statement of Context-Dependent Priority Problem . . . 41
5.3.4 Statement of Ambiguous Assignment Problem . . . 42
5.4 Solution for Context-Dependent Priority Problem . . . 43
5.5 Solution for Ambiguous Assignment Problem . . . 46

6 Algorithms Developed for Conflict Resolution **48**
6.1 Resolution of Reduce/Reduce Conflicts . . . 48
6.2 Resolution of Shift/Reduce Conflicts . . . 48
6.3 Resolution of Pop/Reduce Conflicts . . . 55
6.4 Resolution of Pop/Pop Conflicts . . . 56

7 Implementation **58**
7.1 Adding Syntactical Support for Priorities Section Definitions 60
7.2 Conflict Resolution Implementation 60
7.3 Grammar Report Generation 61
7.3.1 Generation of Conflict Reports 62
7.3.2 Parser Representation Format 63

8 Conflict Resolution in Grammars of JAVA and JAMOOS **64**
8.1 Shift/Reduce Conflicts 65
8.1.1 Array Creation Conflict 65
8.1.2 Modifiers Conflict 66
8.1.3 Dangling Else Conflict 68
8.2 Reduce/Reduce Conflicts 68
8.3 Pop/Reduce Conflicts 70
8.3.1 Choice Expressions in JAMOOS 70
8.3.2 Nested Choice Expressions Conflict 71
8.4 Pop/Pop Conflicts 73
8.4.1 Optional Expressions in JAMOOS 73
8.4.2 Optional-Choice Conflict 74

9 Conclusions **76**

A JAVA Language Grammar **79**

List of Figures

1 Parse trees derived for unambiguous input string 5
2 Parse trees derived for expressions `2+3*5` and `3*5+2` according to a grammar for which left recursion elimination technique was applied 8
3 Parse tree derived for the expression `6-4-1` according to an LL grammar that describes arithmetic expressions 9
4 Parse trees derived for the same sting according to a production which contains regular expressions 18
5 Parse tree for BNF grammar 24
6 Parse tree for EBNF grammar 24
7 Generation and execution of an object oriented parser by JAMOOS parser generator 31
8 Reduce/Reduce conflict 34
9 Shift/Reduce conflict 35
10 Pop/Reduce conflict 36
11 Pop/Pop conflict 37
12 Resolution of a Pop/Reduce conflict in favor of the pop action 55
13 Resolution of a Pop/Reduce conflict in favor of the reduce action 56
14 Resolution of a Pop/Pop conflict in favor of a shorter alternative 58
15 Resolution of a Pop/Pop conflict in favor of a longer alternative 58
16 Array creation conflict in JAVA 66
17 Modifiers conflict in JAVA 67
18 Resolution of the nested choice expression conflict in JAMOOS in favor of the pop action 72
19 Resolution of the nested choice expression conflict in JAMOOS in favor of the reduce action 72
20 Resolution of the optional-choice conflict in JAMOOS in favor of choice expression interpretation 75
21 Resolution of the optional-choice conflict in JAMOOS in favor of general optional expression interpretation 75

List of Algorithms

1 Madsen and Kristensen's generalized algorithm for LR parsing of Extended Context Free Grammars 30
2 Intermediate step in resolution of a Shift/Reduce conflict: split productions that contain the shifted token to two groups according to the token's priority and associativity with respect to the priority of the reduced production 45
3 Intermediate step in resolution of a Shift/Reduce conflict (lookahead): generate set of tokens that can potentially follow a given token 46
4 Intermediate step in resolution of a Shift/Reduce conflict (lookahead): generate two sets of tokens that can follow a given token; each group corresponds to a group of productions built in Algorithm 2 47
5 Reduce/Reduce conflict resolution 49
6 Priority assignment to a production according to the priority of its rightmost token . 50
7 Identification of the rightmost token in a production 51
8 Identification of the rightmost token in a list 52
9 Shift/Reduce conflict resolution 53
10 Priority and associativity assignment to a token according to the priority and associativity of a production in which the token appears 54
11 Pop/Reduce conflict resolution 57
12 Pop/Pop conflict resolution 59

Abstract

Conflict resolution is a fundamental problem in the theory of compilation. YACC resolves conflicts by means of priorities and associativity. The main disadvantage of YACC is that it is based on BNF grammars which do not provide intuitive way of a formal language description. Extended BNF (EBNF) improves BNF by involving regular expressions such as lists, alternatives etc. Many conflicts of BNF grammars do not occur in EBNF grammars. Still, EBNF grammars bear conflicts that should be resolved. Furthermore, in addition to standard types of conflicts (Shift/Reduce and Reduce/Reduce) two other types (Pop/Reduce and Pop/Pop) may occur.

Obviously, conflicts can be eliminated by the grammar redesign. Also, the user can resolve conflicts by extending the parser's functionality with special conflict resolution subroutines that would be unique for each grammar given. Both these user-provided solutions are extremely time consuming; the user should demonstrate deep knowledge in the parsing theory and large experience with the particular parser generator. A good conflict resolution method would be built-in the system, while a high-level user interface would be provided to decide about resolution strategies.

This thesis is the first work that proposes a built-in conflict resolution method that is universal for all four types of conflicts which may occur in EBNF grammars. To our knowledge, a built-in method for resolution of Pop/Reduce conflicts is first proposed in this thesis. Our method generalizes the popular approach of priorities and associativity. We provide original user interface for priorities and associativity assignment. In our method priorities can be explicitly specified for productions and production components as well as for tokens. In addition, Automatic Priority and Associativity Inference (APAI) is proposed for implicit assignment of priorities and associativity. Combined with explicit priority assignment, APAI allows managing priorities of abstract entities such as arithmetic operations. A runtime APAI is also proposed that performs a one-symbol lookahead for determining the actual parsing route in order to identify priority of the shifted token when more than one possible parsing routes exist for the input. The runtime APAI enriches the context information and provides the power of LR(2) without enlarging parsing tables. A set of original algorithms are designed for supporting the proposed conflict resolution method.

We implement these algorithms in a framework of a large ongoing project JAMOOS – a new object-oriented parser generator. In addition to natural advantages of its pure object-oriented software design, JAMOOS represents the ideal combination of a strong LR parsing technique and beneficial Extended BNF approach. Powered by our universal conflict resolution method, JAMOOS becomes the unique state-of-the-art modern system for parser generation.

Notation

L	Language
G	Grammar
P	Production
ε	Empty right-hand side of a production
t	Token
x	Either token or variable
$\mathcal{N}$	Set of nonterminal symbols
Σ	Set of terminal symbols
$\mathcal{P}$	Set of productions
$\mathcal{S}$	Start symbol of a grammar
A, B, D	Nonterminals
α, β, γ, ω, φ, ψ	Expressions consisting of tokens and variables
$RE(\mathcal{N} \cup \Sigma)$	Set of regular expressions over $\mathcal{N} \cup \Sigma$
ϵ	Set consisting of empty string
$\emptyset$	Empty set
$\Rightarrow$	Derivation
$\overset{*}{\Rightarrow}$	Derivation performed in zero or more steps
T	Parse tree
$\bullet$	LR-marker (dot)
s	Parser state
ϕ	Phrase of a language
$\#$	Indication on the action to be performed during parsing process. Used in algorithm for parsing EBNF grammars only
E_R	Regular expression
r	Reduce item
l	List regular expression
e_l	Repeated element of the list l
a	Alternative regular expression
ch	Choice in an alternative regular expression
$P_t(s)$	Set of all productions which yielded LR-items in the state s with the LR-marker directly before the token t
C	Conflict
π	Priority
ρ	Associativity
PT	Priority and associativity table
c	Component of a production
S_C	Set of LR-items from the parser state in which a conflict C occurs

APAI	Automatic Priority and Associativity Inference
BNF	Bachus-Naur Form
CFG	Context Free Grammar
EBNF	Extended Bachus-Naur Form
ECFG	Extended Context Free Grammar
LR parsing	A parsing technique. “L” stands for left-to-right scanning of the input, “R” stands for constructing rightmost derivation
MSc	Master of Science

1 Introduction and Related Works

A process of compilation consists of a number of stages: lexical analysis, *parsing*, semantic analysis, intermediate code generation, optimizations, assembly and linkage. This thesis is focused on the parsing stage of the compilation process. The parsing (or syntactic analysis) bears especial importance since all further compilation stages depend on its successful execution.

Given an input in a language L a *parse tree* of the input is built at the parsing stage. The parsing is based on a *grammar* G that describes the syntax of the language L. In most cases syntax rules of a language are described in a recursive meta-language called Backus-Naur Form (BNF).

All languages are classified into two groups: ambiguous and unambiguous. A language is ambiguous if it appears to express more than one possible meaning. For example, all natural languages are ambiguous. Consider the following sentence:

> She sent the apples to the buyers before they were ready.

The above sentence has two meanings. According to the first meaning the apples were not ready while according to the second one the buyers were not ready.

Programming languages may be ambiguous too. Probably the most famous ambiguity which appears in programming languages is related to conditional statements. For example, the following conditional statement is ambiguous:

```
if(i > 10) if(j < 0) k++ else k--
```

The **else** branch in the example above may be equally associated with both conditions. This ambiguity is called a *dangling else problem*. It appears in programming languages such as JAVA and C++. Actually dangling else problem is not the only ambiguity which appears in these languages [32, 33, 18].

Similar to languages grammars may be either ambiguous or unambiguous. However, the term ”ambiguity“ has a different meaning when it is applied to grammars. A grammar is ambiguous if two or more parse trees can be built for the same input string according to this grammar. In other words, an ambiguous grammar gives more than one way of understanding an input.

If a language is ambiguous all grammars which describe the language will be ambiguous. However, if a grammar is ambiguous it does not necessarily mean that the language described by this grammar is ambiguous.

Example 1. *Consider the following grammar which describes a list of identifiers separated by commas*[1]*:*

List → Id;	(1)
List → List OptComma Id;	(2)
List → List “,”;	(3)
OptComma → “,”;	(4)
OptComma → ε ;	(5)

The above grammar is ambiguous though the language described by the grammar is unambiguous. Figure 1 shows two parse trees obtained for the input `a , b`.

[1]The implementational part of this research was carried out in the framework of the JAMOOS project. JAMOOS is an object oriented language for grammars, all grammars which will be discussed in this work, will be written in JAMOOS's syntax unless other is specified. JAMOOS and its syntax are discussed in details in Section 3.

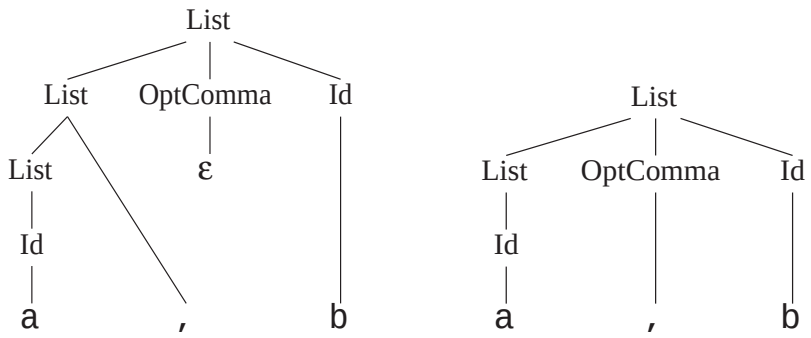

Figure 1: Parse trees derived for unambiguous input string.

Obviously ambiguous grammars cannot be employed in the compilation process: two or more parse trees for the same input program implies two or more executables with different meanings for this program. Unfortunately, even unambiguous grammars sometimes are unsuitable for parsing because they are too complicated for the parsing algorithm. For the purposes of this discussion, we call such grammars *confusing*.

Example 2. *Consider the following grammar that describes strings which consists of letters* a *only and whose length is odd:*

A → “a”; (1)
A → “a” A “a”; (2)

The grammar above is unambiguous: for each input string there exist a unique parse tree. Still, a parser generator such as YACC *that builds parsers which read their inputs strictly left-to-right cannot handle this grammar.*

According to the grammar a parsing process should start at the middle of an input. However, a parser which read an input from left to right cannot locate the middle of the input because it uses a finite lookahead.

Parsers built for both ambiguous and confusing grammars contain conflicts. There exist several approaches to dealing with conflicts.

One of the approaches implies redesign of a grammar. There are several disadvantages in this approach. First, grammars must provide a suitable framework for associating semantic meaning with the input. Rewriting of a grammar in most cases hampers the structure of the grammar and causes to rethinking the semantic attribution. Another problem is that given an ambiguous grammar G that describes a language L, it is undecidable whether there exist a grammar G' that describes L and does not contain conflicts [19]. It follows that grammar rewriting does not necessarily lead to eliminating of conflicts.

Another approach implies redesign of a language. For example, the dangling else problem would not occur if a terminating keyword such as **end** was used at the end of each conditional statement. The main disadvantage of this approach is that it often complicates the language. The **end** keyword which should be written after each conditional statement is an extra rule that anyone learning the language has to remember.

Many conflicts can be eliminated by changing a parsing algorithm. The decision that a parser repeatedly makes is: given what it has already read of an input, and the grammar productions it has already applied, which production should be applied next? The more context information the

parser can obtain before it has to make the decision, the more likely it is to be able to avoid a conflict. Different parsing techniques provide different amount of a context information to a parser. A considerable amount of conflicts, in particular those which occur in confusing grammars, can be eliminated by choosing a more powerful parsing algorithm.

Conflicts that cannot be eliminated by rewriting a grammar, language redesign or changing the parsing technique must be resolved. Each conflict is represented by two or more actions which a parser may perform at the same stage of a parsing process. A user or the parser itself decides which of these actions will be performed at parsing time. This decision can be made statically at the parser generation stage or dynamically at parsing time.

In the following sections we describe parsing techniques that can be employed for parsing formal languages, explain how conflicts occur, discuss existing conflict resolution methods, and present original conflict resolution method which is universal for all existing types of conflicts.

1.1 LL parsers

LL parsers are often called ”predictive“ or ”top-down“ because they start building a parse tree from its root. At each step of a parsing process an LL parser chooses a production which should be applied. The only context the parser uses in order to choose the production is k current symbols of an input string.[2]

Aho et al. [9] describe a simple procedure which calculates for each production a set of k symbol long strings. This set is called a ***Select** set*. At each step of the parsing process the parser chooses a production whose **Select** set contains a string which is equal to k current symbols of the input. If two or more productions have the same *left-hand side* and there exists a string which appears in **Select** sets of all these productions a conflict occurs.

The most popular approach to the conflict resolution in LL parsers is based on increasing the amount of the available context. There exist many different implementations of this approach. For example, one of conflict resolution methods in CppCC system [1] allows a number of lookahead symbols to be explicitly set for conflicting productions. SLK parser generator [6] builds parsers which in case of a conflict gradually increase a lookahead until a production that should be applied will be unequivocally determined.

A more sophisticated method implemented in ANTLR [27] and CppCC [1] systems employs syntactic predicates. A *syntactic predicate* defines a context in which certain production should appear. The context is defined as a simplified production. When a conflict between two productions P_1 and P_2 occurs a parser attempts to parse an input according to a syntactic predicate attached to P_1. If the parser succeeds the conflict is resolved in favor of P_1 because the context in which P_1 appears was found. Otherwise the conflict is resolved in favor of P_2.

Example 3. *Consider the following productions which describe statements in* E *language:*

```
Statement → List;            (1)
Statement → List "=" List;   (2)
```

The first production describes a list of variables. The second production describes parallel assignments: variables from the second list are assigned to variables from the first list.

The productions above contain a conflict: since both productions starts identically a parser with one symbol lookahead cannot decide which one to choose. Actually this conflict cannot be resolved by a parser with any fixed lookahead because the length of the variables list is not constant.

[2] In most cases LL parsers use only one symbol of lookahead.

The conflict can be resolved employing a syntactic predicate that would describe the context in which the production (2) *should appear. Since in case of parallel assignments a list of variables must be followed by the assignment operator the syntactic predicate may be defined as follows:*

List "="

The syntactic predicate above means that when the conflict occurs the parser will look in an input string for a list of variables followed by an assignment operator. If such a pattern will be found the production (2) *will be applied, otherwise the production* (1) *will be chosen.*

Other methods of conflict resolution involve backtracking: a parser tries to apply all conflicting production one after another, the first production which leads to a parse tree without errors is chosen [6, 23]. Often conflicts are resolved in favor of the longest match and not in favor of just the first successful one [7, 27, 21]. A *dangling else problem* is a notorious example of a conflict which can be effectively resolved by the longest match method.

Relatively small group of existing LL parser generators employs semantic information for a conflict resolution. This approach involves extracting of the meaning of the input. For example, ANTLR [27], CppCC [1] and LLGen [17] systems allow to attach a user-supplied function to conflicting productions. The resolution of the conflict depends on the value returned by this function. Assume, for example, that a conflict occurs between productions P_1 and P_2, and P_1 has a function attached to it. A parser will execute the function at parsing time. If the function returns true, then the conflict will be resolved in favor of P_1, otherwise it will be resolved in favor of P_2.

The most exotic conflict resolution approaches involve:

- Choosing the very first conflicting production [6].
- Finding all possible interpretations of an input [27]. All possible parse routes are proceeded in parallel. When all parse trees are constructed a user should choose from them the right one.
- Ignoring conflicts. For example, T-Gen [16] and Oops [22] parser generators deal only with grammars which contain no conflicts.

Even a superficial study of the variety of parser generators is enough to show that LL parsers are quite popular. The LL parsing method is fast and simple; it produces economical and readable parsers for small grammars. LL parsers are ideal for parsing tricky lines of input, date formats, simple nested data etc. LL parsers can also be constructed for certain programming languages such as C, ADA or PASCAL because these languages were specially designed to be easy for parsing [21]. However, LL parsers have several disadvantages which make them improper for parsing more complicated programming languages such as JAVA or C++.

One of the most annoying disadvantages of LL parsers is their inability to deal with *left recursion* and common prefixes in productions. As it was shown in the Example 3 productions which have common prefixes lead to conflicts. Aho et al. [10] showed a simple *left factoring* technique which eliminates common prefixes.

Left recursion occurs if a *nonterminal* symbol which appears in the left-hand side of a production P also is the leftmost symbol of the *right-hand side* of P.

Example 4. *Consider the following grammar which describes arithmetic expressions:*

Expression → Expression "+" Expression; (1)
Expression → Expression "-" Expression; (2)
Expression → Expression "*" Expression; (3)
Expression → Number; (4)

There is a left recursion in productions (1)*,* (2) *and* (3) *because the nonterminal* Expression *which appears in the left-hand sides of the productions is the leftmost symbol of their right-hand sides.*

None of top-down parsers can deal with left recursion because it makes them enter an infinite loop while parsing an input. A well-known technique shown by Aho et al. [10] helps to eliminate left recursion by employing right recursion instead. This technique though makes the resulting productions hardly understandable. For example, the grammar from the Example 4 will be transformed into the following grammar:

Expression → Number A;
A → B A;
A → ε ;
B → "+" Expression;
B → "-" Expression;
B → "*" Expression;

Besides of unreadability the grammar above has another problem: it does not allow to associate with operators any information about their precedence. Figure 2 shows that parse trees derived for the expressions `2+3*5` and `3*5+2` according to the grammar above are absolutely identical. It follows that though these expressions have the same meaning they will be interpreted differently. The expression `2+3*5` will be interpreted as `(2+(3*5))` while the expression `3*5+2` will be interpreted as `(3*(5+2))`.

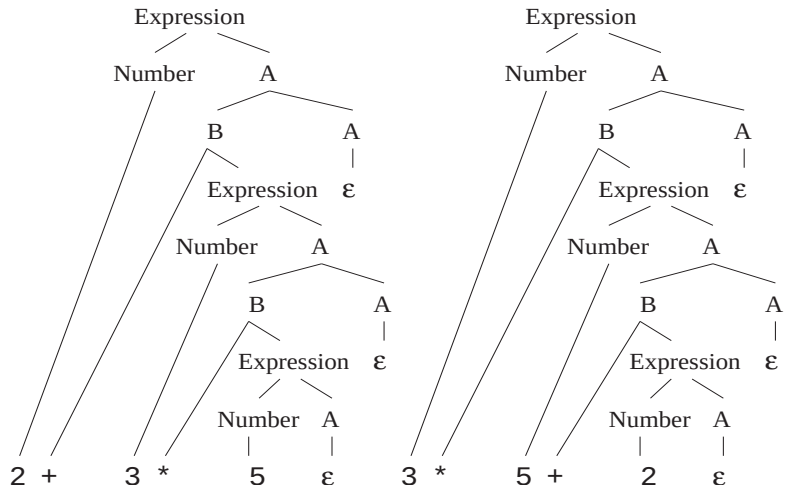

Figure 2: Parse trees derived for expressions `2+3*5` and `3*5+2` according to a grammar for which left recursion elimination technique was applied.

Aho et al. [10] show a technique which introduces operator precedences to an LL grammar. According to this technique operations with lower precedence are described through operations with higher precedence. For example, the grammar from the Example 2 will be modified as follows:

Expression → AddExpr I;
Expression → SubExpr I;
I → "+" AddExpr I;

I → “-” SubExpr I;
I → ε ;
AddExpr → MultExpr P;
SubExpr → MultExpr P;
P → “*” MultExpr P;
P → ε ;
MultExpr → Number;

The technique described above has two significant disadvantages. Firstly, it produces unnatural and confusing grammars when the number of arithmetic operations which should be described is greater than 40 (which is the case in all modern programming languages). Secondly, resulting grammars fail to handle left associative operators. For example, the subtraction operator will be wrongly interpreted as a right associative one according to the grammar above. Figure 3 shows that the expression `6-4-1` will be interpreted as `(6-(4-1))`.

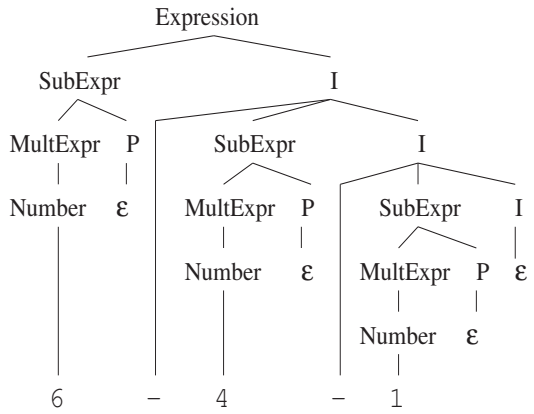

Figure 3: Parse tree derived for the expression `6-4-1` according to an LL grammar that describes arithmetic expressions.

The infamous problem of handling left associativity in LL grammars does not have any solution [10]. There exist a few techniques, however, which help to overcome this problem at the parser level: after the parser is created a user should manually extend its functionality to support left associativity.

The main problem of LL grammars that cannot be solved even manually is their weakness in terms of parsing: the group of languages that can be described by LL grammars is relatively small. There exist languages which cannot be recognized by LL parsers even if an infinite lookahead is available [10].

Example 5. *Consider the following LL grammar which describes strings consisting of n, $n \geq 0$, letters* **x** *followed by n letters* **y**. *A string ends with the letter* **e** *if n is even. A string ends with the letter* **o** *if n is odd.*

S → E “e”; (1)
S → O “o”; (2)
E → “x” O “y”; (3)
E → ε ; (4)
O → “x” E “y”; (5)

The grammar above contains a conflict. At the very first step of a parsing process either production (1) *or* (2) *should be chosen. Since the first symbol which will be derived from both nonterminals* `E` *and* `O` *is* `x` *the parser cannot decide which of the two productions to choose. A parser with* k, $k > 1$, *symbols lookahead is useless in this case: for each* k *there exist an input string in which the number of symbols* `x` *is* $k + 1$.

A conflict described in Example 5 occurs because LL parsers have access to few context information. The context for an LL parser is restricted to k lookahead symbols only. The parser must uniquely predict which production to apply basing on these lookahead symbols. The problem arises when several productions can be applied at the same time and the parser must choose one of them.

The problem of choosing the relevant production from several applicable ones would not occur if the parser could attempt to match all applicable productions at the same time and postponed making a decision until sufficient input had been seen. This is the strategy employed in LR parsers; many conflicts such as the one described in the Example 5 simply would not occur if an LR parser were used. It follows that LR parsers unlike LL ones can be employed for parsing complicated programming languages such as JAVA or C++.

1.2 LR parsers

LR parsers are often called ”bottom-up“ because they build a parse tree starting from its leaves. While scanning an input string an LR parser tries to find a production right-hand side that matches a part of the input which has been already seen. When the parser finds such a production it replaces this part of the input with a nonterminal on the production's left-hand side. The parser proceeds until it the whole input string will be replaced by a start nonterminal symbol.

The process of input scanning is called *shifting*. The process of replacing of some part of an input with a nonterminal symbol is called *reducing*. These two actions are essential in LR parsing strategy; they even gave another name to LR parsers – ”shift-reduce“ parsers.

LR parsers are stronger than LL parsers which were described in Section 1.1 because they use more context information. The context for an LR parser consists of all grammar productions consistent with the previously seen input and k symbols of lookahead [3]. An LR parser attempts to apply several productions at the same time and postpones making a decision until sufficient input has been seen.

Besides the input buffer an LR parser has a stack on which it keeps a list of states it has been in. The parser also has two tables that tell to which new state the parser should shift or by which production it should reduce given the state the parser is currently in and a terminal/nonterminal symbol it has just seen in the input.

The parser may reach a configuration in which basing on the stack content and the current input token it cannot decide whether to shift or reduce. In this case a *Shift/Reduce conflict* occurs. There also may exist a configuration in which the parser may reduce by several productions. In this case a *Reduce/Reduce conflict* occurs [4].

The most popular approach to conflict resolution implies that a user should prompt a parser in favor of which action a conflict should be resolved. One of implementations for this approach employs priorities and associativity. Simple yet powerful and elegant this method has gained a wide recognition [20, 4, 8, 30, 28, 32].

[3] In most cases LR parsers use only one symbol of lookahead.

[4] A detailed description of conflict types is given in Section 4

According to this method priority and associativity can be assigned both to tokens and productions. In most cases priority and associativity are assigned to a token explicitly while a production derives priority from the rightmost token in its right-hand side.

If a Shift/Reduce conflict occurs, priorities of the conflicting token and production are compared. If the token has a higher priority, the conflict is resolved in favor of shift. If the production has a higher priority, the conflict is resolved in favor of reduce. If priorities are equal, associativity is used: left associativity leads to choosing reduce, while right associativity leads to choosing shift.

A Reduce/Reduce conflict is resolved in favor of a production with the highest priority.

This method provides a natural solution for the problem of precedence and associativity of arithmetical operators (see Section 1.1).

Example 6. *Consider the grammar for arithmetic expressions from the Example 4:*

```
Expression → Expression "+" Expression;   (1)
Expression → Expression "-" Expression;   (2)
Expression → Expression "*" Expression;   (3)
Expression → Number;                      (4)
```

The grammar above contains a few Shift/Reduce conflicts. Consider some of these conflicts:

- *A conflict between a token "+" and a production* (3) *is actually a conflict between addition and multiplication operators. This conflict can be easily resolved by assigning higher priority to the token "*" which is associated with the multiplication operator. The production* (3) *will derive the priority from the token "*" because it is the rightmost token in the production's right-hand side. It follows that the conflict will be resolved in favor of the reduce action: multiplication will be performed before addition.*

- *A conflict between a token "-" and a production* (2) *is a conflict between two subtractions. This conflict can be resolved by assigning a left associativity to the token "-" which is associated with the subtraction operator. Since priorities of both the production* (2) *and the token "-" are equal associativity will be used for the conflict resolution. The conflict will be resolved in favor of the reduction because the token "-" is left associative.*

A slightly different method of assigning priorities implemented in SGLR system [32] employs *preference attributes*. A preference attribute is a *prefer* keyword which may follow any production. When a conflict occurs it is resolved in favor of a production to which a preference attribute is attached. In the example below the *dangling else conflict* is resolved so that an **else** branch will be associated with the innermost condition:

```
Statement → if Expression then Statement |
            if Expression then Statement else Statement {prefer};
```

Parser generators, such as PRECC [11] and BTYACC [12] employ a backtracking approach to conflict resolution. The obvious disadvantage of this approach is that it leads to highly inefficient parsers. On the other hand backtracking allows the user to refrain from dealing with conflict resolution as such. The entire responsibility for conflict resolution is upon a parser which eventually will come up with a correct parse tree.

According to this approach conflicts are resolved at parsing time as follows:

1. When the parser enters a conflict state, it remembers the content of the stack and the current input token.

2. The parser chooses one of the conflicting actions and continues parsing process.

3. If the parser runs into an error, it backtracks to the most recent conflict point and tries a different action.

The parser tries different parse routes until it finds a successful one or there is no more routes to try.

The order of productions in a grammar are important in this method, because if a conflict occurs, a parser examines the productions in order of their appearance. Therefore by setting productions in a certain order a user actually assigns a priority to them.

For example, in order to resolve the *dangling else problem* the user should define the choice that contains the **`else`** branch first:

```
Statement → if Expression then Statement else Statement |
            if Expression then Statement;
```

When the conflict will occur the parser will start to examine choices from the first alternative which will lead to associating the **`else`** branch with the innermost condition.

Some parser generators involve subgrammars for conflict resolution. For example, in AnaGram [28] a user can define a separate grammar for parsing logically independent pieces of an input such as regular expressions. Separating a grammar into several grammars constricts contexts in which productions may appear. The smaller a production context, the less likely that another production will appear in the same context which means that the number of conflicts decreases.

An interesting method of conflict resolution, proposed by Rus and Jones [29], involves analyzing of the context in which a conflict occurs.

Consider a Reduce/Reduce conflict between productions P_1 and P_2. When the conflict occurs the parser calculates two sets of tokens for both productions. One set is called *Follow* and contains all tokens that may appear after the left-hand side nonterminal of a production. Another set is called *Precede* and contains all tokens that may appear before the left-hand side nonterminal of the production.

The parser compares the current context with those in which P_1 and P_2 may appear. The conflict is resolved in favor of a production whose context is equal to the current one. If a one-symbol context is not enough to resolve the conflict, the parser calculates *Follow* and *Precede* sets that contain strings which consist of two tokens and so on.

A similar algorithm is used for Shift/Reduce conflicts resolution. Consider a Shift/Reduce conflict between the token t and the production P. When the conflict occurs the parser calculates two *Follow* sets. One set contains all tokens that may appear after t. Another set contains all tokens that may appear after the left-hand side variable of P. The conflict is resolved by comparing the contexts of t and P with the current one.

Surprisingly large number of parser generators do not employ any method of conflict resolution [28, 16, 2]. If a grammar contains conflicts a user is proposed to rewrite it. The main idea behind this approach is that it yields the highest level of confidence in the resulting parser [28]. This approach, however, has several disadvantages which were discussed in the beginning of this section.

In spite of the fact that LR parsers use much context information a considerable number of conflicts occur because a parser is not able to obtain enough context information in order to decide how to act.

Example 7. *Consider the following grammar that describes a declaration of a function in* E *programming language:*

FunctionDecl → TypeName Id "(" Params ")"; (1)
FunctionDecl → TypeName Id "(" ")"; (2)
Params → Param; (3)
Params → Params "," Param; (4)
Param → TypeName Ids; (5)
Ids → Id; (6)
Ids → Ids "," Id; (7)

Each function in E *language should return a value. The nonterminal* TypeName *in productions* (1) *and* (2) *describes the type of the returned value.*

A function may receive parameters. If two or more parameters have the same type they can be declared in one group; in this case the type is specified only once before the names of these parameters. Parameters (or groups of parameters) should be separated from each other by commas. Here is an example of a function declaration in E*:*

```
void foo(int a, b, short c)
```

The grammar contains a Shift/Reduce conflict. Assume that the above string is being parsed. When the substring

```
void foo(int a
```

has been read and comma is the current input symbol a parser can perform two actions: shift and reduction by the production (5)*. Performing the shift action in this situation will lead to reduction by the production* (7)*.*

Productions from Example 7 describe strings which represent nested repeated patterns. Identifiers separated by commas represent an inner pattern R. Pairs `(type name,` R`)` also separated by commas represent an outer pattern. The only way to describe such strings is to employ a recursive technique. It follows that a parser will have to reduce each time it founds an identifier though more natural behavior in this case would be to reduce only once after the entire expression has been seen.

Performing reductions on early stages of a parsing process often leads to conflicts. In the example above the conflict would not occur if the parser could read the whole expression before reduction.

There exist a natural way to avoid early reductions which employs Extended BNF (EBNF) for productions definitions. EBNF employ regular expressions for a grammar definition which allows to describe such complicate patterns as lists in one production. Consequently, a reduction is delayed to the later stages of a parsing process.

1.3 Employing EBNF for parsing

EBNF can be considered as an extension of BNF. The very first version of BNF was created by John Backus. Shortly after it was improved by Peter Naur; this improved version was publicly used for the first time, to define ALGOL 60 [26]. Many extensions to BNF were used to define programming languages after ALGOL 60, all slightly different.

In 1977 Niklaus Wirth proposed a single formulation of BNF which employed regular expressions [35]. Later this formulation got a special name – Extended BNF (EBNF). During the last thirty years large number of EBNF versions were created. In the most general case EBNF can be viewed as any variation on the basic BNF notation with the following additional constructs:

- Repetition operator ($*$ or $+$)
- Alternative operator (|) [5]

Here is a list of main EBNF dialects:

- **Wirth** According to Wirth [35] a left-hand side of a production P should be separated from its right-hand side by an assignment symbol ($=$); P should be ended with a dot. Terminal symbols should be embraced in double quotes. Terminals and nonterminals in the right-hand side of P should be separated by one or more spaces. Choices of an alternative should be separated by vertical bars (|). Optional sections should be embraced in square brackets. Lists should be embraced in curly brackets. The notations for both a list whose items may appear zero or more times and a list whose items may appear one or more times are equal. Wirth proposed the following technique for describing lists of the latter type [6]:

  ```
  List = Listitem { Listitem }.
  ```

  Here is an example of a production written in Wirth's EBNF dialect. The production describes the syntax of a URL:

  ```
  Url = ("http" | "ftp") "://" Domain "/" [ "~" UserName "/" ] {Field "/"}.
  ```

- **SAIF** Spatial Archive and Interchange Format (SAIF) employs a different version of EBNF [5]. In this version a left-hand side of a production should be separated from its right-hand side by a double semicolon followed by an assignment symbol (::=). Nonterminal symbols should be embraced in triangular brackets while terminal symbols should be embraced in double quotes. Terminals and nonterminals in the right-hand side of P should be separated by one or more spaces. Notations for alternatives, optional sections and lists are identical to those in Wirth's version.

  The following example shows the URL production from the previous paragraph written in SAIF dialect:

  ```
  ⟨Url⟩ ::= ("http" | "ftp") "://" ⟨Domain⟩ "/" [ "~" ⟨UserName⟩ "/" ] {⟨Field⟩ "/"}
  ```

- **Farrel** According to Farrel [14] a left-hand side of a production should be separated from its right-hand side by a semicolon followed by a double assignment symbol (:==). Terminal symbols should be embraced in single quotes.

  There exist two notations for alternatives. The first notation is identical to the one of Wirth: choices of an alternative should be separated by vertical bars (|). According to the second

[5] An alternative operator first was employed as a part of BNF as early as in ALGOL 60 [26]. However, in BNF the operator was employed only as a succinct notation for productions with identical left-hand sides. In EBNF this operator is also employed for defining productions whose right-hand sides contain alternatives.

[6] This example is written in Wirth's EBNF dialect.

notation the entire alternative is embraced in square brackets while its choices are separated from each other by commas. The second notation is similar to the one for optional sections which also should be embraced in square brackets. Such a similarity emphasizes the fact that an optional section can be considered as an alternation with two choices. The first choice is an item of the optional section while the second one is empty.

Lists should be embraced in curly brackets. A star which follows a close curly bracket indicates that items of a list may appear zero or more times.

The following example shows the URL production from the previous paragraph written in Farrel dialect:

```
Url :== ['http' , 'ftp'] '://' Domain '/' [ '~' UserName '/' ] {Field '/'}*
```

- **Pascal** Welsh and Elder employed an original version of EBNF for the description of PASCAL language [34]. According to this version a right-hand side of a production should be written under its left-hand side. There is no any specific symbol which would separate these two sides from each other. Similar to SAIF dialect nonterminal symbols should be embraced in triangular brackets. However, when a nonterminal appears on a left-hand side of a production it should not be embraced in brackets. Terminal symbols may be either embraced in double quotes or written in a bold font. Notations for alternatives, optional sections and lists are identical to those in Wirth's version.

  The following example shows the URL production from the previous paragraph written in Pascal dialect:

```
Url
(http | ftp) "://" ⟨Domain⟩ "/" [ "~" ⟨UserName⟩ "/" ] {⟨Field⟩ "/"}
```

- **ISO/IEC** In 1996 International Organization for Standardization (ISO) and International Electrotechnical Commision (IEC) approved a standardized version of EBNF [3]. In this version a left-hand side of a production P should be separated from its right-hand side by an assignment symbol ($=$). P should be ended with a semicolon. Terminals and nonterminals in the right-hand side of P should be separated by commas; ISO/IEC is the only version of EBNF in which symbols in a right-hand side of a production are explicitly separated from each other.

  Terminal symbols may be embraced either in single or in double quotes.

  Notations for alternatives and optional sections are identical to those in Wirth's version. There exist though different notations for lists. Lists should be embraced in curly brackets. A minus symbol ($-$) which follows a close curly bracket indicates that items of a list may appear one or more times. Items which should appear in an input exactly n times are prefixed by the number n and a following star. For example, the following production describes a string which consists of 4 letters a.

```
A = 4*"a";
```

ISO/IEC version contains a comment facility; each comment should be prefixed by a open bracket and a star ($(*$) and should be ended by a star and close bracket ($*)$).

This version also allows to define productions which specify exceptional cases. A minus symbol ($-$) is employed for this purpose. For example the following production defines that every symbol except of a semicolon may appear as a part of a comment:

```
CommentChar = Char - ";";
```

The following example shows the URL production from the previous paragraph written in ISO/IEC version of EBNF:

```
Url = "http" | "ftp" , "://" , Domain , "/" , [ "~" , UserName , "/" ] , {Field , "/"};
```

- **Jamoos** This research was carried out in the framework of the JAMOOS project which defines its own version of EBNF notation [31]. There is a direct correspondence between this notation and main principles of JAMOOS (see Section 3 for details).

  In JAMOOS version a left-hand side of a production P should be separated from its right-hand side by an arrow ($\rightarrow$). P should be ended with a semicolon. Terminal symbols may be embraced either in double or single quotes or may not be embraced by anything.

  A notation for optional sections is identical to the one in Wirth's version.

  There exist two different notations for alternatives. In both cases choices of an alternative should be separated by a bar ($|$). However, one of the notations allows specifying names for the choices while another does not.

  The notation for lists is rather sophisticated yet handy. As in all previously discussed EBNF dialects lists should be embraced in curly brackets. In contrast to all previous EBNF versions the JAMOOS notation allows to define sequences of symbols which may appear before and after a list, and symbols which may separate items of the list [31]. One or more plus symbols ($+$) may follow a list. The number of pluses actually defines a number of mandatory repetitions. For example, the following production describes strings which consist of at least three letters a:

```
A → {"a"}+++;
```

  JAMOOS version contains a comment facility; each line started by two minus symbols is considered as comment. The following example shows the URL production written in JAMOOS dialect:

```
Url → http OF "http" | ftp OF "ftp" "://" Domain "/" [ "~" UserName "/" ] {Field "/" ...};
```

EBNF is not more powerful than BNF; lists, alternatives and optional sections can be expressed in plain BNF using extra productions. However, employing EBNF for a grammar definition improves its readability because EBNF allows to describe many language constructs in a more natural and succinct way.

There exist two approaches to parsing EBNF grammars:

1. Conversion into equal plain BNF grammar and then employ the algorithm described by Aho et al. [9].
2. Employ the parsing algorithm that works directly on EBNF grammars [7].

The first approach has several disadvantages:

- Conversion into plain BNF destroys the semantic meaning of a grammar. It is a complicated task to split semantic actions among productions of a converted grammar, so that their resulting meaning will be preserved.
- Conversion into plain BNF may lead to appearance of new conflicts (see Example 7 and Example 8).

Employing second approach helps to avoid the occurrence of some conflicts. There exist two main reasons for this phenomena. First, EBNF allows to avoid null productions which are the main source of Reduce/Reduce conflicts. Second, EBNF allows to describe in one production complicate phrases which would require several productions if plain BNF was employed. It means that parsers which work directly on EBNF often perform reductions on later stages of a parsing process. Late reductions provide more context information to a parser which helps to avoid many conflicts.

Example 8. *Function declarations described in Example 7 can be defined by one EBNF production as follows:*

```
FunctionDecl → TypeName Id "(" { (TypeName { Id "," ... }+) "," ... } ")";
```

According to the production above parameters of a function should be embraced by brackets. The parameters may appear in groups where each group is described as follows:

```
TypeName { Id "," ... }+
```

In the production the expression above is followed by a comma embraced by quotation marks which means that the groups of parameters should be separated by commas. The dots that follows the comma mean that the groups of parameters may appear in the input several times.

The same syntax is used for description of a list of identifiers also separated by commas:

```
{ Id "," ... }+
```

The plus sign which follows the list means that at least one identifier should appear in the list. At the same time the outer list (list of parameter groups) is not followed by pluses which means that the parameters list may be empty.

The conflict described in the Example 7 in this case does not occur, because the parser performs reduction only when the entire expression is matched.

[7] The detailed description of such an algorithm is given in Section 2.5

The main drawback of late reductions is that a reduction procedure for the same production can be different depending on the input. Intuitively, employing regular expressions for description of a language L implies that there can be a situation when one production may derive the same string σ, $\sigma \in L$, differently.

Consider the following production:

```
A → a | aa {a ...};
```

The above production describes strings which consist of letters **a** only. Consider the string aaa. Obviously, this string is derived by the production A; still, the alternation in the right-hand side of the productions allows two different parse trees to be obtained for the string (see Figure 4).

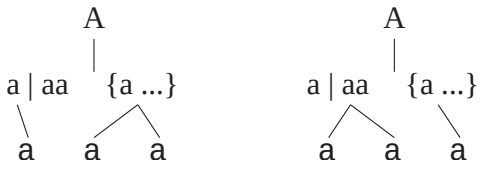

Figure 4: Parse trees derived for the same sting according to a production which contains regular expressions.

Madsen and Kristensen [24] showed that conflicts of two new types may occur in EBNF grammars:

- If a production P contains a list, the number of list elements that should be popped during reduction by P may be different. While performing a reduction by P, a parser may reach a configuration in which it must decide whether a symbol on the top of the stack is an element of the list or not. In this case we say that a *Pop/Reduce conflict* occurs.
- If a production P contains an alternative, different choices of it can be popped while reducing by P. While performing a reduction by P, a parser may reach a configuration in which two or more choices of the same alternative can be popped. In this case we say that a *Pop/Pop conflict* occurs.

Madsen and Kristensen [24] also generalized LR parsing theory introduced by Aho et al. [9] so that it can be applied to EBNF grammars. They proofed that in addition to Shift/Reduce and Reduce/Reduce conflicts only conflicts of the two types described above may occur in EBNF grammars [24, Theorem 2].

Surprisingly large number of parser generators which employ EBNF does not provide any means for resolution of Pop/Reduce and Pop/Pop conflicts explaining this by relatively rare occurrence of these conflicts [28, 16, 13]. We are aware of only two parser generators (Accent and LLGen) that do allow resolution of such conflicts.

Accent parser generator allows to resolve Pop/Pop conflicts by assigning priorities to conflicting choices of an alternative [30]. Each conflicting choice can be attached a number with the help of the keyword *%prio*. The conflict is resolved in favor of a choice whose attached number is the biggest. Accent does not provide any means for resolution of Pop/Reduce conflicts.

LLGen provides advanced method of conflict resolution which employs *conflict resolvers* [17]. Conflict resolver is a user-supplied routine which is called at parsing time when a conflict occurs. The resolution of the conflict depends on a boolean value which the conflict resolver returns.

In order to resolve a Pop/Pop conflict conflict resolvers are attached to conflicting choices of an alternation with the help of the keyword *%if*. When the conflict occurs the resolvers are executed in order of appearances of choices associated with them. The conflict is resolved in favor of the first choice whose resolver returns *true*.

The resolution of Pop/Reduce conflicts is different. The keyword *%while* is used to attach a conflict resolver to the element of a list. Each time the conflict occurs the conflict resolver is executed. If the resolver returns *true*, the symbol on the top of the stack is considered as an element of the list. If the conflict resolver returns *false*, the symbol on the top of the stack is not considered as an element of the list.

The method used by LLGen for conflict resolution has three significant drawbacks:

- The method is inflexible. For instance, it does not offer an intuitive way for supporting different levels of precedence.
- It is user-unfriendly. The user should be deeply involved in inner aspects of the parsing process.
- It is inefficient. The conflicts are resolved at runtime.

1.4 Contributions

It is difficult to compose an unambiguous readable grammar which would also provide a convenient framework for attaching semantic actions. The more complex a language is, the more complex is the problem of composing a grammar for this language.

In the previous sections we have shown that different parsing techniques provide different levels of assistance to a user with the problem of composing unambiguous grammars. The direct parsing of EBNF grammars discussed in Section 1.3 is the most powerful parsing method ever developed for parsing formal languages. This method minimizes the necessity for the user to redesign a grammar in order to eliminate conflicts. The less grammar redesign is needed, the more readable and intuitive the grammar remains.

Unfortunately, even employing of the most powerful parsing technique does not guarantee that a grammar will contain no conflicts, these conflicts should be resolved. A large number of conflict resolution methods have been developed. Most of the methods require that the user demonstrate a deep knowledge in the parsing theory. Often the methods are also intimately connected to a particular parser generator which complicates their employment by the user who has little experience with this parser generator. A good conflict resolution method would be built-in the system; minimal involvement of the user would be suggested.

In this thesis we propose a built-in conflict resolution method that is universal for all four types of conflicts which may occur in EBNF grammars. Our method generalizes the popular approach of priorities and associativity. Here are the most important characteristics of our conflict resolution method which comprise contributions of this work:

1. *User-friendly interface for priorities and associativity assignment.* In our method priorities can be explicitly assigned to productions and production components as well as to tokens. Priority and associativity can be assigned to a production by specifying a nonterminal symbol which appears on the left-hand side of the production (see Section 5). This mechanism is more intuitive and flexible than mechanisms which use special keywords for assigning priority, or mechanisms which derive production's priority and associativity from those of a certain token.

Another mechanism of priority assignment was developed as a part of algorithm for resolution of Pop/Reduce and Pop/Pop conflicts. Such conflicts occur between production components: Pop/Reduce conflict occurs between a list and its repeated item while Pop/Pop conflict occurs between choices of an alternative. We propose two mechanisms for assigning priority to production components:

(a) Assigning priorities to list and its repeated item by employing their user-defined names (see Section 5.1).

(b) Assigning priorities to choices of an alternative by specifying priority groups (see Section 5.2).

2. *Automatic Priority and Associativity Inference (APAI).* Often it is useful to treat a token and a production in which the token appears as a single unit that has priority and associativity. For example, a production that defines a multiplication operation and a token " *" obviously should have equal priority and associativity. We propose the APAI method based on this interdependent relationship between a production and tokens which appears in it.

   According to APAI method a token derives priority and associativity of the production in which it appears (see Section 5.3.2). On the other hand a production derives priority and associativity of the rightmost token in its right-hand side (see Section 5.3.1). The meaning of a rightmost token in an EBNF production is not well defined because in addition to nonterminals and tokens the production may contain lists, alternations and optional sections. We give a formal definition of the rightmost token in an EBNF production and provide algorithms for its identification (Algorithms 7 and 8).

   The formal description of an algorithm for APAI for tokens is presented in Algorithm 10. The formal description of an algorithm for APAI for productions is given in Algorithm 6.

3. *Runtime Automatic Priority and Associativity Inference (Runtime APAI).* In many cases the same token may have different meanings. For example, in C++ language the same token " *" is used both in multiplication operations and in pointer definitions. Obviously, in a grammar describing C++ this token appears in two different productions which means that it may infer priority and associativity differently.

   We propose a dynamic method for priority and associativity inference that disambiguates priority and associativity assignment for polysemantic tokens. In order to find an actual production from which a token derives its priority and associativity a parser attempts to determine a parse route that will be chosen for a current input. For this purpose the parser increases an available amount of context information by performing one symbol lookahead during the parsing time (see Section 5.4).

   A set of algorithms which support Runtime APAI has been developed (see Algorithms 2, 4, 3, and 9)

4. *Resolution of Pop/Reduce and Pop/Pop conflicts.* We propose a method for resolution Pop/Reduce and Pop/Pop conflicts (see Sections 6.3 and 6.4). The formal description of Pop/Reduce conflict resolution algorithm is given in Algorithm 11. The formal description of Pop/Pop conflict resolution algorithm is presented in Algorithm 12.

Outline The following section makes some pertinent definitions. All conflict resolution techniques presented in this thesis were implemented in the framework of the JAMOOS project which

is described in Section 3. Different conflict types are discussed in Section 4. Section 5 describes techniques for conflict resolution in EBNF grammars within the framework of JAMOOS project. Formal description of proposed conflict resolution algorithms is given in Section 6. Implementation of developed conflict resolution techniques and algorithms in JAMOOS project is discussed in Section 7. Conflict resolution techniques developed in this work were tested while generating parsers for JAMOOS and JAVA programming language; obtained conflicts and their resolution are subject of Section 8. Finally, Section 9 gives brief conclusions.

2 Preliminaries and Definitions

A discussion on the conflict resolution implies understanding of the most basic concepts of the compilation theory. In this section terms of the compilation theory necessary for the further discussion will be presented.

The concept of the conflict resolution is inseparable from the one of a grammar which is subject of the Section 2.1. All conflicts which will be discussed in this work will be illustrated by two or more possible parse trees. The concept of parse trees is based on the concept of a *derivation*. Definitions of a derivation and a parse tree are presented in Section 2.2 and Section 2.3 accordingly.

A resolution of conflicts takes place at a stage of a compilation process called parsing. Conflict resolution algorithms presented in this thesis will be described in terms of LR parsing. These algorithms, however, can be easily adapted to the LL parsing method. An explanation of the LR parsing process essentials is given in Section 2.4.

A special parsing algorithm for direct parsing of EBNF grammars is discussed in Section 2.5.

2.1 Grammars

A *context free grammar* (CFG) is an inductive definition of a formal language.

Definition 9 (CFG). *A* Context-Free Grammar *(CFG) is a 4-tuple* $G = (\mathcal{N}, \Sigma, \mathcal{P}, \mathcal{S})$ *where*

- $\mathcal{N}$ *is a finite set of* nonterminal symbols.
- Σ *is a finite set of* terminal symbols, *disjoint from* $\mathcal{N}$.
- $\mathcal{P}$ *is a finite set of* productions. *Each production has the form* $A \rightarrow \alpha$, *where* $A \in \mathcal{N}$ *and* $\alpha \in (\mathcal{N} \cup \Sigma)^*$.
- $\mathcal{S}$ *is a distinguished symbol in* $\mathcal{N}$ *called the* start symbol.

Each production $A \rightarrow \alpha$ consists of two parts: A is called *left-hand side* of the production and α is called *right-hand side* of the production. A production is called an *epsilon production* if its right-hand side is empty[8]. An epsilon production is denoted as follows:

$$A \rightarrow \varepsilon$$

An *extended context free grammar* (ECFG) is CFG in which the right-hand sides of the productions may contain *regular expressions*, i.e., repetition operators ($*$ or $+$) and alternative operator ($|$) may be used to express the right-hand side of the productions.

Here is a formal definition of regular expressions.

[8] In some sources a production with an empty right-hand side is called a *lambda production*.

Definition 10 (Regular expressions). *Let* $\{, \}, *, +, |$ *be symbols not in* $(\mathcal{N} \cup \Sigma)$. *The set of* regular expressions *over* $\mathcal{N} \cup \Sigma$, *denoted* $RE(\mathcal{N} \cup \Sigma)$ *is defined as follows:*

- *If* $\alpha \in (\mathcal{N} \cup \Sigma)$, *then* $\alpha \in RE(\mathcal{N} \cup \Sigma)$.
- *If* $\alpha, \beta \in RE(\mathcal{N} \cup \Sigma)$, *then* $\alpha\beta \in RE(\mathcal{N} \cup \Sigma)$.
- *If* $\alpha \in RE(\mathcal{N} \cup \Sigma)$, *then* $\{\alpha\}^* \in RE(\mathcal{N} \cup \Sigma)$.
- *If* $\alpha \in RE(\mathcal{N} \cup \Sigma)$, *then* $\{\alpha\}^+ \in RE(\mathcal{N} \cup \Sigma)$.
- *If* $\omega_1, \omega_2, \ldots, \omega_n \in RE(\mathcal{N} \cup \Sigma)$, *then* $\{\omega_1 \mid \omega_2 \mid \ldots \mid \omega_n\} \in RE(\mathcal{N} \cup \Sigma)$.
- $\epsilon \in RE(\mathcal{N} \cup \Sigma)$.
- $\emptyset \in RE(\mathcal{N} \cup \Sigma)$.

The expression $\{\alpha\}^*$ means that α can be repeated zero or more times, while the expression $\{\alpha\}^+$ means that α can be repeated one or more times. The expression $\{\omega_1 \mid \omega_2 \mid \ldots \mid \omega_n\}$ means precisely one of ω_1, ω_2, $\ldots$, ω_n. The symbol ϵ denotes the set $\{\epsilon\}$ consisting of the empty string and the symbol $\emptyset$ denotes empty set.

Definition 11 (ECFG). *An* extended context free grammar *(ECFG) is a 4-tuple* $G = (\mathcal{N}, \Sigma, \mathcal{P}, \mathcal{S})$ *where*

- $\mathcal{N}$ *is a finite set of* nonterminal symbols.
- Σ *is a finite set of* terminal symbols, *disjoint from* $\mathcal{N}$.
- $\mathcal{P}$ *is a finite set of* productions. *Each production has the form* $A \rightarrow \alpha$, *where* $A \in \mathcal{N}$ *and* $\alpha \in RE(\mathcal{N} \cup \Sigma)$.
- $\mathcal{S}$ *is a distinguished symbol in* $\mathcal{N}$ *called the* start symbol.

2.2 Derivation

Every string of a formal language can be generated employing the set of productions of a CFG that defines that language. A string can be generated by replacing left-hand sides of productions with their right-hand sides. This process is called *derivation*.

The symbol $\Rightarrow$ is used to denote a derivation. The expression $\alpha \Rightarrow \beta$ means that α *derives* β *in one step*. If a derivation is performed in several steps, the symbol $\overset{*}{\Rightarrow}$ is used to denote such a derivation. The expression $\alpha \overset{*}{\Rightarrow} \beta$ means that α *derives* β *in zero or more steps*.

For example, consider the following grammar for arithmetic expressions, with the nonterminal E representing an expression:

$$\text{E} \rightarrow \text{E} + \text{E} \mid \text{E} * \text{E} \mid (\ \text{E}\ ) \mid \textbf{id}$$

String `(id + id)*id` can be derived as follows:

$$\text{E} \Rightarrow \text{E} * \text{E} \Rightarrow \text{E} * \textbf{id} \Rightarrow (\ \text{E}\ ) * \textbf{id} \Rightarrow (\ \text{E} + \text{E}\ ) * \textbf{id} \Rightarrow (\ \text{E} + \textbf{id}\ ) * \textbf{id} \Rightarrow (\ \textbf{id} + \textbf{id}\ ) * \textbf{id}$$

In a *leftmost derivation*, the leftmost nonterminal is replaced at each step. In a *rightmost derivation*, the rightmost nonterminal is replaced at each step. The derivation of the example above is a rightmost derivation.

A non-terminal A is called *nullable* if $A \stackrel{*}{\Rightarrow} \epsilon$.

The derivation process is different for ECFG: not only nonterminal symbols can be replaced at each derivation step, but also regular expressions can be replaced by corresponding strings [24].

Example 12. *Consider the following grammar that describes a variables definition section in a* PASCAL*-like language.*

VarDef → {Type {VarName}* ;}*
Type → **int** | **char**
VarName → **id**

Consider the following statements which define three variables. Two of the variables are of the integer type while the third one is of the character type.

```
int a b;
char c;
```

Here is the derivation that corresponds to the above statements:

VarDef	⇒ { Type { VarName }* **;** }*	// VarDef *is replaced by the list of pairs*
		// of the following form:
		// (Type name, list of variable names)
	⇒ Type { VarName }* **;** Type { VarName }* **;**	*// The list of the pairs is replaces by two pairs*
		// according to the input
	⇒ Type { VarName }* **;** Type VarName **;**	*// The list of variable names is replaced by*
		// one variable name according to the input
	⇒ Type { VarName }* **;** Type **id ;**	
	⇒ Type { VarName }* **; CHAR id ;**	
	⇒ Type VarName VarName **; CHAR id ;**	*// The list of variable names is replaced by*
		// two variable names according to the input
	⇒ Type VarName **id ; CHAR id ;**	
	⇒ Type **id id ; CHAR id ;**	
	⇒ **INT id id ; CHAR id ;**	

2.3 Parse Trees

A *parse tree* is an alternative method to show the derivation process for some input.

Given a context free grammar $G = (\mathcal{N}, \Sigma, \mathcal{P}, \mathcal{S})$, a *parse tree* is a tree with the following properties:

- The root is labeled by $\mathcal{S}$.
- Each leaf is labeled by a token $t \in (\Sigma \cup \epsilon)$.
- Each interior node is labeled by a nonterminal $A \in \mathcal{N}$.

- Consider an interior node labeled by A, where $A \in \mathcal{N}$. Consider also n nodes labeled by $A_1, A_2, \ldots, A_n$, where $A_1 \in (\Sigma \cup \mathcal{N})$, $A_2 \in (\Sigma \cup \mathcal{N}), \ldots, A_n \in (\Sigma \cup \mathcal{N})$. If $A_1, A_2, \ldots, A_n$ are children of A (from left to right) then $A \to A_1 A_2 \cdots A_n$ is a production from $\mathcal{P}$.

Example 13. *Consider the following grammar:*

E → E + E | E ∗ E | (E) | **id**

The parsing tree for the input `(id + id)*id` *is shown in Figure 5:*

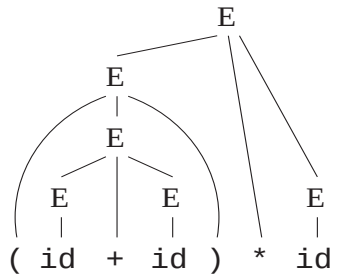

Figure 5: Parse tree for `(id + id)*id` according to the grammar in Example 13.

A parse tree T, which is built according to ECFG, will look different: since productions in ECFG may contain regular expressions in their right-hand sides, interior nodes of T may be labeled by regular expressions.

Example 14. *Consider the following grammar:*

VarDef → { Type { VarName }* **;** }*
Type → { **INT** | **CHAR** }
VarName → **id**

The parsing tree for the input `INT a b;` *is shown in Figure 6:*

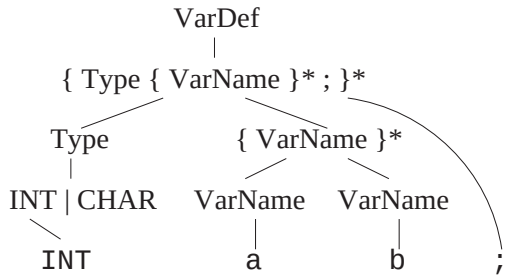

Figure 6: Parse tree for `INT a b;` according to the grammar in Example 14.

2.4 LR Parsing

Parsing is a process of determining whether a string of tokens can be derived from the start symbol of a grammar. A technique that is called *LR parsing* is used to parse context free grammars; "L" stands for left-to-right scanning of the input, "R" for constructing rightmost derivation.

The main idea behind the concept of parsing is that a language L described by a context free grammar G can be represented as a finite set of its possible configurations. The configurations are called *states* and built on the basis of productions in G.

Example 15. *Consider a language L that consists of the string a b only. Here is a grammar G which describes L:*

S → a b

The following three states can be built on the basis of G:

1. S → • a b
   Nothing has been read yet, the symbol **a** *is expected.*
2. S → a • b
   The symbol **a** *has been read, the symbol* **b** *is expected.*
3. S → a b •
   The string **a b** *has been read.*

The dot symbol (•) indicates the part of a string that has been already read. The dot also indicates a token of L which will appear next.

The states obtained from the grammar G can be organized in a directed graph. Nodes of the graph are the states, while edges are legal transitions between states. Each edge is labeled with a token or a variable of G. An edge labeled x goes from state s_1 to s_2 if s_1 contains a production with a dot before x and s_2 contains a production with a dot after x. This transition describes a basic step of parsing: in state s_1 the element x has not been yet read but has been expected, while in state s_2 the element x has already been read.

To summarize the above, the parsing process is travelling over the graph of states while keeping the travelling history: storing all the nodes visited and all the labels on the transition edges. When the parsing process eventually enters a state in which the dot is located after the end of the production (i.e. $A \to \alpha\bullet$) the right-hand expression α should be flushed off the parsing history and the left-hand variable A should be stored instead. The action of replacing the right-hand expression α with the left-hand variable A is called *reduction by production* $A \to \alpha$.

After reading the entire input the parsing history will contain only the start symbol of the grammar. In this case we say that the parsing process has been successfully finished: the parsed phrase ϕ belongs to the language L. However, if on a certain stage of the parsing process the current state is not connected to any other state by an edge labeled with the currently seen input token, the failure of parsing the input is reported.

The LR parser consists of:

- Stack
- Parsing table
- Parsing algorithm

2.4.1 Actions

The following actions may be performed during the parsing process:

- *Accept*. The parsing process succeeded. An input string is empty and the parser stack contains only one symbol, i.e., the start symbol.
- *Error*. No valid state transition exists for the current input token.
- *Shift*. Push the next input symbol to the parser stack.
- *Reduce*. The top of the parser stack holds the right-hand side of a production. The parser pops it off the stack and pushes the left-hand side of the production to the stack.

2.4.2 Stack

The stack stores pairs of the following form: (*grammar symbol, state*). A grammar symbol represents terminal or nonterminal symbol of the grammar. A state summarizes the information contained in the stack below it. The combination of the state number on the top of the stack and the current input symbol is used to index the parsing table and determine the action that should be performed.

2.4.3 Parsing Table

A parsing table has two parts: *action table* and *goto table*.

The *action table* is a table with rows indexed by parsing states and columns indexed by terminal symbols. For each parser state s and for each terminal symbol t the action table determines the action that should be applied if t is the current terminal symbol of the input string while the parser is in the state s.

The *goto table* is a table with rows indexed by parser states and columns indexed by nonterminal symbols. For each parser state s and for each nonterminal symbol A goto table determines the state number to which the parser should transit if reduction to A was performed in the state s.

The process of building the parsing table (see [9]) consists of two stages:

1. Creating the set of *LR-items* and grouping them into the parser states.
2. Constructing action and goto tables on the basis of the parser states.

An *LR-item* of a grammar G is a production of G with a LR-marker (dot) at some position of the right side. LR-marker indicates how much of a production has been already parsed. For example, production $A \rightarrow BD$ yields the three LR-items:

$A \rightarrow \bullet BD$
$A \rightarrow B \bullet D$
$A \rightarrow BD\bullet$

The first item indicates that a string derivable from BD is expected to appear next in the input. The second item indicates that a string derivable from B has been already parsed and a string derivable from D is expected to appear next in the input.

2.5 The Algorithm for LR-parsing Extended Context Free Grammars

LR-parsing algorithm for CFG [9] is based on the fact that the right-hand side of a production has fixed length and content. Here are two main characteristics of this algorithm:

1. Number of parser states is always finite.

   The number of productions in a grammar is finite. Since the right-hand side of a production P has fixed length, the number of possible LR-items yielded by P is finite. Therefore the number of all LR-item combinations is also finite. From these LR-item combinations only those which are legal input configurations are parser states.

2. Reduction procedure always is fixed sequence of pops.

   Consider a production $A \rightarrow \alpha$. Since α has fixed content, the procedure of reduction by A is fixed: pop from the stack all symbols in α in the reversed order.

In contrast to CFG right-hand side of a production in ECFG may have:

- Variable length, if list is used.
- Variable content, if alternation or optional section are used.

The algorithm for parsing CFG fails to parse ECFG because of the following reasons:

1. Number of parser states may be infinite.
2. Reduction procedure for the same production may be different.

Madsen and Kristensen [24] showed how the algorithm for parsing CFG can be modified to works directly on ECFG. The essential changes are:

1. New rules for moving the LR-marker through a regular expression for the purpose of computing LR-items.
2. Parsing algorithm is extended to work on tables constructed from the new type of LR-items. The difference is that new algorithm does not know the length of the right-hand side of an applied production.

The LR-items are constructed as usual [9] employing the following additional rules:

1. Any LR-item of the form

   $$A \rightarrow \alpha \bullet \{\beta\}^* \gamma$$

   is replaced by

   $$A \rightarrow \alpha \# \{\beta\}^* \gamma$$
   $$A \rightarrow \alpha \{\bullet \beta\}^* \gamma$$
   $$A \rightarrow \alpha \{\beta\}^* \bullet \gamma$$

2. Any LR-item of the form

   $$A \rightarrow \alpha \{\beta \bullet\}^* \gamma$$

   is replaced by

$$A \to \alpha\{\beta\#\}^*\gamma$$
$$A \to \alpha\{\bullet\beta\}^*\gamma$$
$$A \to \alpha\{\beta\}^* \bullet \gamma$$

3. Any LR-item of the form

$$A \to \alpha \bullet \{\beta\}^+\gamma$$

is replaced by

$$A \to \alpha\#\{\beta\}^+\gamma$$
$$A \to \alpha\{\bullet\beta\}^+\gamma$$

4. Any LR-item of the form

$$A \to \alpha\{\beta\bullet\}^+\gamma$$

is replaced by

$$A \to \alpha\{\beta\#\}^+\gamma$$
$$A \to \alpha\{\bullet\beta\}^+\gamma$$
$$A \to \alpha\{\beta\}^+ \bullet \gamma$$

5. Any LR-item of the form

$$A \to \alpha \bullet \{\omega_1 \mid \omega_2 \mid \ldots \mid \omega_n\}\beta$$

is replaced by

$$A \to \alpha\{\omega_1 \mid \omega_2 \mid \ldots \mid \bullet\omega_i \mid \ldots \mid \omega_n\}\beta$$

for $i = 1, 2, \ldots, n$.

6. Any LR-item of the form

$$A \to \alpha\{\omega_1 \mid \omega_2 \mid \ldots \mid \omega_i\bullet \mid \ldots \mid \omega_n\}\beta$$

is replaced by

$$A \to \alpha\{\omega_1 \mid \omega_2 \mid \ldots \mid \omega_i\# \mid \ldots \mid \omega_n\}\beta$$
$$A \to \alpha\{\omega_1 \mid \omega_2 \mid \ldots \mid \omega_n\} \bullet \beta$$

The idea behind the rules is to keep track of which productions are applicable at a given point in the input string, in the manner consistent with the meanings of regular expressions. New symbol $\#$ indicates what is the current content of the stack. If this symbol appears at the end of a repeated pattern (when $*$ or $+$ operators are used) it means that the pattern is now on the top of the stack and it should be popped. Accordingly, if the symbol appears after some alternative (when $|$ operator is used) it is the sign that this alternative should be popped now. On the contrary, if this symbol appears before a repeated pattern it means that there is no such pattern on the stack.

An LR-item that contains symbol $\#$ is called *reduce item*.

The parsing algorithm is alike the common algorithm [9] except when a reduction is applied.

Assume that the production $A \to \alpha$ is applied. Right-hand side of the production can be split into $\alpha_1, \alpha_2, \ldots, \alpha_n$ ($n \geq 1$) such that $\alpha = \alpha_1\alpha_2 \ldots \alpha_n$ and each α_i has the form:

1. $\alpha_i \in (N \cup \Sigma)^*$.
2. $\alpha_i = \{\beta\}^*$ for some β.
3. $\alpha_i = \{\beta\}^+$ for some β.
4. $\alpha_i = \{\omega_1 \mid \omega_2 \mid \ldots \mid \omega_m\}$ for some ω_j ($j = 1, 2, \ldots, m$).

The reduction can be split into popping first $|\alpha_n|$ symbols, then $|\alpha_{n-1}|$ symbols etc., and finally $|\alpha_1|$ symbols. If α_i represents regular expression (the last three forms of α_i) similar splitting have to be done since α_i may contain nested regular expressions.

The formal description of the reduce function is presented in Algorithm 1.

3 JAMOOS

JAMOOS is object oriented language for grammars [31]. JAMOOS exploits the correspondence between grammar productions and object oriented classes. The main principle of this correspondence is that a grammar production also defines a class. The left-hand side of the production declares the name of the class, and the right-hand side defines the structure of the class. A production of the form `A` $\rightarrow$ `B | C` defines an inheritance relationship between classes. A production of the form `X` $\rightarrow$ `BC` defines an aggregation relationship between classes.

JAMOOS receives an EBNF grammar G that describes a language L and creates for it an object oriented parser. Terminal symbols of G are called *tokens*. Non-terminal symbols of G are called *variables*.

The object oriented parser consists of two parts:

- **Set of classes**. For each production in G JAMOOS defines a corresponding class.
- **Parser**. The parser identifies grammatical phrases of L in the input stream. For each phrase ϕ there is a corresponding production P_ϕ in G. Each time the parser identifies ϕ it creates an object of a class that corresponds to P_ϕ.

A parse tree produced by the object oriented parser can be viewed from grammatical and object oriented points of view. From the grammatical point of view the parse tree is an hierarchical tree that shows the parsing process (see Section 2.3). From the object oriented point of view the parse tree is an object with a recursive structure. This object corresponds to the root of the tree, and all the nested objects correspond to its subtrees.

Figure 7 shows the processes of generation and execution of the object oriented parser.

3.1 Extended BNF in JAMOOS

There exist two approaches to parsing EBNF grammars:

1. Conversion into equal plain BNF grammar and then employ the algorithm described by Aho et al. [9].
2. Employ the parsing algorithm that works directly on EBNF grammars.

The first approach has several disadvantages:

Algorithm 1 Reduce function of the parsing algorithm that works directly on ECFG. The algorithm does not examine the case when right-hand site of an applied production contains regular expression of the form: $\{\alpha\}^+$. Such regular expressions should be treated in exactly the same manner as regular expressions of the form: $\{\alpha\}^*$.

Input: α – a string of terminal symbols, nonterminal symbols and regular expressions

Procedure Reduce

1: Split α into $\alpha_1, \alpha_2, \ldots, \alpha_n$
2: **for** $i = n, \ldots, 1$ **do**
3: $\varphi = \alpha_1 \ldots \alpha_{i-1}$
4: $\psi = \alpha_{i+1} \ldots \alpha_n$
5: **if** $\alpha_i \in (N \cup \Sigma)^*$ **then**
6: Pop $|\alpha_i|$ symbols from the stack
7: **else**
8: **if** $\alpha_i = \{\beta\}^*$ **then**
9: **repeat**
10: $s \leftarrow$ Stack.top.state *// Parser state whose number is on the top of the stack*
11: **if** s contains both items $A \rightarrow \varphi\{\beta\#\}^*\psi$ and $A \rightarrow \varphi\#\{\beta\}^*\psi$ **then**
12: **Report conflict**
13: **Exit**
14: **end if**
15: **if** s contains item $A \rightarrow \varphi\{\beta\#\}^*\psi$ **then**
16: **Reduce (**β**)**
17: **end if**
18: **until** s contains item $A \rightarrow \varphi\#\{\beta\}^*\psi$
19: **else** *//* $\alpha_i = \{\omega_1 \mid \omega_2 \mid \ldots \mid \omega_m\}$
20: $s \leftarrow$ Stack.top.state
21: **if** s contains both items $A \rightarrow \varphi\{\omega_1 \mid \ldots \mid \omega_j\# \mid \ldots \mid \omega_m\}\psi$ and $A \rightarrow \varphi\{\omega_1 \mid \ldots \mid \omega_s\# \mid \ldots \mid \omega_m\}\psi$, $j \neq s$ **then**
22: **Report conflict**
23: **Exit**
24: **end if**
25: **if** s contains item $A \rightarrow \varphi\{\omega_1 \mid \ldots \mid \omega_j\# \mid \ldots \mid \omega_m\}\psi$ **then**
26: **Reduce (**ω_j**)**
27: **end if**
28: **end if**
29: **end if**
30: **end for**

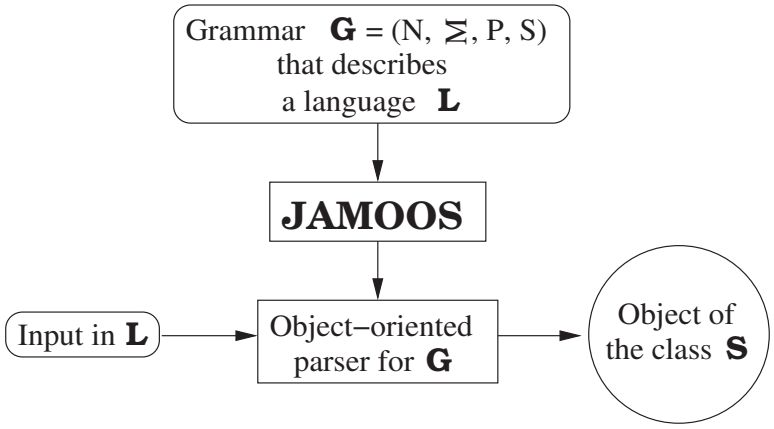

Figure 7: Generation of an object oriented parser by JAMOOS and execution of the parser.

- Conversion into plain BNF destroys all the semantic meaning of the grammar, i.e., classes and fields defined by the JAMOOS productions. It is a complicated task to redefine all the classes according to the converted grammar, so that their resulting meaning will be preserved.
- Conversion into plain BNF may lead to appearance of new conflicts (see Example 7 and Example 8).

JAMOOS uses the second approach for parsing EBNF grammars. It implements the algorithm developed by Madsen and Kristensen [24] (Section 2.5 provides a detailed discussion on the algorithm).

3.2 Production Types

There are two types of productions *common* and *ignore*. *Common* productions represent productions of EBNF grammar. *Ignore* productions describe elements such as blanks, tabulations, newlines and comments that should be ignored by a parser.

Common productions are divided into *inheritance* and *composition* productions. *Inheritance* productions define inheritance: classes, defined by the non-terminals in the right-hand side of a production P, inherit from the class, defined by the non-terminal in the left-hand side of P. For example, the following production:

Expression → AddExpr | MultExpr | Number;

defines four classes: *Expression*, *AddExpr*, *MultExpr* and *Number*, where the last three classes inherit from the class *Expression*.

Composition productions define classes. Right-hand side of a composition production consist of JAMOOS *language components*. Each JAMOOS language component defines a field in the class described by the production. Further we will refer to JAMOOS language component simply as component.

Definition 16 (Component). Component *is defined as follows:*

- *If a is a token, then a is a component.*

- *If a is a variable, then a is a component.*
- *If a_1, a_2, ..., a_n are components, then $a_1 a_2 \dots a_n$ is a component.*
- *If a, a_I, a_S and a_T are components, then $\{a_I \setminus a\ a_S \dots a_T\}\tilde{+}$ is a component.*

  In this component a_I, a_S and a_T may be omitted. By $\tilde{+}$ we represent a sequence of symbols $+$ that may have arbitrary length.
- *If a is a component, then $[a]$ is a component.*
- *If a_1, a_2, ..., a_n are components and x_1, x_2, ..., x_n are strings, then $x_1\ OF\ a_1 | x_2\ OF\ a_2 | \dots | x_n\ OF\ a_n$ is a component.*

The component $a_1 a_2 \dots a_n$ is called a *sequence* component.

The component $\{a_I \setminus a\ a_S \dots a_T\}\tilde{+}$ is called *list* component. It means than the a component is repeated as many times as the symbol $+$ appears in $\tilde{+}$. The component a is called *repeated item of the list*. Repeated items may be separated by the component a_S. The component a_I may appear before the sequence of repeated items. The component a_T may appear before the sequence of repeated items. Here is an example of a production that contains a list:

```
VarDecl → {Type\ VarName "," ... ";"}+;
```

The component $[a]$ is called *optional section*. It means that the component a may appear zero or one time. Here is an example of a production that contains an optional section:

```
ConditionalStmt → if Expression then Stmt [else Stmt];
```

The component $x_1\ OF\ a_1 | x_2\ OF\ a_2 | \dots | x_n\ OF\ a_n$ is called *alternation*. It means that only one component a_i, $i = 1, \dots, n$, can appear. The component a_i, $i = 1, \dots, n$, is called *choice*. Each choice a_i, $i = 1, \dots, n$, has a name x_i. Here is an example of a production that contains an alternation:

```
Operator → plus OF "+" | mult OF "*";
```

Components may be given names. If a component has a name the corresponding class field has the same name. A component and its name are separated by semicolon. For example:

```
Program → "Program" progName:Id body:{Statement ";" ...}+ "End";
```

In the example above:

- The component Id has a name *progName*
- The list {Statement ";" ...}+ has a name *body*
- The component Statement has no name.

4 Types of Conflicts

The common parsing algorithm [9] is based on the fact that the right-hand side of a production has fixed length and content. This fact gives a possibility to build a set of parser states; each parser state unequivocally defines a part of a production that:

- has been already seen
- is expected to appear next in the input

If a production P contains a regular expression E_R, the exact length (if E_R is a list) or content (if E_R is an alternative) of the right-hand side of P is not known. The algorithm for parsing EBNF grammars [24] uses special hints that help to parse regular expressions. Such hints are represented as reduce items in parser states (see 2.5).

While parsing a list a parser has to decide when the list ends. Consider the following production:

$$A \to \{\alpha\}*$$

The production above yields two reduce items with different meanings:

- $A \to \{\alpha\#\}*$
  This reduce item hints to the parser that the current symbol in the stack is the element of the list.
- $A \to \#\{\alpha\}*$
  This reduce item hints to the parser that the current symbol in the stack is not the element of the list.

Definition 17 (Conflicting list reduce items). *Reduce items r_1, r_2 are called* conflicting list reduce items *if the following conditions are hold:*

1. *Both r_1 and r_2 are yielded by the same production P.*
2. *The right-hand side of P has the form $\alpha\{\beta\}^*\gamma$ or $\alpha\{\beta\}^+\gamma$.*
3. *r_1 has the form $\alpha\#\{\beta\}^*\gamma$.*
4. *r_2 has the form $\alpha\{\beta\#\}^*\gamma$.*

While parsing an alternative a parser has to decide which choice should be popped. Consider the following production:

$$A \to \alpha \mid \beta$$

The production above yields two reduce items with different meanings:

- $A \to \alpha\# \mid \beta$
  This reduce item hints to the parser that the choice α should be popped from the stack.
- $A \to \alpha \mid \beta\#$
  This reduce item hints to the parser that the choice β should be popped from the stack.

Definition 18 (Conflicting alternative reduce items). *Reduce items r_1, r_2 are called* conflicting alternative reduce items *if the following conditions are hold:*

1. *Both r_1 and r_2 are yielded by the same production P.*
2. *The right-hand side of P has the form $\alpha\{\omega_1 \mid \omega_2 \mid \ldots \mid \omega_n\}\beta$.*
3. *r_1 has the form $\alpha\{\omega_1 \mid \omega_2 \mid \ldots \mid \omega_i\# \mid \ldots \mid \omega_n\}\beta$, $i = 1, 2, \ldots, n$.*
4. *r_2 has the form $\alpha\{\omega_1 \mid \omega_2 \mid \ldots \mid \omega_j\# \mid \ldots \mid \omega_n\}\beta$, $j = 1, 2, \ldots, n$ and $j \neq i$.*

We say that ECFG contains *conflicts* if the LR parser that was build for the grammar contains *conflict states*.

Definition 19 (Conflict state). *A parser state s of a parser that works directly on ECFG is called* conflict state *if at least one of the following conditions holds:*

- *Multiple actions are defined for s and a terminal symbol in the action table of the parser.*
- *s contains conflicting list reduce items.*
- *s contains conflicting alternative reduce items.*

There exist four types of conflicts. Two of them, *Reduce/Reduce* and *Shift/Reduce*, occur between several actions. Another two types of conflicts, *Pop/Reduce* and *Pop/Pop*, differ from the two previous types: they occur if there exist more than one possibility to execute a reduction.

4.1 Reduce/Reduce Conflicts

Reduce/Reduce conflicts occur in a conflict state for which two or more reduce actions are defined. In other words, this conflict occurs if there are two or more productions that can be applied to the same input string. For example, consider the following grammar $G_{Friends}$:

Friends → boys **OF** (BoyName **and** “John”) | girls **OF** (GirlName **and** “Jane”);
BoyName → “Alex”;
GirlName → “Alex”;

Consider an input *Alex and John*. The string *Alex* can be reduced to both BoyName and GirlName productions. Figure 8 shows two possible parse trees which can be produced.

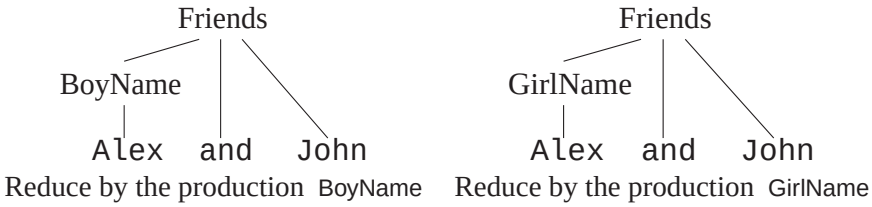

Figure 8: Reduce/Reduce conflict which occurs in the grammar $G_{Friends}$

4.2 Shift/Reduce Conflicts

Shift/Reduce conflicts occur in a conflict state for which both shift and reduce actions are defined. In other words, this conflict occurs if the content of the stack matches right-hand side of a production (the reduce action can be performed) and at the same time it is also legitimate to shift the current token, because this would lead to a reduction by another production.

For example, consider the following grammar G_{Expr} for arithmetical expressions:

```
Expression → AddExpr | MultExpr | Number;
AddExpr → Expression "+" Expression;
MultExpr → Expression "*" Expression;
Number → ⟨[1-9][0-9]*⟩;
```

The grammar G_{Expr} contains several Shift/Reduce conflicts. One of them is between the token + and the production MultExpr.

Consider the input $2 * 6 + 3$. When the substring $2 * 6$ has been already seen and the token + is the current input token, both shift and reduce actions can be performed. The corresponding parse trees are shown in Figure 9:

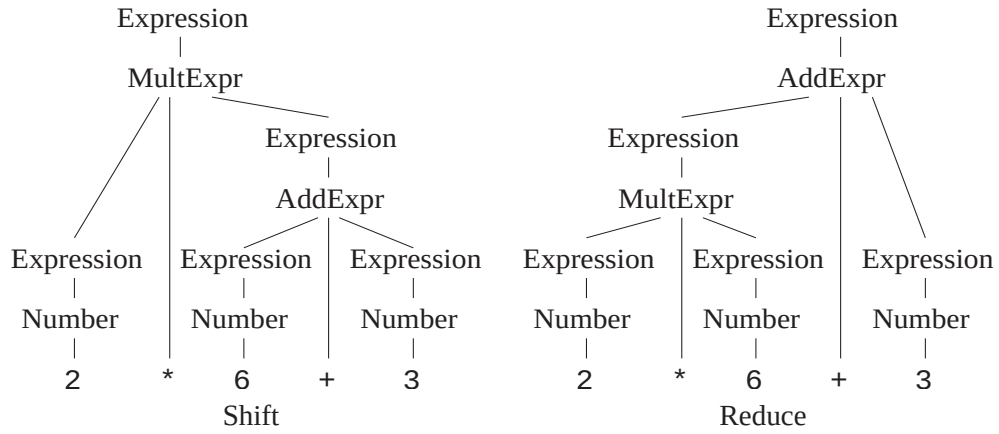

Figure 9: Shift/Reduce conflict between the token + and the production MultExpr which occurs in the grammar G_{Expr}

4.3 Pop/Reduce Conflicts

Pop/Reduce conflicts occur in a conflict state that contains conflicting list reduce items.

Consider a production P which contains a list l in its right-hand side. While parsing the same input there may exist two or more ways to reduce by P so that the content of l would be different after each reduction. In this case we say that Pop/Reduce conflict occurs.

For example, this conflict occurs in the following grammar which describes strings that consist of the symbols a only:

```
A → [ "a" ] { "a" ... };
```

Consider an input that consists of two symbols a. The parser for the grammar above will read the input, put both symbols to the stack and start to reduce by A. The reduction by A is performed in two stages: first, all elements of the list should be popped from the stack, second, the element of the optional section should be popped.

Consider the first reduction stage. Since the optional section contains only one element, the second symbol a in the input obviously is the part of the list, so, the parser will pop it from the stack. Then the parser should decide whether the remaining symbol a is an element of the list or an element of the optional section.

In the first case the parser will pop the symbol a from the stack. At this point of the parsing process the stack will be empty. This will cause the parser to finish the first reduction stage and to start the second one.[9] When the reduction will be finished, the list will contain two elements and the optional section will be empty.

In the second case the parser will assume that all list elements have already been popped and will start the second reduction stage. When the reduction will be finished, the list will contain one element and the optional section will contain an element.

Figure 10 shows two possible parsing trees which can be produced for the input *aa*.

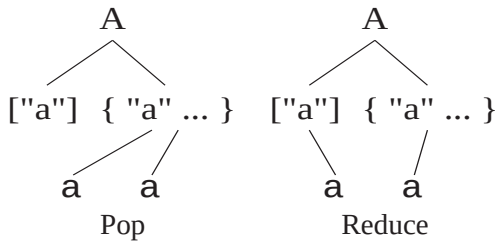

Figure 10: Pop/Reduce conflict which occurs in production A → [“a”] { “a” … };

4.4 Pop/Pop Conflicts

Pop/Pop conflicts occur in a conflict state that contains conflicting alternative reduce items.

Consider a production P which contains an alternative in its right-hand side. While parsing the same input there may exist two or more ways to reduce by P so that different choice will be made at each reduction. In this case we say that Pop/Pop conflict occurs.

For example, this conflict occurs in the following grammar G_A:

```
A → alt1:(c1 OF “a” | c2 OF “ab”)
    alt2:(c3 OF “bc” | c4 OF “c”);
```

Consider the input *abc*. The parser for G_A will read the three tokens of the input, push them to the stack and start to reduce by A. The reduction by A is performed in two stages: first, the alternative **alt1** should be reduced, second, the alternative **alt2** should be reduced. At the first stage either choice “bc” or “c” should be popped from the stack. At the second stage either choice “a” or “ab” should be popped from the stack.

[9] In this case nothing will be done at the second reduction stage. Since the stack is empty, the optional section will also be considered empty.

Consider the first reduction stage. Both choices "bc" and "c" are legitimate. If the parser pops choice "bc" then the choice "a" should be popped at the second reduction stage. If the parser pops choice "c" then the choice "ab" should be popped at the second reduction stage.

Figure 11 shows two possible parsing trees which can be produced for the input *abc*.

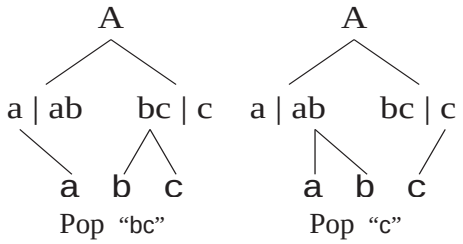

Figure 11: Pop/Pop conflict which occurs in the grammar G_A

5 JAMOOS Extension for Priorities and Associativity

The Shift/Reduce, Reduce/Reduce, Pop/Reduce and Pop/Pop conflicts are resolved employing priorities and associativity. JAMOOS allows to assign priority and associativity to both tokens and nonterminal symbols. By assigning priority and associativity to a nonterminal symbol, these priority and associativity are assigned to the production which has the nonterminal symbol on its left-hand side.

JAMOOS allows to assign priority to components. This can be done by assigning priority to the name of a component. Consider the following production:

VarDeclaration → Type var_list:{ var_name:Var ... }+;

There are three components in the production above:

1. Variable Type. This component is unnamed.
2. List { var_name:Var ... }+. The name of this component is **var_list**.
3. Variable Var. This component is the repeated item of the list { var_name:Var ... }+. The name of this component is **var_name**.

By assigning priority to the name **var_list**, the priority is assigned to the list. By assigning priority to the name **var_name**, the priority is assigned to the element of the list.

Priorities and associativity in JAMOOS can be assigned in the priorities section (see Sec. 5.1). JAMOOS also provides additional method of priority assigning: locally in connection to a specific production (see Sec. 5.2).

5.1 Priorities Section

In the priorities section groups of tokens, nonterminal symbols and component names are declared in a certain order, so that a group has lower priority than the priority of the groups declared afterwards. Within each group, the tokens, nonterminal symbols and component names have the same priority.

For each group of tokens, nonterminal symbols and component names the associativity of its elements should be defined. The associativity can be declared employing keywords **LEFT** or **RIGHT**.

Component names do not have associativity. Although a component name appears in a group with a certain associativity, this associativity is not assigned to the component name.

There are two methods of assigning priority to a component name:

- **Specifying simple name**. Consider a component name id_1 that appears in only one production. A priority can be assigned to id_1 just by declaring id_1 in the priorities section.
- **Specifying qualified name**. Consider a component name id_2 that appears in the right-hand sides of the productions $A \rightarrow \alpha$ and $B \rightarrow \beta$. Assume that a priority should be assigned to the component id_2 which appears in the production $A \rightarrow \alpha$. The priority can be assigned by specifying in the priorities section a qualified component name which includes the variable A and the component name id_2 separated by dot.

Below is a JAMOOS definitions that describes the syntax of the priorities section:

```
PrioritiesSection →
    { "PRIORITIES" /
        ( left of "LEFT" | right of "RIGHT"
          { token of Token |
            NT_symbol of Name |
            qualified_name of (Name "." Id) ... }+ )
      ... "END" }+ ;

Token → strng OF StringExp | keyword OF Keyword | regexp OF RegExpression;

StringExp → ⟨("(([^"\\])|(\\[.]))*")|('(([^'\\])|(\\[.]))*')⟩;
-- A string surrounded by quotation marks " or '

Keyword → ⟨[a-z][_a-zA-Z0-9]*⟩;
-- A string started from the small letter

RegExpression → PARSE("<",RegularExpression(),">");
-- Regular expressions are described in the special grammar

Name → ⟨[A-Z][_a-zA-Z0-9]*⟩;
-- A string started from the capital letter
```

Example 20 (Declaring priorities and associativity in priorities section). *Consider the following* JAMOOS *program which describes arithmetical expressions:*

```
GRAMMAR Expression()
PRIORITIES
LEFT "+" AddExpr.exp2
LEFT MultExpr
RIGHT num
END -- Priorities Section
    Expression → AddExpr | MultExpr | Number;
```

```
    AddExpr → exp1:Expression "+" exp2:Expression;
    MultExpr → exp1:Expression "*" exp2:Expression;
    Number → num:⟨[1-9][0-9]*⟩;
END -- Program
```

From the declaration of the priorities section in this program it follows that:

1. *Priorities of the token* "+" *and the component* **exp2***, which appears in the production* AddExpr*, are equal.*
2. *The priority of the token* "+" *is lower than the one of the variable* MultExpr*.*
3. *The priority of the component* **exp2***, which appears in the production* AddExpr*, is lower than the one of the variable* MultExpr*.*
4. *The priority of the variable* MultExpr *is lower than the one of the component* **num** *which appears in the production* Number*.*
5. *Both token* "+" *and variable* MultExpr *are left-associative.*

5.2 Local Priority Assignment

In JAMOOS the alternation operator (|) is used in two cases:

1. Declaration of classes that inherit from another class. For example, consider the following production:

   ```
   Expression → MultExpr | AddExpr;
   ```

   This production defines three classes: Expression, MultExpr and AddExpr. Both classes MultExpr and AddExpr inherit from the class Expression.

2. Specifying choices in an alternative section. For example:

   ```
   Expression → mult OF MultExpr | add OF AddExpr;
   ```

   This production does not define an inheritance. It defines a class Expression with only one field which has a variable content. Depending on input the content can be either MultExpr or AddExpr.

In both cases a priority order can be defined among the choices. This can be done by using the double vertical bar separator (||). The bar divides the choices into groups. The priority of all choices in a group x is higher than the one of choices in a group to the right of x. Within a group all choices have equal priority.

Example 21 (Local declaration of priorities). *Consider the following productions:*

```
Expression → MultExpr | DivExpr || AddExpr | SubExpr || AssignExpr;      (1)
Operator → mult OF "*" | div OF "/" || add OF "+" | sub OF "-" || assign OF "=";   (2)
```

In both production priorities are assigned locally. In the production (1) *priorities are assigned to the non-terminals, whereas in the production* (2) *priorities are assigned to the choices of the alternative section.*

According to the priority assignment in the production (1)*:*

1. *Non-terminals* MultExpr *and* DivExpr *have equal priority.*
2. *Non-terminals* AddExpr *and* SubExpr *have equal priority.*
3. *The priority of non-terminals* MultExpr *and* DivExpr *is higher than the one of non-terminals* AddExpr *and* SubExpr*.*
4. *The priority of non-terminals* AddExpr *and* SubExpr *is higher than the one of the non-terminal* AssignExpr*.*

According to the priority assignment in the production (2)*:*

1. *The choices* **mult** *and* **div** *have equal priorities.*
2. *The choices* **add** *and* **sub** *have equal priorities.*
3. *The priority of the choices* **mult** *and* **div** *is higher than the one of the choices* **add** *and* **sub***.*
4. *The priority of the choices* **add** *and* **sub** *is higher than the one of the choice* **assign***.*

5.3 Problems with Priorities in JAMOOS

5.3.1 Assigning Priority and Associativity to a Production

In JAMOOS there are two methods for assigning priority and associativity to a production:

1. Explicitly, by assigning a priority and associativity to a nonterminal symbol which represents the left-hand side of the production.
2. Implicitly, basing on the priority and associativity of a rightmost terminal symbol in the right-hand side of the production.

Example 22 (Explicit assignment of priorities and associativity to a production). *Consider the following grammar for arithmetical expressions:*

Expression → AddExpr | MultExpr | Number; (1)
AddExpr → Expression "+" Expression; (2)
MultExpr → Expression "*" Expression; (3)
Number → ⟨[0-9]*⟩; (4)

Assume that we want to declare the production (2) *as left-associative and to assign it a priority that is lower than the priority of the production* (3)*.*

Such assignment can be made explicitly as follows:

```
PRIORITIES
    LEFT AddExpr
    LEFT MultExpr
END
```

The nonterminal AddExpr *is declared before the nonterminal* MultExpr*, therefore the production* (2) *has a lower priority than the production* (3)*.*

Example 23 (Implicit assignment of priorities and associativity to a production). *Consider the grammar from the Example 22 and consider that the following declaration takes place:*

```
PRIORITIES
    LEFT "+"
    LEFT MultExpr
END
```

Here the production (2) *derives its priority and associativity from its rightmost token " +". The production* (2) *has lower priority than the production* (3)*, because the token " +" was declared earlier than the nonterminal* MultExpr *in the* **PRIORITIES** *section.*

5.3.2 Assigning Priority and Associativity to a Token

In JAMOOS there are two methods for assigning priority and associativity to a token:

1. Explicit declaration.
2. Implicitly, basing on the priority and associativity of a nonterminal symbol which represents the left-hand side of a production that contains the token.

In the Example 23 token " +" is explicitly declared as left-associative and its priority is lower than the priority of the production (3).

In the Example 22 token " *" derives its priority and associativity from the production:

MultExpr → Expression "*" Expression;

5.3.3 Statement of Context-Dependent Priority Problem

Consider the grammar from the Example 22. The token " +" appears only in the production:

AddExpr → Expression "+" Expression;

therefore by assigning a priority to the production we unequivocally assign this priority to the token.

A problem arises when there is more than one production that contains a token. In this case a parser has to determine the production from which the token derives its priority at a given stage of the parsing process. We call this problem *Context-Dependent Priority Problem.*

Example 24 (Context-Dependent Priority Problem). *Consider the following grammar for arithmetical expressions.*

```
Expression → AddExpr | MultExpr | IncExpr | Number;   (1)
AddExpr → Expression "+" Expression;                  (2)
MultExpr → Expression "*" Expression;                 (3)
IncExpr → Expression "+" "+";                         (4)
Number → ⟨[1-9][0-9]*⟩;                               (5)
```

The Shift/Reduce conflict occurs between reduction by the production (3) *and shift of the token* +. *The conflict can be resolved by assigning priorities to the production* (3) *and to the token* +. *Assume that the priorities were assigned as follows:*

```
PRIORITIES
    LEFT AddExpr
    LEFT MultExpr
    LEFT IncExpr
END
```

Depending on the input, the token + *may derive its priority from the production* (2) *or from the production* (4). *In the first case the conflict is resolved by choosing reduce by* MultExpr. *In the second case the conflict is resolved by choosing shift action.*

For example, consider the expression $2 * 3 + +$. *When the substring* $2 * 3$ *has already been parsed and the token* + *appears next in the input, the parser has to decide which action to perform. The decision depends on the priority of the token* +. *The parser has to recognize that in the current situation the token derives its priority from the production* IncExpr, *therefore its priority is higher than the priority of* MultExpr, *and to perform the shift action.*

Consider another input: $2*3+5$. *In this case the parser must recognize that the token* + *derives its priority from the production* AddExpr. *Since the priority of this production is lower than the one of the production* MultExpr, *the conflict should be resolved by choosing reduce.*

The solution of the Context-Dependent Priority Problem is described in Section 5.4.

5.3.4 Statement of Ambiguous Assignment Problem

The *Ambiguous Assignment Problem* occurs when different methods are combined while assigning priority/associativity to a production or a token.

Example 25 (Ambiguous Assignment Problem). *Consider the following* JAMOOS *code:*

```
PRIORITIES
    RIGHT "+"
    LEFT AddExpr
END
 ...
Expression → AddExpr | MultExpr | Number;
AddExpr → Expression "+" Expression;
 ...
```

Here the token " + *" was explicitly declared as right-associative. Implicitly the token derives its associativity from the production:*

```
AddExpr → Expression "+" Expression;
```

and the production was declared as left-associative.
Also the priority of the token is not well defined.

The solution of the Ambiguous Assignment Problem is described in Section 5.5.

5.4 Solution for Context-Dependent Priority Problem

Assume that the parser is currently at a state s and a Shift/Reduce conflict occurs in this state: the reduce by the production P may be performed and, at the same time, it is legitimate to shift the current input symbol t. The parser state s contains a set of LR-items. The token t may derive its priority only from productions which yielded LR-items with the dot directly before t.

Example 26. *Consider the following grammar G_{Expr} for arithmetical expressions:*

Expression → AddExpr | AssignExpr | CompoundAssign | Var; (1)
AddExpr → Expression "+" Expression; (2)
AssignExpr → Expression "=" Expression; (3)
CompoundAssign → Expression "+""=" Expression; (4)
Var → ⟨[a-z]+⟩; (5)

The grammar G_{Expr} contains several Shift/Reduce conflicts. One of them is between the token = *and the production* (2)*. The corresponding conflict state looks like follows:*

AddExpr → Expression "+" Expression; • (a)
AddExpr → Expression • "+" Expression; (b)
AssignExpr → Expression • "=" Expression; (c)
CompoundAssign → Expression • "+""=" Expression; (d)

The token = *appears in both* (c) *and* (d) *LR-items. However, the token derives its priority from the production* (3)*, because the only LR-item with the dot directly before* = *is* (c) *and the item was yielded by the production* (3)*.*

Let us define $P_t(s)$ as the set of all productions which yielded LR-items in s with the dot directly before t. Assume that priorities are defined for all productions in $P_t(s)$. Otherwise implicit priority and associativity assignment to t is impossible, the conflict will be resolved by default.

Having $P_t(s)$ the conflict can be easily resolved if priority of each production in $P_t(s)$ is higher than the priority of P. In this case it is not important to exactly determine the production from which t derives its priority. The priority of t will be higher than the priority of P, so, the conflict should be resolved by choosing the shift. Accordingly, if priority of each production in $P_t(s)$ is lower than the priority of P, the conflict should be resolved by choosing the reduce. If each production in $P_t(s)$ has a priority that equals to the priority of P, the associativity should be used to resolve the conflict.

If $P_t(s)$ contains productions with priority that is higher than the priority of P as well as productions with lower or equal priority, it is necessary to exactly determine the production from which t derives its priority. Such a production cannot be determined at compile-time. The selection of the production depends on the given input: the fact that in s several LR-items with dot before t exist implies that parsing process can go over different routes depending on the input. JAMOOS uses *Runtime Automatic Priority and Associativity Inference (Runtime APAI)* to find a production from which t derives its priority and associativity.

Runtime APAI is actually performed in two steps:

1. **Compile-time.** Productions in $P_t(s)$ are divided into two groups P_{Shift} and P_{Reduce}. If priority of a production P', $P' \in P_t(s)$, is higher than priority of P, then P' is putted to P_{Shift}. If priority of a production P', $P' \in P_t(s)$, is lower than priority of P, then P' is putted to P_{Reduce}. If both productions P', $P' \in P_t(s)$, and P have equal priorities associativity is used. If P' is left-associative it is putted to P_{Reduce}, otherwise P' is putted to P_{Shift}.

The formal description of the algorithm for division of $P_t(s)$ into groups P_{Shift} and P_{Shift} is presented in Algorithm 2.

Having two production groups P_{Shift} and P_{Reduce} JAMOOS forms two sets of tokens T_{Shift} and T_{Reduce}. The set T_{Shift} contains tokens that may be derived immediately after t by productions from P_{Shift}. The set T_{Reduce} contains tokens that may be derived immediately after t by productions from P_{Reduce}.

Definition 27 (*FOLLOW(A,t)*). *Let A be a nonterminal symbol, t be a terminal symbol. Assume that there exists production $A \rightarrow \alpha$ such, that α contains t.*

FOLLOW(A,t) *is the set of terminal symbols that can appear immediately to the right of t.*

It follows that:

$$T_{Shift} = \bigcup_{p_i \in P_{Shift}} FOLLOW(p_i, t) \tag{1}$$

$$T_{Reduce} = \bigcup_{p_i \in P_{Reduce}} FOLLOW(p_i, t) \tag{2}$$

The algorithm for computation the *FOLLOW(A,t)* set uses *FIRST* and *FOLLOW* functions described in Aho et al. [9]. To compute the *FOLLOW(A,t)* set, the following steps should be applied until no more terminals can be added to the set:

(a) If t' is a terminal symbol and $A \rightarrow \alpha t t' \beta$ is a production, then add t' to *FOLLOW(A,t)*.

(b) If B is a nonterminal symbol and $A \rightarrow \alpha t B \beta$ is a production, then add *FIRST(B)* to *FOLLOW(A,t)*.

(c) If B is a nonterminal symbol, B is nullable and $A \rightarrow \alpha t B \beta$ is a production, then change production to $A \rightarrow \alpha t \beta$ and continue the algorithm.

(d) If a production is $A \rightarrow \alpha t$, i.e., t is the last symbol in the production, then add *FOLLOW(A)* to *FOLLOW(A,t)*.

The formal description of the algorithm for computing *FOLLOW(A,t)* set is presented in Algorithm 3.

The same token t' may appear in both T_{Shift} and T_{Reduce} sets, which means that one-symbol lookahead is not enough to exactly determine the applied production. In this case the preference is given to productions from the P_{Shift} group. The token t', when it will appear next to t in the input, will be considered as indication that a production from P_{Shift} is applied and the conflict will be resolved in favor of shift.

The formal description of the algorithm for calculating the T_{Shift} and T_{Reduce} token sets is presented in Algorithm 4.

When the sets T_{Shift} and T_{Reduce} are formed, a parser can be created. The parser stores the Shift/Reduce conflict and both token sets which are associated with the conflict.

2. **Runtime.** When the Shift/Reduce conflict occurs, the parser performs one-symbol lookahead. Assume that a token t' appears next to t in the input. If $t' \in T_{Shift}$ the token t derives its priority and associativity from a production in P_{Shift}, which means that the conflict should be resolved in favor of shift. If $t' \in T_{Reduce}$ the token t derives its priority and associativity from a production in P_{Reduce}, which means that the conflict should be resolved in favor of reduce.

Algorithm 2 Given a Shift/Reduce conflict C between a production P and a token t. Implicit priority and associativity assignment is required for resolution of C. The token t may derive priority and associativity from several productions which form group $\mathcal{P}$. Depending on a production from which t derives priority and associativity C can be resolved differently. This function divides $\mathcal{P}$ into groups P_{Shift} and P_{Reduce}. If t derives priority and associativity from a production from P_{Shift} group, C is resolved in favor of Shift. If t derives priority and associativity from a production from P_{Reduce} group, C is resolved in favor of Reduce.

Input: $\mathcal{P}$ – set of productions from which the token t may derive priority and associativity. Each production in the set has priority π and associativity ρ;
π_P – priority of a production P
Output: P_{Shift} and P_{Reduce}
Function FormShiftReduceProductionGroups

$n \leftarrow$ size of $\mathcal{P}$
for $p_i \in \mathcal{P}, i = 1, \ldots, n$ **do**
  if $p_i.\pi > \pi_P$ **then**
    Add p_i to P_{Shift}
  end if
  if $p_i.\pi < \pi_P$ **then**
    Add p_i to P_{Reduce}
  end if
  if $p_i.\pi == \pi_P$ **then**
    if $p_i.\rho$ is $LEFT$ **then**
      Add p_i to P_{Reduce}
    else
      Add p_i to P_{Shift}
    end if
  end if
end for
return P_{Shift} and P_{Reduce}

5.5 Solution for Ambiguous Assignment Problem

JAMOOS does not assure that priority/associativity which was assigned to a token or nonterminal explicitly is equal to the priority/associativity which was derived implicitly. The user is expected to bear the responsibility for unambiguous assignment of priority/associativity.

Assume that Ambiguous Assignment Problem arises for a token/nonterminal x. If x is involved in a conflict, JAMOOS will use for the conflict resolution the priority/associativity that was assigned to x explicitly.

Example 28. *Consider the following* JAMOOS *code:*

```
PRIORITIES
    LEFT AddExpr
    LEFT "+"
END
Expression → AddExpr | MultExpr | Number;    (1)
AddExpr → Expression "+" Expression;         (2)
MultExpr → Expression "*" Expression;        (3)
Number → ⟨[0-9]*⟩;                           (4)
```

The Shift/Reduce conflict occurs between the token "+" and the production (2)*. To resolve the conflict* JAMOOS *will compare priorities of the token and the production.*

Here it is desirable to implicitly assign the priority to the production. In this case priorities of the token and the production would be equal. It means that associativity would be used for the conflict resolution and since "+" is left-associative, the reduce would be chosen.

Algorithm 3 The recursive function calculates *FOLLOW(A,t)* set. The function uses *FIRST* and *FOLLOW* functions described in Aho et al. [9].

Input: $A \rightarrow \alpha$ – a production;
t – a token which is contained in the right-hand side of the production $A \rightarrow \alpha$
Output: Σ_t – set of tokens that may be derived by the production A immediately after t
Function Follow
1: Split α into three parts so that $\alpha \equiv \alpha_1 t \alpha_2$
2: **if** α_2 has the form $t'\beta$, where t' is a token **then**
3: Add t' to Σ_t
4: **end if**
5: **if** α_2 has the form $B\beta$, where B is a non-terminal symbol **then**
6: Add $FIRST(B)$ to Σ_t
7: **if** B is nullable **then**
8: $\Sigma' \leftarrow$ **Follow**($A \rightarrow \alpha_1 t \beta$, t)
9: Add Σ' to Σ_t
10: **end if**
11: **end if**
12: **if** α_2 is empty **then**
13: *// t is the last symbol in the right-hand side of A*
14: Add $FOLLOW(A)$ to Σ_t
15: **end if**
16: **return** Σ_t

Algorithm 4 Given a Shift/Reduce conflict C between a production P and a token t. Implicit priority and associativity assignment to t is required for resolution of C. The token t may derive priority and associativity from several productions. These productions are divided into two groups P_{Shift} and P_{Reduce}. If t derived priority and associativity from a production which belong to P_{Shift}, C should be resolved in favor of shift. If t derived priority and associativity from a production which belong to P_{Reduce}, C should be resolved in favor of reduce. In order to determine the actual production from which t must derive priority and associativity a parser performs one-symbol lookahead at parsing time. This function calculates two groups of tokens T_{Shift} and T_{Reduce} that may appear after t in the input. A token from the group T_{Shift} that appears after t indicates that t must derive priority and associativity from a production from P_{Shift} group. A token from the group T_{Reduce} that appears after t indicates that t must derive priority and associativity from a production from P_{Reduce} group. This function calls **Follow** function for calculating the set of tokens that may appear after t. The **Follow** function is described in Algorithm 3.

Input: t – a token;
P_{Shift} – a set of productions. If t derives priority and associativity from a production from this set, the conflict is resolved in favor of shift;
P_{Reduce} – a set of productions. If t derives priority and associativity from a production from this set, the conflict is resolved in favor of reduce

Output: T_{Shift} – set of tokens. If a token from this set appears in the input next to t, the conflict should be resolved in favor of shift;
T_{Reduce} – set of tokens. If a token from this set appears in the input next to t, the conflict should be resolved in favor of reduce

Function FormTokenSets

1: $n \leftarrow$ size of P_{Shift}
2: **for** $p_i \in P_{Shift}$, $i = 1, \ldots, n$ **do**
3: $\Sigma_t \leftarrow$ **Follow**(p_i, t)
4: Add Σ_t to T_{Shift}
5: **end for**
6: $n \leftarrow$ size of P_{Reduce}
7: **for** $p_i \in P_{Reduce}$, $i = 1, \ldots, n$ **do**
8: $\Sigma_t \leftarrow$ **Follow**(p_i, t)
9: Add Σ_t to T_{Reduce}
10: **end for**
11: *// Tokens that appear both in T_{Shift} and T_{Reduce} groups are deleted from T_{Reduce} group*
12: $T_{Reduce} \leftarrow T_{Reduce} \setminus (T_{Shift} \cap T_{Reduce})$
13: **return** T_{Shift} and T_{Reduce}

However, Ambiguous Assignment Problem arises for the nonterminal AddExpr*, therefore explicit assignments will be used. Basing on explicit assignments, the priority of the token is higher than the priority of the production. The conflict will be resolved by choosing shift.*

6 Algorithms Developed for Conflict Resolution

6.1 Resolution of Reduce/Reduce Conflicts

Consider a Reduce/Reduce conflict C that occurs between n productions P_1, P_2, ..., P_n. This conflict will be resolved in favor of the production P_i, $i = 1, \ldots, n$ with the highest priority.

If more than one conflicting production have the highest priority, default resolution is performed. By default, C is resolved in favor of a production whose left-hand side variable is the first in alphabetical order.

If priority is not assigned to a production P_i, $i = 1, \ldots, n$ explicitly in the priorities section, JAMOOS attempts to assign the priority implicitly basing on the priority of the rightmost token in the right-hand side of P_i (see Section 5.3.1). If JAMOOS also fails to assign the priority implicitly, it resolves C by default.

The formal description of the Reduce/Reduce conflict resolution algorithm is presented in Algorithm 5.

6.2 Resolution of Shift/Reduce Conflicts

Consider a Shift/Reduce conflict C between a token t and a production P. Actually a Shift/Reduce conflict may occur between a token t and a set of productions $\mathcal{P}$. This kind of conflict can be reduced to C by resolving the Reduce/Reduce conflict between the productions in $\mathcal{P}$.

Assume that P has priority π_P and t has priority π_t. If π_t is higher than π_P the shift action should be performed. If π_t is lower than π_P the reduce action should be performed.

If π_t is equal to π_P then associativity should be used[10]. If t is left associative the reduce action should be performed. If t is right associative the shift action should be performed.

If JAMOOS fails to determine π_P or π_t, default resolution is performed. By default C is resolved by choosing shift action.

If priority and associativity are not assigned to a token t explicitly in the priorities section, JAMOOS attempts to assign them implicitly, basing on the priority and associativity of a production which contains t (see Section 5.3.2).

While attempting to assign priority and associativity to t implicitly, the Context-Dependent Priority problem may occur (see Section 5.3.3). The resolution of this problem is described in the Section 5.4.

If priority is not assigned to a production P explicitly in the priorities section, JAMOOS attempts to assign the priority implicitly basing on the priority of the rightmost token in the right-hand side of P (see Algorithm 6).

Formal description of Shift/Reduce conflict resolution algorithm is presented in Algorithm 9. The algorithm for implicit priority assignment to token is presented in Algorithm 10. Formal description of Runtime APAI is presented in Section 5.4.

[10]The associativity of both t and P can be used to resolve C. Since both P and t have equal priorities, they should be declared in the same group in the priorities section and have equal associativity.

Algorithm 5 The function that resolves a Reduce/Reduce conflict between productions P_1, P_2, ..., P_n. The function calls the **AssignPriorityToProduction** function when it is required to assign a priority to a production implicitly. This function is described in Algorithm 6.

Input: P_1, P_2, ..., P_n – conflicting productions sorted in an alphabetical order by variables on their left-hand sides;
PT – priority and associativity table that contains all variables and tokens which appear in the priorities section, together with their priorities and associativity values

Output: $P_i, i = 1, \ldots, n$ – production in favor of which the conflict is resolved

Function ResolveReduceReduceConflict

1: **for all** $P_i, i = 1, \ldots, n$ **do**
2: **if** variable on left-hand side of P_i appears in PT **then**
3: $\pi_i \leftarrow$ priority of P_i
4: **else**
5: *// Attempting to assign a priority to P_i implicitly.*
6: *// In case of success the priority of P_i is set to π_i and the function returns **true**.*
7: *// In case of failure the function returns **false**.*
8: $res \leftarrow$ **AssignPriorityToProduction**(P_i, PT, π_i)
9: **if** $res == false$ **then**
10: **return** P_1 *// Default resolution*
11: **end if**
12: **end if**
13: **end for**
14: $z \leftarrow max(\pi_1, \pi_2, \ldots, \pi_n)$
15: *// Initiate the variable that will contain the number of a production which has the highest priority*
16: $k \leftarrow 0$
17: **for all** $P_i, i = 1, \ldots, n$ **do**
18: **if** $\pi_i == z$ **then**
19: **if** $k == 0$ **then**
20: $k \leftarrow i$
21: **else**
22: **return** P_1 *// More than one production has the highest priority. Apply default resolution.*
23: **end if**
24: **end if**
25: **end for**
26: **return** P_k

Algorithm 6 The function infers priority of a production basing on the priority of the rightmost token in its right-hand side. The function fails to infer priority of the production if it does not contain any token in the right-hand side or if the priority of its rightmost token is not defined in priorities section. In order to find the rightmost token the function calls auxiliary function **LastToken**. This function is described in Algorithm 7.

Input: P – production for which priority should be assigned;
PT – priority and associativity table that contains all variables and tokens which appear in the priorities section together with their priorities and associativity values;
π – a variable which will contain the priority of P in case of successful priority assignment

Output: In case of successful priority assignment the function returns **true**. Otherwise it returns **false**.

Function AssignPriorityToProduction

1: **if** P is a Composition **then**
2: $n \leftarrow$ number of components in P
3: **let** c_i, $i = 1, \ldots, n$, be a component of P
4: **for** $i = n, i - -$ **do**
5: *// Attempting to find a rightmost token t in the component c_i.*
6: *// In case of success the function returns **true**. In case of failure the function returns **false**.*
7: $res \leftarrow$ **LastToken**(c_i, t)
8: **if** $res == true$ **then**
9: **if** t appears in PT **then**
10: $\pi \leftarrow$ priority of t
11: **return true**
12: **else**
13: **return false**
14: **end if**
15: **end if**
16: **end for**
17: **end if**
18: **return false**

Algorithm 7 The function searches the rightmost token in a production component. The function fails to find such a token if the component does not contain any token. The function calls an auxiliary function **ListLastToken** (see Algorithm 8) in order to find the rightmost token in a List component.

Input: c – component whose rightmost token should be found;
t – a variable which will contain the rightmost token of C in case of successful function completion.
Output: The function returns **true** if the rightmost token of c was successfully found. Otherwise the function returns **false**.

Function LastToken

1: **if** c is Token **then**
2: $t \leftarrow c$
3: **return true**
4: **end if**
5: **if** c is Variable **then**
6: **return false**
7: **end if**
8: **if** c is Sequence **or** c is Optional **then**
9: $n \leftarrow$ number of components in c
10: **let** c_i, $i = 1, \ldots, n$, be a component of c
11: **for** $i = n, i - -$ **do**
12: $res \leftarrow$ **LastToken**(c_i, t)
13: **if** $res == true$ **then return true**
14: **end for**
15: **return false**
16: **end if**
17: **if** c is List **then**
18: $res \leftarrow$ **ListLastToken**(c, t)
19: **if** $res == true$ **then return true**
20: **return false**
21: **end if**
22: **if** c is an Alternative **then**
23: $n \leftarrow$ number of choices in c
24: **let** c_i, $i = 1, \ldots, n$, be a choice of c
25: **for** $i = n, i - -$ **do**
26: $res \leftarrow$ **LastToken**(c_i, t')
27: **if** $res == true$ **then return true**
28: **end for**
29: **return false**
30: **end if**

Algorithm 8 The function searches the rightmost token in a List component. The function fails to find such a token if the component does not contain any token.

Input: c_l – List component whose rightmost token should be found;
t – a variable which will contain the rightmost token of c_l in case of successful function completion.
Output: The function returns **true** if the rightmost token of c_l was successfully found. Otherwise the function returns **false**.

Function ListLastToken

1: **if** c_l has a terminal component **then**
2: $c \leftarrow$ the terminal component of c_l
3: $res \leftarrow$ **LastToken**(c, t)
4: **if** $res == true$ **then**
5: **return true**
6: **end if**
7: **end if**
8: $c \leftarrow$ the repeated item of c_l
9: $res \leftarrow$ **LastToken**(c, t)
10: **if** $res == true$ **then**
11: **return true**
12: **end if**
13: **if** c_l has a separator component **then**
14: $c \leftarrow$ the separator component of c_l
15: $res \leftarrow$ **LastToken**(c, t)
16: **if** $res == true$ **then**
17: **return true**
18: **end if**
19: **end if**
20: **if** c_l has a initial component **then**
21: $c \leftarrow$ the initial component of c_l
22: $res \leftarrow$ **LastToken**(c, t)
23: **if** $res == true$ **then**
24: **return true**
25: **end if**
26: **end if**
27: **return false**

Algorithm 9 The function resolves Shift/Reduce conflict C between the token t and the production P. When it is required to assign priority and associativity to t implicitly, auxiliary function **AssignPriorityToToken** (see Algorithm 10) is called. When it is required to assign priority and associativity to P implicitly, auxiliary function **AssignPriorityToProduction** (see Algorithm 6) is called.

Input: P – production by which reduction can be performed;
t – token that can be shifted;
PT – priority and associativity table that contains all variables and tokens which appear in the priorities section together with their priorities and associativity values;
S_C – set of LR-items from the parser state in which C occurs

Output: The function may return three values: **Shift**, **Reduce** and **VariableAction**. If the function returns **Shift**, C is resolved by shifting t. If the function returns **Reduce**, C is resolved in favor of reduction by P. If the function returns **VariableAction**, C is resolved by performing Runtime APAI.

Function ResolveShiftReduceConflict

if left-hand variable of P appears in PT **then**
  $\pi_P \leftarrow$ priority of P
else
  // Attempting to assign priority to P implicitly.
  // In case of successful assignment priority of P will be set to π_P.
  $res \leftarrow$ **AssignPriorityToProduction**(P, PT, π_P)
  if $res == false$ **then return Shift** *// Default resolution*
end if
if t appears in PT **then**
  $\pi_t \leftarrow$ priority of t
else
  // Attempting to assign priority and associativity to t implicitly. In case of successful
  // assignment priority of t will be set to π_t and associativity of t will be set to ρ.
  $res \leftarrow$ **AssignPriorityToToken**(t, PT, S_C, π_P, π_t, ρ)
  if $res == lookahead$ **then return VariableAction** *// Runtime APAI should be performed*
  if $res == false$ **then return Shift** *// Default resolution*
end if
if $\pi_P < \pi_t$ **then return Shift**
if $\pi_P > \pi_t$ **then return Reduce**
if $\pi_P == \pi_t$ **then**
  if ρ is $LEFT$ **then return Reduce**
  if ρ is $RIGHT$ **then return Shift**
end if

Algorithm 10 The function infers priority and associativity of a token t basing on the priorities and associativity of one or more productions in which t appears. The token t may derive priority and associativity only from a production involved in a Shift/Reduce conflict C which occurs between t itself and a production P.

Input: t – token to which priority and associativity should be assigned;
PT – priority and associativity table;
S_C – set of LR-items from the parser state in which C occurs;
π_P – priority of the production P;
π_t – variable that will contain the priority of t;
ρ – variable that will contain the associativity of t

Output: The function may return three values: **true**, **false** and **lookahead**. The function returns **true** in case of successful priority and associativity assignment. The function returns **false**, if priority of one or more productions from which t may derive its priority is not defined in a priorities section. The function returns **lookahead**, if Runtime APAI should be used.

Function AssignPriorityToToken

let R_C be a set of LR-items
$R_C \leftarrow$ LR-items from S_C which have dot before t
$n \leftarrow$ size of R_C
// Attempting to find priorities and associativity of all productions involved in C
for $item_i \in R_C$, $i = 1, \ldots, n$ **do**
    $A \leftarrow$ left-hand variable of $item_i$
    if A appears in PT **then**
        $\pi_i \leftarrow$ priority of A, $\rho_i \leftarrow$ associativity of A
    else
        return false
    end if
end for
$s \leftarrow true$, $r \leftarrow true$
for i, $i = 1, \ldots, n$ **do**
    if $\pi_i < \pi_P$ **or** ($\pi_i == \pi_P$ **and** ρ_i is $LEFT$) **then** $s \leftarrow false$
    if $\pi_i > \pi_P$ **or** ($\pi_i == \pi_P$ **and** ρ_i is $RIGHT$)) **then** $r \leftarrow false$
end for
if $s == false$ **and** $r == false$ **then**
    return lookahead *// t may derive priority higher than π_P as well as priority lower than π_P*
else
    $\pi_t \leftarrow \pi_1$, $\rho \leftarrow \rho_1$
    return true
end if

6.3 Resolution of Pop/Reduce Conflicts

Consider a Pop/Reduce conflict C between repeated element e_l of a list l and the list itself. While reducing l a parser must decide whether a symbol x on the top of the stack is an element of l or not. In the first case the parser pops x and considers it as e_l. In the second case the parser stops reduction of l and considers x as a part of some other component (see detailed description of the conflict in Section 4.3).

Since both e_l and l are components of JAMOOS language, priorities can be assigned to them. Assume that e_l has priority π_{elem} and l has priority π_l. If π_{elem} is higher than π_l, the parser will continue popping symbols from the stack as elements of l. If π_{elem} is lower than π_l, the parser will finish reduction of l.

Example 29. *Consider the grammar G_A that describes strings which consist of the symbol a only:*

```
GRAMMAR A()
    A → ["a"] aList:{ listElem:"a" ... } ;
END
```

A Pop/Reduce conflict occurs in the grammar above. While parsing a sequence of symbols a, the parser has to decide whether the first symbol in the sequence is a part of the list or an element of the optional section.

The conflict can be resolved by assigning priorities to both the list and its repeated item. If it is required always to consider the first symbol a as a part of the list, higher priority should be assigned to the repeated item of the list. In this case the priorities section will look like follows:

```
PRIORITIES
    LEFT aList
    LEFT listElem
END
```

The parse tree for the input aa is shown in Figure 12. The parse tree is typical for this conflict resolution: optional section always is considered empty, all symbols a are considered as elements of the list.

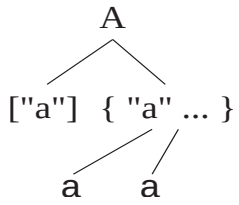

Figure 12: Parse tree for the input *aa* described by the grammar G_A. A Pop/Reduce conflict which occurred in G_A while parsing the input was resolved by choosing to pop the first symbol *a* as the list element.

If the first symbol a always should be considered as an element of the optional section, higher priority should be assigned to the list. In this case the priorities section will look like follows:

```
PRIORITIES
```

```
    LEFT listElem
    LEFT aList
END
```

The parse tree for the input aa is shown in Figure 13. The parse tree is typical for this conflict resolution: the first symbol a is always an element of the optional section.

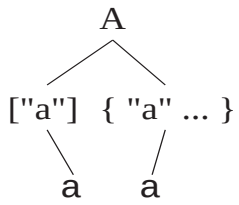

Figure 13: Parse tree for the input *aa* described by the grammar G_A. A Pop/Reduce conflict which occurred in G_A while parsing the input was resolved by choosing to pop the first symbol *a* as the element of the optional section.

If priorities of a list or of its element are not defined explicitly in the priorities section, default resolution is performed. By default, the parser will continue popping symbols from the stack as elements of the list.

Formal description of the Pop/Reduce conflict resolution algorithm is presented in Algorithm 11.

6.4 Resolution of Pop/Pop Conflicts

Consider a Pop/Pop conflict C that occurs between two choices ch_1 and ch_2 in an alternative section a. While reducing a the parser can pop both ch_1 and ch_2, because the stack content matches both choices. In this case the parser must decide which choice to pop (see detailed description of this conflict in Section 4.4).

The conflict C can be resolved by assigning priorities to ch_1 and ch_2. Priorities can be assigned to choices locally (see Section 5.2). The conflict is resolved in favor of the choice with the highest priority.

If both ch_1 and ch_2 have equal priority C default resolution is performed. By default C is resolved in favor of an alternative which appears first in a.

Example 30. *Consider the following grammar $G_{Friends}$:*

```
GRAMMAR Friends()
    Friends → boys:(john OF "John" | boys OF ("John" and "Alex")) and Girls;
    Girls → jane OF "Jane" | girls OF ("Alex" and "Jane");
END
```

A Pop/Pop conflict occurs in the grammar above. While parsing the following input

John and Alex and Jane

it is legitimate to pop both choices of the production Girls*. If the choice* **girls** *is popped while reducing by the production* Girls*, then the choice* **john** *will be popped when reduction by the*

Algorithm 11 The function resolves Pop/Reduce conflict.

Input: $list$ – a name of the list component;
$elem$ – a name of a component which is the repeated item of the $list$;
PT – priority table that contains all component names which appear in the priorities section together with their priority values

Output: The function may return two values **pop** and **reduce**. The function returns **pop**, if the conflict is resolved in favor of popping: a symbol on the top of the stack will be popped as the repeated item of the list. The function returns **reduce**, if the conflict is resolved in favor of reduction: a symbol on the top of the stack will not be considered as the repeated item of the list, and the reduction of the list will be completed.

Function ResolvePopReduceConflict

1: **if** $list$ appears in PT **then**
2: $\pi_{list} \leftarrow$ priority of $list$
3: **else**
4: **return pop** *// Default resolution*
5: **end if**
6: **if** $elem$ appears in PT **then**
7: $\pi_{elem} \leftarrow$ priority of $elem$
8: **else**
9: **return pop** *// Default resolution*
10: **end if**
11: **if** $\pi_{elem} > \pi_{list}$ **then**
12: **return pop**
13: **else if** $\pi_{elem} < \pi_{list}$ **then**
14: **return reduce**
15: **else**
16: **return pop** *// Default resolution*
17: **end if**

production Friends *will be performed. If the choice* **jane** *is popped while reducing by the production* Girls*, then the choice* **boys** *will be popped when reduction by the production* Friends *will be performed.*

The conflict can be resolved by assigning priorities to the choices **jane** *and* **girls**. *If it is required that* Alex *always should be considered as a boy, the production* Girls *should be rewritten as follows:*

Girls → jane **OF** "Jane" || girls **OF** ("Alex" **and** "Jane");

The parse tree for the above input is shown in Figure 14.

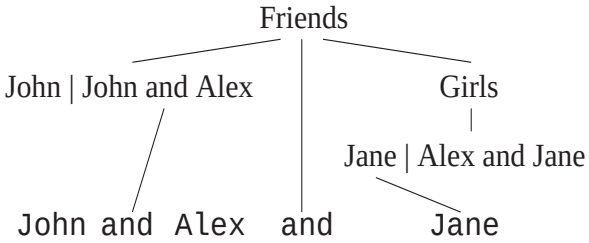

Figure 14: Parse tree for the input *John and Alex and Jane* described by the grammar $G_{Friends}$. A Pop/Pop conflict which occurred in $G_{Friends}$ while parsing the input, was resolved in favor of the choice **jane**.

If it is required that Alex *always should be considered as a girl, the production* Girls *should be rewritten as follows:*

Girls → girls **OF** ("Alex" **and** "Jane") || jane **OF** "Jane";

The parse tree which will be obtained in this case is shown in Figure 15.

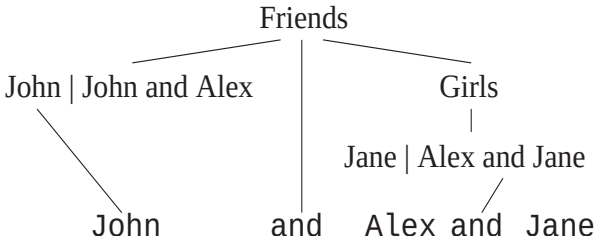

Figure 15: Parse tree for the input *John and Alex and Jane* described by the grammar $G_{Friends}$. A Pop/Pop conflict which occurred in $G_{Friends}$ while parsing the input was resolved in favor of the choice **girls**.

Formal description of the Pop/Pop conflict resolution algorithm is presented in Algorithm 12.

7 Implementation

In this section we will discuss details of conflict resolution implementation in the framework of JAMOOS project. The implementation stages included:

Algorithm 12 The function resolves Pop/Pop conflict.

Input: a – an alternative section in which Pop/Pop conflict occurred.
a has the form $\alpha_1\|\ldots\|\alpha_i\|\ldots\|\alpha_n$, $n \geq 1$.
α_i, $i = 1, \ldots, n$, has the form $\beta_{i1}|\ldots|\beta_{ij}|\ldots|\beta_{im_i}$, $m_i > 1$;
ch_1, ch_2 – conflicting choices of a sorted in order of their appearance in a
Output: ch – a choice in favor of which the conflict is resolved

Function ResolvePopPopConflict

1: **for** i, $i = 1, \ldots, n$ **do**
2: **for** j, $j = 1, \ldots, m_i$ **do**
3: **if** $\beta_{ij} \equiv ch_1$ **then**
4: $\pi_1 \leftarrow i$
5: **end if**
6: **if** $\beta_{ij} \equiv ch_2$ **then**
7: $\pi_2 \leftarrow i$
8: **end if**
9: **end for**
10: **end for**
11: **if** $\pi_1 > \pi_2$ **then**
12: **return** ch_1
13: **end if**
14: **if** $\pi_1 < \pi_2$ **then**
15: **return** ch_2
16: **end if**
17: **if** $\pi_1 == \pi_2$ **then**
18: **return** ch_1 *// Default resolution*
19: **end if**

- Adding syntactical support for priorities section definitions. Another bootstrapping step was performed in order to build the version of JAMOOS which would allow priorities section definitions.
- Implementation of Conflict Resolution module and its implantation into JAMOOS project
- Automatic generation of conflict reports

7.1 Adding Syntactical Support for Priorities Section Definitions

JAMOOS was developed gradually employing a bootstrapping technique: JAMOOS was used to define the grammar of the JAMOOS language itself and generate a parser for the grammar. BISON (the GNU version of YACC) was used to initiate the bootstrapping process. Then at each step new language features have been added to the grammar employing already existed ones [31].

By the time this work has been started syntax of JAMOOS has not allowed definition of a priorities section in JAMOOS programs. Double bar in alternations, which is necessary for the resolution of Pop/Pop conflicts (see Section 5.2), was supported by JAMOOS, though no semantic action was associated with it.

In order to obtain a parser which would accept JAMOOS programs that contain priorities section another step of bootstrapping was performed. Productions that describe priorities section were added to the existed JAMOOS grammar J. Then Jamoos was used to generate a set of classes and a parser for the extended JAMOOS grammar J'.

Since most of the productions from J have been transferred to J' without any changes, the classes generated for them by JAMOOS have not been changed either. Therefore the code that implemented semantic actions associated with these productions was almost entirely reused.

7.2 Conflict Resolution Implementation

By the time this work has been started already implemented parts of JAMOOS included: automatic translation of a grammar productions into C++ classes and generation of an object oriented parser for the grammar. The Parser Generator module has provided an information about conflicts that occur in the grammar. For each type of conflicts the following characteristics have been provided:

- **Shift/Reduce conflict**
  1. State in which the conflict has been occurred
  2. Number of a token by which shift action can be performed
  3. Number of a parser state to which the parser would move in case if the shift action was chosen
  4. List of numbers of productions by which reduce action can be performed
- **Reduce/Reduce conflict**
  1. State in which the conflict has been occurred
  2. List of numbers of productions by which reduce action can be performed
- **Pop/Reduce conflict**
  1. Production in which the conflict has been occurred

2. List in the production which contains the conflict

- **Pop/Pop conflict**

  1. Production in which the conflict has been occurred
  2. List of alternatives in the production among which the conflict has been occurred

The main tasks of the Conflict Resolution module designed and implemented in this work were as follows:

- Establish a connection with the Parser Generation module employing the interface described above and obtain the information about conflicts which occur in a grammar.
- Obtain information about user-provided priorities and associativity of tokens, variables and tags. Build priority and associativity tables.
- Resolve the conflicts basing on priorities and associativity defined by a user in priorities section of the grammar.

Several classes were designed to accomplish the tasks above. Here is a description of the most important of them.

PrioAssocTable This class describes priority and associativity table which contains priorities and associativity defined by a user in priorities section. The table consists of three independent tables: for tokens, variables and tags.

The tables are created on the basis of the information extracted from the class *PrioritySection* automatically generated by JAMOOS. The structure of the *PrioritySection* class reflects the structure of the corresponding production. Tokens, variables and tags are stored in nested lists in arbitrary order. Priority is defined by order of the lists: if the list l_1 appears before the list l_2, then tokens, variables and tags from l_2 have the higher priority than the ones from l_1. Associativity is defined in a special field associated with each list. The *PrioAssocTable* class extracts information about priority and associativity of tokens, variables and tags, and sort it into three corresponding tables.

ConflictResolver This class implements algorithms of conflict resolution described in Section 6.

PAT_Facade This class defines a higher-level interface that makes the priority and associativity tables easier to use.

Tokens, variables and tags are represented in the *PrioAssocTable* class as objects, while in the *ConflictResolver* class they are represented as numbers (token/variable/tag id). The *PAT_Facade* class provides to the *ConflictResolver* class convenient interface that allows the latter to use the priority tables defined in class *PrioritySection*.

7.3 Grammar Report Generation

Along with a set of classes and a parser JAMOOS generates for a grammar an HTML file which contains a *grammar report*. A *grammar report* for a grammar G is a document that provides different kinds of information about G, such as statistics about G, description of states of a parser generated for G, relationships between tokens, variables and productions of G etc.

The main purpose of a grammar report is to provide a handy way to follow the parsing process in order to find inaccuracies or errors in the grammar definition. The problem of tracing the parsing

process becomes even more crucial when conflicts occur. In order to understand how a conflict has occurred it is not enough to observe the conflict parser state only. Often a conflict between certain productions is actually caused by another productions which even may not appear in the conflict state. The Modifiers conflict that occurs in JAVA language is an example of such a conflict (see Section 8.1.2).

A proper conflict resolution requires information about all parsing paths which led to the conflict. A compact and easy-to-understand format of a parser representation is therefore of a great importance. In this section we will describe how a Documentation Generation module of JAMOOS was extended by conflict report generation methods. The improvements in the parser representation format will be discussed also.

7.3.1 Generation of Conflict Reports

A grammar report is generated by the Documentation Generation module of JAMOOS. This module was extended by methods which provide a detailed information about all conflicts that have occurred in the grammar.

For each Shift/Reduce or Reduce/Reduce conflict C the following information is provided:

- States in which C occurs.
- Tokens which are read by a parser by the time C occurs. In case of Shift/Reduce conflicts they are tokens by which the shift action can be performed.
- Productions by which the reduction might be performed.
- Resolution of C. There are three ways to resolve a conflict:
  - **Shift**
  - **Reduce** In this case a production by which the reduce action will be performed, is specified.
  - **Variable Action** Runtime APAI will be used to resolve C (see Section 5.4). In this case two lists of tokens that may appear in the input right after the current input token are provided. One list contains tokens which indicate that the shift action must be performed. Another one contains tokens which indicate that the reduce action must be performed.

For each Pop/Reduce conflict C the following information is provided:

- States in which C occurs.
- A list l that contains C.
- A production in which l appears.
- Resolution of C. There are two ways to resolve a conflict:
  - **Continue** Resolution in favor of the pop action. A parser will pop the current symbol from the stack as an element of l.
  - **Stop** Resolution in favor of the reduce action. A parser stops the reduction of l. The current symbol in the stack will be considered as a part of the component which will be reduced after l.

For each Pop/Pop conflict C the following information is provided:

- States in which C occurs.
- Conflicting alternatives.
- A DataType component a that contains the conflicting alternatives.
- A production in which a appears.
- Resolution of C. An alternative, which will be popped, is specified.

7.3.2 Parser Representation Format

A parser is represented as a set of parser states which in turn are represented as a set of LR-items. Since parser states often contain a huge number of LR-items, it is crucial to find the most compact and understandable representation of the items. Different fonts and colors were used for this purpose. Here is the list of improvements which were made in the representation of LR-items:

1. Reduce items are printed in *italic* font. The left-hand side variables of reduce items are printed in a bright green color, which facilitates locating them among the regular LR-items.
2. The LR-marker (dot) in LR-items and the sharp symbol (#) in reduce items are printed in a red color.
3. String tokens are printed in a green color.
4. Keyword tokens are printed in a blue color.
5. Regular expressions are printed in a light brown color.

Each parser state s is associated with a row in a parse table. The parse table is divided into Action and Goto tables.

For each token t in a grammar G there is a column in the Action table associated with it. The action that should be performed in s when t is the current input symbol appears in the corresponding cell of the Action table. If the cell contains more than one action then a conflict occurs.

For each variable A in G there is a column in the Goto table associated with it. Assume that when a string described by A is derived the parser is in s. When the reduction by A is performed the parser moves to the state s'. The number of s' appears in the cell of the Goto table which corresponds to s and A.

Entries to the Action and Goto tables play the key role in tracing a parsing process. In order to gain maximal understandability of the entries the following improvements in their representation were made:

1. Separate presentation of the entries to Action and Goto tables
2. Entries are presented as tables rather than lists. The Action table entry for s is represented as a table $Action_s$ with two columns. The first column contains tokens of G, while the second one contains actions corresponding to s and the tokens.[11] The Goto table entry for s is represented as a table $Goto_s$ with two columns. The first column contains variables of G, while the second one contains numbers of states to which the parser moves after reductions.

[11] If the same action should be performed on several tokens, the entries corresponding to these tokens are merged.

3. Cells that contain conflicts are represented by several rows in $Action_s$: each row corresponds to a different action of a conflict. However, only the action which will be performed at parsing time is printed in regular font. The actions which are not relevant are printed in *italic* font.

8 Conflict Resolution in Grammars of JAVA and JAMOOS

In this chapter we will show how conflict resolution techniques described in chapter 6 were used for the resolution of conflicts in grammars of JAVA language and JAMOOS.

As the base of our discussion on conflict resolution in the JAVA language, we choose a JAVA grammar composed by Gosling et al. [15, §16]. This grammar then went through a number of revisions:

- Syntax errors were corrected. For example, in certain cases the initial grammar described statements that missed a semicolon afterwards, which contradicts with the JAVA syntax.
- The grammar was extended with several necessary productions. For instance, the initial grammar lacked description of certain constructions of the JAVA standard, such as literals (numbers, characters, strings etc.).
- A few productions were replaced by the ones that exactly correspond to the JAVA standard. For example, a group of productions that specifies arithmetic and logic expressions in the initial grammar has no possibility to describe complex expressions without parenthesis, such that $2 + 5 * x$.
- The grammar was translated into the JAMOOS format, given in Section 3.

The full text of the JAVA grammar in JAMOOS notation can be found in Appendix A.

The resulting grammar contains only one Reduce/Reduce conflict and several Shift/Reduce conflicts. It contains no Pop conflicts.

Most of Shift/Reduce conflicts occurred in a grammar part which describes arithmetic expressions. The nature of these conflicts is trivial, no special discussion on it will be held in this work. The resolution of these conflicts, however, emphasizes one of important characteristics of our conflict resolution method. Our APAI mechanism empowered by the original interface for assigning priorities and associativity to productions allows to assign priority and associativity to abstract entities such as arithmetical operations. In contrast, YACC allows to assign priorities and associativity to operators only. For example, in YACC priorities and associativity are assigned to operators "=", "*", "++" while in our method priorities and associativity are assigned directly to arithmetical operations assignment, multiplication, postfix increment.

We will demonstrate the application of our Pop conflict resolution techniques on resolution of Pop conflicts which occur in a grammar of JAMOOS. For this purpose we have chosen the JAMOOS grammar composed by Tsoglin [31][12]. The grammar contains four Pop conflicts: one Pop/Reduce conflict and three Pop/Pop conflicts. We will show the resolutions of the Pop/Reduce conflict and one of the Pop/Pop conflicts[13].

[12] Tsoglin defined the complete JAMOOS grammar. Since JAMOOS is an ongoing project the current version of JAMOOS grammar that was considered in Section 3 is incomplete and lacks a lot of features. The syntax of the complete JAMOOS grammar differs from the current JAMOOS grammar syntax.

[13] The rest of the Pop/Pop conflicts have the similar resolution and therefore they are not of interest for the current discussion.

8.1 Shift/Reduce Conflicts

8.1.1 Array Creation Conflict

This conflict occurs while parsing JAVA statements in which a multidimensional array is created. Consider the following statement:

```
new int[4][i*5]
```

Two interpretations of this statement are possible. One of them considers the statement as a creation of a 2-dimensional array.

Another interpretation considers a sub-expression `new int[4]` as a creation of a 1-dimensional array. An array creation statement is considered as a kind of the simplest JAVA expressions, among other simplest expressions such as names of variables. These expressions are described by Primary production:

```
Primary →
    reference OF ({ Identifier "." ... }+ [ IdentifierSuffix ]) |
    instance_creation OF (new Creator);
Creator → Type ArrayCreatorRest;
```

The grammar allows the Primary nonterminal to have an array access operator afterwards.

```
LeftHandSide → Primary { Selector ... };
Selector → "[" Expression "]";
```

In the example above the sub-expression `[i*5]` is interpreted as an access to the $(i*5)$-th element of the array `new int[4]`.

The ambiguous interpretation of the array creation statements leads to Shift/Reduce conflict between the token "[" and the production ArrayCreatorRest.

```
ArrayCreatorRest → { ("[" Expression "]") ... }+ BracketsOpt;
BracketsOpt → { "[]" ... };
```

In the example above the conflict occurs when the sub-expression `new int[4]` has been already parsed and the token "[" appears next in the input. If shift on token "[" is performed, the entire statement will be considered as a creation of 2-dimensional array. If reduce by ArrayCreatorRest is performed, the statement will be considered as array access.

Figure 16 shows two possible parsing trees which can be produced for the statement from the example above.

According to JAVA semantic rules, Primary expression, which has an array access operator after it, cannot be an array creation expression [15, §15.13]. It means that the right parsing tree on Figure 16 is erroneous. While parsing array creation statements, the **Shift** action should be performed to obtain semantically correct parsing trees.

The conflict is resolved by means of priorities. The priority is assigned both to the token "[" and to the nonterminal ArrayCreationRest. The priority of the token is higher than the priority of the nonterminal, and this causes to the shift action to be performed at parsing time.

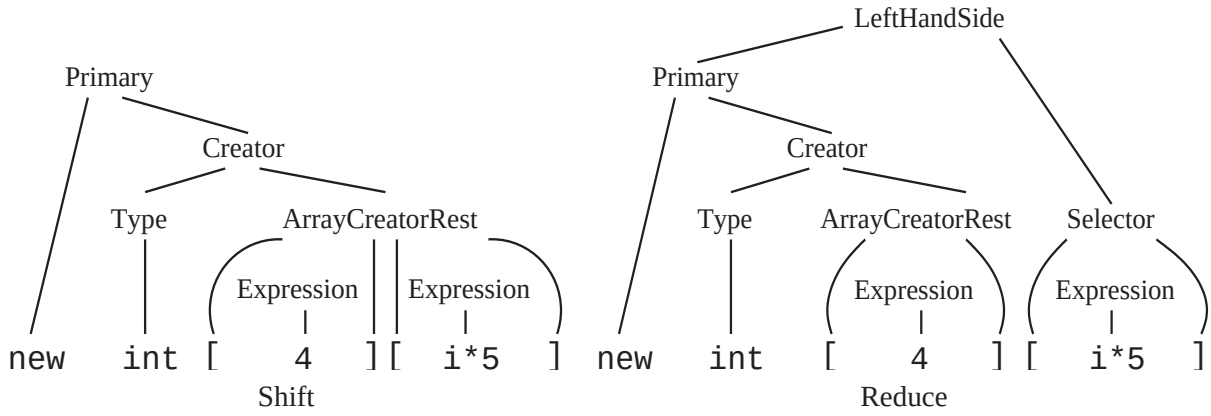

Figure 16: Shift/Reduce conflict which occurs while parsing array creation statements in JAVA language.

8.1.2 Modifiers Conflict

Class members in JAVA can be declared with different access control attributes. Special keywords which are called *modifiers* are used for this purpose. Consider the class `Point`:

```
class Point {
    protected int x, y;
    public void move(int dx, int dy) { x += dx; y += dy; }
}
```

The fields `x` and `y` are declared *protected*. The modifier *protected* means that the fields are accessible only in objects of the class `Point` or in its subclasses. The method `move` of the class `Point` is declared *public*. The modifier *public* means that the method is available to any code that uses an object of type Point.

Class members are described by the following production[14]:

```
ClassBodyDeclaration → ModifiersOpt MemberDecl;
ModifiersOpt → { Modifier ... };
Modifier →
    pub OF public |
    prot OF protected |
    priv OF private |
    abstract OF abstract;
```

A class member, in turn, can be a class or an interface. A class, whose declaration occurs within the body of another class or interface, is called a *nested class*. Both nested class and interface can be specified with their own access control attributes.

```
MemberDecl → ClassOrInterfaceDeclaration;
ClassOrInterfaceDeclaration → ModifiersOpt
    class_decl OF ClassDeclaration | intrf_decl OF InterfaceDeclaration;
```

[14]For sake of simplicity, only part of the modifiers list is shown.

Consider the following declaration of a nested class `Nested`:

```
class TopLevel {
    public class Nested { ... }
}
```

While parsing the nested class declaration we receive the following derivation according to the productions above:

ClassBodyDeclaration ⇒
ModifiersOpt MemberDecl ⇒
ModifiersOpt ClassOrInterfaceDeclaration ⇒
ModifiersOpt ModifiersOpt ClassDeclaration ⇒
ModifiersOpt ModifiersOpt **`class`** `Nested { ... }`

The list of modifiers ModifiersOpt can be empty. Therefore the modifier **`public`** of the class `Nested` can be considered as a part of either ClassBodyDeclaration or ClassOrInterfaceDeclaration.

This ambiguity leads to the Shift/Reduce conflict between the tokens which represent modifiers, and the nonterminal ModifiersOpt.

While parsing the declaration of the class `Nested` in the example above, the conflict occurs when the token **`public`** appears in the input. If shift action is performed at this stage, the token **`public`** will be considered as a part of ClassBodyDeclaration. If reduce by the nonterminal ModifiersOpt is performed, the token **`public`** will be considered as a part of ClassOrInterfaceDeclaration.

Figure 17 shows two possible parsing trees which can be derived for the `Nested` class declaration.

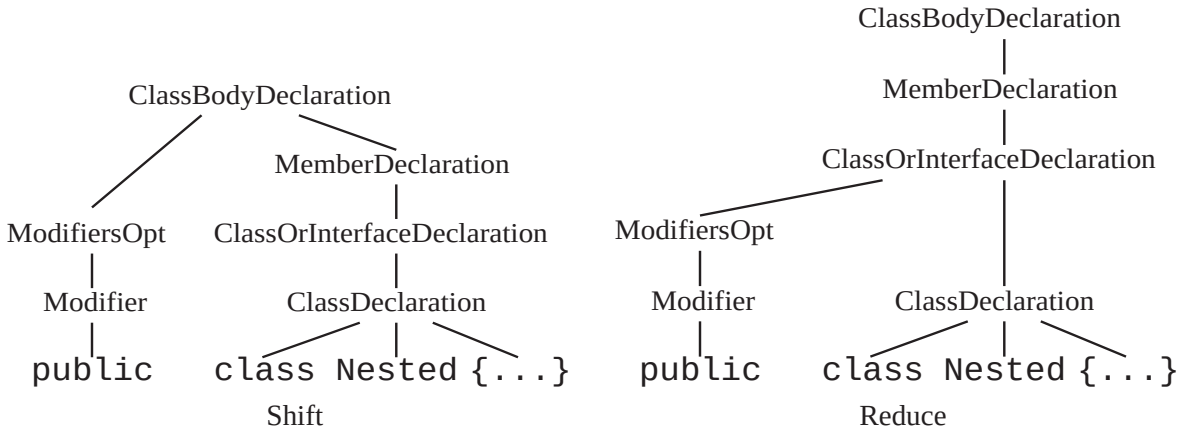

Figure 17: Shift/Reduce conflict which occurs while parsing declarations of nested classes and interfaces in JAVA language. The conflict takes place only if access modifiers are specified for a nested class or interface.

Both parsing trees are legitimate. Still the conflict is resolved by choosing **Reduce** which means that the modifiers will be considered as a part of ClassOrInterfaceDeclaration. This decision provides the common treatment to all class or interface declarations.

To resolve the conflict priorities are assigned to all the tokens which represent modifiers and to the nonterminal ModifiersOpt. The priority of the nonterminal is higher than the priorities of the tokens, and this causes to the reduce action to be performed at parsing time.

8.1.3 Dangling Else Conflict

Consider the following input:

```
if (i > 10)
    if (j < 5)
        // do something...
    else
        // do something else...
```

Here the **else** branch can be associated with both conditional statements. This ambiguity leads to the Shift/Reduce conflict between the token **else** and the nonterminal Statement:

Statement → **if** ParExpression Statement [**else** Statement];

In the example above the conflict occurs when the conditional statements have been already parsed and the token **else** appears next in the input. If the shift action is performed, the **else** branch will be bound to the innermost conditional statement.

If the reduce action is performed, parsing error will occur. The token **else** appears only in the production Statement. If the reduce action is performed each time when the token appears in the input, the parsing flow never will enter to the optional section of the Statement production. It follows that the token **else** will cause to parsing errors each time it appears in the input.

This conflict is resolved by binding **else** branch to the innermost conditional statement. Priorities are assigned both to the token **else** and to the nonterminal Statement. The priority of the token is higher that the priority of the nonterminal, which means that **Shift** will be performed.

8.2 Reduce/Reduce Conflicts

Consider the following block:

```
{
    Telephone = 123456;
}
```

This block statement represents an assignment expression. The identifier `Telephone` is a variable name.

The following derivation corresponds to the block statement from the example above:

Statement ⇒
Assignment ; ⇒
LeftHandSide AssignmentOperator AssignmentExpression ; ⇒
LeftHandSide = 123456; ⇒
Primary = 123456; ⇒
{ Identifier "." ... }+ [IdentifierSuffix] = 123456; ⇒
Telephone = 123456;

Consider another block:

```
{
    Telephone tel = 123456;
```

}

This block statement represents a declaration of a local variable `tel`. Here the identifier `Telephone` is a type name of the variable.

The following derivation corresponds to this block statement:

LocalVariableDeclarationStatement ⇒
[**final**] Type VariableDeclarators `;` ⇒
Type `tel = 123456;` ⇒
Reference `tel = 123456;` ⇒
QualifiedIdentifier BracketsOpt `tel = 123456;` ⇒
{ Identifier "." ... }+ `tel = 123456;` ⇒
`Telephone tel = 123456;`

In the JAVA grammar both Statement and LocalVariableDeclarationStatement are kind of BlockStatement:

```
BlockStatement  →
    local_decl  OF  LocalVariableDeclarationStatement  |
    simple  OF  Statement;
```

It means that parsing of the statements, which are described by both Statement and LocalVariableDeclarationStatement nonterminals, is started from the same parser state. These statements, as we saw in the examples above, may have the same beginning: a qualified identifier. While parsing the qualified identifier, the parsing flow will be identical for both assignment statements and local variable declarations. When the qualified identifier is parsed it can be considered as:

- Variable name. The entire statement is considered as an assignment statement.
- Type name. The entire statement is considered as a declaration of a local variable.

This ambiguity leads to the Reduce/Reduce conflict between the following productions:

```
Primary  →  {  Identifier  "."  ...  }+  [  IdentifierSuffix  ];
QualifiedIdentifier  →  {  Identifier  "."  ...  }+;
```

If reduce by Primary production is performed, the identifier is considered as a variable name. If reduce by QualifiedIdentifier production is performed, the identifier is considered as a type name.

This conflict cannot be resolved by means of priorities and associativity, because both reduce actions may take place at parsing time. However, this conflict can be resolved by exploiting semantic information about the qualified identifier. If the identifier is a type name, one of the three following options should take place:

- The identifier is a basic type name. In this case it should appear in the list of the language keywords.
- The identifier is a name of a type that appears in one of the standard packets imported. In this case the type name can be found in the list of types that appear in the imported packet.
- The identifier is a user-defined type name. In this case it should have already appeared in the program symbol table.

Thus, in all the cases if the identifier is a type name, it should have already been known at the current parsing step. If it is unknown, it should be interpreted as a variable name. Therefore, this conflict can be unequivocally resolved employing the semantic information enclosed in the JAVA program.

8.3 Pop/Reduce Conflicts

In this section we will discuss the Pop/Reduce conflict which occurs while parsing nested JAMOOS *choice expressions*, and demonstrate the resolution of the conflict.

8.3.1 Choice Expressions in JAMOOS

Consider the following JAMOOS program which checks if a number is divisible by 5.

```
GRAMMAR DivisibleBy5()
    DivisibleBy5 → num:Number;
    FEATURES
        return := [[ /* if the last digit of num is 5? */ ]] ? t1:true |
                  [[ /* if the last digit of num is 0? */ ]] ? t2:true |
                                                               f:false ?
                      yes: [[ printf("yes") ]] |
                      no:  [[ printf("no") ]];
    END;
    Number → ⟨[1-9][0-9]*⟩;
END
```

The program reads a number and stores it in the field **num** of the production DivisibleBy5. Then the program applies rules of divisibility by 5 and prints `yes` if the number is divisible by 5 or `no` if it is not.

A number is divisible by 5 if its last digit is either 5 or 0. In order to implement these two rules choice expressions were used in the program above.

The JAMOOS *choice expression* allow selection among multiple choices; it is similar to alternations in regular expressions. The choices are separated from each other by a vertical bar. Each choice has a name which appears immediately to the left of the choice and is separated from it by a colon.

Each choice in a choice expression may have a corresponding condition. The condition appears immediately to the left of the choice's name and is separated from it by a question-mark. In this case the evaluation of the choice expression is similar to the one of C++ **`switch`** statement. The conditions are evaluated in order of their appearance; the first choice whose condition is **true** will be chosen.

A last choice in a choice expression is often defined without corresponding condition. In this case the meaning of this choice is equal to the one of the **`default`** case in C++. If none of the conditions was true the last choice is chosen.

The syntax of choice expressions is described by the following production:

```
ChoiceExpression → {(condition:[Expression "?"] tag:Id ":" Expression) "|" ... }+;
```

Choice expressions are a kind of compound type expressions which in turn are a kind of JAMOOS expressions:

```
CompoundTypeExpression → SequenceExpression | ConstantSizeListExpression |
                         OptionalExpression | ChoiceExpression;
Expression → CompoundTypeExpression | TestExpression | EmbeddedCPP;
```

It follows from the ChoiceExpression production definition that conditions can be described by any type of JAMOOS expressions.[15] In the program above conditions corresponding to two first choices are sections of C++ code[16]:

```
[[ /* if the last digit of num is 5? */ ]]
```

and

```
[[ /* if the last digit of num is 0? */ ]]
```

In JAMOOS sections of C++ code are surrounded by double square brackets.
Another condition from the program above is described by a choice expression, as follows:

```
[[ /* if the last digit of num is 5? */ ]] ? t1:true |
[[ /* if the last digit of num is 0? */ ]] ? t2:true |
                                             f:false
```

Assume that we would like to check if 100 is divisible by 5. When the number 100 is read JAMOOS will calculate the value of the **return** field in the production DivisibleBy5. For this purpose JAMOOS will evaluate the outer choice expression. It means that the first condition, which in turn is described by a choice expression, must be evaluated.

In order to evaluate the inner choice expression the first C++ code section will be executed. Since the last digit of the number 100 is not 5 JAMOOS will proceed to the second condition and will execute the second C++ code section. The last digit of the number 100 is indeed 0, therefore the choice **t2**, which is **true**, will be chosen.

The value of the outer choice expression is **true** and therefore the choice named **yes** will be chosen. This choice is described by a C++ code section that outputs `yes`.

8.3.2 Nested Choice Expressions Conflict

While parsing the choice expression ce from the JAMOOS program described in the Section 8.3.1, a Pop/Reduce conflict occurs. For sake of simplicity, let us denote first and second conditions in the inner choice expression of ce as **c1** and **c2** accordingly. Let us also denote expression **t1** as **e1**, **t2** as **e2**, **f** as **e3**, **yes** as **e4** and **no** as **e5**. The outer choice expression now will look like follows:

```
c1 ? t1:e1 | c2 ? t2:e2 | f:e3 ? yes:e4 | no:e5
```

There exist two different interpretations of this choice expression:

- The inner choice expression consists of three choices: c1 ? t1:**e1**, c2 ? t2:**e2** and f:**e3**. The outer choice expression, in turn, consists of two choices: c1 ? t1:**e1** | c2 ? t2:**e2** | f:e3 ? yes:**e4** and no:**e5**.

[15] The only restriction on conditions is that they must return a boolean value.
[16] The real C++ code which checks if the last digit of a number is 5 or 0 is omitted for sake of simplicity.

- The inner choice expression consists of two choices: c2 ? t2:**e2** and f:**e3**. The outer choice expression consists of three choices: c1 ? t1:**e1**, c2 ? t2:**e2** | f:e3 ? yes:**e4** and no:**e5**.

This ambiguity comes out as a Pop/Reduce conflict at the parsing time. When a parser pops choices of the inner choice expression and reaches the choice c1 ? t1:**e1**, it does not know whether this choice is a part of inner or outer choice expression. In the first case the parser will continue popping the choices of the inner choice expression. It means that the choice c1 ? t1:**e1** will be considered as a part of the inner choice expression. In the second case the parser will stop popping the choices of the inner choice expression, reduce choices that have been already popped, i.e., c2 ? t2:**e2** and f:**e3**, to the Expression and will continue reduction of the outer choice expression. It means that the choice c1 ? t1:**e1** will be considered as a part of the outer choice expression.

The Figure 18 shows the parse tree which is derived if the Pop/Reduce conflict is resolved in favor of pop. The Figure 19 shows the parse tree which is derived if the Pop/Reduce conflict is resolved in favor of reduce.

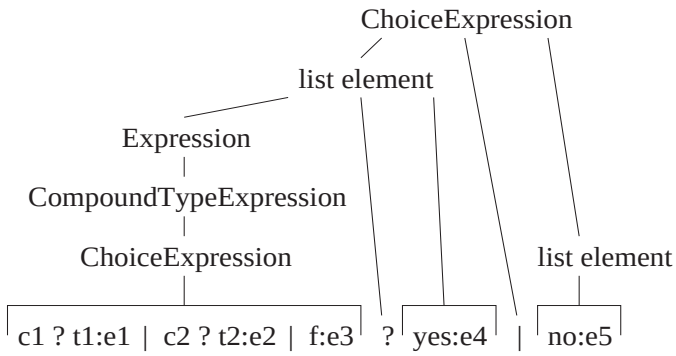

Figure 18: Parse tree which is derived if the Nested Choice Expression conflict is resolved in favor of the pop action. The first alternative is considered as a part of the inner choice expression.

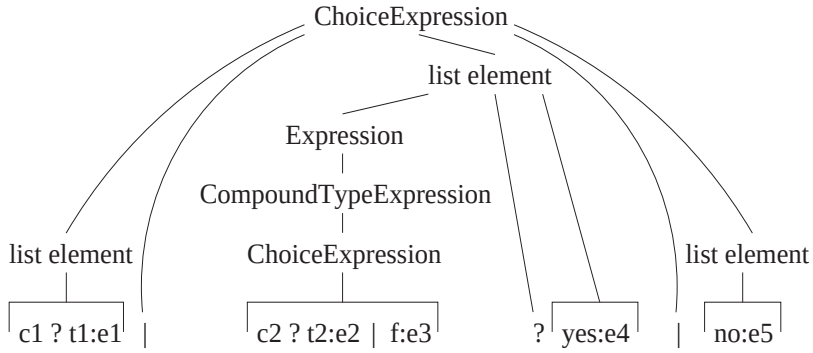

Figure 19: Parse tree which is derived if the Nested Choice Expression conflict is resolved in favor of the reduce action. The first alternative is considered as a part of the outer choice expression.

Since both resolutions are equally legitimate we should define a convention of choosing between them. We decide to choose such resolution that leads to a more compact parse tree. In our case the resolution in favor of the pop action leads to a more compact parse tree.

In order to resolve the conflict in favor of pop we should:

1. Change the ChoiceExpression production so, that names will be assigned to both its list and the element of the list. For example:

   ChoiceExpression → choices:{elem:(condition:[Expression "?"] tag:Id ":" Expression) "|" ... }+;

2. Assign higher priority to the element of the list as follows:

   PRIORITIES
   LEFT choices
   LEFT elem
   END

8.4 Pop/Pop Conflicts

In this section a Pop/Pop conflict which occurs while parsing JAMOOS choice expressions will be discussed and the resolution of the conflict will be demonstrated. This conflict occurs as a result of an ambiguity raised by productions that describe the choice expressions and *optional expressions* in JAMOOS.

The Section 8.4.1 presents a brief explanation on JAMOOS optional expressions in general, and in particular on *general optional* expressions[17]. The understanding of this kind of JAMOOS optional expressions is important, because the production which describes them causes the Pop/Pop conflict above. A detailed explanation on the conflict and its resolution is given in the Section 8.4.2.

8.4.1 Optional Expressions in JAMOOS

The JAMOOS *optional expression* describes an object that may or may not be created by a JAMOOS program depending on the program input. This type of JAMOOS expressions is similar to an optional section in regular expressions.

There exist several types of optional expressions. All of them, except of general optional expressions, should be embraced by square brackets. Here is an example of a production which contains optional expression in its right-hand side:

Name → f:FirstName m:[MiddleName] s:SecondName;

In the production above two fields, FirstName and SecondName, are mandatory, while the field MiddleName may be absent. Both names, which contain a middle name, and names, which does not contain ones, will be accepted by this production. However, objects of the type Name that will be created in these two cases are different. If a name contains a middle name, an object o_m of the type MiddleName will be created and the field **m** of the Name object will be initialized by the reference to o_m. If a name does not contain a middle name, the value of the field **m** of the Name object will be *NULL*.

JAMOOS *general optional* expression describes an object which is created by a JAMOOS program if a certain condition holds. The object should be defined by a JAMOOS expression. The condition appears immediately to the left of the expression and is separated from it by a question-mark. The following production describes the syntax of general optional expressions:

GeneralOptional → [Expression "?"] Expression;

[17] A detailed information about JAMOOS choice expressions can be found at the Section 8.3.1

General optional expressions are a kind of optional expressions which are subtype of compound type expressions:

```
OptionalExpression → EmptyOptional | GeneralOptional;
CompoundTypeExpression → SequenceExpression | ConstantSizeListExpression |
                         OptionalExpression | ChoiceExpression;
Expression → CompoundTypeExpression | TestExpression | EmbeddedCPP;
```

8.4.2 Optional-Choice Conflict

Consider the following JAMOOS program which checks if a number is divisible by 2:

```
GRAMMAR DivisibleBy2()
    DivisibleBy2 → num:Number;
    FEATURES
        return := [[ /* if the last digit of num is even? */ ]] ?
                  yes: [[ printf("yes") ]] |
                  no:  [[ printf("no") ]];
    END;
    Number → ⟨[1-9][0-9]*⟩;
END
```

The number is read from the input and stored in the variable **num** of the production DivisibleBy2. In order to check if the number is divisible by 2 the choice expression should be evaluated. First, the condition

```
[[ /* if the last digit of num is even? */ ]]
```

will be evaluated[18]. If the condition holds, then the first choice named **yes** will be chosen, which means that the corresponding C++ code section will be executed and `yes` will be printed. If the condition does not holds, JAMOOS will proceed to the choice **no**, execute the corresponding C++ code and `no` will be printed.

While parsing the choice expression from the JAMOOS program above a Pop/Pop conflict occurs.

For sake of simplicity let us denote the condition as **c**, the choices **yes** and **no** as **e1** and **e2** accordingly. Then the choice expression will look as follows:

```
c ? yes:e1 | no:e2
```

There exist two interpretations of the expression above:

- Choice expression that consists of two choices: `c ? yes:`**e1** and `no:`**e2**.
- General optional expression in which **c** is a condition and `yes:`**e1** | `no:`**e2** is an optional expression.

[18]The real C++ code which checks if the last digit of a number is even is omitted for sake of simplicity.

Consider a production that describes JAMOOS choice expressions:

```
ChoiceExpression → choices:{elem:(condition:[Expression "?"] tag:Id ":" Expression) "|" ... }+;
```

The optional condition `Expression "?"` is treated by JAMOOS as the following alternation:

```
Expression "?" | --
```

The symbol `--` in JAMOOS stands for an empty string. The alternation above means that in the input may appear either `Expression "?"` or nothing.

While parsing the expression `c ? yes:`**`e1`** `| no:`**`e2`** both alternatives can be popped. If the alternative `Expression "?"` is popped, the string `c ?` is interpreted as a part of the choice expression. It means the the whole expression is considered as a choice expression. The corresponding parse tree is shown in Figure 20.

If the empty alternative is popped, the whole expression is considered as a general optional expression. The corresponding parse tree is shown in Figure 21.

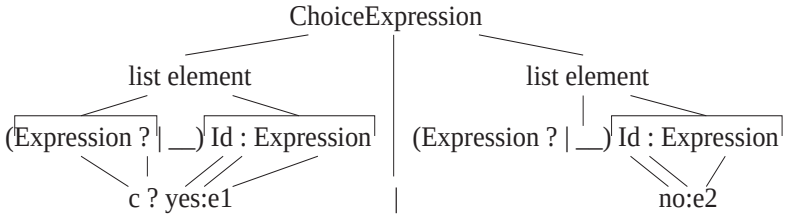

Figure 20: Parse tree which is derived if the Optional-Choice conflict is resolved in favor of choice expression interpretation.

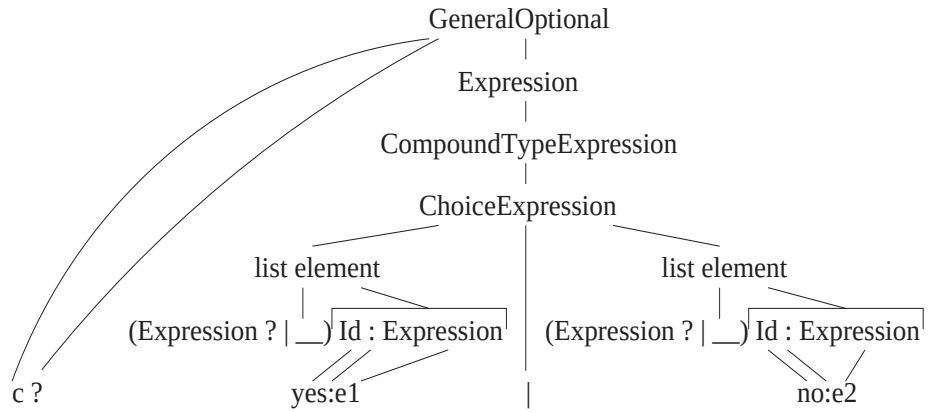

Figure 21: Parse tree which is derived if the Optional-Choice conflict is resolved in favor of general optional expression interpretation.

Since both resolutions are equally legitimate we choose one that leads to more compact parse tree. It means that the conflict will be resolved in favor of the non-empty alternative.

In order to resolve the conflict the following steps should be performed:

1. The optional section in the production ChoiceExpression should be rewritten as an alternation as follows:

   ChoiceExpression → choices:{elem:(condition:(Expression "?" | _)
   tag:Id ":" Expression) "|" ... }+;

2. Priority groups should be defined for the alternatives so, that the priority of the non-empty alternative will be higher the priority of the empty one. The resulting production will look as follows:

   ChoiceExpression → choices:{elem:(condition:(Expression "?" || _)
   tag:Id ":" Expression) "|" ... }+;

9 Conclusions

In this thesis we continue working on programming language JAMOOS that was defined by Tsoglin [31].

JAMOOS was extended for resolving Shift/Reduce and Reduce/Reduce conflicts while generating a parser for a given grammar. The algorithm for Shift/Reduce and Reduce/Reduce conflicts resolution based on priorities and associativity was proposed.

The current JAMOOS's grammar was extended by a production that determines the syntax of a specific JAMOOS program section called "Priorities Section". In this section a user can assign priorities and associativity to tokens and nonterminal grammar symbols.

It was shown that assigning a priority to a token basing on the priority of a production that contains the token is essential for resolving Shift/Reduce conflicts. Since more than one production may contain the token, the parser should determine which particular production is applicable at the given stage of the parsing process. In most cases, the applicable production can be determined at the stage of the parser generation employing the priorities defined by the user and the set of LR-items of the grammar. However, in some cases a next input token should be known to determine the applicable production. The JAMOOS was extended to generate parsers that contain built-in mechanism that determines at run time which production is applicable at the current stage of parsing process.

References

[1] *CppCC – C++ Compiler Compiler*. http://cppcc.sourceforge.net/.

[2] *GOLD – Generalized Oriented Language Developer*. http://www.devincook.com/GOLD Parser.

[3] *Information Technology – Syntactic Metalanguage – Extended BNF*. ISO/IEC 14977:1996(E). http://www.cl.cam.ac.uk/~mgk25/iso-ebnf.html.

[4] *LEMON Parser Generator*. http://www.hwaci.com/sw/lemon.

[5] *SAIF – Spatial Archive and Interchange Format*. http://s2k-ftp.cs.berkeley.edu:8000/sequoia/schema/html/saif/saifHome.html.

[6] *SLK Parser Generator*. http://home.earthlink.net/~slkpg.

[7] *Spirit Parser Framework*. http://spirit.sourceforge.net.

[8] *YAY - Yet Another Yacc*. http://www.thinkage.ca/english/products/product-yay.shtml.

[9] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[10] A. Aho and J. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.

[11] P. Breuer and J. Bowen. A PREttier compiler-compiler: Generating higher order parsers in C. *Software – Practice and Experience*, 25:1263–1297, 1995.

[12] C. Dodd and V. Maslov. *BtYacc – BackTracking Yacc*. Siber Systems. http://www.siber.com/btyacc.

[13] M. A. Ertl. *Gray V 4*. http://students.si.fct.unl.pt/users/pjmlp/en/parserscan.html.

[14] J. A. Farrell. *Compiler Basics*. Addison-Wesley, Aug. 1995.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2000.

[16] J. Graver, V. Hanvivatpong, and D. Wilson. *T-Gen – Translator Generator*, 1992. http://st-www.cs.uiuc.edu/users/droberts/tgen2.2.1/tgen.html.

[17] D. Grune and C. J. H. Jacobs. A programmer-friendly LL(1) parser generator. *Software – Practice and Experience*, 18:29–38, 1988.

[18] A. G. Hartford, V. P. Heuring, and M. G. Main. A new parsing method for non-LR(l) grammars. *Software – Practice and Experience*, 22(5):419–437, 1992.

[19] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts, 1979.

[20] S. C. Johnson. Yacc - yet another compiler compiler. Technical Report Computing Systems Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[21] A. Johnstone and E. Scott. rdp - an iterator-based recursive descent parser generator with tree promotion operators. In *SIGPLAN*, volume 33 of *SIGPLAN Notices*, pages 87–94, Sept. 1998.

[22] B. Kühl and A. Schreiner. An object-oriented LL(1) parser generator. In *SIGPLAN*, volume 35 of *SIGPLAN Notices*, pages 33–40, Dec. 2000.

[23] J. Lampe. *Depot4 – Simple to Use Translator Generator*. http://www.math.tu-dresden.de/wir/depot4.

[24] O. L. Madsen and B. B. Kristensen. LR-parsing of extended context free grammars. *Acta Informatica*, 7:61–73, 1976.

[25] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, Massachusetts, 1999.

[26] P. Naur. Revised report on the algorithmic language ALGOL 60. *ACM*, 6(1):1–17, 1963.

[27] T. Parr. *ANTLR – Framework for Constructing Recognizers, Compilers and Translators*. http://www.antlr.org.

[28] Parsifal Software, Wayland, MA. *AnaGram – LALR Parser Generator*. http://www.parsifalsoft.com/.

[29] T. Rus and J. S. Jones. PHRASE parsers from multi-axiom grammars. *Theoretical Computer Science*, 199(1–2):199–229, 1998.

[30] F. W. Schroer. *Accent*. German National Research Center for Information Technology. http://accent.compilertools.net/index.html.

[31] Y. Tsoglin. JAMOOS—an object oriented language for grammars. Research thesis, The Technion—Israel Institute of Technology, Haifa, Israel, Mar. 2001.

[32] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In N. Horspool, editor, *Proceedings in Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 143–158. Springer-Verlag, 2002.

[33] M. G. J. van den Brand, A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In S. Tilley and G. Visaggio, editors, *6th International Workshop on Program Comprehension*, IEEE Computer Society Press, pages 108–117, 1998.

[34] J. Welsh and J. Elder. *Introduction to Pascal*. International Series in Computer Science. Prentice Hall, 1988.

[35] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions. *ACM*, 20(11):822–823, Nov. 1977.

A JAVA Language Grammar

```
GRAMMAR CompilationUnit ()

PRIORITIES
LEFT    Statement BracketsOpt ModifiersOpt
LEFT    final synchronized public protected private static abstract native
        transient volatile strictfp "[" else
RIGHT   AssignmentExpression
RIGHT   ConditionalExpression
LEFT    ConditionalOrExpression
LEFT    ConditionalAndExpression
LEFT    InclusiveOrExpression
LEFT    ExclusiveOrExpression
LEFT    AndExpression
LEFT    EqualityExpression
LEFT    RelationalExpression
LEFT    ShiftExpression
LEFT    AdditiveExpression
LEFT    MultiplicativeExpression
RIGHT   PrefixUnaryExpression
LEFT    PostfixUnaryExpression
END

_= ⟨[ \n\t]*⟩;
_= "/*" ... "*/";
_= "//" ... ;

QualifiedIdentifier →
    { Identifier "." ... }+ ;

Literal →
    IntegerLiteral |
    FloatingPointLiteral |
    CharacterLiteral |
    StringLiteral |
    BooleanLiteral |
    NullLiteral ;

IntegerLiteral →
    DecimalIntegerLiteral |
    HexIntegerLiteral |
    OctalIntegerLiteral;

DecimalIntegerLiteral → DecimalNumeral;

HexIntegerLiteral → HexNumeral;

OctalIntegerLiteral → OctalNumeral;

BooleanLiteral → tr OF true | fls OF false;

CharacterLiteral → SingleCharacter | EscapeSequence;
```

NullLiteral → **null**;

Expression → AssignmentExpression |
ConditionalExpression |
ConditionalOrExpression |
ConditionalAndExpression |
InclusiveOrExpression |
ExclusiveOrExpression |
AndExpression |
EqualityExpression |
RelationalExpression |
ShiftExpression |
AdditiveExpression |
MultiplicativeExpression |
CastExpression |
PrefixUnaryExpression |
PostfixUnaryExpression;

AssignmentExpression → LeftHandSide
equal **OF** "=" |
plus_eq **OF** "+=" |
minus_eq **OF** "-=" |
mult_eq **OF** "*=" |
div_eq **OF** "/=" |
and_eq **OF** "&=" |
or_eq **OF** "|=" |
xor_eq **OF** "^=" |
mod_eq **OF** "%=" |
shift_left_eq **OF** "<<=" |
shift_right_eq **OF** ">>=" |
zero_ext_eq **OF** ">>>="
Expression;

LeftHandSide → Primary { Selector ... };

ConditionalExpression → Expression "?" Expression ":" Expression;

ConditionalOrExpression → Expression "||" Expression;

ConditionalAndExpression → Expression "&&" Expression;

InclusiveOrExpression → Expression "|" Expression;

ExclusiveOrExpression → Expression "^" Expression;

AndExpression → Expression "&" Expression;

EqualityExpression → Expression equal **OF** "==" | not_equal **OF** "!=" Expression ;

RelationalExpression →
relational **OF** (Expression less **OF** "<" |
more **OF** ">" |
less_eq **OF** "<=" |
more_eq **OF** ">="]
Expression) |

instanceof **OF** (Expression **instanceof** Type) ;

ShiftExpression → Expression
shift_left **OF** "<<" |
shift_right **OF** ">>" |
zero_ext **OF** ">>>"
Expression;

AdditiveExpression → Expression plus **OF** "+" | minus **OF** "-" Expression;

MultiplicativeExpression → Expression
mult **OF** "*" |
div **OF** "/" |
mod **OF** "%"
Expression;

PrefixUnaryExpression → inc **OF** "++" |
dec **OF** "--" |
logical_complitment **OF** "!" |
bitwise_complitment **OF** "~" |
plus **OF** "+" |
minus **OF** "-"
Expression;

PostfixUnaryExpression → primary **OF** Primary |
incdec **OF** (Expression inc **OF** "++" | dec **OF** "--");

CastExpression → "(" Type ")" Expression;

Type → (reference **OF** QualifiedIdentifier | basic **OF** Basic) BracketsOpt ;

Basic → BasicType ;

StatementExpression → Expression ;

ConstantExpression → Expression ;

Selector →
id **OF** ("." Identifier [Arguments]) |
this_ **OF** ("." **this**) |
super **OF** ("." **super** SuperSuffix) |
creation **OF** ("." **new** InnerCreator) |
expr **OF** ("[" Expression "]") ;

Primary →
parenthesized **OF** ("(" Expression ")") |
this_token **OF** (this [Arguments]) |
super_token **OF** (**super** SuperSuffix) |
literal **OF** Literal |
instance_creation **OF** (**new** Creator) |
reference **OF** ({ Identifier "." ... }+ [IdentifierSuffix]) |
class_literals **OF** (Type BracketsOpt "." **class**) |
void_class_literals **OF** (**void** "." **class**) ;

IdentifierSuffix →

```
        brackets_expr OF ( "["
                                empty_brackets_and_class OF ( "]" BracketsOpt "." class ) |
                                array_init OF ( Expression "]" ) ) |
        arguments OF Arguments |
        specified OF ( "."
                        (class_ OF class |
                        this_ OF this |
                        super OF (super Arguments) |
                        creation OF (new InnerCreator) ) ) ;

SuperSuffix →
        arg OF Arguments |
        qulified_superclass_method OF ("." Identifier [ Arguments ]) ;

BasicType →
        byte_ OF byte |
        short_ OF short |
        char_ OF char |
        int_ OF int |
        long_ OF long |
        float_ OF float |
        double_ OF double |
        boolean_ OF boolean ;

ArgumentsOpt → [ Arguments ] ;

Arguments → "(" { Expression "," ... } ")" ;

BracketsOpt → { "[]" ... } ;

Creator → Type
        (array_creator OF ArrayCreatorRest |
        class_creator OF ClassCreatorRest) ;

InnerCreator → Identifier ClassCreatorRest ;

ArrayCreatorRest →
        "["
        array_init OF ( "]" BracketsOpt ArrayInitializer ) |
        expr_in_brackets OF ( Expression "]" { ("[" Expression "]") ... } BracketsOpt) ;

ClassCreatorRest → Arguments [ ClassBody ] ;

ArrayInitializer → "{" { VariableInitializer "," ... } [ "," ] "}" ;

VariableInitializer →
        ArrayInitializer |
        ExpressionStatement ;

ParExpression →  "(" Expression ")" ;

Block → "{" BlockStatements "}";

BlockStatements → { BlockStatement ... };
```

```
BlockStatement →
    local_decl OF LocalVariableDeclarationStatement |
    class_or_interface_decl OF ClassOrInterfaceDeclaration |
    simple OF Statement;

LocalVariableDeclarationStatement → [ final ] Type VariableDeclarators ";" ;

Statement →
    block OF Block |
    if_stmt OF (if ParExpression Statement [ else Statement ]) |
    for_stmt OF (for "(" [ ForInit ] ";" [ Expression ] ";"  [ ForUpdate ] ")" Statement) |
    while_stmt OF (while ParExpression Statement) |
    repeat_stmt OF (do Statement while ParExpression ";") |
    try_stmt OF (try Block  without_fin OF Catches | with_fin OF ([ Catches ] finally Block)) |
    switch_stmt OF (switch ParExpression "{" SwitchBlockStatementGroups "}") |
    synchro_stmt OF (synchronized ParExpression Block) |
    return_stmt OF (return [ Expression ] ";") |
    throw_stmt OF (throw Expression ";") |
    break_stmt OF (break [ Identifier ] ";") |
    continue_stmt OF (continue [ Identifier ] ";") |
    empty_stmt OF ";" |
    expr_stmt OF (ExpressionStatement ";") |
    labeled_stmt OF (Identifier ":" Statement) ;

ExpressionStatement → StatementExpression ;

Catches → { CatchClause ... }+ ;

CatchClause → catch "(" FormalParameter ")" Block ;

SwitchBlockStatementGroups → { SwitchBlockStatementGroup ... } ;

SwitchBlockStatementGroup → SwitchLabel BlockStatements ;

SwitchLabel →
    regular_case OF (case ConstantExpression ":") |
    default_case OF (default ":") ;

ForInit →
    statements OF { StatementExpression "," ... }+ |
    finalized OF ([ final ] Type VariableDeclarators) ;

ForUpdate → { StatementExpression "," ... }+ ;

ModifiersOpt → { Modifier ... } ;

Modifier →
    pub OF public |
    prot OF protected |
    priv OF private |
    stat OF static |
    abstract OF abstract |
    final OF final |
    native OF native |
    synchro OF synchronized |
```

transient **OF** **transient** |
volatil **OF** **volatile** |
strictfp **OF** **strictfp** ;

VariableDeclarators → { VariableDeclarator “,” ... }+ ;

ConstantDeclaratorsRest → ConstantDeclaratorRest { (“,” ConstantDeclarator) ... } ;

VariableDeclarator → Identifier VariableDeclaratorRest ;

ConstantDeclarator → Identifier ConstantDeclaratorRest ;

VariableDeclaratorRest → BracketsOpt [“=” VariableInitializer] ;

ConstantDeclaratorRest → BracketsOpt “=” VariableInitializer ;

VariableDeclaratorId → Identifier BracketsOpt ;

CompilationUnit → [**package** QualifiedIdentifier “;”] { ImportDeclaration ... } { TypeDeclaration ... } ;

ImportDeclaration → **import** { Identifier “.” ... }+ [“.” “*”] “;” ;

TypeDeclaration →
class_or_interface **OF** ClassOrInterfaceDeclaration |
empty **OF** “;” ;

ClassOrInterfaceDeclaration →
ModifiersOpt
class_decl **OF** ClassDeclaration | intrf_decl **OF** InterfaceDeclaration ;

ClassDeclaration → **class** Identifier [**extends** Type] [**implements** TypeList] ClassBody ;

InterfaceDeclaration → **interface** Identifier [**extends** TypeList] InterfaceBody ;

TypeList → { Type “,” ... }+ ;

ClassBody → “{” { ClassBodyDeclaration ... } “}” ;

InterfaceBody → “{” { InterfaceBodyDeclaration ... } “}” ;

ClassBodyDeclaration →
empty **OF** “;” |
block **OF** ([**static**] Block) |
member_decl **OF** (ModifiersOpt MemberDecl) ;

MemberDecl →
method_or_field **OF** MethodOrFieldDecl |
void_return **OF** (**void** Identifier MethodDeclaratorRest) |
ctor_decl **OF** (Identifier ConstructorDeclaratorRest) |
class_or_interface **OF** ClassOrInterfaceDeclaration ;

MethodOrFieldDecl → Type Identifier MethodOrFieldRest ;

MethodOrFieldRest →
    var_decl **OF** (VariableDeclaratorRest ";") |
    method_decl **OF** MethodDeclaratorRest ;

InterfaceBodyDeclaration →
    empty **OF** ";" |
    member_decl **OF** (ModifiersOpt InterfaceMemberDecl) ;

InterfaceMemberDecl →
    method_or_field **OF** InterfaceMethodOrFieldDecl |
    void_return **OF** (**void** Identifier VoidInterfaceMethodDeclaratorRest) |
    class_or_interface **OF** ClassOrInterfaceDeclaration ;

InterfaceMethodOrFieldDecl → Type Identifier InterfaceMethodOrFieldRest ;

InterfaceMethodOrFieldRest →
    constants **OF** (ConstantDeclaratorsRest ";") |
    methods **OF** InterfaceMethodDeclaratorRest ;

MethodDeclaratorRest →
    FormalParameters BracketsOpt [**throws** QualifiedIdentifierList]
    (body **OF** MethodBody | empty_body **OF** ";") ;

InterfaceMethodDeclaratorRest → FormalParameters BracketsOpt [**throws** QualifiedIdentifierList] ";" ;

VoidInterfaceMethodDeclaratorRest → FormalParameters [**throws** QualifiedIdentifierList] ";" ;

ConstructorDeclaratorRest → FormalParameters [**throws** QualifiedIdentifierList] MethodBody ;

QualifiedIdentifierList → { QualifiedIdentifier "," ... }+ ;

FormalParameters → "(" { FormalParameter "," ... } ")" ;

FormalParameter → [**final**] Type VariableDeclaratorId ;

MethodBody → Block ;

Identifier → ⟨[a-zA-Z0-9]*⟩;

DecimalNumeral → ⟨(0)|([1-9][0-9]*)⟩;

HexNumeral → ⟨(0)((x)|(X))([0-9a-fA-F]*)⟩;

OctalNumeral → ⟨0[0-7]*⟩;

Digits → ⟨[0-9]*⟩;

SingleCharacter → ⟨'([^'\\])'⟩;

EscapeSequence → ⟨'(\\b)|(\\t)|(\\n)|(\\f)|(\\r)|(\\")|(\\')|(\\[0-7])|(\\[0-7][0-7])|(\\[0-3][0-7][0-7])'⟩;

StringLiteral → ⟨"(([^"\\]|(\\[.]))*"⟩;
END