Masaryk University
Faculty of Informatics

Generic BNF-driven Parser

Bachelor's thesis

Václav Vacek

Brno, 2008

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Brno, January 3, 2008
Václav Vacek

Advisor: RNDr. Vojtěch Řehák, Ph.D.

Acknowledgement

This thesis has been written in cooperation with ANF DATA, spol. s r.o., I would like to thank namely Petr Gotthard and Radek Sedlaček for their help.

I want to thank Vojtěch Řehák for his support. Finally, I would like to thank Petr Slovák for his help with the implementation of the parser and and also for his useful advice.

Abstract

This thesis describes the bnfparser[2] C++ library. It is a generic runtime-generated parser capable of parsing any context-free language. The parser is primarily meant for verification and diagnosis of telecommunication protocols, but the usage is not restricted to that.

First, the possibility of using tools already available is discussed. The rest of the thesis is then fully devoted to bnfparser[2]. The architecture of the library is described in detail; all the important steps and algorithms are given. This generally involves tranforming the user input into an internal form, generating the parser and parsing. After that, format of input and output, required settings, restrictions are described. Finally the furher improvements of the library are outlined.

Keywords

Parser, parser generator, context-free languages, ambiguous grammars, Backus-Naur form, LALR, GLR, GSS

Contents

1 **Introduction** . . . 2
- 1.1 *Requirements* . . . 2
- 1.2 *Parser Generators* . . . 3
- 1.3 *Bnfparser²* . . . 3
- 1.4 *Structure of the Document* . . . 4

2 **Architecture** . . . 6
- 2.1 *Overview* . . . 6
- 2.2 *Preprocessing* . . . 6
- 2.3 *GLALR Table Generation* . . . 14
- 2.4 *GSS* . . . 18
- 2.5 *Parser* . . . 19

3 **Using the Parser** . . . 21
- 3.1 *Application Interface Class* . . . 21
- 3.2 *Function-call Sequence* . . . 21
- 3.3 *Syntax-description File* . . . 22
- 3.4 *Grammar File* . . . 26
- 3.5 *Semantic Strings* . . . 27
- 3.6 *Description of the Methods* . . . 29

4 **Further Improvements** . . . 32
- 4.1 *Semantic String Generation* . . . 32
- 4.2 *Speedup* . . . 32
- 4.3 *Special Constructions in Grammar Files* . . . 33

5 **Conclusion** . . . 34

Bibliography . . . 35

A **Content of the CD** . . . 36

Chapter 1

Introduction

Syntax analysis (i.e. parsing) is an important part of computer science. Especially the analysis of context-free languages is widely used as a part of compilers and interpreters of programming languages.

A formal syntax is also often defined as a part of technical specifications. Then there arises the need for a tool capable of parsing words in accordance to the grammar description contained in the specification.

In this chapter we will first define requirements on such a tool, then we will discuss the possibility of using existing tools and then a new tool will be outlined – bnfparser^2. The rest of the document is fully devoted to bnfparser^2.

1.1 Requirements

Input

The family of Internet specifications usually uses a modified version of Backus-Naur Form (BNF) for specifying a grammar. The BNF was defined in 1960 for the definition of Algol 60 [2]. This specification language has developed into dozens of variants, for example Augmented BNF – ABNF [3] or Extended BNF – EBNF [5]. The parser should be able to read any variant of BNF, including newly created variants.

The parser should therefore be able to parse any context-free language (including ambiguous languages) since those may be defined in BNF.

Output

The parser is to be used as a quick reference and thus no complex actions are required to be performed during the parsing process. Determining whether a word is correctly formed (and possibly locating an error) is sufficient in this case. For future diagnostic extensions it might be also useful to get a (part of a) derivation tree.

Standalone application

The parser should operate as a standalone application without binding to other applications. Then the simplicity for the end-user is maintained and using as a part of proprietary diagnostic tools is possible. This also allows the parser to be used as a web application.

Absolute traceability

The parser is primarily intended for verification and diagnosis of telecommunication protocols. This requires *traceability* of the verification process – maintaining complete information about every step in a process chain, including preprocessing of language specifications presented in standards. Therefore no modifications should be made to the grammar by the user; manual tranforming grammars into a specific format is extremely time-consuming and makes traceability very expensive.

1.2 Parser Generators

There are tools called *parser generators*, like Bison [4] or ANTLR [8], available. They are well capable of generating parsers, their disadvantage is, however, the fact a parser is not generated in runtime. The tools typically generate the (C++, Java, . . .) source code of the parser. The source code has to be compiled before parsing is possible. Moreover, the grammar has to be input in a rigidly defined format. That makes the tools impossible to be used as quick runtime verifiers. The workflow of a typical parser generator is shown in figure 1.1.

Hapy parser generator [10] uses a different approach. A grammar is input directly in C++. Then the parser has to be compiled and after that it is able to parse words. Again, the format of the grammar is rigidly defined and thus makes user transform his/her grammar into that format, which is unwanted in our case.

1.3 Bnfparser[2]

The tool I've created uses as generic an approach as possible. The program first parses grammar files in accordance to a given description and then parses words in accordance to the previously parsed grammar.

The parser uses the GLALR parsing algorithm from [11], page 39, which had to be slightly modified. It is capable of parsing any context-free language.

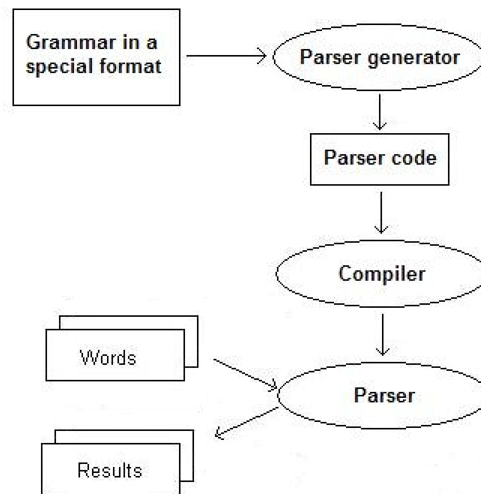

Figure 1.1: Typical parser generator

Thanks to the modifications made to the original algorithm it is possible to obtain a partial syntax tree.

Once the parser is compiled, it can perform all the actions in runtime. The parser is capable of reading grammar specification spanned over multiple files, each of which may use a different syntax. Dependencies between identifiers in different files may be user-defined. Then a GLALR parser is built from the grammar. After that, the parser can parse multiple words without having to be rebuilt. If a word is correctly formed in accordance to the grammar, the parser returns the partial syntax tree, otherwise it localizes the error. Figure 1.2 shows the workflow of bnfparser[2].

1.4 Structure of the Document

Chapter 2 describes the architecture of the parser. First the way the parser is split into files is shown, then the functionality of each part is described. The aim is to give general information – a reader should learn the way the parser works, which algorithms are used and why. Specific information about C++ implementation can be found in the source code.

Chapter 3 describes the usage of the parser. As it is a library, the way it is to be included in a program is described first. Then the way it should be

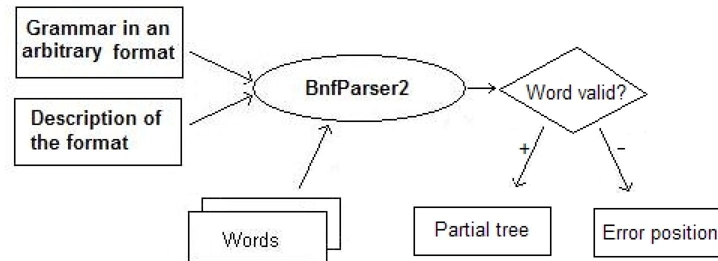

Figure 1.2: BnfParser[2]

used is proposed, the detailed description of the format of input is given as well as the meaning of the output. The public methods of the parser are briefly described in the end.

Finally, chapter 4 summarises possible improvements of the parser.

Chapter 2

Architecture

2.1 Overview

The parser is implemented as a C++ library. It spans over multiple classes, here I'll briefly describe the main classes.

AnyBnfLoad – all the BNF preprocessing steps are performed by this class. It is described in detail in section 2.2.

LalrTable – this class builds the GLALR table from a given BNF grammar. It is described in section 2.3.

GSS – the implementation of a graph-structure stack. It is used by the parser. It simulates a nondeterministic stack. Detailed description can be found in section 2.4.

Parser – the parsing algorithm is implemented in this class. It is described in section 2.5.

BnfParser2 – this class is an encapsulation of the whole parser. It provides an API to the parser library. It is described in section 3.1.

After loading grammar files and syntax-description files, the preprocessing steps are performed by the `AnyBnfLoad` class. The resulting BNF grammar is passed to the `LalrTable` class which computes the GLALR table. The table is then used by `Parser` to parse words supplied by the user.

The figure 2.1 shows the workflow graphically. It doesn't exactly reflect the function calls, it just illustrates the main idea.

2.2 Preprocessing

The preprocessing part of the program deals with transforming user input into internal representation needed by the following modules. User input consists of one or more grammar files and corresponding syntax-description

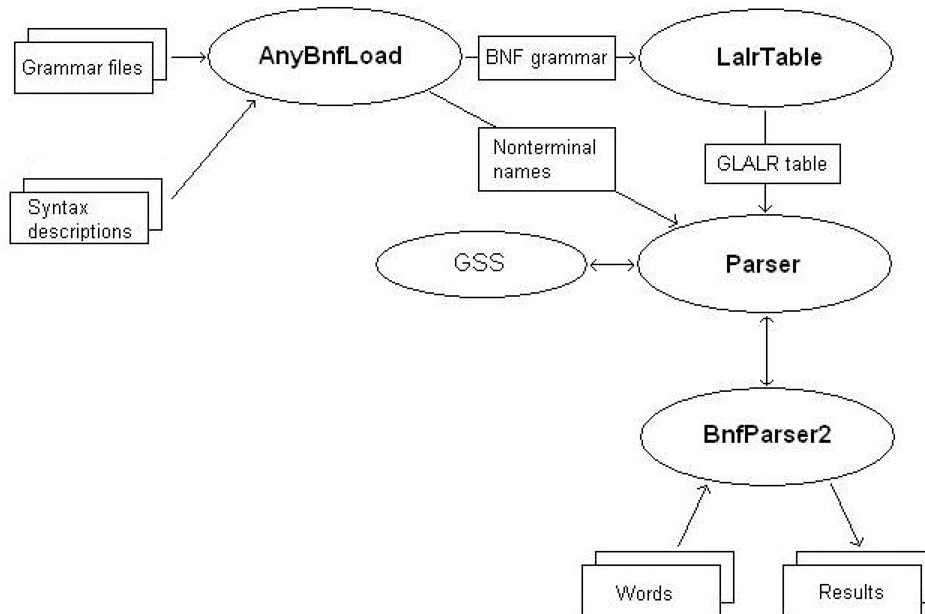

Figure 2.1: Scheme of the parser

files. Each file is processed separately and the result is stored in a common table.

Since each file is processed separately, I will demonstrate the trasformation process on a syntax-description file (figure 2.2) and a simple grammar file (figure 2.3). It describes an ANBF-like syntax and has one extra operator (`#`). The nonterminal `C` is marked in the second rule.

First, the file is loaded into `AnyBnfFile` class. Then the name of the syntax-description file is obtained. It is either set by the user or stored in the grammar file. The syntax-description is loaded into `AnyBnfConf` class. Now, several modifications are made to the input grammar, each taking the output of the previous part as its input.

Processing comments

There may be two types of comments: line comment (`//` in C++) and group comment (`/* */` in C++). Both types of comments are removed, the text of the comments is beforehand searched for eventual directives. The im-

```
rulename=[a-zA-Z][a-zA-Z0-9\-]*
defined==
terminal="
comment=//
concat=
alternative=/
leftgroup=(
rightgroup=)
leftcomment=/*
rightcomment=*/
allbrackets=()
casesensitivestring=false
casesensitiverulename=false
OPERATORS
#\e1=1*3\1
```

Figure 2.2: Syntax-description file

plementation is straightforward, care had to be taken to distinguish valid comment-marks and those contained in strings.

The sample input is transformed as shown in figure 2.4.

Transforming strings

All the strings enclosed in quotation marks are transformed into sequences of ASCII codes in order that all the special characters remaining in the text have actually special meaning (e.g. operators). Besides, each sequence is enclosed in brackets. This stage is shown in figure 2.5.

```
/*
 * Sample grammar file
 */ S = A
A = #B //one, two or three Bs
B = C@
      / *(D D)
C = "//"
D = %x1A-1C
```

Figure 2.3: Input grammar

```
A = #B
B = C@
      / *(D D)
C = "//"
D = %x1A-1C
```

Figure 2.4: Comments removed

```
A = #B
B = C@
      / *(D D)
C = (%d47 %d47)
D = %x1A-1C
```

Figure 2.5: Strings transformed

Condensing rules

Each grammar rule may be spread over multiple lines. It is necessary that each rule is on one line only, thus condensing is performed. The result is shown in figure 2.6.

```
A = #B
B = C@ / *(D D)
C = (%d47 %d47)
D = %x1A-1C
```

Figure 2.6: Rules condensed

Pseudo-lexical analysis

Since the syntax of the identifiers and special characters in the grammar file is known, they may be recognized. As I wanted to benefit from the simple processing using regular expression library in the following part, I had to keep the data stored as standard strings. Therefore I had to mark up the special symbols in the grammar file by certain nonprintable characters.

Following parts of the grammar file are modified:

Names of the nonterminals – each nonterminal is assigned a numeric index (a structure mapping strings to numbers is used), the index is then enclosed by `\036` and `\037` characters. The first index is 2 because 0 is

a terminal symbol and 1 is reserved for the global start nonterminal, which is added later. Each *marked* nonterminal which is assigned the index n is stored as the number `INT_MAX` $-n$.

Terminal symbols – all the ABNF [3] shorthands are transformed to the full form, binary and hexadecimal numbers are transformed to decimal numbers. Finally, each terminal is enclosed by `\036\034` on the left and `\037` on the right.

The "alternative" symbol – each occurrence of the symbol ("/" in ABNF) is replaced by `\035` character.

The "defined" symbol – since it is known that the first nonterminal on a line is a left-hand side of a rule, the "defined" symbol is removed.

The "concatenation" symbol – concatenation is mostly unambiguous, so it may be replaced by a space.

The figure 2.7 shows the result of this phase. The nonprintable characters are made printable as < and > for nonterminals, <− and > for terminals and | for the alternative.

```
<2> #<3>
<3> <2147483643> | *(<5> <5>)
<4> (<-47> <-47>)
<5> (<-26> | <-27> | <-28>)
```

Figure 2.7: Pseudo-lexical analysis

Evaluation of the operators

All the operators defined in the `OPERATORS` section of the syntax-description file are evaluated at this time. For each operator, there is a regular expression finding the occurrence of the operator created. To properly solve bracketing by using regular expressions only the innermost occurrences are searched for and replaced by a single nonterminal (which doesn't contain any brackets any more). The expansion of the operator is then realised in a new grammar rule.

There are three parameter-types available: number parameters are written as `\n<number of the parameter>`, single grammar element or an expression in parentheses written as `\e<number of the parameter>` and `\<number of the parameter>` stands for a sequence of grammar

elements or expressions in parentheses. The parameters and the syntax of operator definition are described in detail in section 3.3.

The reason for introducing two types of element-parameters is following. Some operators have their special brackets. Then the expression inside those brackets has to be found at once. On the other hand, it is completely different with the operators that use standard brackets. If the same type of parameter was used, it could sometimes erroneously match a whole line.

The figure 2.8 shows a highly simplified example. The actual evaluation is a bit more complicated, this just shows the semantics of the process. In the first part, there are three operators defined using different types of parameters. The second part then shows the way the operators are evaluated. Notice that the evaluation of `&` is erroneous.

```
#\e1=1*3(\1)
${\1}=1*3(\1)
&\1=1*3(\1)

#ONE TWO THREE = 1*3(ONE) TWO THREE
#(ONE TWO) THREE = 1*3(ONE TWO) THREE
${ONE TWO} THREE = 1*3(ONE TWO) THREE
&ONE TWO THREE = 1*3(ONE TWO THREE)
&(ONE TWO) THREE = 1*3(ONE TWO THREE)
```

Figure 2.8: Two types of operators

The evaluation proceeds as follows. The innermost occurence of an operator is found. It is replaced with a new, unique nonterminal. The parameters are then substituted to the definition of the operator. The result (in ABNF [3]) is finally stored as a new grammar rule. The process is repeated as long as a change occurs on a certain line. The result is shown in figure 2.9.

```
<2> <6>
<6> 1*3<3>
<3> <2147483643> | *(<5> <5>)
<4> (<-47> <-47>)
<5> (<-26> | <-27> | <-28>)
```

Figure 2.9: Operator evaluation

Translating into BNF

The grammar is now in ABNF. It has to be translated into BNF [2] (with nonterminal naming modified). First, all the brackets are removed (transformed to new grammar rules). Then the ABNF operators are replaced by their BNF definitions. The figure 2.10 shows the result of the process. Finally, the alternatives are split to multiple rules. I will not show the result for all the grammar rules; the figure 2.11 shows just the principle of that.

```
<2> <6>                            <2> <6>
                                   <10> <3>
                                   <10> <3> <3>
                                   <10> <3> <3> <3>
<6> 1*3<3>                         <6> <10>
                                   <11>
                                   <11> <7> <11>
<3> <2147483643> | *<7>            <3> <2147483643> | <11>
<7> <5> <5>                        <7> <5> <5>
<4> <8>                            <4> <8>
<8> <-47> <-47>                    <8> <-47> <-47>
<5> <9>                            <5> <9>
<9> <-26> | <-27> | <-28>          <9> <-26> | <-27> | <-28>
```

Figure 2.10: Translating into BNF

```
<9> <-26>
<9> <-27>
<9> <-28>
```

Figure 2.11: Splitting rules – example

A grammar in this form is stored in the global table – it is common for all the grammar files. Since each grammar file receives its own interval of indexes for nonterminal indexing, there are no naming conflicts. Each grammar actually represents a namespace.

Finalizing

When all the grammar files are loaded into the table, user-defined bindings between nonterminals from different files are set by adding the appropriate grammar rules. Then the start nonterminal is set by adding the rule $X \rightarrow$

⟨user-defined start nonterminal⟩, where X is the new nonterminal with index 1. As the matter of the fact, the resulting grammar is an *augmented grammar* as defined in [1], page 222.

Finally, all the unreachable nonterminals and rules are removed using the algorithm 2.1. The output of the preprocessing part is finally a table containing the rules and a table mapping nonterminal indexes back to their names.

Input: A grammar G and its start nonterminal X
Output: The grammar G with unreachable nonterminals removed

$i = 0$;
$V_i = \{X\}$;
repeat
    $i = i + 1$;
    $V_i = V_{i-1} \cup \{B|\text{there is a rule } A \rightarrow \alpha B \beta \text{ in } G \text{ for some } A \in V_{i-1}\}$;
until $V_i = V_{i-1}$;
Remove all the nonterminals that are not in V_i from G;
Remove all the rules containing nonterminals that are not in V_i from G;

Algorithm 2.1: Removing unreachable nonterminals

Description of the output

At the end of the preprocessing part the grammar resulting from the input grammars is stored as a multimap mapping left sides of rules to vectors of right sides. The data type used is `int`. Hence it is necessary to explain the meaning of certain values.

- Numbers from the interval $[-255; 0]$ are *terminal symbols* – characters. A number $-c$ stands for the character with an ASCII code c.
- Any number from the interval $[1; \frac{\texttt{INT_MAX}}{2})$ is an *ordinary nonterminal symbol*.
- Numbers greater than $\frac{\texttt{INT_MAX}}{2}$ are *marked nonterminal symbols*. The actual index (i.e. the index of the ordinary nonterminal with the same name) of a marked nonterminal n is `INT_MAX` $-n$.
- Nonterminal with the index 1 is the global start nonterminal. It never appears on the right side of any rule.

- An empty vector stands for an ϵ-right side.

2.3 GLALR Table Generation

Due to the parsing algorithm used it is necessary to compute a GLALR parsing table. The procedure takes a grammar in the form produced by the Preprocessing part as an input and produces a pair of tables: $action[state, symbol]$ and $goto[state, symbol]$ with standard meaning defined in [1], pages 216 – 220. Since the usage of the parser is not restricted to LALR grammars, each entry in the $action$ table may consist of multiple actions. The $goto$ table can never contain multiple entries within a single cell – it comes from the way it is computed.

The LALR table is computed in the way described in [1], section 4.7. Then it is modified in accordance with [11] to become the GLALR table.

Similarly to the Preprocessing part, the process consists of several separate steps, all of which are implemented in `LalrTable` class. Now I'll go through them one by one. In this part the marking of nonterminals is completely ignored (however, the information has to be maintained); therefore I'll use the following notation instead of the actual way the grammar is stored:

- Nonterminal symbols are written as upper-case letters early in the alphabet.
- Upper-case letters late in the alphabet represent either nonterminals or terminals.
- Lower-case letters are terminal symbols.
- Lower-case Greek letters represent strings of grammar symbols (terminal and nonterminal).

Precomputations

Before the actual table computation begins, it is necessary to precompute several sets of (non)terminal symbols. For each nonterminal A the following sets are computed:

- $FIRST(A) = \{a | A \Rightarrow^* a\beta\}$. Here, a is either a terminal symbol or an empty word (ε).
- $NONTERMINAL_FIRST(A) = \{B | A \overset{rm}{\Rightarrow^*} B\beta\}$

- $NONEPSILON_FIRST(A) = \{a | A \overset{rm}{\Rightarrow}^* a\beta$ where the last step does not use an ε-production$\}$
- $ENF(A) = \{(B, \overline{F}) | A \Rightarrow^* B\gamma \wedge \overline{F} = \bigcup FIRST'(\gamma)$ for all the possible $\gamma\}$ where ENF stands for *Extended Nonterminal First* and

$$\begin{aligned} FIRST'(\varepsilon) &= \{\varepsilon\} \\ FIRST'(a) &= \{a\} \\ FIRST'(A\beta) &= FIRST(A) \oplus_1 FIRST'(\beta). \end{aligned}$$

and

$$U \oplus_1 V = \begin{cases} (U \setminus \{\varepsilon\}) \cup V & \text{if } \varepsilon \in U \\ U & \text{otherwise} \end{cases}$$

I think this algorithm is not commonly known and so I'll show it here as algorithm 2.2.

The symbol $\overset{rm}{\Rightarrow}^*$ stands for the *rightmost* derivation.

Input: A grammar G and the $FIRST$ set for each nonterminal
Output: The ENF set for the grammar G

```
foreach nonterminal A do
    ENF(A) = {(A, {ε})};

while ENF(X) has changed for some X do
    foreach rule of the form A → Bγ do
        f = FIRST'(γ);
        foreach (C, Q) ∈ ENF(B) do
            if (C, R) ∉ ENF(A) for any R then
                if ε ∈ Q then
                    add (C, (Q \ {ε}) ∪ f) to ENF(A);
                else
                    add (C, Q) to ENF(A);
            else
                if ε ∈ Q then
                    R = R ∪ (Q \ {ε}) ∪ f;
                else
                    R = R ∪ Q;
```

Algorithm 2.2: Extended nonterminal first

Computation of LR(0) items

Since the efficient construction of LALR table is used ([1], page 240), sets of LR(0) items can be represented by their kernels. Then, given a set of items I and a nonterminal X, the kernel of $goto(I, X)$ can be computed as follows:

- If $B \to \gamma \cdot X\delta$ is in I, then $B \to \gamma X \cdot \delta$ is in the kernel of $goto(I, X)$.
- Item $A \to X \cdot \beta$ is also in the kernoel of $goto(I, X)$ if there is an item $B \to \gamma \cdot C\delta$ in the kernel of I and $C \in NONTERMINAL_FIRST(C)$.

As the algorithm is not given in [1], I will introduce it myself as algorithm 2.3.

Input: A grammar G with the start nonterminal S' such that S' only appears in the rule $S' \to S$
Output: The kernels of the LALR(0) collection of sets of items K and the transition function $goto$

```
K_0 = {S' → S};
s = 0; // the index of the processed set of items
num = 1; // the number of sets of items
while s ≠ num do
    gotos(X) = ∅ for all grammar symbols X;
    foreach l ∈ K_s do
        if l = A → β · Xγ then
            gotos(X) += A → βX · γ;
            if X is a nonterminal symbol then
                foreach C ∈ NONTERMINAL_FIRST(X) do
                    foreach C → Yγ ∈ G do
                        gotos(Y) += C → Y · γ;
    foreach grammar symbol X do
        if gotos(X) ≠ ∅ then
            if K_n = gotos(X) for some n then
                goto(s, X) = n;
            else
                goto(s, X) = num;
                K_num = gotos(X);
                num += 1;
```

Algorithm 2.3: Computing LR(0) items

The result of this stage is finally a collection of sets of LR(0) items and the $goto(i, X)$ table. The collection is stored as the vector `m_items`, where each entry in the vector is a set of LR(0) kernel items. Sets of items are referenced by their indexes in the `m_items` vector. The *goto* table is stored as a two-dimensional vector `m_goto`.

Computation of lookaheads

The data from `m_items` are first copied into the `m_ext_items` structure, which stores the same information as `m_items` and also the set of lookahead symbols for each item. Then the algorithm from [1], page 242 – 243 is used for computing the kernels of the LALR(1) collection of sets of items.

Building the table

The $goto$ table is computed during the LR(0) items computation. The task is now to compute the $action$ table. The method is described in [1], page 240. The modification of the table from [11], page 39 is applied since it is required by the parsing algorithm. It has shown to be effective to incorporate the modification right into the original algorithm.

The parsing actions can be computed from the kernel items alone. The following list summarizes the rules used for computing the table. The table is filled separately for each state (set of kernel items). Let us consider the computation for the set of items I is being performed and the number of I is i.

- We shift on input a if there is a kernel item $[B \to \gamma \cdot X\delta, b]$ in I. where X is a nonterminal and $a \in NONEPSILON_FIRST(X)$ or $X = a$. Then the entry $shift(goto[i, a])$ will be in $actions[i, a]$.
- Reduction by a grammar rule $A \to \alpha \cdot \gamma$ will be called on input c if the item $[A \to \alpha \cdot \gamma, c]$ is in I and $\gamma \Rightarrow^* \varepsilon$. The length of the reduction is $|\alpha|$. It is required that $\alpha \neq \varepsilon$. This holds for all the kernel items with the exception of $S' \to \cdot S$. The current implementation allows also $\alpha = \varepsilon$. Although it introduces unwanted ambiguity, it does not affect the result of parsing in any way.
- Reduction by $A \to \alpha$ where $\alpha \Rightarrow^* \varepsilon$ is called on input a iff there is a kernel item $[B \to \gamma \cdot C\delta, b]$ in I such that

  $$(A, p) \in EXTENDED_NONTERMINAL_FIRST(C) \text{ and } a \in p.$$

  The length of such reduction is 0.

Description of the output

The entry $actions[s, a]$ in the $actions$ table consists of a set of actions possible in the state s with the lookahead symbol a. An action is either "shift" or "reduce". For the "shift" action the state to go after the shift is stored. For the "reduce" action the number of the grammar rule to reduce by and the length of the reduction is stored. Both s and a are positive integers and $a \in \{0, \ldots, 255\} \cup \{\$\}$ where the numbers represent ASCII-codes and $\$$ stands for the end of input.

The entry $goto[s, x]$ in the $goto$ table contains the state to go after reading the grammar symbol x when in state s. x can be either terminal or nonterminal. The way the entries are accessed is a bit confusing.

- $x \in \{0, \ldots, 255\}$ stands for a *terminal symbol* with an ASCII-code x.

- $x \geq 256$ denotes a *nonterminal symbol* with an index $x - 256$. It should be noted that $x \neq 256$ since there is not the nonterminal with the index 0.

2.4 GSS

GSS is an abbreviation for *Graph Structure Stack*. It provides an efficient way of simulating nondeterminism in the standard LALR parsing algorithm.

The simple way of simulating nondeterminism is using multiple LALR stacks. Then for n possibilities there are n stacks. This method is, however, inefficient because the nondeterminism is often local. The idea of GSS builds on *sharing* nodes among the stacks.

GSS is a directed acyclic graph where each directed path is a stack. Similarly to a an LALR stack there are two types of nodes: *state* nodes, which are labelled with states of the LALR automaton, and *symbol* nodes, which are labelled with grammar symbols.

Each symbol node contains a grammar symbol, the set of strings representing possible ways of derivation of the grammar symbol (*semantic strings*) and the set of successors of the node (all of which are state nodes).

Each state node contains the number of the state, the set of successors of the node and the *level* within the GSS – the distance from the initial state node measured in the number of characters read from the input.

Detailed description can be found in [11], pages 5 – 12.

2.5 Parser

The final part of the library deals with actual parsing strings supplied by a user. The parser implements the slightly modified parsing algorithm from [11], pages 41 – 42. The only result obtained by the original algorithm is the answer *accept* or *reject*, which is not sufficient. Therefore I had to modify certain parts of the algorithm.

Storing pending reductions Pending reductions are stored at the moment a new state node is created or a new path to a state node is created. For a reduction by $X \to \alpha$ in the state q the triplet $(p, X, |\alpha|)$ is stored in the original algorithm where p is the (new) successor of q. My algorithm stores the four-tuple $(p, i, |\alpha|, l)$ is stored instead where p has the same meaning, i is the identifier of the grammar rule $X \to \alpha$ and l is the identifier of the symbol node between q and p.

Symbol nodes I decided to to augment symbol nodes (which have no role in the original algorithm) by adding the set of *semantic strings* to each of them. Let z be the symbol node with label X between state nodes with levels i and j (where $i \leq j$) and w_{ij} be the substring of the parsed word between i and j. Then the set of semantic strings in z contains strings representing possibilities of the derivation $X \Rightarrow^* w_{ij}$. Semantic strings are discussed in detail later.

Sharing symbol nodes In the original algorithm symbol nodes are shared with no regard of their label. In my version symbol nodes are shared only if their labels are the same.

Searching for the reachable nodes When the reduction by $X \to \alpha$ is set to be performed in the state node p, it must be applied down all paths from p of length $2|\alpha|$. The original algorithm only finds the state nodes which can be reached from p along a path of of length $2|\alpha|$. My algorithm does the same and also collects the semantic information from the symbol nodes along each path. The semantic strings from the nodes (whose labels actually form α) are then used for creating the semantic string for X.

Semantic strings

As further described in section 3.4, the user can mark nonterminals on the right side of the rule. Then the part of the parsed word derived from the

marked nonterminal should be marked in the output string. Here I'll show how it is accomplished.

First, we will suppose the used grammar is LALR(1) and therefore there is no ambiguity. Any symbol node whose label is a terminal symbol a has the set of semantic strings $\{a\}$. Next, since the grammar is LALR(1), the set of semantic strings of a symbol node whose label is a nonterminal contains exactly one element. When applying reduction by $A \to XYZ$ from the state node p, exactly one way is found. It passes through symbol nodes with labels Z, Y, X in the process. Let $\{s_Z\}, \{s_Y\}, \{s_X\}$ be the sets of their semantic strings. If none of X, Y, Z is marked in the rule $A \to XYZ$, the set of semantic strings for A is obviously $\{s_X s_Y s_Z\}$. If any of the symbols is marked, for example Y, the string $\langle Y \rangle s_Y \langle /Y \rangle$ is used in place of s_Y. Let g be the state to go after the reduction. Then the result is stored in the symbol node with the label A.

Now we will move on to ambiguous grammars. Terminal symbols are treated in the same way. Let us consider the reduction by $A \to XYZ$ once again. The ambiguity may show in the following ways:

- More than one state node is found when searching for the possible paths of the length $2|XYZ|$. Then each of them is processed individually.
- The set of semantic strings of some of X, Y, Z contains more than one string. Let $\{r_X, s_X\}$ be the set for X. Then the string

  $$\langle\text{ambiguity}\rangle\langle\text{way}\rangle r_X \langle/\text{way}\rangle\langle\text{way}\rangle s_X \langle/\text{way}\rangle\langle/\text{ambiguity}\rangle$$

  representing a "local ambiguity" is generated for the symbol node with the label X. If X is marked, the above string is enclosed by $\langle X \rangle$ and $\langle /X \rangle$. The resulting semantic string for A is then the concatenation of the strings of X, Y and Z.
- The suitable symbol node with the label A is already present in the GSS. In this case newly computed semantic string is added to the set of the node.

Finally, after finishing, the result of parsing is the set of semantic strings of the symbol node with the label S' between the initial and the accepting state.

Chapter 3

Using the Parser

The parser is implemented as a C++ library. It can be used both in UNIX and Windows environments. After installing the library the file `BnfParser2.h` is to be included. Then the new data type `BnfParser2` is available. This class implements the API of the parser.

Bnfparser[2] is a generic runtime-generated parser capable of parsing any context-free grammar. It is able to use grammars spanning over multiple files and written using different syntax. Then, for a given word, a user is given an answer whether the word is syntactically correct. He/she may select certain parts of the word to be specially marked accordingly to the grammar. In this chapter I will describe the steps needed to run the parser and the way the input should be set to obtain the desired result.

3.1 Application Interface Class

The API of the parser the `BnfParser2` class. The class itself is a wrapper class and calls `Parser`'s methods. This approach has been used in order that the parser may be used as a library. Without a wrapper class, copying all the header files to the system folder would be required to *use* the parser, which is unwanted. The `BnfParser2` class solves the problem by referencing internal header files only in the cpp file. The header file does not contain any #includes and so it is the only file to be copied to the system folder.

3.2 Function-call Sequence

Here I'll show the typical sequence of function-calls when using the parser. The detailed description of the functions is in section 3.6

1. First, `add_search_path()` is used for setting the used directories.
2. Then `add_grammar()` is called to load a syntax specification. It may be called multiple times.

3. `add_referenced_grammars()` can be called to load the grammars that are needed, but not loaded yet.

4. After loading grammars, `set_start_nonterm()` is called to set the start nonterminal.

5. `build_parser()` processes the specifications and builds the parser.

6. `parse_word()` is called to parse a word.

7. `get_error_position()` or `get_semantic_string()` are called to obtain results of the parsing.

Steps 6 and 7 can be repeated multiple times.

3.3 Syntax-description File

Syntax-description file is used by the preprocessing part, which translates grammar files to BNF. The file thus describes the syntax of the grammar file it is loaded with.

The file consists of two parts; the first part describes the basic syntactic rules, the second part allows user to define new (meta)operators he/she wants to use within the grammar file.

Basic rules

The first part of the file is strictly defined. Each line begins with a keyword followed by =. The rest of the line is then the value corresponding to the keyword. I'll describe the meaning of the elements using the syntax-description for ABNF [3] augmented by C-like comments (`/**/`).

```
rulename=[a-zA-Z][a-zA-Z0-9\-]*
```

The value is the standard *Perl regular expression* describing the format of nonterminals. In ABNF nonterminals have to start with a letter and next may contain letters, digits and hyphens.

```
defined==
```

The value is the *string* used for dividing a grammar rule into a left-hand side and a right-hand side. In ABNF [3] rules have the form $X = \gamma$.

`terminal="`

The value represents the *string* terminal strings are enclosed by. In ABNF strings have `"this form"`.

`comment=;`

The *string* introducing line comments (// in C++). It may be left blank in the case the syntax does not allow line comments.

`concat=`

The *string* describing the concatenation mark. If it is whitespace only, the value may be left blank. ANBF [3] uses whitespace ($A = B\ C\ D$) and EBNF [5] uses dots ($A ::= B.C.D$).

`alternative=/`

The *string* describing the alternative mark. In ABNF we have $A = (B/C)\ D$

`leftgroup=(`

The *string* representing the opening group parenthesis.

`rightgroup=)`

The *string* representing the closing parenthesis.

`leftcomment=/*`

The *string* representing the beginning of a comment block.

`rightcomment=*/`

The *string* representing the end of a comment block.

`allbrackets=()`

All the characters used as parentheses are listed here in an arbitrary order. This also applies for the characters used as parentheses with custom operators.

`casesensitivestring=false`

The value must be either true or false. It is set to true iff the text enclosed by the `terminal` elements is to be parsed with respect to the case of characters. In ANBF strings are case-insesitive; the rule $A = "aa"$ actually yields four possibilities: $A = "aa"$, $A = "Aa"$, $A = "aA"$ and $A = "AA"$.

`casesensitiverulename=false`

Determines whether the naming of nonterminals is case-sensitive. In ABNF XY, xY, Xy and xy stand for the same nonterminal.

Custom operators

After the basic settings there must be one line with the text `OPERATORS`. It must be present even if there are no operators defined. Then, there may be any number of custom operators defined, one per line.

Operator syntax

Generally, each operator has the form `pattern=definition`. The pattern may consist of any characters and can contain one or more *parameters*. There are three types of parameters available:

Number – string of digits having the meaning of a number. Thanks to the pseud-lexical analysis digits inside nonterminal names or terminal symbols are not taken into account.
Syntax: `\n<number of the parameter>`

Element – exactly one grammar element in the certain context (terminal, nonterminal or an expression enclosed by grouping parentheses)
Syntax: `\e<number of the parameter>`

String of elements – one or more *elements* (as described above) joint by either as a concatenation or an alternative.
Syntax: `\<number of the parameter>`

The reason for introducing three types of operators is explained in section 2.2. Here, I'll show how to use the parameters.

- Numbers are typically used for defining operators like `n*` in EBNF. Here, the number parameter is used for matching the number, so we get `\n1*` as the first part of the definition.

- If the defined operator *does not use* its own type of parentheses (it uses standard grouping parentheses), the Element parameter type has to be used. The above mentioned operator from EBNF is of that kind. The full pattern for the operator is therefore \n1*\e2.

- Finally, if the defined operator *uses* its own parentheses, the last type of operator is used. For example, when defining the "one-or-more" operator in EBNF, the pattern is {\1}-.

The numbering of parameters must be done manually – there is no control. The numbers have to be in an ascending order, no matter which type of operator is used.

All the parameters are able to match even if there is a whitespace around the element. The whitespace *should not* be present within the pattern unless it is mandatory for the operator defined.

The *definition* of an operator is written in ABNF as a one-line expression. All the parameters from the pattern are used *without* specifying the type. The format is always \<number>. As an example I'll now show the definition of some operators.

\e1?=[\1]	Optional element
#(\1)=*\1	0 or more occurrences
[\1,\2]=\1 *(\2 \1)	List of the first parameter with the second parameter as a separator

The operators are evaluated with the priority determined by the order they are written in the syntax-description file.

Limitations

- There must not be any explicitly written nonterminals or terminals within an operator. Only generic syntax constructions may be used. Therefore the # operator from ABNF defined in [6] cannot be used.

- Since the full lexical analysis is not performed, it is sometimes necessary to adjust the order of operators in the file. With two operators of the form #[\1] and [\1] the expression #[nonterminal] is expanded using [\1] if this operator is written before the other.

- There can be some conficts between nonterminal names and operators. For example, in XBNF the operator O(\1) won't work properly since O is a valid name for a nonterminal. Therefore the expression will be seen as <nonterminal>(<string of elements>) during the operator evaluation and not as O(<string of elements>).

- The line should contain only one = character. This limitation is likely to be removed as soon as possible.

3.4 Grammar File

A grammar file contains the grammar written with accordance to the syntax described in the corresponding syntax-description file. It may also contain ABNF-style terminal symbols (e.g. %x0D) As a result, a grammar file *cannot* contain operators colliding with these terminal symbols.

There can also be additional tags within the comments.

- The reference to the syntax-description file can be written as

  ```
  !syntax("name_of_the_file")
  ```

  Only the first occurrence of the tag is used. The tag is used only if the syntax-description file is not specified in add_grammar procedure. The name must be written without the .conf extension.

  Since the format of comments is not known at the time this tag is searched for, it should be used with care. If there happens to be a terminal string of the above form, it could confuse the program – it would likely report a nonexistent syntax-description file.

- If the grammar uses nonterminals defined in a different file, the relations among them have to be established manually; it cannot be done automatically since each grammar file gets its own namespace and thus there might be duplicate nonterminals that are not to be linked.

  The tag can only be used within *line comments*. Its syntax is

  ```
  !import(1*("<source>" [as "<target>"] ,) "file")
  ```

  where <source> is the name of the nonterminal in the file it is imported from and <target> is the name of the nonterminal in the file it is imported to. If <target> is not specified, it is assumed that <target>=<source> The case-(in)sensitiveness is used accordingly to the syntax-description files for the involved grammar files. Multiple nonterminals may be imported using one tag. For example:

  !import(”ALPHA” as ”<letter>”, ”DIGIT” as ”<digit>”, ”2234.abnf”)
  !import(”telephone-subscriber”, ”rfc2806-2.abnf”)

Marking nonterminals

Any nonterminal on the right side of a rule can be marked. The purpose of marking is explained in section 3.5. Marking a nonterminal is done by writing the `@` symbol *behind* the nonterminal. Nonterminals on the left side, terminals or expressions in parenteheses *cannot* be marked. The grammar can be, however, modified so that nearly the same result is obtained, as shown in the following examples. Having the grammar rule

```
A = (B / C) D / "string"
```

the nonterminal `B` can be marked by writing

```
A = (B@ / C) D / "string"
```

To mark `A` we have to write

```
A = A'@
A' = (B / C) D / "string"
```

Marking the string can be achieved by writing

```
A = (B / C) D / S@
S = "string"
```

and finally the sequence in the parentheses is marked in the following way:

```
A = P@ D / "string"
P = B / C
```

If a new nonterminal is used for marking a part of the rule, the name of the new nonterminal is used for the purposes of marking.

3.5 Semantic Strings

In this section I will describe the purpose of marked nonterminals and the meaning of semantic strings returned by the parser. There are three types of information stored in a semantic string: the characters of a parsed word, marking of certain parts of the word, and ambiguity marks.

Each syntactically correct word is derived from the starting nonterminal using the grammar rules. Any time a rule with a marked nonterminal is used, the part of the word derived from that nonterminal is marked – it is enclosed by the tags $\langle A \rangle$ and $\langle /A \rangle$ where A is the name of the marked nonterminal. The tags may be nested since the part of the word inside a tag can be also derived using rules with marked nonterminals.

If a part of a word can be derived in multiple ways and *the semantic strings of the ways differ*, an ambiguity is recorded. The part of the word is eclosed by the tags ⟨ambiguity⟩ and ⟨/ambiguity⟩. Then each possible semantic string is marked by ⟨way⟩ and ⟨/way⟩ tags and all the possibilities are listed.

Following simple examples demonstrate semantic strings and marking in practice.
Grammar:
```
S -> (A / B) C
A -> "a"
B -> "a"
C -> "b"
```
Parsed word: ab; semantic string: ab
Grammar:
```
S -> (A@ / B) C
A -> "a"
B -> "a"
C -> "b"
```
Parsed word: ab; semantic string:
<ambiguity><way><A>a</A></way><way>a</way></ambiguity>b
Grammar:
```
S -> A B@
A -> "" / "a" A@
B -> "b"
```
Parsed word: aab; semantic string: a<A>a<A></A></A><B>b</B>
Grammar:
```
S -> "a" S@ "b" / "c"
```
Parsed word: aacbb; semantic string: a<S>a<S>c</S>b</S>b

Finally, I'll give a formal description of the semantic strings. Let w be the parsed word, S the start nonterminal and $A \to B@C$ a grammar rule where the nonterminal B is marked. Suppose further that

$$S \Rightarrow^* \alpha A \beta \Rightarrow^* \alpha B\ C \beta \Rightarrow^*$$

and

$$\begin{aligned} \alpha &\Rightarrow^* w_1 \\ B &\Rightarrow^* w_2 \\ C\beta &\Rightarrow^* w_3 \end{aligned}$$

where $w_1 w_2 w_3 = w$. Then the semantic string has the form

$$w'_1 \langle B \rangle w'_2 \langle / B \rangle w'_3$$

where w'_k is w_k possibly with some other marking.

Next, let X be a nonterminal such that

$$S \Rightarrow^* \alpha X \beta \begin{matrix} \nearrow & \alpha\rho\beta & \searrow \\ \searrow & \alpha\phi\beta & \nearrow \end{matrix} \Rightarrow^* \alpha w_2 \beta \Rightarrow^* w_1 w_2 w_3$$

where $\rho \neq \phi$. Obviously $\rho \Rightarrow^* w_2$ and $\phi \Rightarrow^* w_2$. Let $w'_{2\rho}, w'_{2\phi}$ be w_2 as marked during derivation from X using the ρ- and ϕ-way respectively. The resulting semantic string will be

$$w'_1 w'_{2\rho} w'_3$$

if $w'_{2\rho} = w'_{2\phi}$ and

$$w'_1\langle\text{ambiguity}\rangle\langle\text{way}\rangle w'_{2\rho}\langle/\text{way}\rangle\langle\text{way}\rangle w'_{2\phi}\langle/\text{way}\rangle\langle/\text{ambiguity}\rangle w'_3$$

otherwise, where w'_1, w'_3 have the same meaning as above. Therefore if no marking is done within an ambiuous part of the derivation, the ambiguity is not recored, which is actually good – since the user has not marked any rule involved in the derivation, he/she is not interested in that part. If there are more than two possibilities, all the distinct ones are listed in the same fashion.

3.6 Description of the Methods

void add_search_path(const char *path)

Adds a new directory path to be searched when the parser is looking for grammar files or syntax-description files. Multiple calls are allowed. The path may be written either absolutely or relatively.

void add_grammar(const char *syntax_name, const char *variant_name)

Loads a grammar file and its syntax-description file. Multiple calls are possible. The grammar file is searched in the working directory and the directories specified by the add_search_path method. The same applies for the syntax-description file but only the subdirectory `syntax` of each specified directory is searched. If the syntax-description parameter is omitted, the file is determined from the `!syntax` tag within the grammar file.

Each file gets its own namespace for nonterminals and the relationship among nonterminals from different grammar files has to be set using `!import` tags in the grammar files.

The extension of syntax-description files is `.conf`. Names have to be passed without the extension to the procedure.

void add_referenced_grammars(void)

If a grammar file that has not been added by calling `add_grammar` procedure is referenced from some grammar file, a warning is generated after calling `build_parser` method. To add all such files at once this method is called.

void set_start_nonterm(const std::string& start_name, const std::string& start_grammar_name)

Sets the global start nonterminal for all the added files. It takes the name of the nonterminal and the name of the file it is defined in as its parameters. It may be called even before the specified file is loaded. The name of the grammar file is arbitrary; when omitted, the file containing the specified nonterminal is found automatically.

void build_parser(void)

Processes the set of grammar files previously loaded. First, the start nonterminal is set, then dependencies among the files are resolved, unreachable nonterminals are removed. Finally, the parsing table is built. This procedure may take up to a few seconds.

bool parse_word(const std::string& word)

This procedure performs the actual parsing. The parameter is the word to be parsed. It returns true or false depending on whether the word is syntactically correct with respect to the grammar previously loaded. The procedure must not be called before `build_parser` procedure.

bool get_parsing_result(void)

Returs the value returned by the last call of the `parse_word` method.

unsigned get_error_position(void)

If the result of the last parsing is false, this method returns the position at which the parsing failed.

std::string get_semantic_string(void)

If the last parsing action is successful, it returns the semantic string generated during the parsing. The meaning of a semantic string is described in

section 3.5.

void set_verbose_level(unsigned level)

For debugging purposes: set verbosity to a given level. Only messages of equal or higher importance will be printed. The following importance levels are defined:

0	system is unusable
1	action must be taken immediately
2	critical conditions
3	error conditions
4	warning conditions
5	normal but significant condition (default level)
6	informational
7	debug-level messages
8	function call tracing

int get_verbose_level(void)

Returns the currently set verbosity.

Chapter 4

Further Improvements

4.1 Semantic String Generation

The current implementation does not work properly with certain grammars with right nullable rules, for example:

$$
\begin{aligned}
S' &\rightarrow S \\
S &\rightarrow aB \\
B &\rightarrow C\text{@} \\
C &\rightarrow \varepsilon
\end{aligned}
$$

Here, the reduction is allowed in the state with the item $S \rightarrow a \cdot B$ and so the semantic string of B (which should be $\langle C \rangle \langle / C \rangle$) is not recorded. It seems that further modifications to the algorithm have to be made in order that all semantic strings are properly generated.

4.2 Speedup

There are two major areas in which the speedup seems to be possible.

Building the parser – building the GLALR table can be faster if bit-sets are used instead of std::set for storing sets of terminal symbols (i.e. fixed range of numbers). The computation of the table for the very large SIP grammar takes about 15 seconds at the moment.

Semantic string generation – parsing is extremely slow for complicated grammar files and long parsed words (parsing long SIP [9] messages takes up to a few minutes). The reason is the way semantic strings are generated. Any time a reduction is performed, the corresponding part of the input is copied into another place. Since the length of each reduction is constant, there are $O(n)$ reductions performed when

parsing a word of the length n. Because strings are copied during each reduction, the complexity of the entire process is $O(n^2)$.

There is actually no need to store input characters within semantic strings. It seems to be sufficient to store only the left end and the right end of each portion of input text present in a semantic string, as shown in the example.

The input word (letter-spaced) and the indexes of its letters:

```
s  a  m  p  l  e  i  n  p  u  t  w  o  r  d
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
```

A possible semantic string:

```
sample<C>input</C>word
```

The way it is stored:

```
[0;5]<C>[6;10]</C>[11;14]
```

I believe this modification will significantly increase the parsing speed.

4.3 Special Constructions in Grammar Files

Since ABNF [3] is used as a core syntax, there are some umsupported special constructions. I would like to implement syntactic-exception from EBNF [5] by introducing a new operator and making it usable within syntax-description files and the support for nonterminal names significantly different from ABNF (e.g. multiple words with whitespace).

Next, the `#` operator from [6] cannot be used due to the way operators are evaluated. Grammar rules newly created during the operator evaluation are not fully processed; therefore no grammar symbols may be used within it. However, the definition of `#` requires both symbols:

```
#\e1=(*LWS \1 *( *LWS "," *LWS \1))
```

The preprocessing part has to be slightly modified in order that the definition of such operators is possible.

Chapter 5

Conclusion

My task was to design and implement a runtime-generated parser driven by BNF-like grammars. This document describes such a tool – the C++ library bnfparser^2. It was given the name bnfparser^2 because of the way it works; First, a grammar is *parsed* in accordance to a given syntax specification, then words are *parsed* in accordance to that grammar.

Thanks to its generality it can be easilly used as a quick reference tool for telecommunication protocols – e.g. the syntax of SIP [9] messages is defined using ABNF [3]. Since context-free languages have great importance throughout the computer world, the parser seems to be a very versatile tool not restricted to internet protocols. It could also serve as an educational tool.

As the input format of a grammar is arbitrary, the first part of the library performs translation into simple BNF – this part is my authorial work (coauthored by Petr Slovák). I would like to highlight the process of operator evaluation. I've come up with the algorithm that uses regular expressions. The algorithm allows the evaluation of operators without syntax analysis of grammar files.

Once a grammar is translated, a GLALR table is computed. The process involves several steps; all the steps are well documented, but mostly not their implementation. Therefore I had to design my own algorithms for certain precomputations and also for the computation of a collection of sets of LR(0) items.

When a GLALR table is ready, it is possible to parse words. The parser uses the GLALR parsing algorithm. I had to slightly modify the original algorithm so that maximum amount of information is gained during the parsing process; the original algorithm only allows to obtain the answer whether a word is syntactically correct. My version also provides the information about the way a word is derived.

The parser has been tested using SIP messages; it has shown that the performance is not satisfactory. A longer SIP message (thousands of characters) takes up to a few minutes to be parsed. Thence increasing the speed is going to be the first improvement made.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, techniques, and tools*, Addison-Wesley Publishing Company, 1986.

[2] John W. Backus et al., *Report on the algorithmic language ALGOL 60*, Communications of the ACM **3** (1960), no. 5, 299–314.

[3] Dave Crocker and Paul Overell, *Augmented BNF for syntax specifications: ABNF*, RFC 4234, 2005.

[4] Paul Eggert and Akim Demaille, *Bison – GNU parser generator*, http://www.gnu.org/software/bison.

[5] *Information technology – Syntactic metalangue – Extended BNF*, ISO/IEC 14977:1996(E), 1996.

[6] Roy T. Fielding et al., *Hypertext transfer protocol – http/1.1*, RFC 2616, 1999.

[7] Philip Hazel, *PCRE – perl compatible regular expressions*, http://www.pcre.org/.

[8] Terence Parr, *ANTLR Parser Generator*, http://www.antlr.org.

[9] Jonathan Rosenberg, Henning Schulzrinne, Gonzalo Camarillo, Alan Johnston, Jon Peterson, Robert Sparks, Mark Handley, and Eve Schooler, *SIP: Session initiation protocol*, RFC 3261, 2002.

[10] Alex Rousskov, *Hapy parser generator*, http://hapy.sourceforge.net.

[11] Elizabeth Scott, Adrian Johnstone, and Shamsa Sadaf Hussein, *Tomita-style generalised LR parsers*, Tech. Report CSD-TR-00-A, Department of Computer Science, Royal Holloway University of London, 2000.

[12] Robert J. Sparks, *The session initiation protocol (SIP) refer method*, RFC 3515, 2003.

Appendix A

Content of the CD

thesis.pdf	the pdf version of this document
thesis.zip	the archive containing the LaTeX source of this document
bnfparser2-0.1.0.tar.gz	the latest release of bnfparser[2]

Description of the content of the release can be found in the README file.