Multicore SIP

Ivan Brešković, Vedrana Janković, Zvonimir Pavlinović, Željko Rumenjak

Zagreb, October 2009

Table of Contents

1 Summary

The goal of the Multicore SIP parser project was the development of a functional SIP parser that efficiently utilizes advantages of a multicore processor. The Multicore SIP parser is generated via developed parser generator and is 100% SIP compliant, which is significantly better in comparison to the tested open source SIP stacks. Multicore parsing is implemented in two ways: by executing multiple parser instances, one per core, and by splitting the SIP message into independent segments and parsing them on different cores. Performance measurements have shown 3.4 times performance gain on multicore processors in comparison to singlecore performance of the Multicore SIP parser, which is significantly better in comparison to the tested open source solutions, which have an average performance gain of 1.5 times. The developed parsers are consisted of lexical and syntax analyzers, which are interconnected in order to dissolve grammar ambiguities and conflicts, and can easily be extended with semantic actions. Additionally, tools for the ABNF to BNF grammar conversion, as well as the parsing tree visualizer were developed.

2 Acknowledgement

The work presented in this document was conducted as part of "Parallel message processing on multicore processors" prototype which is being implemented in Research department of Research and design center of Ericsson Nikola Tesla d.d. The authors of the document wish to acknowledge the help provided by Dr. Sc. Ivan Skuliber, the mentor assigned to the project.

3 Introduction

3.1 Lexical Analysis

Lexical analysis is the text analyzing process that is consisted of scanning the input message and grouping the characters into tokens [1]. Lexical analysis reads the input message mark by mark and breaks the message into tokens described by regular expressions. Each time a sign sequence is recognized by a regular expression, it is classified into one of the corresponding lexical classes. Lexical analysis tries to find the longest sign sequence for which a regular expression is defined. In some cases, depending on message specification, this solution is not satisfactory. In that case the lexical analysis sends every possible token to the syntax analyzer. After a token has been recognized and classified, its class and value form a new lexical unit called lexeme. Every lexeme is stored into the lexeme table. This table keeps the message structure for following processes because lexical analysis is the only process that operates with the original code. Some lexical analyzers save only the token class into the main table, but they also generate a table for each class in which they save token values and parameters needed for the following processes (e.g., semantic analyzer). This method is typically used in compiler implementation [3].

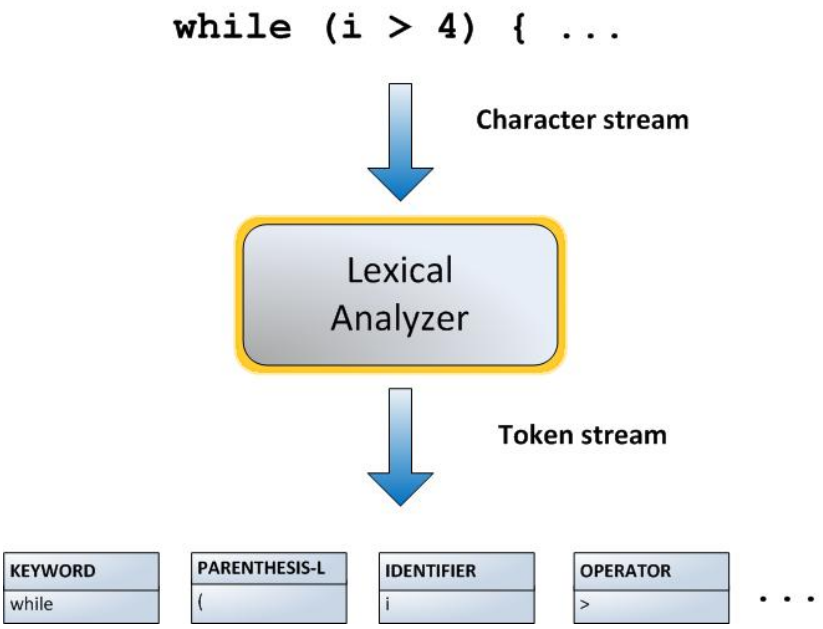

Figure 3-1 Lexical Analysis

The lexical analyzer needs to recognize lexical errors and, if possible, correct them. As its output, the lexical analyzer gives lexeme tables used as input for syntax analysis. The lexical analyzer can completely analyze the input message or it can be called as a function every time the syntax analysis needs a new token.

3.2 Formal Grammar

A formal grammar is a set of rules that define a formal language. These rules describe how to form strings that are valid according to language syntax. It is important to note that these rules do not describe the meaning of those strings [3].

The grammar is usually thought of as a language generator, but it can also be used to determine if a string belongs to the language that the grammar describes.

The formalization of generative grammars was first proposed by Noam Chomsky in 1956. According to him, a grammar G is formally defined as the ordered quad-tuple (N, Σ, P, S) [1]:

- a finite set N of nonterminal symbols,
- a finite set Σ of terminal symbols that is disjoint from N,
- a finite set P of production rules,
- a distinguished symbol S $\in$ N that is the start symbol.

When working with formal grammars, nonterminals are usually represented by uppercase letters, terminals by lowercase letters, and the start symbol by S.

Noam Chomsky also classified grammars into types now known as the Chomsky hierarchy. The difference between these types is that they have increasingly strict production rules and can express fewer formal languages [1]. Table 3-1 shows formal grammar types defined by Noam Chomsky with class of languages that each can describe, the minimal automaton required to recognize this type of grammar and the form its rules must have.

Table 3-1 The Chomsky hierarchy

Grammar	Languages	Automaton	Production rules (constraints)
Type 0	Recursively enumerable	Turing machine	$\alpha \to \beta$ (no restrictions)
Type 1	Context-sensitive	Linear-bounded non-deterministic Turing machine	$\alpha A \beta \to \alpha \gamma \beta$
Type 2	Context-free	Non-deterministic pushdown automaton	$A \to \gamma$
Type 3	Regular	Finite state automaton	$A \to a$ and $A \to aB$

Two of the most important types of formal grammars are context-free grammars (Type 2) and regular grammars (Type 3). Although much less powerful than unrestricted grammars (Type 0), these two types of grammar are often used because parsers can be efficiently implemented for them. For example, all regular languages can be recognized by a finite state machine, and there are well-known algorithms to generate efficient LL parsers and LR parsers for useful subsets of context-free grammars to recognize the corresponding languages those grammars generate [1]. The relation between different grammar types based on their ability to express different languages is shown on Figure 3-2.

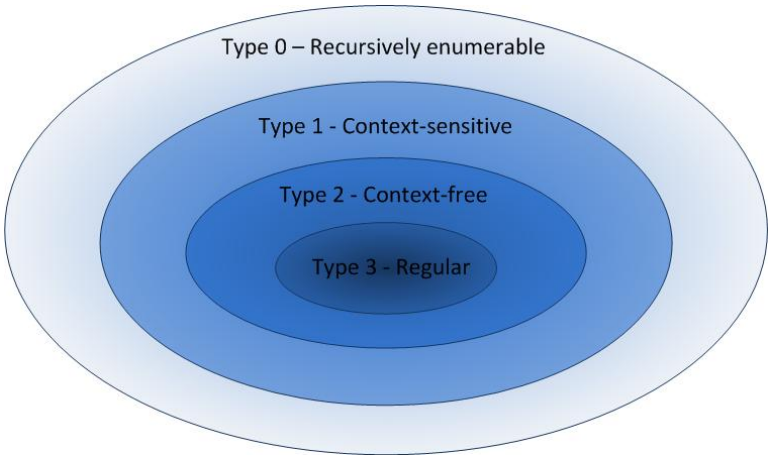

Figure 3-2 The relation between different grammar types from the Chomsky hierarchy

3.3 LR Parsing

LR parser reads the input file from left to right and produces the rightmost derivation. It performs bottom-up parsing because it constructs parse tree starting form tree leaves [2]. LR(k) parsers parse messages with *k* look ahead input symbols that are used in making parsing decisions. LR(k) parsers are used to parse context-free grammars, such as formal SIP grammar.

LR parser is built upon a set of rules that trigger actions in specific parser situation. The parser constructs a parse tree bottom-up and produces the *rightmost* derivation [3]. The rightmost derivation represents the replacement of the right side of the production with the left side of the production. For example, for the next grammar production:

(1) S -> a b c

the rightmost derivation would replace “S” with “a b c” where tokens “a”, ”b” and ”c” would represent the leaves of the tree. The sub tree for this derivation would look like as it is shown on Figure 3-3.

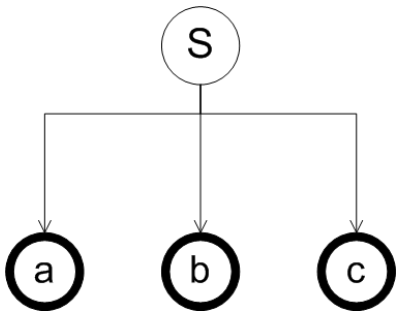

Figure 3-3 Parse tree produced from rightmost derivation of production S-> a b c

The parser consists of three major elements that make action decisions: *input message*, *parsing table* and *stack*.

The input message no longer represents the original message, but the list of tokens which are the result of lexical analysis [3].

Parsing table consists of a set of actions to be executed in a specific parser situation. Decision of what action to trigger is made upon parser state, which is a simple number and is situated on the top of the stack, and an input token. The actions are `ShiftX`, `ReduceX` , `GotoX` and `Accept` [3]. The `ShiftX` action pushes the input token on the stack and then puts ”X”, which is the next parser state, onto the stack. The token pointer then moves to the next token. `ReduceX` action replaces the right side of the grammar production number ”X”, that is situated on stack, with the left side of grammar production number ”X”. `ReduceX` action is also used to construct tree because it runs the rightmost derivation. At this moment two elements that are on the top of the stack are the left side of the production number ”X” and some previous parser state. Those two elements determine `GotoX` action in parsing table. `GotoX` action pushes state ”X” on the top of the stack. Token pointer does not move to the next token, yet it stays on the current one. The `Accept` action accepts the input message meaning the parsing was successful.

The parsing stack, as it is described in the previous section, is used to store parser states and input tokens. On the top of the stack the current parser state is situated. At the beginning of the parsing process the parser pushes number ”0” onto the stack, which represents the initial state.

Parsing tables that are produced from complex grammars often have conflicts. That means that for some parser state and some input token the parsing table has multiple actions. There are two types of conflicts : *reduce/reduce* and *shift/reduce*. Conflicts make parsers non-deterministic. If the parser decides to use one action and ignores other actions, it is possible it will not parse the message successfully although it should. There is no way of knowing in advance that the ignored action would lead to successful parsing. Taking this fact into matter, it is necessary to copy the entire parser and run it, if the parser for the first action was unable to parse the given message.

3.4 Parser Generator

A parser generator is a tool that creates a parser from some form of formal description. It produces the LR parsing table according to the grammar given as input, using specific algorithms. In this project the parsing table is written in XML form. Parser generators are generally used to construct LR tables when dealing with complex grammars, since those result in too many states for the procedure to be done manually. Figure 3-4 Parser generator schemeFigure 3-4 represents a simplified parser generation process.

Figure 3-4 Parser generator scheme

3.5 The Session Initiation Protocol

The Session Initiation Protocol is a protocol that initiates and manages interactive user sessions involving voice, video, instant messaging and other multimedia sessions [4]. The increasing demand put by services such as video conferencing, unified messaging and voice chatting emphasizes the development of protocol that can safely and fast enable usage of mentioned services over the Internet. SIP is based on Hypertext Transfer Protocol (HTTP) and it handles the signaling part of a communication session. After the establishment of the session, Session Description Protocol (SDP) and Real-time Transport Protocol (RTP) takes care of data transport between two endpoints [4]. If SIP messages are not correctly processed, the session will not be established and no communication will be possible. The major advantage of SIP is in its support for both IP and conventional telephone communication and the fact that the combination of both implementations works properly. Also, SIP is text based and it can modify session in progress. The SIP protocol is a TCP/IP-based Application Layer protocol and it requires no implementation on network level. It is the standard protocol for signaling on 3G cell phone networks by the Third Generation Partnership Project (3GPP), meaning that all voice over IP signaling might be done through SIP.

In SIP session SIP User Agent (SIP UA) has the main role - it is a logical network endpoint that creates or receives SIP messages on request/response model. SIP UA can be SIP UA *Client* that sends requests and SIP UA *Server* that receives requests and sends back responses (Figure 3-5).

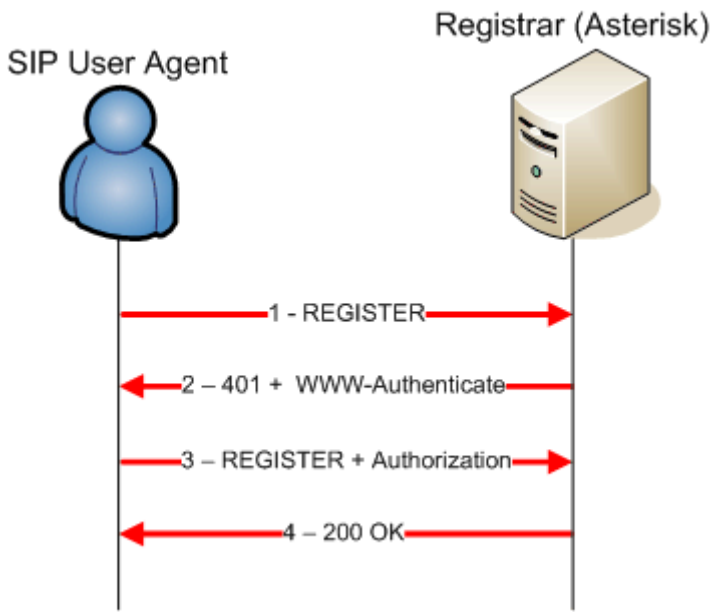

Figure 3-5 Exemplary SIP communication process

From parser perspective, the disadvantages of SIP are complex and abundant grammar that results in parsing conflicts [4]. The conflicts slow down the parser and make the parser use more memory resources, but they make it more accurate. The formal SIP grammar is written in Augmented Backus–Naur Form (ABNF) that makes process of generating parsing table impossible. Form from which this would be possible is Backus–Naur Form (BNF) and because of that within this project an ABNF to BNF Converter is designed and implemented.

3.6 Multicore Processors

A multicore processor is an integrated circuit die (a chip multiprocessor or CMP) to which two or more processors have been attached for enhanced performance, reduced power consumption and more efficient simultaneous processing of multiple tasks. Figure 3-6 represents Intel Core i7 processor architecture.

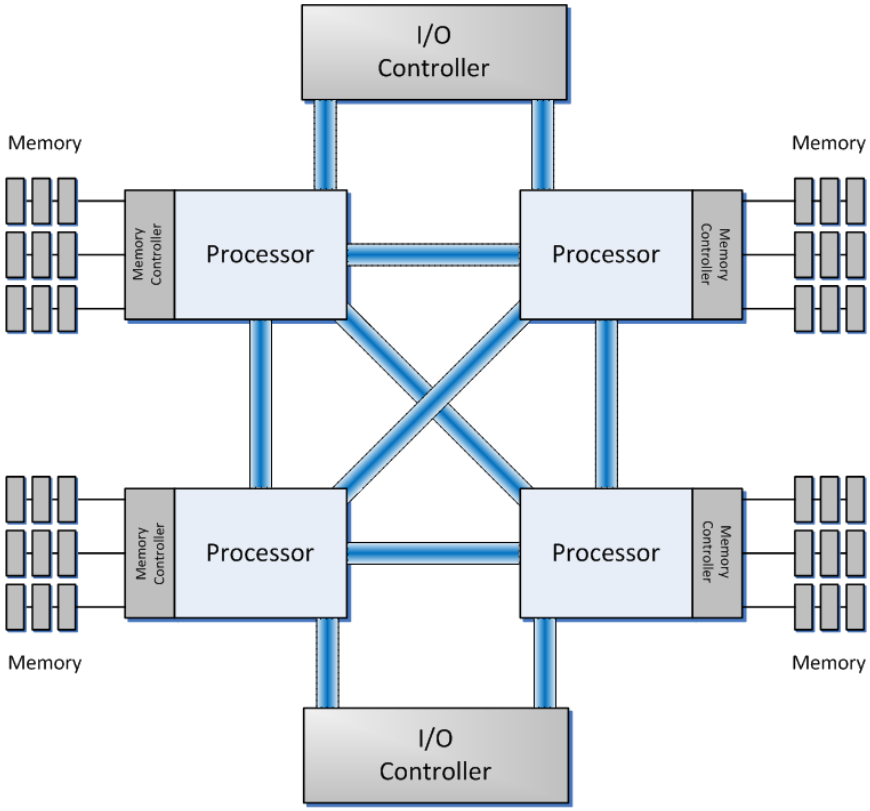

Figure 3-6 Intel Core i7 processor architecture

In recent years, multicore processors have become a growing industry trend as singlecore processors rapidly reach physical limits of possible complexity and speed. The general trend in processor development has been from multicore to many-core: from dual-, tri-, quad-, hexa-, octo-core chips to ones with tens or even hundreds of cores. In addition, multicore chips mixed with simultaneous multithreading, memory-on-chip, and special-purpose "heterogeneous" cores promise further performance and efficiency gains, especially in processing multimedia, recognition and networking applications. There is also a trend of improving energy efficiency by focusing on performance-per-watt with advanced fine-grain or ultra fine-grain power management and dynamic voltage and frequency scaling.

The amount of performance gained by the use of a multi-core processor is strongly dependent on the software algorithms and implementation. In particular, the possible gains are limited by the fraction of the software that can be "parallelized" to run on multiple cores simultaneously.

Parsing processes are sequential in nature, a consequence of which is a limited performance gain when executed on multicore processors.

4 Comparison To Existing Open-Source Solutions

Currently there exist several open-source SIP parsers, which mostly do not subsist as the self-contained programming units, but as parts of the complete SIP stacks. During this project, several of those tools have been tested for the purpose of the performance comparison to the parsers developed as the part of this project. The parsers used for the comparison are:

- PJSIP Parser
- sipXecs parser
- Sofia SIP parser

All of the mentioned parsers have been successfully tested, with the exception of the Sofia SIP parser, due to unresolved header incompatibilities, as detail described in the further chapters.

The tests were performed parallel on all of the tested parsers, including the SIP Multicore parsers, testing singlecore and multicore performance on the restricted set of the SIP messages. All of the parsers were tested in the same way, which is further described in the following text.

For the purpose of the performance measurements, all of the parsers were given 8840 typical SIP messages. All the messages were taken from the RFC 4475 and RFC 5118 torture tests and contained many valid SIP messages, but also several types of invalid messages, such as:

- messages containing long lines,
- messages containing non-printable characters,
- messages containing long repeating strings,
- messages containing invalid use of the % escaping mechanism,
- multipart MIME messages,
- messages containing unusual or empty reason phrase,
- messages containing unknown methods,
- messages containing negative content-length,
- messages containing malformed URI or no URI at all,
- messages containing multiple IP addresses or no address at all, etc.

It is important to mention that, in comparison to the other parsers, the Multicore SIP parser parsed successfully all of the given messages, giving correct answer on validity of all the input messages. On the other side, PJSIP parser could not parse several valid SIP messages, such as those containing long lines and long repeating strings. SipXecs was showed as the worst of the tested parsers, parsing successfully only about 80% of the valid messages. However, those parsers are faster than the Multicore SIP parser, which is most evident on the PJSIP Parser. Nevertheless, it is important to note that the Multicore SIP parser is only a proof of a concept implementation.

The second part of the test was the performance acceleration test of the parsing on the quad-core processor in the comparison to the singlecore performance. The testing was performed by starting four instances of the singlecore parser, therefore forcing the operating system to perform parsing on each of the four processor cores. Each of the parser was given a set of 2210 SIP messages which each of the parsers parsed on the separate core. The time needed for the parsing was afterwards compared to the singlecore performance time. The Multicore SIP parser showed a performance gain significantly better than the performance gain on other parsers.

Parsing techniques of the open-source parsers and performance measurements results are depicted in the following chapters.

Due to the unsatisfying results of the performance tests performed on the existing open-source parsers, which are the missing of the performance gain on multicore machines, as well as the fact that the parsers are not 100% SIP compliant, it is obvious that the development of the new parsers was necessary.

4.1 PJSIP Parser

PJSIP is a SIP stack supporting many SIP extensions/features, with the following key benefits, such as portability, small footprint, good support for embedded development and general applications, and finally high performance [6].

In comparison to the Multicore SIP parser, which uses the LR parsing method, parser of the PJSIP stack is a recursive descent parser, a top-down parser built from a set of procedures where each such procedure implements some of the production rules of the grammar, i.e. for each header of the SIP message. PJSIP is written in C programming language and can be run on most of the operating systems. Due to the SIP stack complexity, code is complex and hard to follow. However, documentation of the stack is detailed and helpful.

In comparison to the parser described in this document, PJSIP parser parses messages much faster, but the parsing speed does not increase when the task is split on more processor cores, as depicted in Figure 4-1.

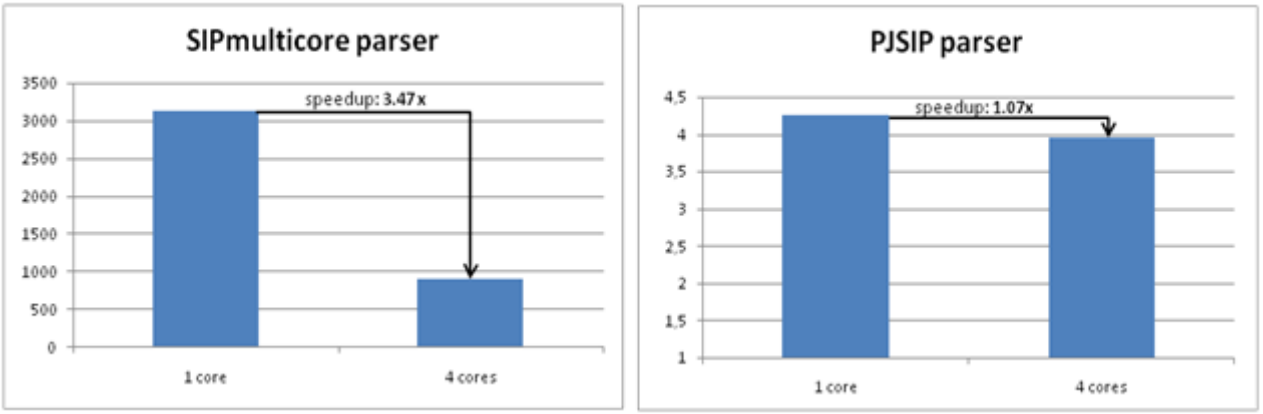

Figure 4-1 Comparison of SIP Multicore and PJSIP parsers

Unlike the Multicore SIP parser, which parses 100% of SIP messages valid according to the grammar, PJSIP parser does not parse successfully all of the SIP messages from the RFC torture tests (RFC 4475 and RFC 5118). One such message is given in Appendix A.

4.2 SipXecs Parser

SipXecs is an application that simulates communication using SIP. The parsing part of that application was extracted and compared to the Multicore SIP parser.

SipXecs parser is developed just for SIP messages and it is not using LR parsing [7]. Unlike the Multicore SIP parser, which parsed all of the messages successfully, sipXecs parsed only 81% messages.

Because it is developed just for SIP and the fact that it is written in programming language C, which is much faster than C#, sipXecs parser parses messages faster.

As mentioned, both parsers were given 8440 valid SIP messages and their performance compared on singlecore and quad-core processor. The Multicore SIP parser showed greater acceleration than sipXecs parser. SipXecs has the acceleration of 1.75 times, while the Multicore SIP parser has the acceleration of 3.47 times, as depicted on the Figure 4-2.

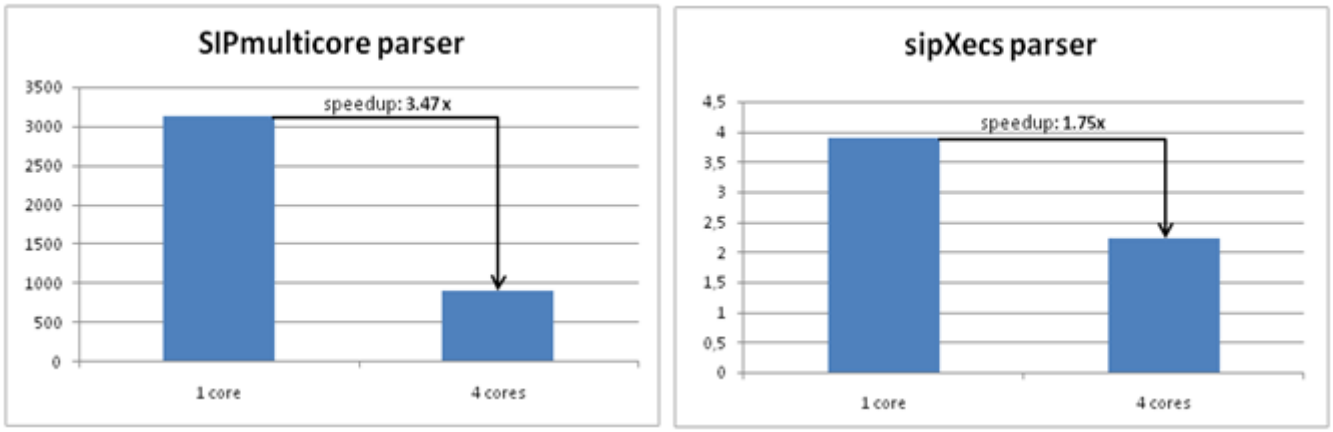

Figure 4-2 Comparison of SIP Multicore and sipXecs parsers

SipXecs was not able to parse three out of sixteen messages that were taken from RFC 4475 and RFC 5118 documents. Two examples of valid SIP messages that sipXecs was not able to parse are listed here:

```
INVITE sip:user@[2001:db8::10] SIP/2.0
To: sip:user@[2001:db8::10]
From: sip:user@example.com;tag=81x2
Via: SIP/2.0/UDP [2001:db8::20];branch=z9hG4bKas3-111
Call-ID: SSG95599055239970 77@hlau_4100
Contact: "Caller" <sip:caller@[2001:db8::20]>
CSeq: 8612 INVITE
Max-Forwards: 70
Content-Type: application/sdp
Content-Length: 268
```

```
INVITE sip:UserB@biloxi.com SIP/2.0
Via: SIP/2.0/TCP client.atlanta.com:5060;branch=z9hG4bK74bf9
Max-Forwards: 70
From: BigGuy <sip:UserA@atlanta.com>;tag=9fxced76sl
To: LittleGuy <sip:UserB@biloxi.com>
Call-ID: 3848276298220188511@atlanta.com
CSeq: 1 INVITE
Contact: <sip:UserA@client.atlanta.com;transport=tcp>
Content-Type: application/sdp
Content-Length: 143
```

4.3 Sofia SIP Parser

The performance of the developed SIP Multicore parser could not be compared to Sofia-SIP parser since the latter could not be extracted from Sofia-SIP stack due to unresolved header incompatibilities. A partial example of using just the parsing functionality of Sofia-SIP is presented in the sofia-sip-devel mailing list: http://www.mail-archive.com/sofia-sip-devel@lists.sourceforge.net/msg02568.html and listed here:

```
#include <stdio.h>
#include <sofia-sip/sip_parser.h>
#include <sofia-sip/sip_tag.h>

int main()
{
        char data[] = { "BYE sip:[EMAIL PROTECTED] SIP/2.0 \
        Via: SIP/2.0/UDP sip.example.edu;branch=d7f2e89c.74a72681 \
        Via: SIP/2.0/UDP pc104.example.edu:1030;maddr=110.213.33.19 \
        From: Bobby Brown <sip:[EMAIL PROTECTED]>;tag=77241a86 \
        To: Joe User <sip:[EMAIL PROTECTED]>;tag=7c6276c1 \
        Call-ID: [EMAIL PROTECTED] \
        CSeq: 2" };

        msg_t* msg = msg_make(sip_default_mclass(), 0, data, -1);
        sip_t* sip = sip_object( msg );
        printf( "1 sip: %p\n", sip );
        printf( "2 sip_err: %s\n", sip->sip_error->er_name );
        printf( "3 sip_from: %p\n", sip->sip_from );
        printf( "4 display: %s\n", sip->sip_from->a_display );
}
```

5 SIP Multicore Project

The goal of the project was to develop a SIP multicore parser that would be 100% SIP compliant in order to demonstrate the advantages of multicore parsing approach. Additionally, other tools were developed so to ease the parsing process, as well as to allow easier system verification.

Within the project the following elements were developed:

1. ABNF to BNF grammar converter
2. Parser generator
    a. Lexical analyzer generator
    b. Syntax analyzer generator
3. Parser template
4. Parsing tree visualizer

Figure 5-1 represents the developed project segments and their interconnectedness. Each of the segments is described in the following chapters.

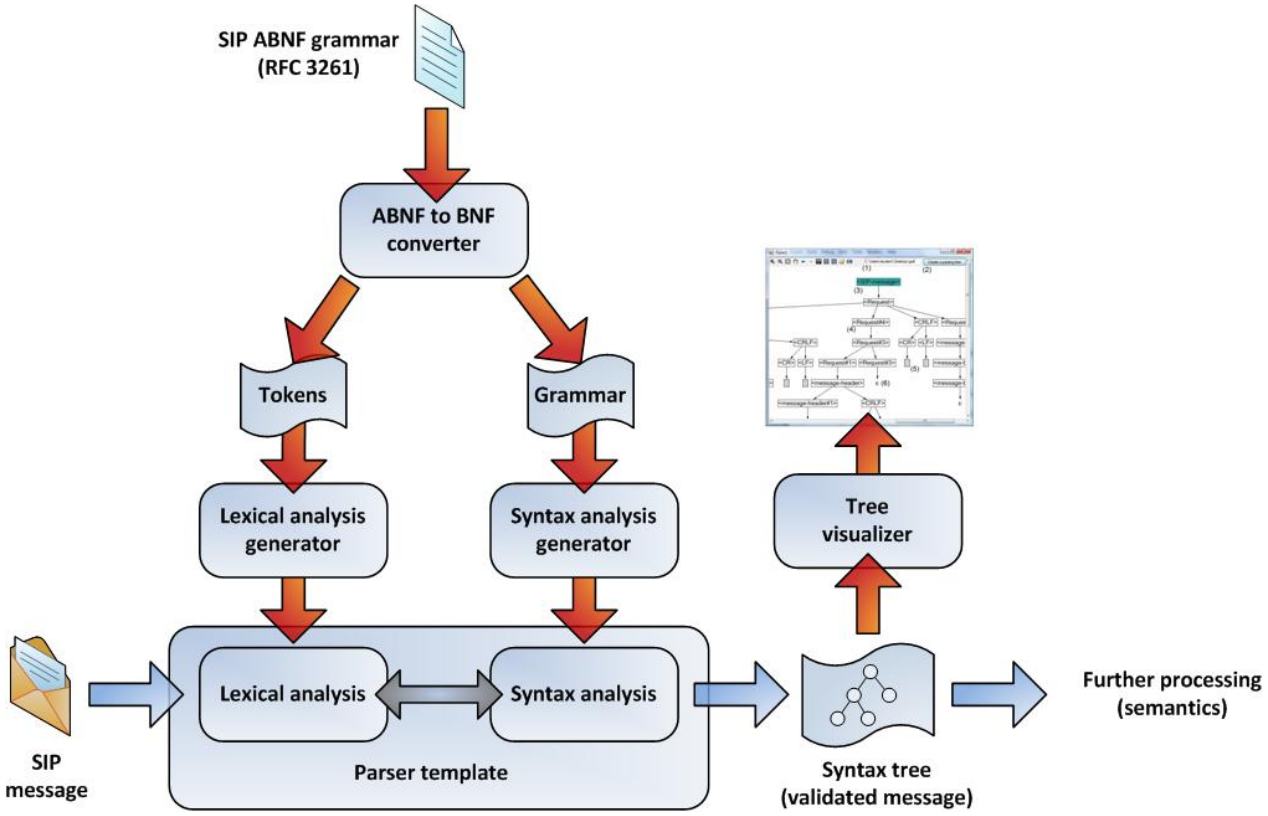

Figure 5-1 SIP Multicore project overview

5.1 ABNF to BNF Conversion

5.1.1 Motivation

ABNF to BNF converter is a tool for conversion of the input grammar written in Augmented Backus–Naur Form (ABNF) to Backus–Naur Form (BNF). ABNF is a meta-language based on BNF, but consisting of its own syntax and derivation rules [2].

An ABNF specification is a set of derivation rules, written as

```
rule = definition ; comment CR LF
```

where rule is a nonterminal, the definition consists of sequences of symbols that define the rule, a comment for documentation, and ending with a carriage return and line feed. The ABNF grammar form differs from the BNF in its basic syntax, such as different definition of terminals (value ranges, hexadecimal values...), but also usage of more complex rules, such as usage of variable repetitions, sequence groups, and optional sequences [2]. Such syntax enables easier definition of complex grammar rules, but also makes the parsing process significantly more complex. Therefore, it is more feasible to transform the ABNF grammar into its BNF form before the parsing process.

Formal SIP grammar is defined in ABNF, and building a parser that deals with that form of grammar would be highly complicated. Since there does not exist an open source solution, it was necessary to develop a tool that would convert the grammar into its BNF equivalent.

5.1.2 Implementation

As mentioned, ABFN grammar form differs from the BNF form in several basic syntax rules, but also in some complex definitions. For an example, angle brackets (“<”, “>”) are not required around rule names, as they are in BNF.

Terminals are specified by one or more numeric characters. Numeric characters may be specified as the percent sign “%”, followed by the base (x = hexadecimal), followed by the value, or concatenation of values (indicated by “.”). For example a carriage return is specified by %x0D in hexadecimal. A carriage return followed by a line feed may be specified with concatenation as %x0D.0A.

Literal text is specified through the use of a string enclosed in quotation marks ("). These strings are case-insensitive. For a case-sensitive match the explicit characters must be defined: to match “aBc” the definition would be %x61 %d42 %d63.

ABNF Operators

- **White space** is used to separate elements of a definition; for space to be recognized as a delimiter it must be explicitly included by hexadecimal value (%x20).
- **Alternation** symbol “|” in BNF is written as the symbol “/” in ABNF. Incremental alternatives, which are additional alternatives added to a rule through the use of “=/” between the rule name and the definition, are not supported by the converter and cannot be converted to BNF by the tool.

- **A value range** may be specified through the use of a hyphen ("-").
  E.g. the rule
  `OCTAL = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7"`
  is equivalent to
  `OCTAL = %x30-37`
- **Sequence group:** elements may be placed in parentheses to group rules in a definition. To match "elem fubar snafu" or "elem tarfu snafu" the following rule could be constructed:
  `group = elem ( fubar / tarfu ) snafu`
- **Variable repetition:** to indicate repetition of an element the form <a>*<b>element is used. The optional <a> gives the minimum number of elements to be included with the default of 0. The optional <b> gives the maximum number of elements to be included with the default of infinity. It is possible to use `*element` for zero or more elements, `1*element` for one or more elements, and `2*3element` for two or three elements.
- **Specific repetition:** to indicate an explicit number of elements the form `<a>element` is used and is equivalent to `<a>*<a>element`. It is possible to use `2DIGIT` to get two numeric digits and `3DIGIT` to get three numeric digits.
- **Optional sequence:** to indicate an optional element the following constructions are equivalent:
  `[ fubar snafu ]`
  `*1( fubar snafu )`
  `0*1( fubar snafu )`
- **Other operators:** other ABNF operators, such as alternation and comments, are equal to the corresponding operators in BNF, and are not subject of any change in the converter.

ABNF to BNF Converter

First step in conversion of ABNF to BNF grammar is loading the original ABNF grammar from the input file. Input file is written as a standard form of ABNF grammar, where each production is written in a form:

```
rule = definition ; comment CR LF
```

All operators are of the form as explained in the former chapter. Each operator is separated by a single space, including round and square brackets, with the exception of the variable repetition number.

Class `Grammar` represents the complete grammar loaded from the input file. Input file is given to the constructor of the class. The grammar contains list of productions and list of terminal values (tokens) included in the input grammar. Tokens are all terminal values which occur in the input grammar (string and hexadecimal values and ranges). `Grammar` saves the start symbol of the grammar, which is the symbol on the left-hand side of the first production in the grammar.

The constructor of the `Grammar` class splits the input file into lines and generates new productions by calling the constructor of the `Production` class and adding new productions into the list of all grammar productions.

`Production` class represents a single production of the grammar. Each production saves its head symbol and list of body symbols, as well as the reference to the grammar which contains the production. Each production symbol (head symbol and body symbols of the production) are objects of the type `ProductionSymbol`, which contains the value of the symbol, as well as its type and number of repetitions. Production symbol can be one of the several possible types, such as string, value range, bracket, comment, etc. Constructor of the `Production` class parses the input string (production of the input grammar) by determining the head symbol and body symbol, i.e. by calling the constructor of the class `ProductionSymbol`.

The constructor method of the `ProductionSymbol` class saves the value of the symbol and calls two methods: `DetermineRepetition` and `DetermineType`. Method `DetermineRepetition` checks if the symbol contains repetition symbol, which can be one of several forms:

1. *<symbol>
2. *n<symbol>
3. n*<symbol>
4. n*n<symbol>
5. n<symbol>

The method recognizes the minimal and maximal repetition of the symbol (e.g. minimal repetition of `n*<symbol>` is `n`, and maximal is infinity) and saves the value of the symbol. Method DetermineType determines the type of the production symbol. Possible types are:

1. String,
2. ValueRange,
3. Nonterminal,
4. Alternation,
5. Comment,
6. Equality,
7. LeftRoundBracket,
8. RightRoundBracket,
9. LeftSquareBracket,
10. RightSquareBracket.

After determing the type of the symbol, the method `ChangeABNFtoBNFform` is being called. The method changes the ABNF to BNF notation:

- Alternation symbols is changed from "/" to "|",
- Square brackets („<" and „>") are added to nonterminals,
- The form of hexadecimal values and value ranges is changed:
    - Value %xDD is changed to \xDD
    - Value %xAA.BB.CC is changed to \xAA\xBB\xCC
    - Value range %xAA-BB is changed to \xAA-\xBB
- Quotation marks from the terminals are removed,
- Equals symbol ("=") is changed to "::=".

Parsing the input grammar starts by calling the method `Parse` of the `Grammar` class. The method goes through the list of the productions and calls the `Parse` method for the each production. This method parses the production in five steps by calling five different methods:

1. `DeleteComments` removes all the comments from the production. It finds the first symbol of the type `Comment` and deletes all following body symbols.

2. `SeparateTokens` separates terminals into the list of tokens the grammar. Tokens are presented as objects of the class `Token` which contains the name and the value of the token. The method searches through the list of body symbols, and for each terminal (symbol of the type `String` or `ValueRange`) calls the method `CreateOrGetExistingToken` of the `Grammar` class. This method creates a new for the new token, which is created by merging the prefix „TOKEN_“ and the value of the terminal, and checks if there already exists a token with the same value in the list of grammar tokens. If there already exists one, the method finds the token and returns its name and, if there does not exist that token, it creates a new token, adds it into the list of all tokens and returns the name of the new token.
   `SeparateTokens` gets the name of the token and replaces the value of the terminal in the production with the name of the token.

3. `SeparateGroupings` creates new productions for the grouped symbols of the production. The method is going through the body symbols of the production, looking for the opened bracket (symbol of the type `LeftRoundBracket`). When it finds it, it calls the recursive method `SeparateGroupingsRecursive` which collects all the symbols which are in one group (in one level) and calls the method `CreateProduction` of the `Grammar` class which creates a new production for the group of symbols and returns the head symbol of the production. The complete group in the primal production is changed by the head symbol of the new production.
   When giving the name to the new production, which is the result of the groupings inside the production, the name is made by merging the name of the head symbol of the “mother” production, symbol “#” and the number of the production.
   E.g., from the production
   `A = "a" ( "b" ( "c" "d" ) "e" )`
   two new productions are created and the primal production is changed to the following form:
   `<A> ::= TOKEN_a <A#1>`
   `<A#1> ::= TOKEN_b <A#2> TOKEN_e`
   `<A#2> ::= TOKEN_c TOKEN_d`

4. `SeparateOptionals` creates new productions for the optional group of the symbols of the production. The method is going through the body symbols of the productions, looking for the opened square bracket (symbol of the type `LeftSquareBracket`). When it finds it, it calls the recursive method `SeparateOptionalsRecursive` which collects all the symbols which are in the optional group and calls the method `CreateOptionalProduction` of the `Grammar` class which creates new production for

the optional group of the symbols and returns the head of the production. The complete optional group in the primal production is changed by the head symbol of the new production.
New production is made in the way that it contains the body of the optional group of the primal production and an epsilon-production as the alternative.
When giving the name to the new production, which is the result of the optional variables inside the production, the name is made by merging the name of the head symbol of the "mother" production, symbol "#" and the number of the production.
E.g., from the production

```
A = "a" [ "b" [ "c" "d" ] "e" ]
```

two new productions are created and the primal production is changed to the following form:

```
<A> ::= TOKEN_a <A#1>
<A#1> ::= TOKEN_b <A#2> TOKEN_e | €
<A#2> ::= TOKEN_c TOKEN_d | €
```

5. `SeparateRepetitions` goes through the body symbols and looks for those symbols which have minimal or maximal repetition different than 1 and calls the method `CreateRepetitionProduction` of the `Grammar` class for that symbol.
   There are five possible types of repetitions, which can be grouped into two groups:
   1. maximal repetition is determined:
      a) *b<symbol> - minimal 0, maximal b,
      b) a*b<symbol> - minimal a, maximal b,
      c) a<symbol> - minimal and maximal a,
   2. maximal repetition is infinite
      a) a*<symbol> - minimal a,
      b) *<symbol> - minimal 0.

   For the first group of repetitions, two new productions are created. First production contains the minimal number of the repeated symbol and a jump to the second new production which enables from 0 to the maximal-minimal numbers of repeating of the symbol. E.g., for the production

   ```
   B = "a" 2*4"b"
   ```

   new productions created are

   ```
   <B> ::= TOKEN_a <B#2>
   <B#2> ::= TOKEN_b TOKEN_b <B#1>
   <B#1> ::= TOKEN_b | TOKEN_b TOKEN_b | €
   ```

   For the second group of repetition symbols, two new productions are created. First production contains the minimal number of the repeated symbol and a possible jump to the second new production which can be an epsilon production, or go to the repeated symbol. E.g., for the production

   ```
   B = "a" 2*"b"
   ```

   new productions created are

   ```
   <B> ::= TOKEN_a <B#2>
   <B#2> ::= TOKEN_b TOKEN_b <B#1>
   <B#1> ::= TOKEN_b <B#1> | €
   ```

At the end of the parsing, grammar is being simplified and productions are sorted by alphabet, with the exception of the start symbol, which productions are placed on the first place of the list of productions.

Grammar simplification is carried out in six steps:

1. `RemoveAlternation` is the method that breaks all the productions into its alternation groups and creates new productions for each alternation group, so that it is easier to simplify the grammar. E.g., from the production
   `A = "a" "b" / "c" "d"`
   these productions are generated
   `<A> ::= TOKEN_a TOKEN_b`
   `<A> ::= TOKEN_c TOKEN_d`

2. `RemoveDeadSymbols` removes all dead symbols from the grammar. Dead symbols are those from which it is impossible to create a list of terminals. The algorithm contains four steps:
   a. As live symbols add those symbols which on the right side of production have no nonterminals.
   b. As live symbols add those which on the right side have all live symbols.
   c. All the symbols that are not in the list of live symbols are dead
   d. Remove all productions that contain dead symbols from the list of grammar productions.

3. `RemoveUnreachableSymbols` removes all unreachable symbols from the grammar. Unreachable symbols are those that cannot be reached from the start symbol by applying any of the productions. The algorithm contains four steps:
   a. As reachable add the start symbol of the grammar.
   b. If the head symbol of a production is reachable, all body symbols of that production are also reachable.
   c. All the symbols that are not reachable are unreachable.
   d. Remove all unreachable symbols from the grammar.

4. `RemoveEpsilonProductions` removes epsilon-productions from the grammar. The algorithm contains several steps:
   a. Get all empty symbols. Empty symbols are those from which can be reached an epsilon -production if some of productions applied. Empty symbols are collected by the method `GetEmptySymbols`, which finds empty symbols in two steps:
      1. Add head symbols of epsilon -productions in the list of empty symbols.
      2. In the list of empty symbols add head symbols of productions which contain only empty symbols as body symbols.
   b. Check each grammar symbol of every production. If the symbol is equal to the empty symbol, we find all productions that on their left-hand side have the empty symbol. We check all the productions that we found, and we get two options:

1. production is epsilon-production (new production is created – production equal to the production that on its right-hand side contains the empty symbol, but without the empty symbol)
2. production is not epsilon-production (no action)

c. Delete epsilon-productions.

d. If there is just one production that on its left-hand side contains the empty symbol, and that production is an epsilon-production, delete productions that on their right-hand side contain the empty production.

5. `RemoveUnitProductions` removes unit productions from the grammar. The algorithm is carried out in these steps:

a. Find unit production, i.e. production that on its right-hand side contains only one, non-terminal symbol.

b. For each unit production find all production that on their left-hand side contain the right-hand symbol of the unit production and for each such production that is not an unit production create a new production which will on the left-hand side contain the left-hand side of the unit production, and on the right-hand side the right-hand side of the found production.

c. When found, delete all unit productions.

6. `RemoveMultipleProductions` removes multiple productions from the list of productions of the grammar, i.e. those productions that exist more than once in the list.

It is important to notice that although simplifying grammar creates additional productions in the grammar and enlarges the number of created states in the LR table of the generated parser, it significantly accelerates parsing. However, with completely simplified grammar it might be more difficult to insert semantic actions into the parser. The problem is mostly connected to the removal of unit productions, when those unnecessary productions are being removed, because of the fact that those productions sometimes make the grammar more readable to humans easier to follow. Therefore, when adding semantic actions to the parser, it might be useful to compare fully simplified and non-simplified grammars. For an example, when observing the following header of a SIP message:

```
To: sip:user@example.com
```

and following productions of the SIP grammar, responsible for parsing the header:

```
<message-header>   ::= <message-header#1> <CRLF>
<message-header#1> ::= <Accept>
<To>               ::= <To#1> <HCOLON> <To#2> <To#5>
<To#1>             ::= TOKEN_To
```

Second of the mentioned grammar productions is a unit production and would be removed at the time of the grammar simplifying. However, that production is very important when adding semantic actions, because the parser could recognize the type of the header when reaching the symbol `<Accept>`. When this production is removed, each of headers will start with the symbol `<message-header#1>`, and it is necessary to recognize headers in an alternative way, such as

recognizing first token in the production (in this case token TOKEN_To). This method is significantly more complex for the programmer and possibly for the parser.

As the output, the converter generates two files – “grammar.txt” and “LexInputFile”. First file contains all the productions of the grammar written in the Backus–Naur Form, in the format readable to the syntax analyzer of the SIP parser generator. The second file contains definitions of the token, which is written in the form readable to the lexical analyzer of the SIP parser generator. Forms of those files are described in further chapters.

5.2 Parser Generator

5.2.1 Motivation

Existing parser generators generate singlecore sequential parsers, which cannot easily be converted into their multicore equivalents. Additionally, existing solutions generate lexical and syntax analysis separately, the result of which is that lexical analysis must be performed prior to syntax analysis. However, by enabling dynamical coupling between lexical and syntax analyzers it is possible to resolve grammar ambiguities and conflicts.

5.2.2 Lexical Analysis Generator Implementation

Generation of the lexical analyzer of the generated parser starts with the method `GenerateLex` of the `LexGenerator` class. The method parses the input file and generates list of tokens, and at the end modifies the parser template.

The input file, called `LexInputFile`, contains definitions of the tokens for the lexical analyze of the parser. The file is of the form:

```
<token definition>\t:\t<token name>
```

i.e. token definition and token name separated by three symbols: tab (\t), colon (:) and another tab.

Token definitions can be written as string values and as hexadecimal values, as well as ranges and concatenation of hexadecimal values. Following example of the input file presents all possible forms of the token definitions.

```
Accept    :          TOKEN_Accept
\x21      :          TOKEN_!
\x41\x43  :          TOKEN_AC
\x30-\x39 :           TOKEN_0-9
```

All tokens written as strings are case insensitive, and all written in hexadecimals are case sensitive. E.g., the definition of the token of the value “IN” and the token name “TOKEN_IN” can be written as:

```
IN        :          TOKEN_IN
```

if the value is case insensitive, or

```
\x49\x4E :          TOKEN_IN
```

if the value is case sensitive.

Method `ParseInputFile` of the class `LexGenerator` parses input file and generates list of objects of the class `TokenRegex`, which holds the definition (regular expression) of the token, name (class) of the token, and information if the value of the token, for which the token is defined, must be case sensitive.

Beside the input for the lexical analysis, the parser gets the parser template as another input file. The goal of the generator of the lexical analyzer is to modify the parser template so to insert definition of the lexical units. The parser generator finds the following line in the parser template:

```
//[<TOKEN-REGEX>]
```

and changes it by definitions of the tokens. Definitions are added in the form of initialization of the objects of the type `TokenRegex`. At the start of the execution, parser loads those definitions and determines next steps in parsing.

Result of the generator of the lexical analysis is, therefore, modified parser template, which is saved as the output file LexOutput.cs.

Generated parser contains the method `NextTokens` of the class `RegexProcessor` which gets the list of tokens found in the source code (input text of the parser) at the defined position in the source. This method represents the whole lexical analysis in the parser. Syntax analyzer calls the method when it needs the next token at the exact position in the file, giving the list of possible tokens to be recognized. As the result of the method invocation, syntax analyzer gets the list of recognized tokens and its values at the given position in the source file. Each token (object of the type Token) contains the position of the next token that should be recognized in the source code, so that the parser can know with which index (position in the input file) it can call the method `NextTokens` in the next iteration.

The method itself finds all definitions of tokens which are one of those which could be recognized (given as the input parameter). It checks all the definitions (regular expressions) at the exact position in the input file and as the result of invocation returns those tokens that have been recognized. In the case that one of the possible tokens can be the end of input file (symbol $), it checks if the given position is at the end of the input file and sends the end symbol as the only recognized token.

The method uses methods of the .NET `Regex` class to find tokens which definitions (regular expressions) are satisfied.

5.2.3 Syntax Analysis Generator Implementation

Syntax analysis generator creates the LR table for a parser based on the input BNF grammar. Required input file for the syntax analysis generator is a text file with the input grammar. Additionally,

some other input files may be required if the created LR table is going to be written as a part of the parser template. In that case the generator program expects a parser template, which can be written in any programming language, and files with the source code for creation of the parser actions (shift, reduce, goto…). All input files are described in the following chapter. After that, the generator itself is described, and, in the end, possible output files are being described.

Input File Specification

Grammar Input File

Grammar input file consists of three parts:

1. Information about priorities of the terminal grammar symbols
2. Information about associativities of the terminal grammar symbols
3. Grammar productions

These parts must appear in the given order and each part of the grammar input file must be delimited from the next one with one empty line. The information about priorities and associativities of the terminal symbols may be used by the parser to resolve the grammar ambiguities. This information is not required by the syntax analysis generator and can be left empty in the input file, but the empty lines that are used as delimiters must be present. For example, if the input file doesn't contain the information about priorities, nor the information about the associativities of the grammar symbols, it must start with two empty lines: one to delimit the priorities part from the associativities part (both of which are empty) and one to delimit associativities part from the grammar productions.

Information about priorities of the symbols must be provided in the following format: terminal symbol, single space and an integer number that represents the priority of that symbol, one entry per line. For example, if the terminal symbol `TOKEN_ACK` has priority 5, and the terminal symbol `TOKEN_Allow` has priority 3, then the lines in the input file which contain that information would look like this:

```
TOKEN_ACK 5
TOKEN_Allow 3
```

As was mentioned before, the information about priorities is not used in the syntax analysis generation process, so the meaning of the numbers that represent priorities depends only on the parser template. In one template, lower number may mean higher priority, while in the other it may mean lower priority, that makes no difference to the syntax analysis generator and the input file must be adapted for the parser template that is going to be used.

Information about associativities of the symbols is formatted in the same way. The only difference is that there is no integer number for the associativity, but the associativity is given with a single character: L – if the symbol is left associative, R – if the symbol is right associative, and U – if associativity is undefined. If the associativity for a given symbol is undefined, then that symbol doesn't even need to be listed in the associativities part of the input file. For example, if the terminal symbol

TOKEN_ACK is left associative, TOKEN_Allow is right associative, and the associativity for TOKEN_A-Z is undefined, then the lines in the input file that contain that information may look like this:

```
TOKEN_ACK L
TOKEN_Allow R
```

In the above example TOKEN_A-Z is not listed, because it has undefined associativity. Equivalent associativities section of a file in which TOKEN_A-Z is listed may look like this:

```
TOKEN_ACK L
TOKEN_Allow R
TOKEN_A-Z U
```

The final section of the grammar input file are the productions of the BNF grammar. This section is the only part of the input file that is used in the LR table generation process.

Productions consist from terminal and nonterminal grammar symbols. Nonterminal grammar symbols must be surrounded with the less than (<) and the greater than (>) signs. All other symbols, except the special symbols, will be considered terminal. Epsilon symbol is represented with the euro sign (€). The first production in the file must be the production that has the initial grammar symbol on its left side, the order of all other productions is irrelevant.

The symbol used to discern the left and the right sides of a single production is considered as special. That symbol consists of the two colons and an equals sign (::=).

Space is a special symbol that is used as delimiter. Each terminal or nonterminal grammar symbol must be delimited from its neighbors with at least one space. The productions are listed one per line, but the vertical line character (I) can be used to list more than one right side for a given nonterminal symbol. With the use of that symbol, productions of a single nonterminal left hand side symbol can be written in a number of ways. For example, if for the nonterminal symbol <A> there exist three productions: <A> → <B><C>, <A> → a, <A> → <D>, a part of the input file might look like this:

```
<A> ::= <B> <C>
<A> ::= a
<A> ::= <D>
```

The other one might look like this:

```
<A> ::= <B> <C> | a
<A> ::= <D>
```

While the third one may look like this:

```
<A> ::= <B> <C> | a | <D>
```

All of these files are equivalent, and the resulting LR table will be the same, no matter which one of them is used.

A simple grammar input file is listed hereafter. That file contains a BNF grammar that can generate simple arithmetic expressions. Operators * and / have higher priority than + and –, so higher number means higher priority in this case. All operators are left associative. The grammar has four different nonterminal and 6 terminal symbols. The grammar can generate empty sentences; this is accomplished by the epsilon symbol in one of the productions of the initial grammar symbol.

Sample grammar input file:

```
+ 0
- 0
* 1
/ 1

+ L
- L
* L
/ L

<S> ::= <E>
<S> ::= €
<E> ::= <T> + <E> | <T> - <E> | <T>
<T> ::= <F> * <T> | <F> / <T> | <F>
<F> ::= digit | ( <E> )
```

Parser Template And Actions

Syntax analysis generator can insert LR table and the information about priorities and associativities directly into the parser template code. The general idea when building syntax analysis generator was to build it in such a way that the parser template can be written in any programming language. To accomplish that, syntax analysis generator requires some additional files.

Those files are text files that can contain arbitrary code and the placeholders which will be used by the generator to insert the required information in the parser template. Parser template should also contain placeholders which will be replaced by the generated code. Generator will, for each parser action, generate a block of code in which the contents of the corresponding action file will be repeated once for each action of that type in the LR table and the placeholders from the action file will be replaced with the action's data. Placeholders in the parser template will be replaced by those blocks of code. All of the placeholders are optional and can be repeated any number of times. Each occurrence of the placeholder will be replaced by the corresponding source code.

To reduce the number of parameters to the syntax analysis generator, all filenames have been hardcoded and the generator should only be given a path to the folder with those files. Table 5-1 shows the filenames of required files with short descriptions of the each file's role. As was mentioned before, those files are necessary only if the LR table will be written into the parser template.

Table 5-1 Filenames and their descriptions

File	Description
init.txt	Contains the source code for the parser initialization (e.g., creation of the empty LR table).
priorityAction.txt	Contains the source code that is used to define the priority for a single terminal grammar symbol.
associativityAction.txt	Contains the source code that is used to define the associativity for a single terminal grammar symbol.
shiftAction.txt	Contains the source code that is used to define the shift action in the parser.
gotoAction.txt	Contains the source code that is used to define the goto action in the parser.
reduceAction.txt	Contains the source code that is used to define the reduce action in the parser.
reduceInstruction.txt	Contains the source code that is used to define the number of symbols that should be removed from the stack and the nonterminal symbol that should be pushed onto the stack to make a certain reduction.
acceptAction.txt	Contains the source code that is used to define the accept action in the parser.

Files from the previous table can have a number of different placeholders in them. Table 5-2 lists all the available placeholders and gives a short description of the data that the generator will insert in their place.

Table 5-2 All available placeholders in action files

Action placeholder	Description
[<NOS>]	**Number of states** – an integer that represents total number of parser states.
[<CS>]	**Current state** – an integer that represents the current parser state.
[<NS>]	**Next state** – an integer that represents the parser state after the action is executed.
[<GS>]	**Grammar symbol** – a string that represents terminal or nonterminal grammar symbol, depending on the action in which it is used.
[<PN>]	**Production number** – an integer that represents the number of a production that is going to be reduced.
[<PH>]	**Production head** – a string that represents the left hand symbol of a production, that symbol is going to be pushed onto the parsing stack after the reduction.
[<PBC>]	**Production body count** – an integer that represents the number of the grammar symbols on the right side of a production, this is the number of the symbols that are going to be removed from top of the stack during a reduction.
[<P>]	**Priority** – an integer that represents the priority of a terminal grammar symbol.
[<A>]	**Associativity** – a string that represents the associativity of a terminal grammar symbol, it can have a value of “Left” or “Right”.

Table 5-3 shows which placeholders will be recognized by the generator for each action file, in other words, which placeholders can be used in which action input file.

Table 5-3 Available placeholders for each action file

File	Available placeholders
init.txt	[<NOS>]
priorityAction.txt	[<GS>], [<P>]
associativityAction.txt	[<GS>], [<A>]
shiftAction.txt	[<CS>], [<GS>], [<NS>]
gotoAction.txt	[<CS>], [<GS>], [<NS>]
reduceAction.txt	[<CS>], [<GS>], [<PN>], [<PH>], [<PBC>]
reduceInstruction.txt	[<PN>], [<PH>], [<PBC>]
acceptAction.txt	[<CS>], [<GS>]

Placeholders in action files will be replaced by the syntax analysis generator only with their values; generator won't insert any additional characters, e.g., quotation marks around strings. It is user's responsibility to ensure that the code in the action file takes that into consideration. For example, surround the [<GS>] placeholder with quotation marks so that the inserted string would also be surrounded with quotation marks.

Syntax analysis generator treats the placeholders inside the parser template a bit differently. It removes the whole line in which it finds the placeholder from the template and in that place puts the generated block of code with the parser actions. The parser template placeholders are treated in such a way so that the placeholders could be placed in single line comments. That allows the parser template to be successfully compiled with the placeholders inside the file, which can be useful in some situations. Table 5-4 lists all the available placeholders that can be placed in the parser template and for each one there is a short description of what the generator will insert in its place.

Table 5-4 Parser template placeholders and the description of their replacements

Placeholder	Replacement
[<INIT>]	The code which is needed for parser initialization.
[<PRIORITIES>]	Block of code which defines the priorities for the grammar's terminal symbols.
[<ASSOCIATIVITIES>]	Block of code which defines the associativities for the grammar's terminal symbols.
[<SHIFT ACTIONS>]	Block of code which defines all the shift actions for the parser.
[<GOTO ACTIONS>]	Block of code which defines all the goto actions for the parser.
[<REDUCE ACTIONS>]	Block of code which defines all the reduce actions for the parser.
[<REDUCE INSTRUCTIONS>]	Block of code with all the instructions that define the number of symbols that need to be taken off the parsing stack and which symbol should be put onto the parsing stack in order to reduce a certain production.
[<ACCEPT ACTION>]	Block of code which defines the conditions for which the parsing process will be successful.
[<PRODUCTIONS LIST>]	The list of the grammar's productions. This placeholder should be placed inside a multiline comment.

The [<PRODUCTIONS LIST>] placeholder should be placed inside a multiline comment, because its replacement doesn't contain any source code. As was mentioned before, the whole line in which this placeholder is placed will be replaced, so the characters for the start and the end of the comment need to be placed before and after that line. Hereafter are given two examples, the first one shows incorrect placement of this placeholder, and the other shows the correct placement of this placeholder.

Incorrect placement:

```
/* [<PRODUCTIONS LIST>] */
```

Correct placement:

```
/*
[<PRODUCTIONS LIST>]
*/
```

Program Logic

Syntax analysis generator is implemented in the programming language C#. The classes are placed in three namespaces: `GrammarUtil`, `ParserGenerator` and `Output`.

`GrammarUtil` namespace contains utility classes used for working with the grammar and its productions: `GrammarSymbol`, `Production`, `GrammarLoader` and `Grammar`.

`GrammarSymbol` class represents terminal or nonterminal grammar symbol. It contains information about its type and value.

Production class represents a single grammar production. Each production has a head and a body. Productions head is the nonterminal `GrammarSymbol` that is found on the left side of the production and body is a list of `GrammarSymbol` instances that contains all the grammar symbols found on the right side of a production.

`GrammarLoader` is a static class that contains all the logic necessary to parse a grammar input file. The class has one public static method, `LoadGrammar` method, which takes as argument the path to the grammar input file and returns a new instance of the `Grammar` class. Class defines and three other methods, one for parsing each part of the grammar input file.

`Grammar` class contains all the information about a grammar:

- Grammar's productions
- Terminal and nonterminal symbols of a grammar
- Information about priority of the terminal symbols
- Information about associativity of the terminal symbols

Class also defines methods for easy access to this information, e.g., method that gets all the grammar's productions, method that gets all the productions for a certain nonterminal grammar symbol, etc.

`ParserGenerator` namespace contains the `Generator` class that implements syntax analysis generator and some additional classes that are specific to the parser generation process.

Item class describes a single LR(1) item, which is a production with a dot at some position in a body and a grammar symbol called the lookahead symbol, which is used to determine if a reduction should or shouldn't be made. Item class contains an instance of a production class, one integer that represents the position of the dot in a production's body and an instance of a `GrammarSymbol` class that represents a lookahead symbol.

`SetOfItems` class represents a set of LR(1) items. `SetOfItems` class is backed by a list, but ensures that there are no duplicate values in the collection. An instance of the `SetOfItems` class is used to represent a parser state in the parser generation process.

`LRTable` class is used to store the information about the transitions between states, or in other words the parser actions. For every state, three dictionaries are defined, one for each action type (shift, goto, reduce), except the accept action. There can be only one accept action, so it is defined separately. These dictionaries map the grammar symbol with an integer number, or a list of productions in the case of the reduce action. The list is used for the reduce actions to allow for possible reduce/reduce conflicts. Shift/reduce conflicts are also allowed because of the separate dictionary for the shift and reduce actions.

Generator class is the class that contains the required methods for creation of the parser. All of the algorithms used in the parser generation process can be found in [3].

Generator class contains the methods that calculate three different sets required in the parser generation process, these are: FIRST, CLOSURE and GOTO sets. Each method used to calculate one of those sets is called by the set it calculates.

The parser generation process starts with the creation of all of the parser states. As was mentioned before, each parser state is in fact a set of LR(1) items. So to generate all of the states of the future parser, one needs to generate all possible sets of items. For that, the `GenerateSetsOfItems` method is used.

The final method in the Generator class is the `GenerateParserActions` method. This method starts by calling the `GenerateSetsOfItems` method to get all the parser states. After that it creates a new instance of the `LRTable` class and starts adding the parser actions to it. The method loops through every state and checks the conditions defined by the algorithm to determine if a shift, goto, reduce or accept action should be created for that state.

To increase the performance of the syntax analysis generator few improvements were made. In the parser generation process the FIRST set if often calculated for the same grammar symbol. The algorithm that calculates the FIRST set is recursive in nature, so to avoid unnecessary work, the

calculated FIRST sets are saved to a dictionary. If it is required to calculate the FIRST set that was already calculated, the result is simply retrieved from the dictionary.

The other improvement, that significantly increased performance, was made in the code used to determine the next state when calculating shift and goto actions. The generator calculates the next state as a set of items, because this is how the states are represented in the generator. After the next state is calculated the generator needs to find the number of that state to be able to save the transition in the LR table. This means that the generator must compare the calculated set of items with all the sets of items that are in the list of parser states, until it finds a match. The number of parser states can easily grow to a few thousand, or even tens of thousands, and this process becomes very time consuming. To speed this up, the parser states were sorted according to the number of items in them and binary search was used to search the list of states.

`Output` namespace contains classes used to write the resulting LR table to a file. The table can be written as an XML or as a generated code in the parser template.

Syntax analysis generator works with the `OutputFile` interface and all of the classes that are used for writing the output files should implement this interface. This allows for easy replacement of the class used for writing the output file.

`XMLOutputFile` class is used to write the LR table as an XML file and the `ParserTemplate` class is used to write the LR table in the parser template. Both of those classes implement the `OutputFile` interface and the generator uses those classes only through this interface. The `ParserTemplate` class reads the parser template line by line and writes those lines into a new file that will be the output of the generator, in other words, it copies the parser template. When it encounters one of the placeholders, it writes the block of code that should replace that placeholder and continues with the process of copying the parser template.

Output Files

XML

XML created by the syntax analysis generator has the `<table>` element as its root element. In this element there are `<state>` elements. Each of those elements contains all the actions from the LR table that the parser can make if it is in this state. The `<state>` element has one required attribute; this is the value attribute whose value is an integer number that represents the number of the parser state that the element describes. The state with the number 0 is the initial parser state.

Actions in the <state> element are defined with the `<character>` element. That element can contain a number of attributes, depending on the action that it describes. The required attributes are the value, type and action attributes. The value attribute is used to define the value of the terminal or nonterminal grammar symbol for which the action is defined. The type attribute is used to define the type of this grammar symbol and can have the value of either “TERMINAL” or “NONTERMINAL”. Finally the action attribute defines the type of the action that the element describes and can have four possible values: “GOTO”, “SHIFT”, “REDUCE” and “ACCEPT”.

If the action type is “GOTO” or “SHIFT” than one other attribute needs to be defined, the state attribute, which is used to define the next state of the parser.

If the action type is “REDUCE” than two other attributes need to be defined, the lhs and the `rhs_count` attribute. The lhs (left hand side symbol) attribute defines the symbol that will be put on the parsing stack as a result of the reduce action. The `rhs_count` (right hand side count) attribute defines the number of the grammar symbols on the right side of the production that is going to be reduced. This attribute is needed by the parser so it knows how many symbols need to be taken off the parsing stack to make a reduction.

If the action type is “ACCEPT” then no other attributes need to be defined.

Part of the XML file with that defines the LR table is listed hereafter. Only the actions for the first three parser states are listed, and the grammar used to generate this file was the grammar defined in the chapter Grammar Input File.

```
<?xml version="1.0" encoding="utf-8"?>
<table>
 <state value="0">
  <character value="digit" type="TERMINAL" action="SHIFT" state="1" />
  <character value="(" type="TERMINAL" action="SHIFT" state="2" />
  <character value="&lt;S&gt;" type="NONTERMINAL" action="GOTO" state="3" />
  <character value="&lt;E&gt;" type="NONTERMINAL" action="GOTO" state="4" />
  <character value="&lt;T&gt;" type="NONTERMINAL" action="GOTO" state="5" />
  <character value="&lt;F&gt;" type="NONTERMINAL" action="GOTO" state="6" />
  <character value="$" type="TERMINAL" action="REDUCE" lhs="&lt;S&gt;"
         rhs_count="0" />
 </state>
 <state value="1">
  <character value="-" type="TERMINAL" action="REDUCE" lhs="&lt;F&gt;"
         rhs_count="1" />
  <character value="$" type="TERMINAL" action="REDUCE" lhs="&lt;F&gt;"
         rhs_count="1" />
  <character value="*" type="TERMINAL" action="REDUCE" lhs="&lt;F&gt;"
         rhs_count="1" />
  <character value="/" type="TERMINAL" action="REDUCE" lhs="&lt;F&gt;"
         rhs_count="1" />
  <character value="+" type="TERMINAL" action="REDUCE" lhs="&lt;F&gt;"
         rhs_count="1" />
 </state>
 <state value="2">
  <character value="digit" type="TERMINAL" action="SHIFT" state="7" />
  <character value="(" type="TERMINAL" action="SHIFT" state="8" />
  <character value="&lt;E&gt;" type="NONTERMINAL" action="GOTO" state="9" />
  <character value="&lt;T&gt;" type="NONTERMINAL" action="GOTO" state="10"
         />
  <character value="&lt;F&gt;" type="NONTERMINAL" action="GOTO" state="11"
         />
 </state>
 <state value="3">
  <character value="$" type="TERMINAL" action="ACCEPT" />
```

```
 </state>
…
</table>
```

Parser Template

If the LR table is going to be written into the parser template, than the parser template needs to have the placeholders defined. Those placeholders will be replaced by the generated code. The process used to generate the resulting parser and the required input files are explained in the previous chapters.

Action input files can be used in a number of different ways. They can contain the source code as simple as a call of a method to save the transition in the parser's internal table or can contain the whole code required to execute a certain type of the parser action. In this chapter an example of the input action file is given and the resulting parser is shown if the LR table is inserted in a very simple parser template.

Let's say that the parser template contains only two methods: a method called Initialize in which only the shift actions will be defined and a method for adding a new shift action in the parser's table called `AddShiftAction`. Initialize method, written in C#, might look something like this:

```
public void Initialize() {

  //[<SHIFT ACTIONS>]

}
```

If the shift action input file would look like this:

```
AddShiftAction([<CS>], "[<GS>]", [<NS>]);
```

And, if the generated LR table would contains only the first two shift actions from the LR table given as an example in the previous chapter, then the generated parser would look like this:

```
Public void Initialize() {

  AddShiftAction(0, "digit", 1);
  AddShiftAction(0, "(", 2);

}
```

5.3 Parser Templates

5.3.1 Motivation

Beside the parser template designed for multicore processors, which was the main goal of the project, another parser was developed, that runs on one processor core only. This was done in order to enable performance comparison between singlecore and multicore parsers.

5.3.2 Singlecore Parser Template Implementation

The initial parser template was created both as a preliminary template model and as a completely functional parser for a reduced SIP grammar. In this way it serves as a template for the parser generator and as a validation method for the generated parser, enabling the comparison of parsing procedures via generated and manually written parser.

The program logic of the manually developed functional parser was implemented on the basis of a manually written LR parsing table. The tokens were also defined explicitly. The following grammar was used:

```
<Response> ::= <Status-Line> <Message-Header> CRLF "body"
<Message-Header> ::= <Call-Id> <Message-Header>
<Message-Header> ::=
<Call-Id> ::= "Call-Id:user@user"
<Call-Id> ::= "i:user@user"
<Status-Line> ::= <SIP-Version> SP "200" SP <Reason-Phrase2> CRLF
<SIP-Version> ::= "SIP/" <DIGIT> "." <DIGIT>
<Reason-Phrase2> ::= <Reason-Phrase> <Reason-Phrase2>
<Reason-Phrase2> ::=
<Reason-Phrase> ::= "reserved"
<Reason-Phrase> ::= "unreserved"
<Reason-Phrase> ::= "escaped"
<DIGIT> ::= <BROJ><NAS>
<NAS> ::= <BROJ> <NAS>
<NAS> ::=
<BROJ> ::= 0
<BROJ> ::= 1
<BROJ> ::= 2
<BROJ> ::= 3
<BROJ> ::= 4
<BROJ> ::= 5
<BROJ> ::= 6
<BROJ> ::= 7
<BROJ> ::= 8
<BROJ> ::= 9
```

An example of a valid message is shown below:

```
SIP/298.234 200 reserved
Call-Id:user@user
body$
```

The definitions of parsing actions – the reduce, goto and shift actions were explicitly coded into the parser. The initial parser template differs from the initial functional parser by having a defined place in the code where the parser generator injects generated actions, instead of having them explicitly coded.

5.3.3 Multicore Parser Template Implementation

Parser's input files are parsing table and input SIP messages. The parsing table is written in XML form, as described in previous chapters. At the beginning, the parser loads the parsing table into data structures so it can easily access parsing actions. The parsing table consists of list of states in which actions for possible tokens are defined.

The following is a parsing table data structure:

```
class LRtable
    {

        private List<State> states = new List<State>();

        public List<State> States
        {
            get { return states; }
            set { states = value; }
        }
  }
```

Piece of code written in *C#* that was used to catch data from XML file:

```
xmlreader.MoveToAttribute("value");
string sign = xmlreader.Value;

xmlreader.MoveToAttribute("action");
string action = xmlreader.Value;

xmlreader.MoveToAttribute("state");
string  nextState = xmlreader.Value;

if (action == "SHIFT")
{
     shiftAction(index, sign, nextState);
}
else if (action == "GOTO")
{
     gotoAction(index, sign, nextState);
}
else if (action == "REDUCE")
{
     reduceAction(index, sign,indexOfReduce.ToString());
}
else
{
     acceptAction(index, sign);
}
```

The function `MoveToAttribute` moves a pointer to the attribute which name is submitted as function parameter. Property `Value` returns current attribute value. After all the data is collected, depending on action value, parsing table data structures are filled.

Parsing process is implemented as a single function that is called at the beginning of the program and every time a parsing attempt ends unsuccessfully. Function parameters are the index of the current token, the parser stack, the node stack, that stores the data for making and visualizing the parse tree, and the list of parser copies. The parser completely ends when there are no more available tokens or list of parser copies is empty. Every time the parser needs input token it calls lexical analysis methods. When parser ends successfully it returns the parsing tree as its output. The parser model is shown in Figure 5-2.

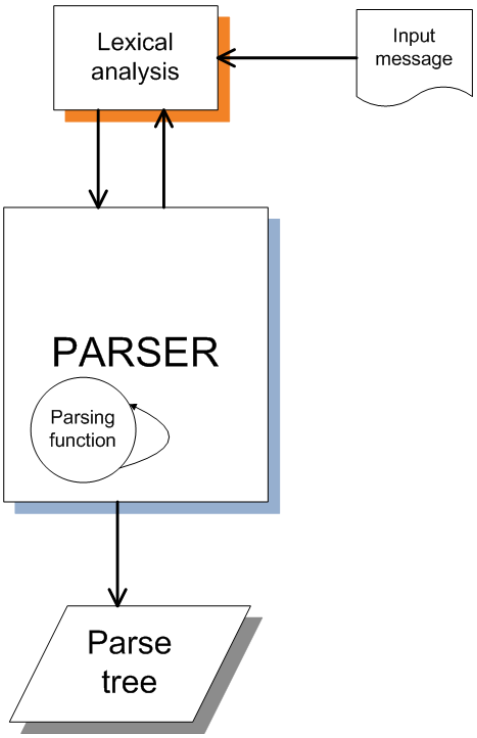

Figure 5-2 Parser model

Every time the parser finds conflicts, it automatically copies itself. The parser uses a special list in which it marks the current action as used, in order not to use the same action when it restores this copy. Each time one of the parser copies is restored, it searches mentioned list for available actions. The parser copy has several elements that unambiguously determine in which situation the parser is: current state, stack, token index, node stack, list for marking used actions and index of previously used token. Grammar complexity can lead to the lexical non-determinism. It is possible that the lexical analyzer can recognize several tokens at the particular moment.

There is no way of knowing in advance which token is right or wrong, so the parser tries to parse the message with each token. Each time this type of conflict occurs, the parser makes a copy of itself marking which token it used in previous step. The data structure that was used to copy the parser is shown hereafter:

```
public class CopiedState
{
     public Stack<string> stack;
     public string stateToCheck;
     public Dictionary<string, List<bool>> isUsedInpreviousState;
```

```
    public int tokenIndex;
    public Node root;
    public Stack<Node> nodeStack;
    public int indexOfPreviousToken;
}
```

Some parsers have *priority* and *associativity* rules that are related to conflict occurrences [3]. Rules are applied only if shift/reduce conflict occurs. Two elements relevant for the mentioned rules are the last token that was pushed to the stack and the input token. If the last pushed token and the input token are equal, the associativity rule applies. Associativity characteristic of the last pushed token determines which action is going to be executed. If associativity is "left" then reduction applies, otherwise shift applies. Priority rules are applied when the last pushed token and the input token are not equal. If priority of the last pushed token is greater than priority of the input token, then reduction applies, otherwise shift applies. The rules are defined in the grammar file used as the input file to the parser generator.

There are two possible reasons why parsing attempt could end unsuccessfully. The first reason is related to the lexical analysis. As it is said before, every time the parser needs an input token, it calls the lexical analysis. The function that returns a list of possible tokens is named `checkForAvailableTokens`. This function returns possible token, each one of which must be listed in the list of possible tokens that the function receives as the input parameter. This parameter is the list of tokens for which actions for current parser state are defined. So, if the set of tokens defined in the parsing table does not overlap with the set of possible tokens that were found in the lexical analysis, this function returns an empty list and the parsing attempt is unsuccessful. Simplified, if there are no actions defined in the parsing table for the current parser state and the next token, the parsing is unsuccessful.

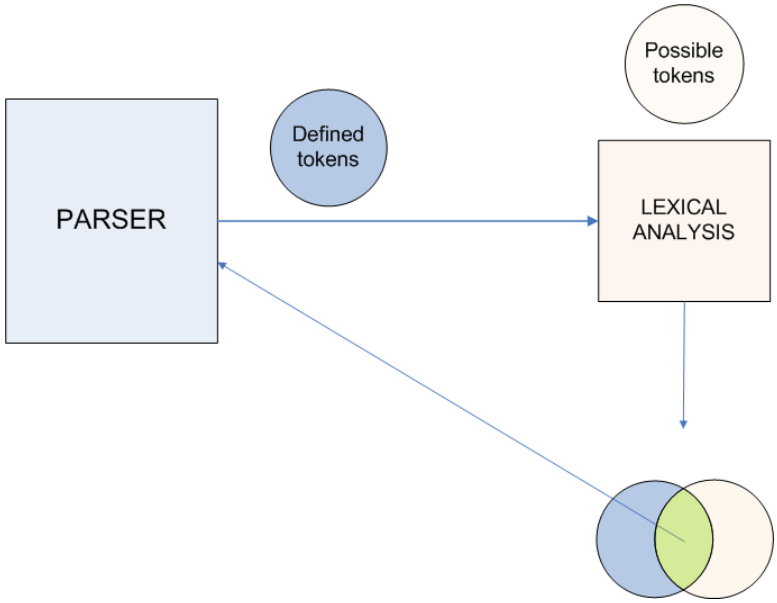

Figure 5-3 Parser to lexical analysis communication model

Other reasons come from the fact that the parsing table can be wrong, if it is written manually. Then the parser can stop at token obtaining, as it is described above, or when GOTO action is applied.

Despite the fact that this table errors come in small quantities, the parser is able to recognize them and start a new parsing attempt.

The parsing attempt is implemented as the `activateNewParsing` function. The function checks for a list of copied parsers named `copiedStates` for a new copy and activates a new parsing process. After that a previous parser copy is saved in `AlreadyUsedParserStates` list and every time a new copy is made, the parser checks this list for an identical copy. If that copy exists, the parser does not save that copy, but it tries to activate a new parsing process. Just because of that, the parser gathers the speed and avoids an infinite loops.

As it is said before, the parser output is a parsing tree. Each node represents the left side of one of the grammar productions except the leaves, which represent tokens. The rightmost derivation produces a tree in theory, but in implementation it is crucial to save the tree structure. Main reasons for saving the tree structure are semantic actions and visualization.

Semantic actions can be executed each time action `ReduceX` applies, but in fact their execution can be manipulated by "walking" through the tree. In our case, semantic actions are implemented as delegate functions that are called every time action `ReduceX` applies. They are similar to the inline functions, but are triggered on an event. Applying the `ReduceX` action or "walking" through every node would trigger an action submitted to the observed node.

Data structure that was used to save parse tree is listed here:

```
public class Node
    {
        private Node parent;
        private List<Node> children;
        private string NodeValue;


       public Node Parent
        {
            get { return parent; }
            set { parent = value; }
        }

        public List<Node> Children
        {
            get { return children; }
            set { children = value; }
        }

        public string Value
        {
            get { return NodeValue; }
            set { NodeValue = value; }
}
```

Each node has to know its parent node and its children nodes. The root node does not have a parent

node and the leaf nodes do not have any children. Each node can have just one parent node, but several children nodes, depending on the grammar. To save the whole tree, all it is necessary is a root node. The root node has children nodes that have children nodes, etc. Recursively, when some nodes children are *null*, node is a leaf, i.e. the token. Example of a simple tree is shown below on Figure 5-4.

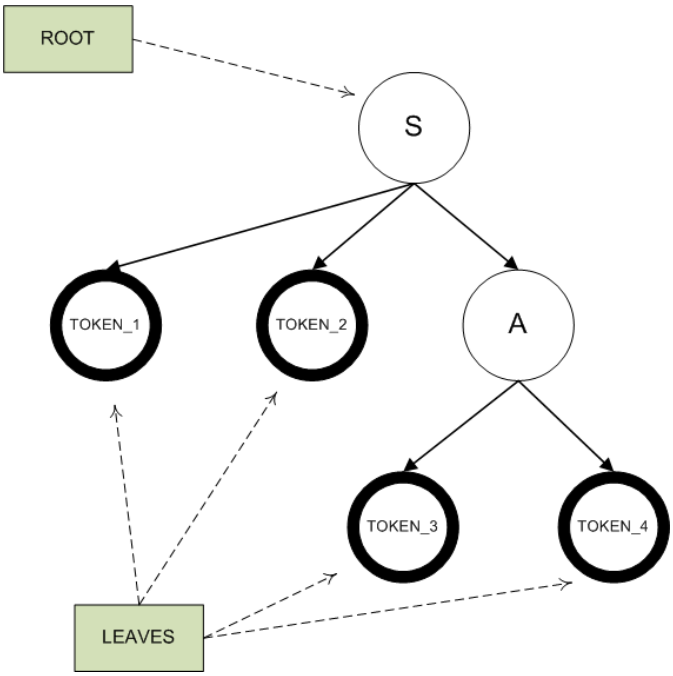

Figure 5-4 Parse tree example

Main task of this project was the development of a LR parser that will work on multicore processors. The main idea was to give each core a parser. Depending on the number of available cores, messages were parsed in parallel. In implementation, each parser is a class implemented in C#. Each core has its own parser class instance, and parsing works in separate thread. Simplified, for each core there exists a thread in which an independent parser parses the messages. This model is shown in the following picture.

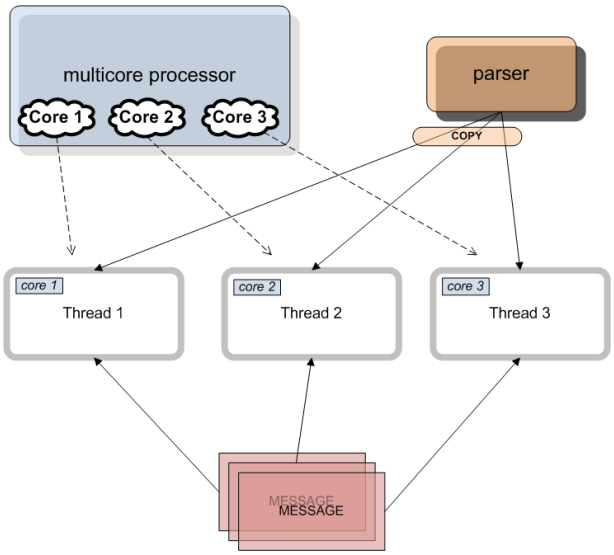

Figure 5-5 Model of a multicore parser with three cores

The program that runs the parser fetches the messages from the project folder. It creates threads depending on the number of available cores. Each thread has its own parser and it parses messages fetched from the mentioned folder. Each thread parses one message and all threads parse in parallel. The rogram measures parsing time and results showed that with greater number of cores, parsing time is smaller. Therefore parsing process is faster.

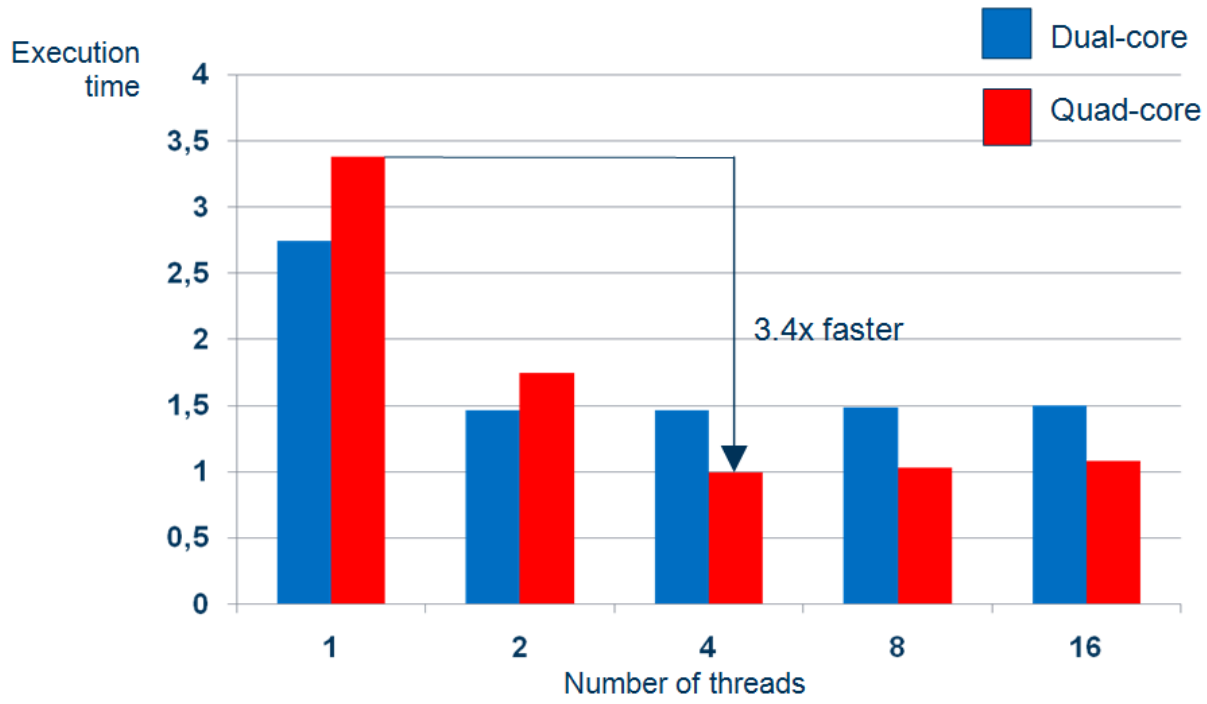

Figure 5-6 Parsing times on various number of cores

The preliminary measurements where the simplified SIP grammar was used are shown in Figure 5-6. The parser parsed 200 messages with 1, 2, 4, 8 and 16 threads. Processors that were used

are Dual-Core (3.00 GHz) and Quad-Core (2.66 GHz). Measurements showed that the parsing with four threads on Quad-Core (each thread on one core) is 3.4 times faster than the parsing with one thread.

5.4 Multicore Multi-instance Parsing

Parallel parsing described in previous chapters was done by running multiple SIP parsers, one for each core of a processor. In this chapter another way of parallel parsing is described, a way that enables parsing of a single SIP message on multiple processor cores. This is done by splitting the initial grammar in groups that we call parallel groups and building a parser for each of those groups. Such parsers are able to parse only some parts of a SIP message, when the message arrives, it is split into the parts that those parsers can parse, each parser parses its part and in the end the results are combined.

In this chapter the parallel groups and the process of creating them is described. After that the process of splitting the initial message and the process of combining the results is described. In the end this solution is compared to the solution described in previous chapters.

5.4.1 Parallel Groups

Parallel group is a group of grammar's productions that is self-sufficient, i.e., it contains all the productions that are required to reduce any sequence of terminal symbols that the group accepts into the nonterminal grammar symbol that is the group's initial symbol. If the grammar is displayed as a tree whose root is the grammar's initial nonterminal symbol, then parallel group could be any subtree of that tree. Parallel group also has its initial nonterminal symbol, which is the root of the subtree that represents that parallel group.

The procedure for constructing a parallel group is similar to the procedure for calculating CLOSURE set and a parallel group can be constructed from any production. The procedure is recursive and starts with a single production. The nonterminal symbol on the left side of that production will be the group's initial symbol.

The procedure for finding the set of all the productions of the parallel group (PG):

1. Add initial production to the set PG
2. For every production P from the set PG
    a. For every nonterminal symbol S on the production's P right side
        i. Add all the productions for symbol S to the set PG
3. Repeat step 2 until no more productions are added in one round

As was mentioned earlier a parallel group can be constructed for every production, but not all groups constructed in such a way are good for constructing parsers that will parse parts of the input message. Parallel groups that are good for that task must fulfill some other requirements.

First of all, parallel groups must be chosen in such a way that the initial SIP message can be easily and efficiently split into the parts which will be parsed by the each parallel group. SIP protocol is a text based protocol so the initial message can be parsed line by line. Because of that, parallel groups

for the SIP grammar are those groups which can parse one or more complete lines of the SIP message. SIP message can easily be split into lines, but some additional information is required to determine which line should be sent to which parser. Because of that the FIRST set for each group is calculated and groups are constructed in such a way that there are no common elements in any two group's FIRST sets. This allows that the decision to which parser a line should be sent can be made based only on the first token of the line that is being processed.

Second requirement is to allow that the results received by the parsers that parse parts of the message can easily be combined into the final result, i.e., that the parse trees received by the parallel group parsers can easily be combined into the final parse tree that describes the whole message. To allow this, there shouldn't be any overlaps between parallel groups. In other words, no two parallel groups should be able to parse the same part of the message. If overlaps would be possible then the received parse trees would need to be checked if they contain some common elements before the final parse tree could be generated. That could be a very time consuming process for more complicated messages. That also means that there can't be any parallel group that is a subgroup of some other parallel group. If the grammar is ordered in a way that the productions that can be reached from the initial grammar symbol are at the beginning, productions that can be reached from the nonterminal symbols on the right side of those productions are after them, and so on, then some productions from the start of the grammar wouldn't be in any group. These productions form a special group that is used for combining the results of the parallel group parsers; this process is described in more detail in the chapter Combining The Results.

Productions of the SIP grammar that are likely to fulfill all of the requirements described above are the productions that parse message headers, productions for status and request line and productions for the message body. Some of those productions may have common elements in their FIRST sets so they need to be put in the same parallel group if we wish to fulfill the first requirement.

A group that is formed from more than one parallel group contains the productions from every one of its parallel subgroups, the FIRST sets of every subgroup are merged and additional productions must be added to link the subgroups together. These additional productions are constructed by adding a new nonterminal symbol to the group that will be the new initial nonterminal symbol of the group. New productions have this new initial symbol on the left side and one of the initial symbols of a subgroup on the right side. For every subgroup one of these productions is constructed. This allows for any two parallel groups to be merged together, which is very useful because we can control how many parallel groups will be created by the algorithm.

Steps to split the grammar into parallel groups:

1. Find the productions that parse one (or more) complete lines of the message
2. Find the parallel group for every one of such productions
3. Check that no two of those parallel groups contain common elements in their FIRST sets
    a. If such groups exist, merge them into a new group
4. (optional) Merge the parallel groups to get the desired number of groups
5. All the productions from the initial grammar that don't belong to any of the created groups form a special group that will be used for merging the results

5.4.2 Implementation

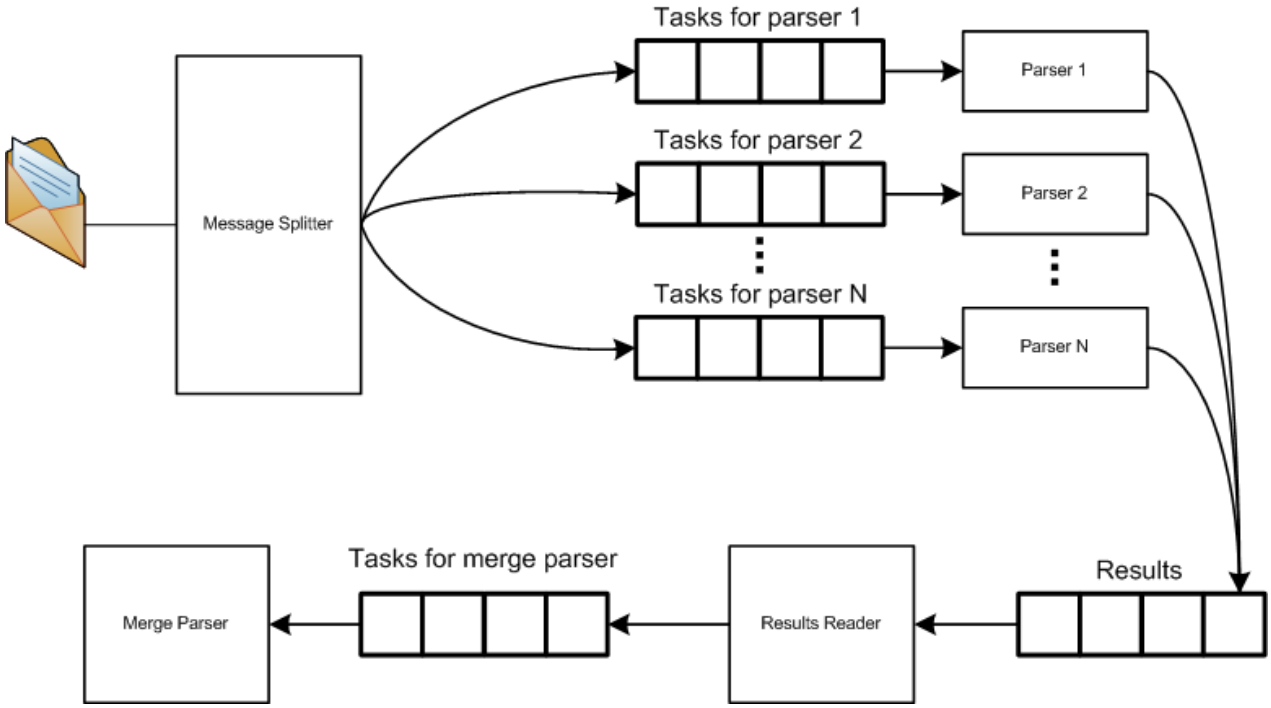

Figure 5-7 Architecture of a parser based on parallel groups

Figure 5-7 shows the architecture of the system that uses parallel groups in order to parse the message. Rectangles with names inside them are the components of the system and every one of them is running in its own thread. These components communicate using queues. Message splitter receives the message and splits it into tasks which will be processed by the parsers, each one of which is able to parse a single parallel group. When a parser completes the parsing of a task it stores the result in the results queue. Results reader monitors that queue and retrieves the results from it. Results reader groups the results that belong to one message and when it collects all the parts of any message it sends those parts to the merge parser. Merge parser is a special kind of parser whose task is to merge the results produced by the parallel group parsers into a single result that is the same as the result which would be produced by a normal SIP parser that is built from the whole SIP grammar.

5.4.3 Splitting The Message

When an input message comes in, it is read line by line and the first token in each line is recognized. A parser that should be able to parse the current line is determined based on that first token and a new parser task is created. Parser tasks are not sent to the parser right away, but are saved at least until the next line is read. That is done because it is possible that the task is not complete, i.e., that the parallel group parses one or more of the next lines in the message. For the SIP grammar that can happen only if the first token in the next line is a space character.

One other problem may occur in this process. SIP grammar has the productions that are designed to allow future extensions to the protocol. If this is to be supported then situations may occur in which it is not clear to which group a task should be sent. Good thing is that the tokens used in those productions are the most basic ones (e.g., A-Z, a-z …) and those are shorter than the tokens that

are found at the beginning of the other productions that form parallel groups. Because of that the first token can be recognized by taking the longest token that the lexical analyzer can find.

5.4.4 Combining The Results

Results of the parallel group parsers are a special kind of tokens. These tokens contain a name and a parse tree that was created by the parser. In order to combine the parse trees that are the results of the parallel group parser, a special parser is constructed.

The grammar used to construct this parser consists of the slightly modified productions of a special group that is created in the last step of the algorithm used to split the grammar into parallel groups, i.e., the productions that don't belong to any of the created groups. Some of the nonterminal grammar symbols in those productions are turned into terminal symbols with the same name. Nonterminal symbols that are modified in such a way are the nonterminal symbols that are the initial nonterminal symbols of any of the generated parallel groups. This allows that the final parse tree is constructed by parsing the results of the parallel group parsers.

A slightly modified version of a parser is constructed in order to achieve this. This parser takes as input the result tokens from the parallel group parser and does everything as a normal parser would do, with one small exception. When a reduction needs to be made, this parser takes the parse trees from the input tokens and combines them by connecting them with a common parent

5.4.5 Performance Comparison

Execution times of a parser that uses parallel groups, a full SIP parser running in one thread and four full SIP parsers running each in one thread are compared in this chapter. Seven different messages from SIP torture tests were selected as input messages. First test is done with these seven messages and for the following tests these messages were copied in order to test the parsers behavior with higher loads. The results are shown in the following figure.

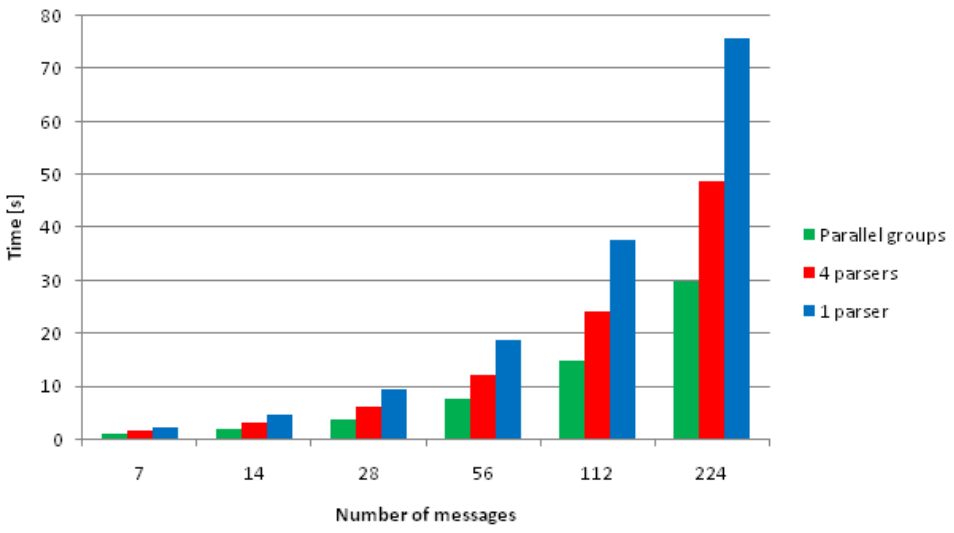

Figure 5-8 Execution times of different parsers

Parallel parsing based on the parallel groups is roughly 1.6 times faster than the 4 full SIP parsers running each in one thread. Implementation based on parallel groups uses five threads for parsing, but one of those threads parses the body of a message and every message used for testing has an empty body. That means that four threads are actually used for parsing. Three other threads are used: one for reading the input message, one for reading the results of the parallel group parsers and one for merging the results. Figure 5-9 shows the number of tasks that each of the parallel groups parsed when parsing initial seven test messages.

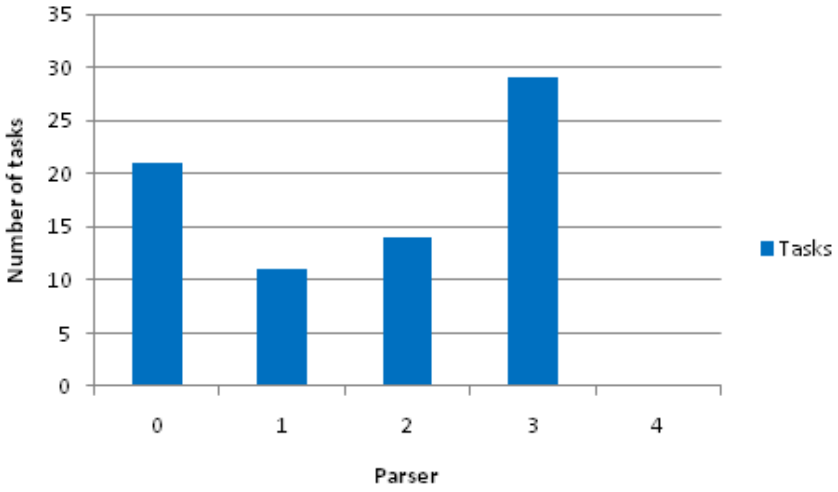

Figure 5-9 Number of tasks per parallel group in seven initial test messages

Previous figure shows that the distribution of tasks among parallel groups is not ideal. If the parallel groups would be organized in a way that the tasks are evenly distributed among them, then the results would probably be even better. This can be achieved if some sort of statistical data about the types of messages, more precisely the types of headers present in SIP messages, would be available. Based on that data, parallel groups could be created so that the expected number of tasks for each group would be about the same.

One additional advantage of parallel groups approach is seen when error messages are parsed. In those situations parser must backtrack and examine all the possible solutions before it can conclude that the message is incorrect. SIP grammar has many conflicts so this greatly prolongs the parsing time. Parallel group approach uses smaller grammar groups, so that means that the parser has much less backtracking to do. The worst case is when the error is at the end of the message, because then the parser has the most backtracking to do. The differences between parallel groups approach and the full SIP parser running in one thread for that case are shown on Figure 5-10. The test was done by measuring the parsing time when parsing one error message, with the error at its end, 14 times.

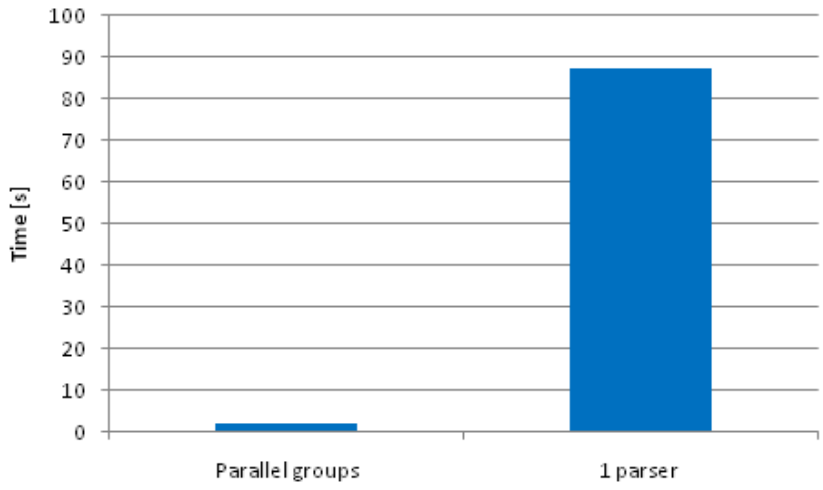

Figure 5-10 Execution time when parsing 14 error messages

5.5 Visualization Of The Parsing Trees

ParsingTreeGraph is a Windows form application which visualizes a tree of a successfully parsed SIP message and is used to simplify the understanding of the parsing process. The graph is visualized using the Microsoft Automatic Graph Layout. MSAGL is a .NET tool for laying out and visualizing directed graphs. A graph is created in three steps: creating the graph object, setting the nodes and edges, and then binding the graph object with the viewer. An example of a tree representing a parsed message is shown below on Figure 5-11. The grammar used is complete SIP grammar, and the parsed message is a standard SIP message.

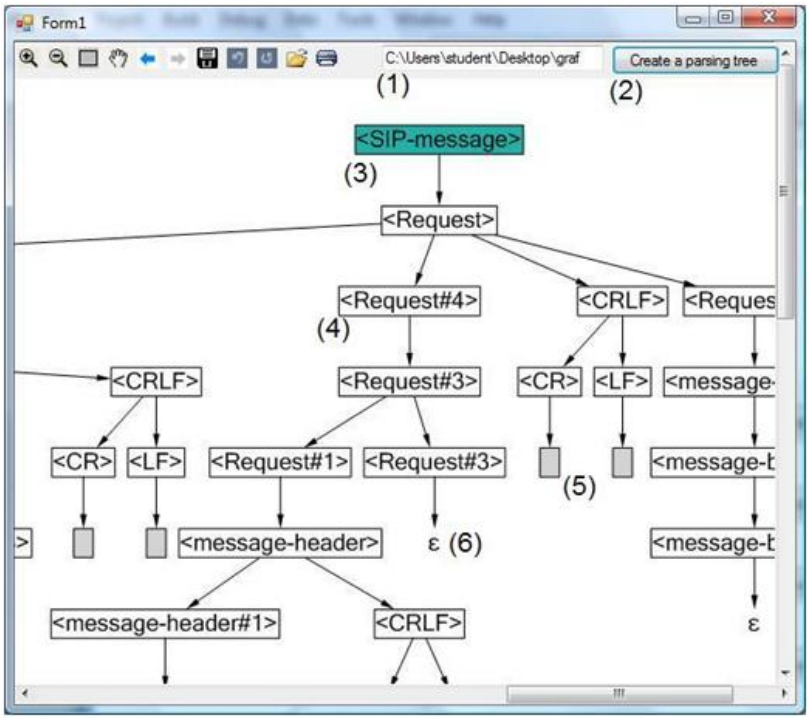

Figure 5-11 Parsing tree graph

The gray nodes (5) represent the final tokens of the grammar, i.e., the lexical units of the parsed SIP message. Read sequentially from left to right, they form the complete SIP message. The ε (epsilon) nodes (6) do not stand for any final token, but symbolize epsilon productions of their precursory nonterminal symbols. The white nodes (4) represent the nonterminal symbols. The sea-green node (3) represents the initial nonterminal symbol. A production is represented via parent node and its children, the parent node being the left-hand nonterminal symbol, and the children nodes being the right-hand symbols of the production.

The program structure and usage is described hereafter. The project references `GeneratedParallelParser` project which enables access to its classes and thus the creation of a parser object. The folder containing the SIP messages to be parsed is specified in the textbox (1). The parser reads the messages sequentially, parses them and creates a list of parsing tree structures of the parsed messages. The created list is passed from the parser object to the program. From this list the root node of one tree to be drawn is chosen (because of the exemplary character of the `ParsingTreeGraph` project, the choice of the first root representing a successfully parsed tree is defined in advance) and passed to the `DrawGraph` function. The function traverses the tree structure, sets the nodes and creates the edges, thus connecting the parent nodes and their children. The nodes are formatted according to their category (the initial nonterminal symbol node, the nonterminal tokens symbol nodes, the final tokens symbol nodes and the epsilon symbol nodes).

6 References

[1] Jezični procesori 1: Uvod u teoriju formalnih jezika, automata i gramatika – Siniša Srbljić, Neven Elezović – Element, 2000

[2] Jezični procesori 2 – Siniša Srbljić – Element, 2002

[3] Compilers: Principles, Techniques and Tools – Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Wesley – Addison, 1986

[4] Internet Communications Using SIP – Henry Sinnreich, Alan B. Johnston, Wiley Computer Publishing, 2001

[5] Online at: http://www.w3schools.com/xml/default.asp

[6] PJSIP SIP Stack, http://www.pjsip.org/

[7] sipXecs - Open Source IP PBX for Unified Communications, http://www.sipfoundry.org/

[8] Sofia-SIP Library, http://sofia-sip.sourceforge.net/

7 Appendices

7.1 Appendix A - Example of a SIP message (Torture Test RFC 5518)

```
INVITE sip:user@example.com SIP/2.0
To: "I have a user name of
extremeextremeextremeextremeextremeextremeextremeextremeextremeextreme
proportion"
sip:user@example.com:6000;unknownparam1=verylonglonglonglonglonglonglong
longlonglonglonglonglonglonglonglonglonglongvalue;longparamnamenamenamen
amenamenamenamenamenamenamenamenamenamenamenamenamenamenamenamenamen
amenamename=shortvalue;verylonglonglonglonglonglonglonglonglonglonglongl
onglonglonglonglonglonglonglonglonglonglongParameterNameWithNoValue
F:
ip:amazinglylongcallernameamazinglylongcallernameamazinglylongcallernameamaz
inglylongcallernameamazinglylongcallername@example.net;tag=12982982982982982
98298298298298298298298298298298298298298298298298298298424;unknownheaderpa
ramnamenamenamenamenamenamenamenamenamenamenamenamenamenamename=unknowhe
aderparamvaluevaluevaluevaluevaluevaluevaluevaluevaluevaluevalue;unknownValu
elessparamnameparamnameparamnameparamnameparamnameparamnameparamnameparamnam
eparamname
Call-ID:
longreq.onereallyreallyreallyreallyreallyreallyreallyreallyreallyreallyreall
yreallyreallyreallyreallyreallyreallylongcallid CSeq: 3882340 INVITE
Unknown-LongLongLongLongLongLongLongLongLongLongLongLongLongLongLong-
Name: unknown-longlonglonglonglonglonglonglonglonglonglonglonglong-
value; unknown-
longlonglonglonglonglonglonglonglonglonglonglonglonglonglonglonglonglong
-parameter-name = unknown-
longlonglonglonglonglonglonglonglonglonglonglonglonglonglonglonglonglong
long-parameter-value
Via: SIP/2.0/TCP sip33.example.com
v: SIP/2.0/TCP sip32.example.com
V: SIP/2.0/TCP sip31.example.com
Via: SIP/2.0/TCP sip30.example.com
ViA: SIP/2.0/TCP sip29.example.com
VIa: SIP/2.0/TCP sip28.example.com
VIA: SIP/2.0/TCP sip27.example.com
via: SIP/2.0/TCP sip26.example.com
viA: SIP/2.0/TCP sip25.example.com
vIa: SIP/2.0/TCP sip24.example.com
vIA: SIP/2.0/TCP sip23.example.com
V :  SIP/2.0/TCP sip22.example.com
v :  SIP/2.0/TCP sip21.example.com
V  : SIP/2.0/TCP sip20.example.com
v  : SIP/2.0/TCP sip19.example.com
Via : SIP/2.0/TCP sip18.example.com
Via  : SIP/2.0/TCP sip17.example.com
Via: SIP/2.0/TCP sip16.example.com
Via: SIP/2.0/TCP sip15.example.com
Via: SIP/2.0/TCP sip14.example.com
Via: SIP/2.0/TCP sip13.example.com
```

```
Via: SIP/2.0/TCP sip12.example.com
Via: SIP/2.0/TCP sip11.example.com
Via: SIP/2.0/TCP sip10.example.com
Via: SIP/2.0/TCP sip9.example.com
Via: SIP/2.0/TCP sip8.example.com
Via: SIP/2.0/TCP sip7.example.com
Via: SIP/2.0/TCP sip6.example.com
Via: SIP/2.0/TCP sip5.example.com
Via: SIP/2.0/TCP sip4.example.com
Via: SIP/2.0/TCP sip3.example.com
Via: SIP/2.0/TCP sip2.example.com
Via: SIP/2.0/TCP sip1.example.com
Via: SIP/2.0/TCP host.example.com;received=192.0.2.5;branch=very<repeat
count=50>long</repeat>branchvalue
Max-Forwards: 70
Contact:
sip:amazinglylongcallernameamazinglylongcallernameamazinglylongcallernameama
zinglylongcallername@host5.example.net
Content-Type: application/sdp
l: 150
```