Development of a Cross-Platform Interpreter for the muLISP Language

Andrey Chernetsov
Dept.of Applied Mathematics and Artificial Intellegence
National Research University "MPEI
Moscow, Russia
0000-0001-7655-2395

Dmitry Shevtsov
Dept.of Applied Mathematics and Artificial Intellegence
National Research University "MPEI
Moscow, Russia
ShevtsovDY@mpei.ru

Oleg Ivanov
Dept.of Applied Mathematics and Artificial Intellegence
National Research University "MPEI
Moscow, Russia
IvanovOlY@mpei.ru

2024 7th International Conference on Information Technologies in Engineering Education (Inforino) | 979-8-3503-7136-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/Inforino60363.2024.10551992

***Abstract*— Lisp is one of the basic functional programming languages. There are many modifications and implementations of language dialects, but stable and simple implementations that have proven their reliability over time are often used in the educational process. Such implementations include muLisp. The paper is devoted to the development of a cross-platform interpreter for the muLISP language. The main points of its internal structure are outlined: analysis of input expressions, representation of built-in functions, the computation process and memory management mechanisms. Numerous software tools used to develop the compiler are presented.**

Keywords*— *interpreter, cross-platform, LISP, context-free grammar, finite state machine

I. Introduction

The existence of different approaches to writing programs has led to the emergence of many programming languages that provide different features for them.

Most languages traditionally belong to the so-called imperative model – this is dictated by the architecture that most modern computers use. However, there are other programming paradigms. An example would be logical or functional.

To study them, simple and at the same time visual programming languages are required. For the functional approach, one of these languages is Lisp (from the English LISt Processing).

Its main features are:

- focus on working with lists;
- focus on processing symbolic information
- the ability to present any complex structured information (through the use of nested sublists);
- self-applicability of the language (the program and data are presented in a uniform list form, which allows you to use the same list in different ways, depending on the situation);
- natural recursiveness (a function calling itself either directly or through other functions);
- simplicity of syntax;
- reference memory organization, automatic release of unused memory;
- dynamic typing.

Lisp has many implementations and dialects, the most significant of which are: Common Lisp [1], Scheme [2], Racket [3], Clojure [4], muLISP [5]. However, most of them are not very suitable for studying, which leads to the fact that implementations of old dialects are often used in the educational process due to their simplicity. Thus, at the Department of Applied Mathematics and Artificial Intelligence (AM&AI) of the National Research University "MPEI", implementations of the muLISP language from the 80s of the 20th century, developed for the MS-DOS operating system (OS), are used. Its use is difficult on modern systems, which creates the need to create a new implementation of this dialect.

II. MULISP Language

muLISP is a dialect characterized by a relatively small set of functions and small size [5]. As a result, it is relatively simple to learn. It is used in the educational process [6-8].

Like any dialect, muLISP has a number of distinctive features [8]:

- availability of a mechanism for auto-quoting atoms (by default, the value of a symbol is equal to its name);
- ability to set methods for processing and calculating arguments when defining a function;
- support for implicit operators (simplified PROGN and COND notation);
- using formal parameters as local variables.

Let us briefly consider the following systems that implement this dialect: muLISP-85, muLISP-87, Frame Oriented Instrumental System (FORIS, based on muLISP-87). There are a number of differences between them in the set of built-in functions, some differences in their implementation:

- muLISP-87 expands the set of built-in functions of muLISP-85, changing the input/output system (including the file one);
- FORIS complements muLISP-87 with many development tools – a window system, graphical tools for working with files, as well as the FRL language [9]. Also in this system there are changes to some built-in functions (for example, DRIVER, GETD, BREAK).

S-expression is the basic data structure in muLISP, a symbolic expression that is an atom or dot pair [5].

A macrosymbol is a symbol for which a special function is specified that is called when read [5]. Such symbols allow the user to dynamically introduce new syntactic constructs into the language, thereby changing the syntax of the language at runtime.

III. Design and Implementation of the Interpreter

An analysis of the structure and features of the muLISP-87 system was carried out, on the basis of which a new muLISP interpreter was designed and implemented, preserving the main functionality of this system.

The main properties that the new interpreter provides are:

- the ability to use the interpreter as a separate program (console application) and as part of another program (for example, an integrated development environment [10]);
- possibility of controlling the calculation from the outside: pausing and continuing, complete stop;
- cross-platform – support for Windows OS and Linux OS;
- implementation of the main features of the language.

Main features of the internal architecture of the interpreter are:

- application of finite state machines in the implementation of built-in functions;
- implementation of syntactic analysis based on context-free (CF) grammar, lexical analysis taking into account macrosymbols and environment settings;
- memory management, combining different methods;
- implementation of the interpreter as a separate class in C++;
- lack of muLISP functionality oriented towards working with MS-DOS OS (for example, screen control or direct work with memory);
- support for loading FRL language [10].

A. Used Software

The tools used to develop the interpreter are described in Table 1.

TABLE I. Tools Used to Develop

Link	TOOL		
	Purpose	***software***	***Version***
	Main programming language.	C++	17
[11]	Cross-platform build system.	CMake	3.8 and upper
	Linux Compiler.	GCC	7.0 and upper
	Windows Compiler.	MSVC	1919 and upper
[12]	A library designed for calculations with floating point, integer and rational numbers with arbitrary precision. Used to implement integer and fractional numbers.	GMP	6.0.0 and upper
[13]	A collection of C++ class libraries that provide a convenient cross-platform interface for everyday programming tasks. Used in the interpreter to implement a reference type.	BOOST	1.76
[14]	A library for working with the JSON text format. In the interpreter it is used to save and load state.	JSON	3.0 and upper
[15]	Lexical analyzer generator. Used in the interpreter to generate state machines for lexical analysis.	RE/FLEX	3.2.12 and upper
[16]	A parser generator that converts an annotated context-free grammar into an LALR(1) parser. In the interpreter it is used to generate a parser.	Bison	3.7.1
	Python programming language. Used to convert files during interpreter build.	Python3	3.11.0

Thus, cross-platform is ensured through the use of cross-platform development tools indicated in Table 1.

B. Parsing Input Expressions

The organization of analysis of input expressions in this paper differs from the implementation in classic muLISP. It is based on a CF grammar and uses a special lexical analyzer that allows you to distinguish lexemes differently depending on the environment settings, as well as recognize macro characters. The general scheme for analyzing expressions is shown in Fig. 1.

The lexical analyzer uses several finite state machines to implement the recognition of macro characters, which can either interrupt identifiers (interrupting macro characters) or not interrupt (non-interrupting ones) [5]. Moreover, finite state machines use transitions not only by a simple symbol, but also transitions that depend on the environment settings (for example, the number system, the presence of an interrupting macrosymbol).

The simplified algorithm of this analyzer is as follows:

1. the first character read, if it is used in the grammar (period, parentheses, comma, etc.), is checked to ensure that it is a macro character. If it is, then it is returned as a token of the macrocharacter, otherwise – as a token representing this character;

2. if the symbol is not used in the grammar and is a macrosymbol, then it is returned as a macrosymbol token;

3. otherwise, further analysis occurs, which uses a set of finite state machines (the set is needed to simplify the analysis scheme).

Syntactic analysis is organized sequentially: the lexeme received from the lexical analyzer (integer number, fractional number, string, identifier, macro or special character) is transferred to the push-pull LALR(1) analyzer [17] generated on the basis of the CF-grammars. The analyzer saves its state between the transmission of lexemes and returns the completion code of the next iteration: analysis is completed, syntax error, analysis continues. While the analysis continues, tokens are read from the input stream and passed to the analyzer, and error handling occurs through the BREAK function.

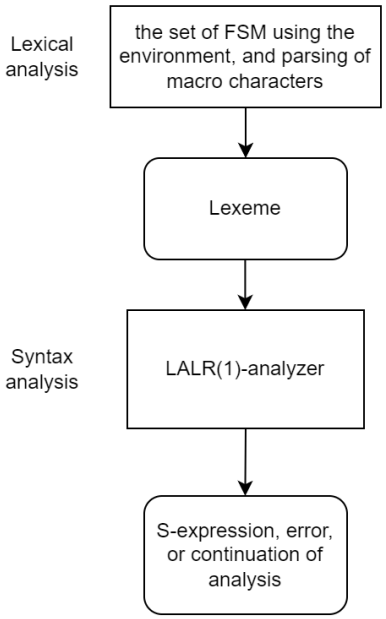

Fig. 1. General Scheme for Analyzing Expressions.

This approach allows the use of a parser generator, and also provides the ability to pause and interrupt parsing expressions externally.

C. Representation of Built-in Functions

One of the main properties that the developed interpreter has is the ability to control calculations from the outside. To implement this property, a special representation of built-in functions is used.

Functions are divided into complex and simple depending on their time computational complexity. A function is classified as simple if it satisfies the following restrictions:

- does not call complex built-in functions;
- does not have long loops (without a pre-known upper bound on the size – depending on the size of the input data).

The process of calculating simple functions is not interrupted, but for complex ones it is divided into sequential stages, the calculation of which is also not interrupted. Subject to the restrictions of a simple function within each stage, the process of calculating a complex function is equivalent to calculating a sequence of simple ones. This partition allows you to interrupt the calculation of a complex function when moving between its stages.

In the implementation of complex functions, the mechanism of finite state machines is used: functions are represented in the form of automata classes that make it possible to save the state of the function execution between stages (states of the machine), transitions between which are carried out using special commands. There are two such classes for each function.

The first class (common for all functions) defines the external state, ensures the fulfillment of restrictions through their external implementation: allows the use of loops and calling other complex functions. It is also used for other actions that do not directly affect the internal data of the function: returning a result, evaluating arguments, etc. To implement them, a clipboard is used between the function and the external control loop. The structure of the machine of this class is shown in Fig. 2.

The second class defines the internal state, which reflects the process of calculating a specific function: it changes its internal data and, using the first class, implements the logic of the function as a whole.

An illustration of the structure of an arbitrary complex function is shown in Fig. 3.

D. Computation Process

The calculation of S-expressions, function calls, macro expansion and other language mechanisms in this architecture are implemented through separate built-in auxiliary functions. The evaluation of any simple built-in function is implemented through a complex auxiliary function. Thus, the computation process is the execution of a given complex built-in function.

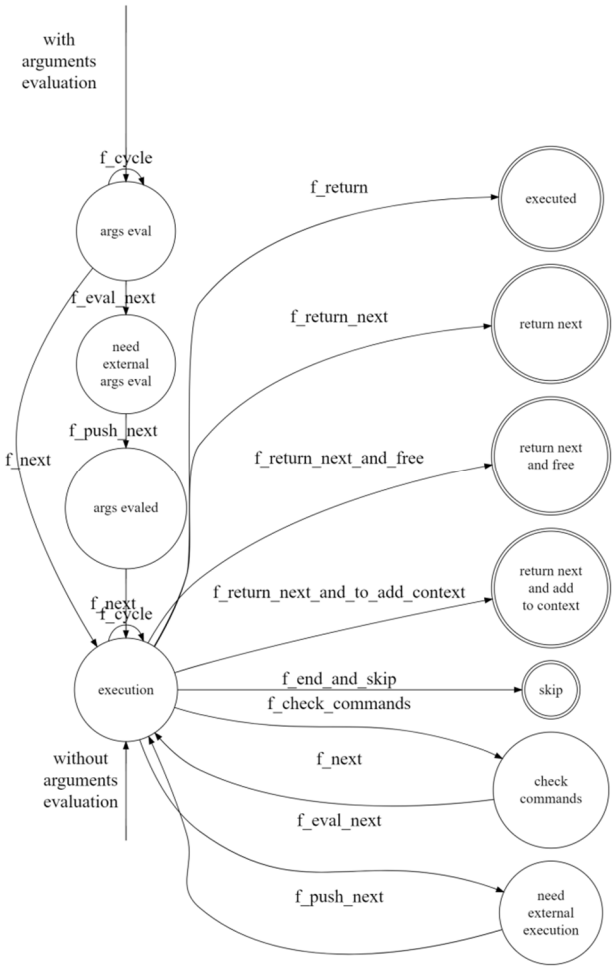

Fig. 2. Structure of an External State Machine of a Complex Function.

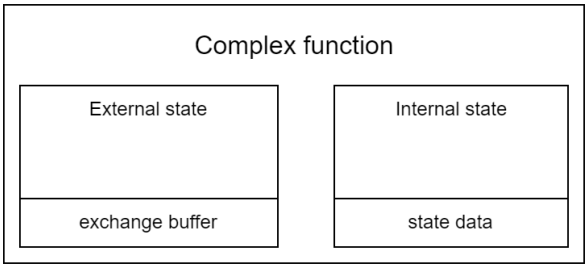

Fig. 3. Structure of a Complex Function.

This process consists of sequential transitions between the states of an external automaton of a complex function, and the processing of its requests, the data of which is transmitted through the clipboard of the external state. This is how all elements of calculation are implemented: calling another function, calculating arguments, organizing a loop and returning the result, etc.

Also during the calculation process, unconditional transitions, out-of-memory errors, and external commands (for example, commands to stop or pause the calculation) are processed.

An illustration of the computation process is presented in Fig. 4.

Main calculation loop

The basis for calculating an arbitrary built-in complex function is the following mechanisms:

- the calculation has a normal flow (without system transitions - unconditional transitions out of memory), which occurs cyclically and ends with the result in the form of an S-expression;
- when computing, there are several stacks: a stack of complex functions and a data stack (used to store data or states of completed functions);
- when a transition occurs or memory shortage occurs, the normal flow of computation is interrupted and processing of the corresponding situation is started. If its processing does not complete the calculation, then the calculation continues in the normal flow.

Normal computation flow

A simplified algorithm for the normal flow of computation is as follows: while the function stack is not empty, a function is retrieved from it and, depending on the state of its external automaton, various actions are performed, leading either to a change in the state of the function or to a change in the stacks. In addition, after a fixed number of iterations, external interpreter commands are processed, allowing the normal computation flow to be adjusted (pause, resume, etc.).

The external function automaton (Fig. 2) acts as the main element of the calculation: through it the calculation of arguments is started, the result is returned, and also requests for additional actions are made – calling other functions, moving to the data stack, additional processing of external commands, deleting from the function stack.

The normal thread loop ends when the last function on the stack has returned a value. This value is the result of the calculation (along with any side effects).

Handling low memory situations

If an interrupt occurs, signaling a lack of memory in the system, the current calculation is terminated abnormally and the stacks are cleared. If the calculation took place in driver mode (i.e. using the top-level DRIVER function), then the DRIVER function is added to the stack. In any mode, the BREAK function is added to the stack with a “MEMORY FULL” error interrupt.

Exception Handling

The exception system (unconditional jumps) consists of the THROW, UNWIND-PROTECT and CATCH functions [5]. When implementing it, the exception mechanism of the C++ language is used, with the help of which the normal flow of calculation is interrupted.

When processing a transition called by the THROW function, the function stack is cleared until a function is found that can handle it. Processing is then passed on to the found function.

If there are no functions left on the stack and the transition is not processed, then information about it is passed to the top.

Processing external commands

External commands are a mechanism that allows you to influence a calculation from the outside.

The interpreter supports 4 external commands:

- Pause – pauses the calculation, waits for the “continue” command;
- Continuation – removes the pause;
- Stop – stops the calculation, the interpreter returns the corresponding completion code;
- Call CONSOLE-INTERRUPT-BREAK – calls the “CONSOLE-INTERRUPT-BREAK” interrupt function.

E. Memory Management

During the calculation, many dynamic objects are used, which are constantly created and destroyed. In this case, memory is allocated and freed on the heap. This process requires a lot of time. Therefore, the problem arises of minimizing the number of such operations.

muLISP uses a two-look garbage collection approach [5], but due to the use of different tools, a different approach to memory management was used in the implementation. It combines reference counting and block memory allocation, as well as an automatic deallocation strategy [18].

Reference counting makes it easier to work with S-expressions, it automatically destroys unused data and simply increases the counter when copying. And block allocation with an automatic release strategy is used to control the memory of complex functions and S-expressions.

The main idea of this approach is that memory allocation occurs in blocks. In this case, the automatic release strategy is responsible for releasing free blocks.

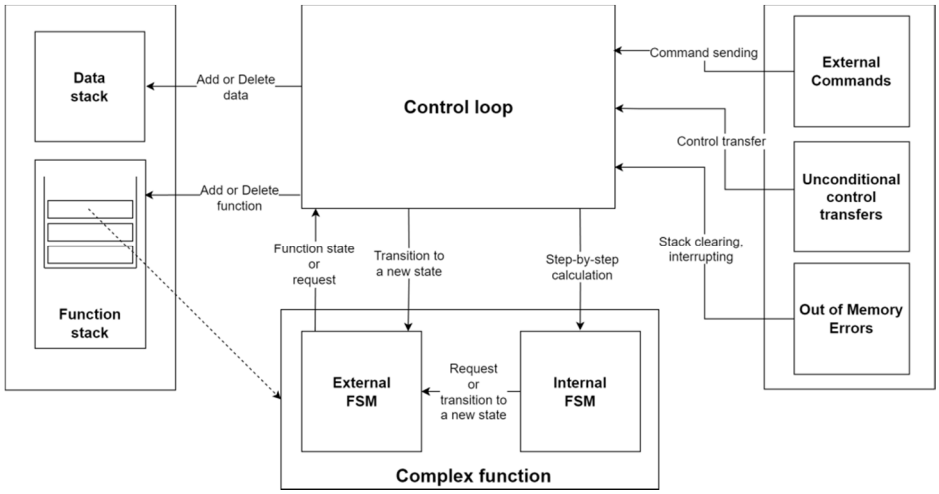

Fig. 4. Illustration of the Computation Process.

In the interpreter, a strategy is a class that implements a certain behavior. To automatically clear free memory blocks, the following strategies are used:

- nothing – does not automatically free memory;
- greater_then – releases memory if the number of free blocks exceeds a certain fixed number;
- predicat – releases memory if the number of free blocks satisfies an arbitrary condition;
- the last strategy has another variation: freeing memory when the number of free blocks exceeds a certain proportion in the total number of blocks. It is used by default in the interpreter.

This achieves a more uniform dynamic allocation and release of memory compared to the implementation of the FORIS system.

F. External Interfaces

The interpreter provides interfaces that allow you to conveniently work with it. These include:

- input and output interfaces: define the interfaces of the input and output streams with which the interpreter works. Based on them, any input/output methods can be implemented – using a buffer, console, etc.;
- control command interface: defines the interface through which the interpreter receives external commands;
- main interface of the interpreter – a class that is the entry point to the interpreter. Allows you to configure its creation and launch. In addition, it supports simultaneous creation of multiple instances, which allows you to run multiple interpreters in one application.

IV. Using the Interpreter and Testing

A. Using the Interpreter

The interpreter contains more than 240 language functions, implementing all the main mechanisms. In addition to the functions of muLISP, the interpreter supports launching with the FRL language loaded. The implementation contains 20 thousand lines of source code and 2.2 thousand comments to it, 147 source code files.

The implementation is made in the form of a C++ library and uses one dynamically linked library – GMP. Thus, to use the interpreter as a separate program (a simple console application), 2 conditions must be met:

- the presence in the system of the GMP library (its dynamic modules) is not lower than the version used when developing the interpreter;
- the presence of a locale in the system with CP1251 encoding.

If the interpreter is used as part of another program, then additional requirements are versions of compilers and the C++ language standard not lower than those used during assembly. Another requirement is the absence of optimizations (at linkage) that could give two functions the same address (for example, /OPT:ICF [19] in MSVC).

To launch the interpreter, it is enough to have classes that implement the input and output stream interfaces, as well as control commands.

B. Testing

Due to the specifics of the application being developed (language interpreter), there is an external specification of how the program should work – a description of each function. Therefore, “black box” testing was used in the work.

As part of testing, 6472 tests were developed for various language functions. Testing was carried out in 6 operating systems: Windows 7, Windows 10, CentOS 7, Manjaro, Alt Linux, Astra Linux. The results of testing on Astra Linux OS are presented in Fig. 5.

V. Conclusion

In this paper, an analysis of the structure and features of the muLISP-87 system was carried out, and the architecture of a new cross-platform muLISP language interpreter for modern operating systems was designed.

Based on the developed architecture, a set of language functions has been implemented, consisting of 245 elements, which also supports loading the FRL language.

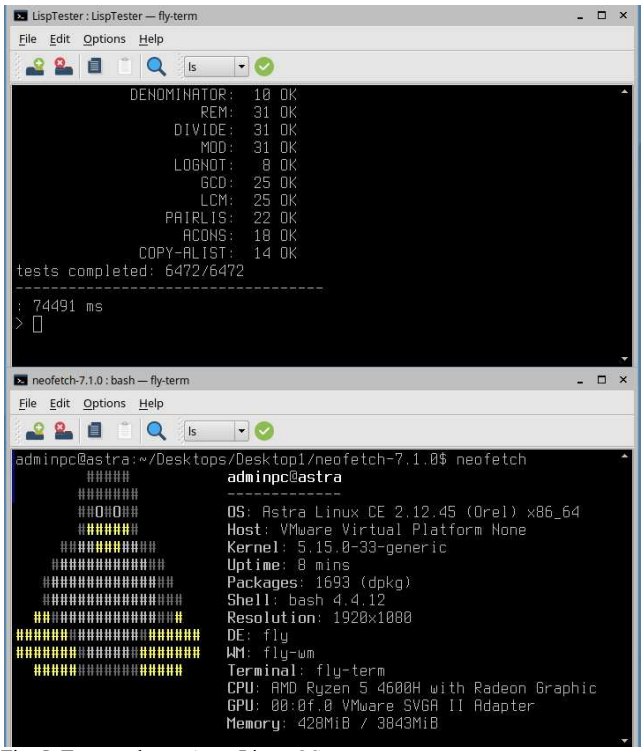

Fig. 5. Test results on Astra Linux OS.

The developed interpreter was tested and debugged, including on various operating systems (Windows 7, Windows 10 [20], CentOS 7 [21], Manjaro [22], Astra Linux [23], Alt Linux [24]).

In total, more than 6.4 thousand tests were developed during testing, and the results of their launch were successful.

The new interpreter can be used in the educational process at the Department of AM&AI within the discipline “Data Structures and Programming Methods” when studying the muLISP and FRL languages instead of the outdated FORIS system.

This paper, after a many-year break, continues research in the field of system software development carried out at the Department of AM&AI of the National Research University "MPEI".

References

[1] [Online] Common Lisp Language URL – https://common-lisp.net/ ([Date accessed: 11/21/2023).

[2] [Online] Scheme Language URL – https://www.scheme.org/ ([Date accessed: 11/21/2023).

[3] [Online] Racket Language URL https://racket-lang.org/ ([Date accessed: 11/21/2023).

[4] Reference Manual muLISP-90: LISP Language Programming Environment / 3-d Edition., July 1990, SoftWarehouse Inc., Honolulu, Hawaii, USA.

[5] Bolshakova E.I., Gruzdeva N.V. Fundamentals of programming in the Lisp language: Textbook. / MAX Press, 2010 – 112 p. (in Russian).

[6] [Online] Introduction to LISP language URL http://it.kgsu.ru/Lisp/lisp0001.html [Date accessed: 05/21/2023] (in Russian).

[7] Baidun V.V., Kruzhilov S.I., Sergievsky A.E., Chernov P.L. “Programming in the LISP language in the muLISP-90 system” / M.: MPEI, 1993. 40 p. (in Russian).

[8] Semenov M. Yu. Lisp language for personal computers: Textbook. allowance / Moscow State University. M. V. Lomonosov. – Moscow: Moscow State University Publishing House, 1989. – 70 p.

[9] R. Bruce Roberts, Ira P. Goldstein. The FRL Manual, MIT 1977 URL - https://dl.acm.org/doi/10.5555/889244

[10] Dmitry Shevtsov, Oleg Ivanov “Name”, “RADIO ELECTRONICS, ELECTRICAL ENGINEERING AND ENERGY: Twenty-ninth Int. scientific-technical conf. students and graduate students (March 16–18, 2023, Moscow): Abstracts. Report” – M.: LLC "Center for Printing Services "Rainbow"", 2023. – 1240 p., pp. 216. (in Russian)

[11] [Online] CMake URL – https://cmake.org ([Date accessed: 05/21/2023).

[12] [Online] GMP URL – https://gmplib.org ([Date accessed: 05/21/2023).

[13] [Online] Boost URL – https://www.boost.org ([Date accessed: 05/21/2023).

[14] [Online] JSON for modern C++ URL – https://github.com/nlohmann/json ([Date accessed: 05/21/2023).

[15] [Online] RE/FLEX URL – https://github.com/Genivia/RE-flex ([Date accessed: 05/21/2023).

[16] [Online] Bison URL – https://www.gnu.org/software/bison/ ([Date accessed: 05/21/2023).

[17] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman Compilers: Principles, Techniques, and Tools / Pearson Education, Inc, 2006.

[18] Andrei Alexandrescu Modern C++ Design: Generic Programming and Design Patterns Applied, Addison-Wesley Professional, 2001.

[19] [Online] MSVC linker reference URL – https://learn.microsoft.com/en-us/cpp/build/reference/opt-optimizations?view=msvc-170 ([Date accessed: 05/21/2023).

[20] [Online] Windows 10 Homepage URL – https://www.microsoft.com/en-us/software-download/windows10%20W ([Date accessed: 11/21/2023).

[21] [Online] CentOS 7 Homepage URL – https://www.centos.org/ ([Date accessed: 11/21/2023).

[22] [Online] Manjaro Homepage URL – https://manjaro.org/ ([Date accessed: 11/21/2023).

[23] [Online] Astra Linux Homepage URL – https://astralinux.ru/en/ ([Date accessed: 11/21/2023).

[24] [Online] Alt Linux Homepage URL – https://en.altlinux.org/Main_Page ([Date accessed: 11/21/2023).