Оригинальная статья / Original article

https://doi.org/10.21869/2223-1536-2023-13-3-135-145

УДК 004.451

Использование метапрограммных средств языка Common Lisp для разработки систем эмуляторов

А. А. Чаплыгин[1] ✉

[1] Юго-Западный государственный университет
ул. 50 лет Октября, д. 94, г. Курск 305040, Российская Федерация

✉ e-mail: alex_chaplygin@mail.ru

Резюме

Цель исследования *заключается в анализе и использовании метапрограммирования на языке Common Lisp при проектировании и реализации эмуляторов, симулирующих аппаратуру вычислительных систем. Рассматриваются виды метапрограммирования, макросредства языка Common Lisp, использование макросов для метапрограммирования.*

Методы. *Язык Lisp характеризуется использованием единообразных S-выражений для представления данных и программ. Таким образом, данные могут являться частью программы, и наоборот: программа может быть данными. Макросредства Common Lisp позволяют напрямую модифицировать абстрактное синтаксическое дерево программы, и возможно создание новых синтаксический конструкций для решения поставленной задачи. При реализации функций эмулятора макросредства языка Common Lisp могут быть использованы для генерации функций, где общая часть функций входит в макрос, а различия функций задаются в параметрах при вызове макросов. Примеры таких макросов включают работу с битовыми регистрами статуса, генерацию арифметических команд, команду сравнения, работу с памятью и др. Таким образом, можно значительно уменьшить размер программы.*

Результаты. *В результате компьютерного моделирования был разработан и реализован симулятор архитектуры NES (процессор MOS 6502) на объектно-ориентированном языке программирования C# и на языке с поддержкой метапрограммирования Common Lisp. В результате симулятор, написанный на языке с поддержкой метапрограммирования, оказался более чем в 2 раза меньше, чем симулятор, написанный на языке C#.*

Заключение. *Использование метапрограммирования (на примере создания эмуляторов) может значительно сократить объем программы, упростить и улучшить архитектуру программы, уменьшить число ошибок и повысить качество программ. Использование предметно-ориентированных языков позволяет существеннее сократить объем кода программы.*

Ключевые слова: *метапрограммирование; макрос; Common Lisp; эмулятор; макроподстановка; предметно-ориентированное программирование.*

Конфликт интересов*: Авторы декларируют отсутствие явных и потенциальных конфликтов интересов, связанных с публикацией настоящей статьи.*

Для цитирования: Чаплыгин А. А. Использование метапрограммных средств языка Common Lisp для разработки систем эмуляторов // Известия Юго-Западного государственного университета. Серия: Управление, вычислительная техника, информатика. Медицинское приборостроение. 2023. Т. 13, № 3. С. 135–145. https://doi.org/10.21869/ 2223-1536-2023-13-3-135-145.

Поступила в редакцию 11.07.2023 *Подписана в печать 08.08.2023* *Опубликована 29.09.2023*

Using Metaprogramming Tools of the Common Lisp Language for the Development of Emulator Systems

Aleksandr A. Chaplygin[1]✉

[1] Southwest State University
50 Let Oktyabrya Str. 94, Kursk 305040, Russian Federation

✉ e-mail: alex_chaplygin@mail.ru

Abstract

The purpose of research *is to analyze and use metaprogramming in the Common Lisp language when designing and implementing emulators that simulate computer system hardware. The metaprogramming, the macro tools of the Common Lisp language and the use of macros for metaprogramming are considered.*

Methods. *The Lisp language is characterized by its use of uniform S-expressions to represent data and programs. Thus, data can be part of a program and vice versa: a program can be data. Common Lisp macro tools allow you to directly modify the abstract syntax tree of a program, and thus it is possible to create new syntactic constructs to solve a given problem. When implementing emulator functions, macro tools of the Common Lisp language can be used to generate functions, where the common part of the functions is included in the macro, and the differences between the functions are specified in the parameters when calling the macros. Examples of this macros are: bit status register macros, generation of ariphmetic commands, comparation commands, memory commands. Using that you can significantly reduce the size of the program.*

Results. *As a result of computer modeling, a simulator of the NES architecture (MOS 6502 processor) was developed and implemented in the conventional object-orientied C# programming language and in the Common Lisp metaprogramming language. As a result, the simulator written in a language with metaprogramming support turned out to be more than 2 times smaller than the simulator written in C#.*

Conclusion. *The use of metaprogramming (using the example of creating emulators) can significantly reduce the size of a program, simplify and improve the program architecture, reduce the number of errors and improve the quality of programs. The use of domain specific languages lets reduce code size even more.*

Keywords: *metaprogramming; macro; Common Lisp; emulator; macro substitution; domain specific languages.*

Conflict of interest: *The Authors declares the absence of obvious and potential conflicts of interest related to the publication of this article.*

For citation: Chaplygin A. A. Using Metaprogramming Tools of the Common Lisp Language for the Development of Emulator Systems. *Izvestiya Yugo-Zapadnogo gosudarstvennogo universiteta. Serija: Upravlenie, vychislitel'naja tekhnika, informatika. Meditsinskoe priborostroenie = Proceedings of the Southwest State University. Series: Control, Computer Engineering, Information Science. Medical Instruments Engineering.* 2023; 13(3): 135–145. (In Russ.) https://doi.org/10.21869/2223-1536-2023-13-3-135-145.

Received 11.07.2023 *Accepted 08.08.2023* *Published 29.09.2023*

***

Введение

Метапрограммирование [1; 2] – это техника программирования, при которой компьютерные программы могут рассматривать другие программы как данные, т. е. читать, генерировать, анализировать или преобразовывать другие

программы. Во многих случаях это позволяет минимизировать число строк кода для решения задачи, соответственно сокращая время разработки. Также это дает гибкость программам, чтобы обрабатывать новые ситуации без перекомпиляции.

Существует три подхода к реализации метапрограммирования. Первый подход – это использование прикладного программного интерфейса (API), с помощью которого предоставляется доступ к внутренним объектам среды времени выполнения [3; 4]. Этот подход используется при генерации промежуточного языка в среде .NET. В языке C# присутствует инкрементальный компилятор, т. е. программа может модифицироваться во время выполнения.

Второй подход метапрограммирования – динамическое выполнение программных выражений в виде строк или других объектов (JavaScript). Это позволяет программе генерировать код и выполнять его очень быстро [5; 6; 7]. При этом следует отметить, что внутренний язык может отличаться от внешнего.

Третий подход заключается в применении компилятора как универсальной системы преобразования кода программы с использованием описаний преобразований. Это дает возможность использовать метапрограммирование с практически любыми целевыми языками, даже с теми, которые не имеют средств метапрограммирования. Впервые этот подход был реализован в языке Scheme [8].

Одно из важных применений метапрограммирования – это предметно-ориентированные языки [9; 10; 11]. Для реализации предметно-ориентированного языка используются генераторы лексических и синтаксических анализаторов. В этом случае язык описывается с помощью регулярных выражений и контекстно-свободных грамматик. Сгененрированная программа эффективно разбирает язык, используя сложные алгоритмы.

Другое применение метапрограммирования – динамический анализ программ [12]. Это включает модульное тестирование, отладку, измерение метрик.

Использование метапрограммирования требует повышенных навыков по сравнению с обычным программированием [13]. Так как метапрограммирование предоставляет большую гибкость и настраиваемость, то любая неточность или некорректное использование может привести к неожиданным ошибкам, которые очень сложно отладить. Это приводит к рискам при разработке и уменьшает надежность при неаккуратном использовании методов метапрограммирования. Поэтому эти методы используют в основном опытные разработчики.

Язык Лисп [14; 15] был первым языком, который позволял обрабатывать программу как код. Он использовал списки как основную структуру данных. Также Лисп обладал свойством гомоиконичности, т. е. текст программы имеет такую же структуру, что и абстрактное синтаксическое дерево, и программа

есть объект данных первого класса. Оператор «цитирование» использовался, чтобы отложить вычисление до времени выполнения. Таким образом, метаязык в Лиспе совпадает с родительским языком, что позволяет расширять уже существующие подпрограммы. Лисп применялся для создания приложений с искусственным интеллектом. Также он получил широкое распространение в системах автоматизированного проектирования [16].

Язык Scheme позволяет определять так называемые гигиенические макросы. Эти макросы гарантируют отсутствие случайных совпадений идентификаторов при раскрытии.

Язык Common Lisp [17; 18; 19] – диалект языка Лисп, который был опубликован как стандарт ANSI. Это язык общего назначения, который представляет собой комбинацию процедурной, функциональной и объектно-ориентированной парадигм. Так как он является динамическим языком, то он поощряет эволюционную и инкрементальную разработку программ с эффективной компиляцией. Важной особенностью Common Lisp является инкрементальная разработка без прерывания работы программы, т. е. добавление и модификация кода работающей программы.

Разработка систем эмуляторов [20] компьютеров требует тщательного проектирования всех подсистем: центрального процессора, памяти, шины, видеоадаптера. Если происходит эмуляция настоящего оборудования, а не виртуального, то разработчик сталкивается с большим объемом информации и с большой сложностью поведения реальной аппаратуры. Это приводит к большой сложности программной системы и, следовательно, к большому объему исходного кода. Использование методов метапрограммирования позволяет значительно уменьшить сложность и объем исходного кода программной системы при разработке систем эмуляторов. Язык Common Lisp обладает развитыми средствами метапрограммирования, в частности развитой системой макросов. Эти макросы могут быть использованы для реализации предметно-ориентированного встроенного языка описания поведения компонентов эмулятора, и таким образом, упростить и уменьшить программный код.

Материалы и методы

Наиболее распространенный способ метапрограммирования в Common Lisp заключается в написании макросов. Макрос – это оператор, который реализуется через трансформацию. При определении макросов мы указываем, каким образом необходимо выполнить преобразование. Само преобразование выполняется компилятором, – это есть раскрытие макроса. Например макрос (1):

```
(defmacro nil! (x)                    (1)
      (list 'setf x nil))
```

при вызове (2):

```
(nil! a)                              (2)
```

будет преобразован в (3):

```
(setf a nil)                          (3)
```

и лишь затем скомпилирован и вычислен.

В макросах мы можем указывать, какую часть абстрактного синтаксического дерева нужно преобразовывать и каким образом. Например, заменить на параметр макроса, или выполнить предварительные вычисления, или вставить несколько параметров как список. Отличие макросов от функций заключается в управлении порядком вычислений, т. е. можно задавать, в какой последовательности будут вычисляться параметры и части кода макроса, а в функции аргументы всегда вычисляются перед вызовом функции. На языке Common Lisp для этого используется обратная кавычка (как оператор предотвращения вычислений) и операторы «запятая» (указание подстановки символа) и «запятая-at» (указание подстановки списка). Например макрос цикла с предусловием можно реализовать таким образом (4):

```
(defmacro while (test &rest body)  (4)
   `(do ()
          ((not ,test))
          ,@body))
```

Здесь сначала тело цикла (остаточный параметр body) собирается внутри списка, этот список вместе с условием test вставляется внутрь макроса, а затем будет выполнен оператор цикла do.

Макрос может быть раскрыт в другой макровызов. Если компилятор встречает такой макровызов, то он раскрывает его до тех пор, пока не останется ни одного макроса.

Эмулятор вычислительного устройства состоит обычно из следующих компонентов: центральный процессор, память, устройства ввода-вывода. Каждый из этих компонент является достаточно сложным в устройстве и в поведении, поэтому необходим индивидуальный подход при проектировании данных компонентов.

Центральный процессор включает в себя регистры. Это ячейки памяти для вычислений. Их моделирование реализуется достаточно просто, за исключением регистра флагов. Здесь данные хранятся в индивидуальных битах, каждый – на своей позиции. Чтобы извлечь или установить бит, необходимо использовать несколько побитовых операций, а так как обращение к флагам происходит очень часто, например после каждой арифметической операции, то получается много повторений одного и того же кода. С помощью макросов мы можем сгенерировать все необходимые функции (чтение флага, установка флага, сброс флага) для каждого флага.

Например, макровызов (5):

```
(make «zero» 1)                                                    (5)
```

приводит к генерации функций (6), (7), (8):

```
(defun get-zero ()                                                 (6)
      (ash (logand ST (ash 1 1)) (- 0 1)))
(defun set-zero ()                                                 (7)
      (setf ST (logior ST (ash 1 1))))
(defun clear-zero ()                                               (8)
      (setf ST (logand ST (lognot (ash 1 1)))))))
```

При реализации логических побитовых команд возникает повторяющийся код. Например, команды AND (побитовое И), EOR (исключающее ИЛИ), ORA (побитовое ИЛИ) проводят следующие действия:

– применяют логическую функцию к содержимому аккумулятора и операнда, записывая результат в аккумулятор;

– устанавливают флаги нуля и знака;

– вычисляют дополнительный цикл процессора.

Эти повторы кода можно заменить, используя макрос, в котором в нужное место для вычисления подставится соответствующая логическая функция, таким образом, после макроподстановки создаются необходимые функции.

Другие функции, которые можно сгенерировать при помощи макросов:

– операции сдвигов (различие в направлениях и обработке флагов);

– операции условных переходов (различаются условиями по состояниям флагов);

– операции сравнения (используют разные регистры);

– операции уменьшения / увеличения регистров;

– операции загрузки из памяти (используют разные регистры);

– генерация таблицы кодов и операций.

Например, для операций условных переходов можно сделать следующий макрос (9):

```
(defmacro make-br (fun flag res)   (9)
  "Команды условного перехода"   ; fun – имя генерируемой функции
                                 ; flag – какой флаг анализируется
                                 ; res – значение флага для перехода
  `(defun ,fun (adr)
    (let ((op (to-signed (mem:rd adr))))   ; чтение операнда
      (when (= (,flag) ,res)               ; условие выполнения перехода
        (setf add-cycle (+ 1 (is-cross PC op)))  ; счетчик циклов процессора
        (setf PC (+ PC op))))))            ; выполнение перехода
```

Тогда команды переходов генерируются следующим образом (10) – (17):

```
(make-br BCC |get-carry| 0) ;Переход если нет переноса          (10)
(make-br BCS |get-carry| 1) ;Переход если перенос               (11)
(make-br BEQ |get-zero| 1) ;Переход если равно                  (12)
(make-br BNE |get-zero| 0) ;Переход если не равно               (13)
(make-br BMI |get-neg| 1) ;Переход если меньше                  (14)
(make-br BPL |get-neg| 0) ;Переход если больше                  (15)
(make-br BVC |get-over| 0) ;Переход если не переполнение        (16)
(make-br BVS |get-over| 1) ;Переход если переполнение           (17)
```

В результате созданы функции: BCC, BCS, BEQ, BNE, BMI. BPL, BVC, BVS. Нет необходимости дублировать повторяющийся код программы.

Результаты и их обсуждение

В качестве эмулятора для реализации был выбран эмулятор NES (процессор MOS 6502). Он был реализован на объектно-ориентированном языке программирования (C#) и с помощью мета программирования (язык Common Lisp). Оба варианта реализуют одинаковые функции эмулятора.

Система NES включает в себя следующие подсистемы:

– центральный процессор;

– оперативная память (адресное пространство);

– постоянная память;

– система ввода-вывода;

– графическая подсистема.

Центральный процессор MOS 6502 имеет более 150 инструкций, из которых уникальных команд около 50, различия заключаются в видах адресации. На обычном языке программирования приходится реализовывать все 150 инструкций, а с помощью метапрограммирования число функций меньше 50, потому что многие команды имеют общие части, которые могут быть перенесены в макросы.

Адресное пространство NES состоит из оперативной памяти, стека, постоянной памяти, отображенных в память регистров ввода вывода и видео. Метапрограммирование позволяет создать таблицу диапазонов адресов и перенаправить операции чтения и записи в память в соответствующие модули.

Постоянная память NES имеет переключающиеся банки. С помощью макросов легко создать необходимые функции по записи данных в нужные области памяти.

Видеоподсистема имеет свой набор регистров, работу с которыми также упрощается при помощи макросов (установка индивидуальных битов).

Результаты реализации показаны в таблице 1.

Таблица 1. Результаты реализации системы NES на объектно-ориентированном языке и с помощью метапрограммирования

Table 1. Results of NES system implementation in object-orientied language and in metaprogramming language

Подсистема	Число строк на языке C#	Число строк на языке Common Lisp	Коэффициент (отношение)
Центральный процессор	1396	543	2,57
Постоянная память	185	66	2,80
Ввод — вывод	56	26	2,15
Память	230	72	3,19
Графический процессор	894	540	1,56
Главная программа	135	38	3,55
Переключение банков	229	91	2,52
Всего	3250	1404	2,31

Анализ проведенных исследований свидетельствует о том, что программа, созданная при помощи метапрограммирования, сократилась по объему программного кода в два раза по сравнению с программой, написанной традиционным способом (объектно-ориентированное программирование).

Выводы

Таким образом, метапрограммирование позволяет существенно сократить объем программы. Это влияет на скорость разработки: скорость разработки значительно повышается. Меньший объем программы – это упрощение программы, улучшение ее архитектуры, а значит, и меньшее количество ошибок, повышенное качество. Метапрограммирование может быть успешно применено не только при написании систем эмуляторов, но и в других прикладных областях.

В качестве недостатков метапрограммирования следует отметить уменьшение скорости работы программы, вызванное необходимостью выполнять преобразование и подстановку макрофункций, а также динамической типизацией данных. Эти недостатки могут быть нивелированы с помощью указаний типов для функций и непосредственной (inline) подстановкой макросов и функций (поддерживается в Common Lisp).

Дальнейшее развитие идей метапрограммирования заключается в применении предметно-ориентированных языков. Они позволяют существеннее упростить программный код за счет использования макрофункций, напрямую отражающих и отображающих предметную область. Предметно-ориентированное программирование – это область дальнейших исследований.

Список литературы

1. Ingebrigtsen E. Metaprogramming in C#. Birmingham, UK: Packt Publishing Limited, 2023. 353 p.

2. Лоторейчик В. Ю. Метапрограммирование на основе текстового препроцессора // Научно-технический вестник Санкт-Петербургского государственного университета информационных технологий, механики и оптики. 2006. № 25. С. 57–65. EDN JURWLF.

3. What APIs Are and Why They Matter. URL: https://www.f5.com/ labs/articles/education/still-mystified-by-apis-what-they-are-really-and-why-they-matter (дата обращения: 26.05.2022).

4. Аникин Д. А. Анализ методов авторизации и аутентификации REST API // Международный журнал информационных технологий и энергоэффективности. 2023. Т. 8, № 5-2(31). С. 120–124. EDN DZKSMQ.

5. Денисов Д. Э. Сравнение Typescript и Javascript для web-разработки // Студенческий вестник. 2023. № 20-8(259). С. 57–58. EDN MQCOML.

6. Назарова О. В. Интерактивный учебник по Java Script // Хроники объединенного фонда электронных ресурсов. Наука и образование. 2017. № 7(98). С. 63. EDN ZRRDGB.

7. Скотт Адам Д., Макдоналд М., Пауэрс Шелли. JavaScript. Рецепты для разработчиков. 3-е изд. СПб.: Питер, 2023. 528 с.

8. Абельсон Х., Сассман Д. Структура и интерпретация компьютерных программ. М.: Добросвет: КДУ, 2012.

9. Эванс Э. Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем. М.: Вильямс, 2020. 448 с.

10. Сухов А. О. Классификация предметно-ориентированных языков и языковых инструментариев // Математика программных систем: межвузовский сборник научных статей / под редакцией А. И. Микова и Л. Н. Лядовой. Пермь: Пермский государственный национальный исследовательский университет, 2012. С. 74–83. EDN TOFMPJ.

11. Fowler M. Domain Specific Language. URL: http://gnetna.com/books/Domain-Specific%20Languages.pdf (дата обращения: 25.05.2023).

12. Тихонов А. Ю., Аветисян А. И. Комбинированный (статический и динамический) анализ бинарного кода // Труды Института системного программирования РАН. 2012. Т. 22. С. 131–152.

13. Davide Di Gennaro. Advanced Metaprogramming in Classic C++. Berkeley, CA: Apress, 2015. 572 p.

14. McCarthy John. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I // Communications of the ACM. 1960. Vol. 3, N 4. P. 184–195.

15. Domkin V. Programming Algorithms in Lisp. Berkeley, CA: Apress, 2021. 392 p.

16. Сосновский Е. И. Использование языка программирования LISP для расширения базовых возможностей бюджетных САПР // Новые компьютерные технологии. 2012. Т. 10, № 1(10). С. 55–59. EDN WHKGLD.

17. Graham P. ANSI Common LISP. Philadelphia, USA: Pearson, 1995. 448 p.

18. Рапортиренко А. М. Common Lisp версия системы компьютерной алгебры REDUCE // Вестник Российского университета дружбы народов. Серия: Математика, информатика, физика. 2010. № 2-2. С. 40–44. EDN MQIMWJ.

19. Сайбель П. Практическое использование Common Lisp. М.: ДМК Пресс, 2023. 488 с.

20. Трещев И., Кожин И. Эмуляторы и симуляторы сетей ЭВМ. Для студентов технических специальностей. М.: Ридеро, 2018. 166 с.

References

1. Ingebrigtsen E. Metaprogramming in C#. Birmingham, UK, Packt Publishing Limited, 2023. 353 p.

2. Lotoreychik V. Y. Metaprogrammirovanie na osnove tekstovogo preprocessora [Metaprogramming based on word processor]. *Nauchno-tekhnicheskij vestnik Sankt-Peterburgskogo gosudarstvennogo universiteta informacionnyh tekhnologij, mekhaniki i optiki = Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2006, no. 25, pp. 57–65. EDN JURWLF

3. What APIs Are and Why They Matter. Available at: https://www.f5.com/ labs/articles/education/still-mystified-by-apis-what-they-are-really-and-why-they-matter. (accessed 26.05.2022)

4. Anikin D. A. Analiz metodov avtorizacii i autentifikacii REST API [Analysis of REST API authorization and authentication methods REST API]. *Mezhdunarodnyj Zhurnal Informacionnyh Tekhnologij i Energoeffektivnosti = International Journal of Information Technologies and Energy*, 2023, vol. 8, no. 5-2(31), pp. 120–124. EDN DZKSMQ

5. Denisov D. Sravnenie Typescript i Javascript dlya web-razrabotki [Comparison of typescript and javascript for web development]. *Studencheskij vestnik = Student Bulletin*, 2023, no. 20-8(259), pp. 57–58. EDN MQCOML

6. Nazarova O. V. Interaktivnyj uchebnik po Java Script [Interactive JavaScript toolbook]. *Hroniki ob"edinennogo fonda elektronnyh resursov. Nauka i obrazovanie = United Fond of Electronic Resources Chronicles: Science and Aducation*, 2017, no. 7(98), p. 63. EDN ZRRDGB

7. Skott Adam D., Makdonald M., Pauers Shelli. JavaScript. Recepty dlya razrabotchikov [JavaScript Cookbook: Programming the Web]. 3rd ed. St. Petersburg, Peter Publ., 2023. 528 p.

8. Abelson H., Sussman D. Struktura i interpretaciya komp'yuternyh programm [Structure and Interpretation of Computer Programs]. Moscow, Dobrosvet, KDU Publ., 2012. 608 p.

9. Evans E. Predmetno-orientirovannoe proektirovanie (DDD). Strukturizaciya slozhnyh programmnyh system [Subject-oriented design (Domain Driven Development). Structuring of complex software systems]. Moscow, Villiams Publ., 2020. 448 p.

10. Suhov A. O. [Classifiaction of domain specific languages and meanings]. *Matematika programmnyh system. Mezhvuzovskij sbornik nauchnyh statej* [Mathematics of software systems. Interuniversity collection of scientific articles]; ed. by A. I. Mikov and L. N. Lyadova. Perm, Perm State National Research University Publ., 2012, pp. 74–83. (In Russ.). EDN TOFMPJ

11. Fowler M. Domain Specific Language Available at: http://gnetna.com/books/Domain-Specific%20Languages.pdf. (accessed 25.05.2023)

12. Tihonov A. Y. Kombinirovannyj (staticheskij i dinamicheskij) analiz binarnogo koda [Combined static and dynamic binary code analisys]. *Trudy Instituta sistemnogo programmirovaniya RAN = Issues of ISP RAN*, 2012, vol. 22, pp. 131–152.

13. Davide Di Gennaro. Advanced Metaprogramming in Classic C++. Berkeley, CA, Apress, 2015. 572 p.

14. McCarthy John. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 1960, vol. 3, no. 4, pp. 184–195.

15. Domkin V. Programming Algorithms in Lisp. Berkeley, CA, Apress, 2021. 392 p.

16. Sosnovsky E. I. Ispol'zovanie yazyka programmirovaniya LISP dlya rasshireniya bazovyh vozmozhnostej byudzhetnyh SAPR [Using Lisp language for adjusting CAD base functions]. *Novye komp'yuternye tekhnologii = New Computer Technologies*, 2012, vol. 10, no. 1(10), pp. 55–59. EDN WHKGLD

17. Graham P. ANSI Common LISP. Philadelphia, USA, Pearson Publ., 1995. 448 p.

18. Raportirenko A. M. Common Lisp versiya sistemy komp'yuternoj algebry REDUCE [Common Lisp version of REDUCE computer algebra system]. *Vestnik Rossijskogo universiteta druzhby narodov. Seriya: Matematika, informatika, fizika = Bulletin of Russian Peoples' Friendship University. Series: Mathematics, Information Sciences and Physics*, 2010, no. 2-2, pp. 40–44. EDN MQIMWJ

19. Sibel P. Prakticheskoe ispol'zovanie Common Lisp [Practical Common Lisp]. Moscow, DMK Press Publ., 2023. 488 p.

20. Treshchev I., Kozhin I. Emulyatory i simulyatory setej EVM. Dlya studentov tekhnicheskih special'nostej [Emulators and simulators of computer networks. For technical students]. Moscow, Ridero Publ., 2018. 166 p.

Информация об авторе / Information about the Author

Чаплыгин Александр Александрович,
кандидат технических наук, доцент кафедры программной инженерии, Юго-Западный государственный университет,
г. Курск, Российская Федерация,
e-mail: alex_chaplygin@mail.ru,
ORCID: 0009-0009-8739-2695

Aleksandr A. Chaplygin, Cand. of Sci. (Engineering), Associate Professor of the Department of Software Engeneering, Southwest State University,
Kursk, Russian Federation,
e-mail: alex_chaplygin@mail.ru,
ORCID: 0009-0009-8739-2695