**Оригинальная статья / Original article**

https://doi.org/10.21869/2223-1536-2024-14-2-181-193



УДК 004.451

# Моделирование интерпретатора функционального языка программирования с возможностями метапрограммирования

**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 исследований*** *заключается в моделировании интерпретатора функционального языка программирования с возможностями метапрограммирования и анализе способов реализации примитивных операторов на основе макросов.*

***Методы.*** *Была разработана формальная модель интерпретатора функционального языка, являющегося подмножеством Common Lisp, с денотационной семантикой, которая позволяет точно описать поведение интерпретатора при вычислении элементов языка, таких как цитирование, обращение к переменным, последовательность действий, ветвление, присваивание, абстракция, аппликация.*

***Результаты.*** *На основе денотационной семантики был разработана архитектура интерпретатора функционального языка с возможностями метапрограммирования. В качестве базовых типов объектов были выбраны числа, символы, пары, строки и массивы. Для хранения объектов была использована теговая архитектура, где младшие биты адреса объекта всегда равны нулю, поэтому в них можно хранить код типа объекта и бит для пометки. Выделение и освобождение объектов происходит автоматически: для сборки мусора использован алгоритм пометки и очистки. С помощью макросов были реализованы операторы ветвления – полные и неполные, оператор выбора, операторы блока связанных переменных.*

***Заключение.*** *В результате работы был реализован интерпретатор функционального языка с возможностями метапрограммирования. С помощью макросов были реализованы примитивные операторы условия, выбора, блока связанных переменных. На примере этих операторов показано, что, используя метапрограммирование, можно встраивать в интерпретатор только базовые формы и примитивы, а остальные операторы могут быть реализованы с помощью метапрограммирования, что позволяет упростить и сократить объем программного кода интерпретатора.*

***Ключевые слова:*** *интерпретатор; Common Lisp; денотационная семантика; метапрограммирование; макрос.*

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

**Для цитирования**: Чаплыгин А. А. Моделирование интерпретатора функционального языка программирования с возможностями метапрограммирования // Известия Юго-Западного государственного университета. Серия: Управление, вычислительная техника, информатика. Медицинское приборостроение. 2024. Т. 14, № 2. С.181–193. https://doi.org/10.21869/ 2223-1536-2024-14-2-181–193

*Поступила в редакцию 18.04.2024* *Подписана в печать 16.05.2024* *Опубликована 28.06.2024*

# Modeling of functional language interpreter with metaprogramming

**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 the research*** *consists of modeling an interpreter for a functional programming language with metaprogramming capabilities and analyzing ways to implement primitive operators based on macros.*

***Methods.*** *A formal model of a functional language interpreter, which is a subset of Common Lisp, was developed with denotational semantics, which allows you to accurately describe the behavior of the interpreter when calculating language elements such as quoting, accessing variables, sequence of actions, branching, assignment, abstraction, application.*

***Results.*** *Based on denotational semantics, the architecture of a functional language interpreter with metaprogramming capabilities was developed. Numbers, symbols, pairs, strings and arrays were chosen as the basic types of objects. To store objects, a tag architecture was used, where the low-order bits of the object address are always zero, so they can store the object type code and the tag bit. Objects are allocated and freed automatically: a mark and cleanup algorithm is used for garbage collection. Using macros, branching operators, complete and incomplete, selection operator, and block operators of related variables were implemented.*

***Conclusion.*** *As a result of the work, a functional language interpreter with metaprogramming capabilities was implemented. Using macros, primitive operators of condition, selection, and a block of related variables were implemented. Using these operators as an example, it is shown that using metaprogramming, only basic forms and primitives can be built into the interpreter, and the other operators can be implemented using metaprogramming, which makes it possible to simplify and reduce the amount of interpreter code.*

***Keywords:*** interpreter; Common Lisp; denotational semantics; metaprogramming; macro.

***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. Modeling of functional language interpreter with metaprogramming. *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.* 2024;14(2):181–193. (In Russ.) https://doi.org/10.21869/2223-1536-2024-14-2-181–193

*Received 18.04.2024* *Accepted 16.05.2024* *Published 28.06.2024*

## Введение

Функциональные языки программирования представляют процесс вычислений как вычисление функции в математике [1; 2; 3; 4]. Входные данные – это аргументы функции, выходные данные – значение функции. Главная функция может зависеть от других функций. В отличие от императивного программирования в функциональном программировании предполагается отсутствие состояния, которое хранится в явном виде (например, в памяти). Также не предполагается изменение этого состояния. Достоинством такого подхода является

отсутствие скрытых побочных эффектов [3; 5; 6], т. е. скрытых зависимостей одного вычисления от другого, в результате чего получаются так называемые чистые функции. При одних и тех же входных значениях чистая функция без побочных эффектов будет всегда возвращать одно и то же значение. Это значение можно кешировать для последующего использования. Также можно менять порядок вызова функций или выполнять их параллельно [7; 6].

Лисп является одним из первых языков программирования вообще и первым функциональным языком программирования [8; 9; 10]. Хотя на нем можно было программировать в функциональном стиле, для практической реализации систем программирования были включены возможности менять состояние программы (глобальное окружение), а также возможности управления потоком вычислений [11].

Реализовать систему программирования можно с помощью интерпретатора или компилятора. Оба способа имеют свои достоинства и недостатки [12; 13; 14]. Интерпретатор языка по сложности гораздо менее сложный, чем компилятор. Например, метациклический интерпретатор Лиспа занимает одну страницу текста [15]. С помощью интерпретатора появляется возможность реализовать интерактивный цикл разработки REPL (Read, Eval, Print Loop), в результате чего можно интерактивно тестировать отдельные функции программы, смотреть состояния переменных. Недостатком интерпретатора является медленная скорость работы по сравнению с компилятором, хотя существуют методы повышения скорости интерпретаторов, такие как предварительная трансляция в байт-код.

Компиляторы за счет многочисленных оптимизаций могут обеспечить высокую скорость работы программы [13; 16; 17], но это требует больших усилий при разработке и тестировании [18]. Также для систем компиляторов невозможна реализация интерактивной консоли разработки, потому что для работы программы требуется полная компиляция всех частей программы, никакая функция программы не может работать по отдельности.

Как было сказано в [19], метапрограммирование способно очень сильно упростить разработку крупных систем. Для поддержки метапрограммирования в систему интерпретатора необходимо добавить возможность создания макрофункций и возможность макроподстановки. Квазицитирование обеспечивает контроль над абстрактными синтаксическими деревьями макрофункций.

## Материалы и методы

В качестве исходного языка возьмем язык, являющийся подмножеством языка Common Lisp. Исходный язык представляется следующей системой грамматик:

$$\begin{gathered} < O >::=< A > | < Q > \\ < A >::=< \text{symbol} > | < \text{number} > \\ < Q >::= ' < S > | < S > \\ < S >::=< A > |(< S > *) \end{gathered} \quad (1)$$

где < O > – вычисляемый объект; < A > – атом, который может быть или символом или числом; < Q > – выражение с цитированием или без него; < S > – списочное выражение или атом.

Для формализации интерпретатора функционального языка воспользуемся денотационной семантикой [20]. Определим математический эквивалент всех конструкций языка. Исходная программа преобразуется в функцию, которая будет являться денотацией программы. Для функциональных языков в качестве базового множества функций традиционно используется λ-исчисление.

Основу интерпретатора составляет функция $\mathscr{E}$. Она имеет следующий тип:

$$\mathscr{E} : \pi \times \rho \times \kappa \rightarrow \varepsilon, \qquad (2)$$

где π – исходная программа; ρ – окружение; κ – продолжение; ε – значение. Текущее продолжение получает результат вычисления, начальное продолжение имеет вид

$$\kappa_0 = \lambda\nu.\nu. \qquad (3)$$

Окружение представим как функцию, которая отображает множество переменных ν на множество значений ε:

$$\rho : \nu \rightarrow \varepsilon. \qquad (4)$$

Значения ε включают в себя множество натуральных чисел, множество пар (ε.ε) и множество функций ϕ.

Основные элементы функционального языка включают в себя:

– цитирование;

– обращение к переменным;

– последовательность действий;

– ветвление;

– присваивание;

– абстракцию;

– аппликацию.

Цитирование – это повторение значения выражения без изменений. Для чисел естественно автоцитирование (неизменность значения). Выражения в функции $\mathscr{E}$ будем заключать в семантические скобки [ и ]. Денотация цитирования показана в формуле (5):

$$\mathscr{E}[\text{quote } \varepsilon] = \lambda\rho\kappa.\varepsilon. \qquad (5)$$

Чтобы получить значение переменной, нужно вычислить функцию окружения [формула (6)]. Функция окружения должна учитывать возможность ошибки – отсутствие переменной в окружении:

$$\mathscr{E}[\nu] = \lambda\rho\kappa.(\rho\nu). \qquad (6)$$

Для денотации последовательности вычисление введем обозначение π+, что означает последовательность из одного или более элементов программы π. Обозначение π∗ означает последовательность из нуля или более элементов программы π. В последовательности все элементы должны вычисляться слева направо. Результатом вычисления последовательности является вычисленное значение последнего элемента последовательности. Введем вспомогательную функцию $\mathscr{E}$+. Тогда денотация последовательности вычислений будет иметь следующий вид:

$$\mathscr{E}[\text{progn } \pi+]\rho\kappa = (\mathscr{E}+[\pi+]\rho\kappa),$$
$$\mathscr{E}+[\pi]\rho\kappa = \mathscr{E}[\pi]\rho\kappa,$$
$$\mathscr{E}+[\pi\pi+]\rho\kappa = \mathscr{E}[\pi]\rho\lambda\varepsilon.(\mathscr{E}+[\pi+]\rho\kappa). \qquad (7)$$

Для денотации ветвления необходимо представить булевы значения в виде лямбда-функций:

$$T = \lambda xy.x,$$
$$F = \lambda xy.y. \quad (8)$$

Тогда ветвление можно записать таким образом:

$$\mathscr{E}[\text{if } \pi\ \pi_1\pi_2]\rho\kappa =$$
$$= \mathscr{E}[\pi]\rho\lambda\varepsilon.(\text{boolify } \varepsilon)\mathscr{E}[\pi_1]\rho\kappa\mathscr{E}[\pi_2]\rho\kappa. \quad (9)$$

В формуле (9) функция boolify должна приводить значение к булевому типу. Сначала вычисляется условие, затем результат передается в продолжение, где используется как функция с двумя аргументами, которые представляют собой вычисления, выполняемые по истине и по лжи.

Для денотации присваивания введем обозначение $\rho[\nu \to \varepsilon]$, которое означает расширение окружения $\rho$, куда добавляются новая переменная $\nu$ со значением $\varepsilon$. Тогда денотация присваивания будет иметь вид

$$\mathscr{E}[\text{setq } \nu\ \pi]\rho\kappa = \mathscr{E}[\pi]\rho\lambda\varepsilon.\kappa\varepsilon\rho[\nu \to \varepsilon]. \quad (10)$$

Абстракция представляет собой передачу в продолжение функции как объекта. При применении функции должно произойти расширение окружения аргументами функции $\varepsilon*$ и подстановка аргументов $\varepsilon*$ вместо параметров $\nu*$:

$$\mathscr{E}[\text{lambda } (\nu*)\ \pi+]\rho\kappa =$$
$$= \kappa\lambda\varepsilon*\kappa_1.\mathscr{E}+[\pi+]\rho[\nu* \to \varepsilon*]\kappa_1. \quad (11)$$

Аппликация – это применение функции к ее аргументам. В начале должны быть вычислены выражения аргументов, затем вычисляется выражение функции (абстракция), и вычисленные аргументы подставляются в абстрактную функцию. При вычислении аргументов $\mathscr{E}*$ обозначим пустую последовательность как $\langle\rangle$, а конкатенацию последовательностей как §. Тогда денотация аппликации будет иметь следующий вид:

$$\mathscr{E}[\pi\pi*]\rho\kappa = \mathscr{E}[\pi]\rho\lambda\phi.\mathscr{E}*[\pi*]\rho\lambda\varepsilon*.\phi\varepsilon*\kappa,$$
$$\mathscr{E}*[]\rho\kappa = \kappa\langle\rangle,$$
$$\mathscr{E}*[\pi\pi*]\rho\kappa =$$
$$= \mathscr{E}[\pi]\rho\lambda\varepsilon.\mathscr{E}*[\pi*]\rho\lambda\varepsilon*.\kappa\langle\varepsilon\rangle\S\varepsilon*. \quad (12)$$

Макросы вычисляются как вычисление результата подстановки параметров. Это можно определить как два вычисления тела макроса:

$$\mathscr{E}[\text{macro } \pi\ \varepsilon]\rho\kappa = \mathscr{E}[\mathscr{E}[\pi\ \varepsilon]\rho\kappa]\rho\kappa. \quad (13)$$

Чтобы выполнить макроподстановку, введем дополнительные операции:

‘ – квазицитирование (частичное цитирование);

, – вычисление внутри квазицитирования;

, @ – вычисление списка внутри квазицитирования.

Квазицитирование выполняется как цитирование для атомов и рекурсивно для пар.

$$\mathscr{E}[‘\pi]\rho\kappa = \mathscr{E}‘[\pi]\rho\kappa,$$
$$\mathscr{E}‘[\varepsilon]\rho\kappa = \kappa\varepsilon,$$
$$\mathscr{E}‘[]\rho\kappa = \kappa\langle\rangle,$$
$$\mathscr{E}‘[(\pi\pi*)]\rho\kappa = \kappa(\mathscr{E}‘[\pi]\rho\kappa.\mathscr{E}‘[\pi*]\rho\kappa). \quad (14)$$

Вычисление выражения внутри квазицитирования имеет вид

$$\mathscr{E}‘[,\pi]\rho\kappa = \mathscr{E}[\pi]\rho\kappa. \quad (15)$$

При вычислении списка внутри квазицитирования полученный список конкатенируется с остальными частями списка:

$$\mathscr{E}`[\pi 1*,@\pi\ \pi 2*]\rho\kappa = \mathscr{E}`[\pi 1*]\rho\kappa\S\mathscr{E}[\pi]\S\mathscr{E}`[\pi 2*] \quad (16)$$

## Результаты и их обсуждение

Архитектура интерпретатора показана ниже (рис. 1).

Главный цикл интерпретатора (модуль main.c) представляет собой традиционный цикл чтения данных, вычисления и печати результата (REPL). Когда данные заканчиваются (конец потока), то цикл прерывается. Обработка ошибок основана на сохранении состояния в начале цикла (функция setjmp). При возникновении ошибочной ситуации печатается соответствующее сообщение и происходит безусловный переход на начало цикла (функция longjmp).

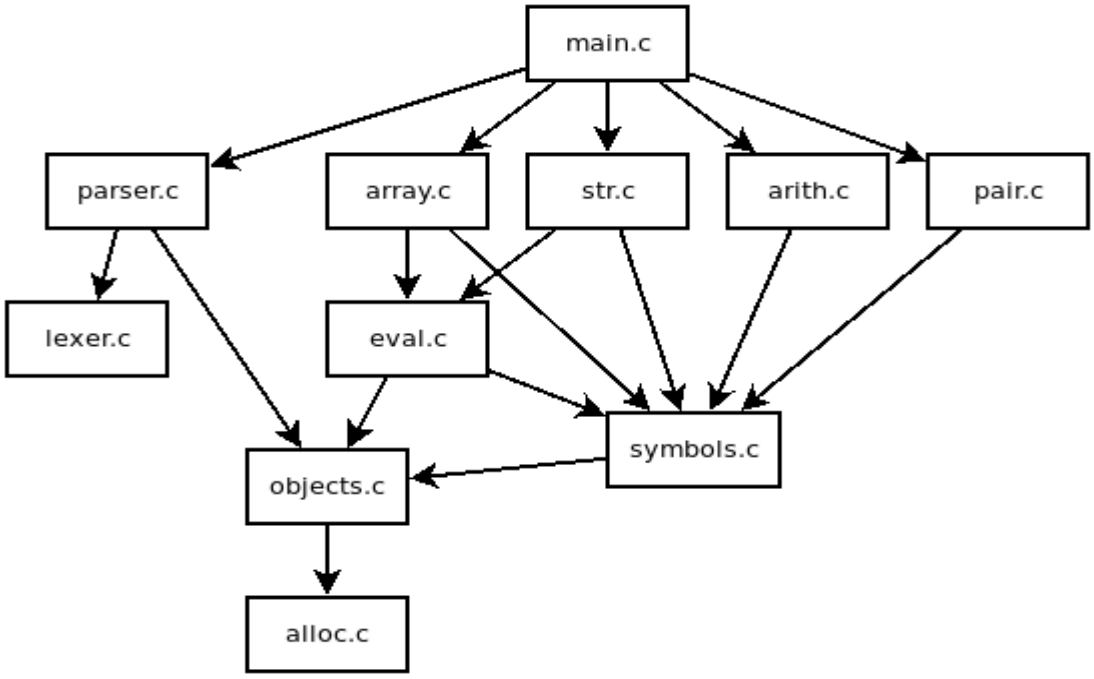


**Рис. 1.** Архитектура интерпретатора

**Fig. 1.** Interpreter architechture

Чтение входного потока данных включает в себя этапы лексического и синтаксического анализа. На этапе лексического анализа (модуль lexer.c) входной поток символов распознается как лексемы: целые, вещественные числа, символьные и обычные строки, скобки, служебные символы. Далее синтаксический анализатор (модуль parser.c) проверяет синтаксис языка на корректность и строит объект, содержащий введенное выражение.

Объектами языка являются числа, символы, пары, строки и массивы. Модуль objects.c отвечает за создание и освобождение объектов. Объекты создаются автоматически (без принудительного указания программиста) и

освобождаются по мере необходимости (сборка мусора). В данном интерпретаторе для сборки мусора использован алгоритм пометки и очистки.

Модуль alloc.c отвечает за низкоуровневое выделение и освобождение памяти для объектов. Все объекты выравниваются по границе памяти в 16 байт. Тип объекта представляет собой указатель, у которого младшие четыре бита всегда равны нулю. Три младших бита используются как тег (тип объекта), четвертый бит используется для пометки при сборке мусора. Числа, которые помещаются в оставшиеся биты, кроме этих четырех, являются маленькими числами, для которых не нужны дополнительные структуры данных и не нужна сборка мусора. Для больших чисел используется отдельный тип объекта и дополнительные структуры данных. Символы хранятся в хеш-таблице для быстрого поиска (модуль symbols.c).

Главная часть интерпретатора – это вычислитель (модуль eval.c). Он выполняет вычисление исходных выражений-объектов ($\pi$ в денотации), в результате которого получается результат – тоже объект. Вычислитель использует окружение $\rho$ для лексических переменных. Динамические переменные хранят свои значения в структуре символа (поле value). Это образует отдельную область видимости для переменных.

Абстракции функций $\lambda$ также хранятся в структуре символа (поле lambda). Встроенные функции делятся специальные формы и обычные функции. Для обычных функций всегда перед вызовом функции вычисляются аргументы (слева направо), а у специальных форм аргументы передаются в форму без вычислений. Имена функций образуют область видимости отдельную от переменных (т. е. возможно существование переменной и функции с одним именем). Встроенные примитивы используют поле func структуры символа для адреса внутренней функции, которая вызывается при применении формы примитива. Сами примитивы распределены между различными модулями. Модуль arith.c содержит примитивы для арифметических и побитовых операций. Модуль pair.c содержит функции для работы с точечными парами. Модуль str.c содержит функции для работы со строками. Модуль array.c содержит функции для работы с массивами.

Макросы используют отдельное поле macro в структуре символа. Область имен макросов совпадает с областью имен функций. Макровызов происходит в два этапа. Сначала происходит макроподстановка, а затем ее результат вычисляется тем же вычислителем. Если результат вычисления также содержит макросы, то происходит рекурсия.

С помощью полученного интерпретатора с возможностями метапрограммирования возможно реализовать новые синтаксические конструкции, которые будут работать так же, как и встроенные формы. Возьмем встроенную специальную форму cond. Ее параметрами является список из пар (pnen), где pn –

предикат, а en – выражение. Значением специальной формы cond является первое выражение en, для которого предикат pn равен истине. На основе этой специальной формы можно сделать традиционный для программирования условный оператор if (рис. 2).

```
(defmacro if (test true false)
  "Условный ␣ оператор"
  "test ␣ – ␣ условие"
  "true ␣ – ␣ выражение ␣ по ␣ истине"
  "false ␣ – ␣ выражение ␣ по ␣ лжи"
  ‘(cond (,test ,true)
    (t ,false)))
```

**Рис. 2.** Условный оператор

**Fig. 2.** Conditional operator

Недостатком данного оператора является обязательное указание выражения по истине и по лжи, а также, если необходимо использовать последовательность операторов, тогда приходится использовать форму progn. Эти недостатки можно устранить, если определить неполный условный оператор, тело которого может состоять из нескольких выражений, причем мы можем использовать уже определенный оператор if (рис. 3).

```
(defmacro when (test &rest body)
  "Условный ␣ неполный ␣ оператор"
  "test ␣ – ␣ условие"
  "body ␣ – ␣ список ␣ выражений ␣ по ␣
истине"
  ‘(if ,test (progn ,@body) nil))
```

**Рис. 3.** Неполный условный оператор

**Fig. 3.** Incomplete conditional operator

По аналогии определим условный оператор, который выполняет множество выражений, если условие не выполняется (рис. 4).

```
(defmacro unless (test &rest body)
  "Условный ␣ неполный ␣ оператор"
  "test ␣ – ␣ условие"
  "body ␣ – ␣ список ␣ выражений ␣ по ␣
лжи"
  ‘(if ,test nil (progn ,@body)))
```

**Рис. 4.** Неполный условный оператор с невыполнением условия

**Fig. 4.** Incomplete conditional operator on false condition

Среди операторов ветвления существует также оператор выбора, который по результату вычисления выражения выбирает значение из пар, а также имеет значение по умолчанию (рис. 5).

```
(case k
  ((1 2)
   (2 3)
   (otherwise 4)))
```

**Рис. 5.** Пример оператора выбора

**Fig. 5.** Selection operator example

Это соответствует следующему выражению (рис.6).

```
(if (equal k 1) 2
    (if (equal k 2) 3 4))
```

**Рис. 6.** Оператор выбора выраженный через if

**Fig. 6.** Selection operator expressed with if

Для реализации оператора выбора удобно воспользоваться формой cond, которая похожа на оператор выбора, а

также сделать вспомогательную функцию, которая создает пару из предиката и выражения. Для создания тела cond используется функция отображения map (рис. 7).

```
(defmacro case (val &rest list)
  `(cond ,@(map 'list (lambda (x) (case-func
x val)) list)))

(defun case-func (p val)
  "(e v) -> ((equal val e) v)"
  "(otherwise v) -> (t v)"
  (if (eq (car p) 'otherwise)
      (list t (cadr p))
      (list (list 'equal (car p) val) (cadr p))))
```

**Рис. 7.** Макрос оператора выбора

**Fig. 7.** Selection operator macro

После макроподстановки выражение на рисунке 5 будет преобразовано в следующее выражение (рис. 8).

```
(cond ((equal k 1) 2)
      ((equal k 2) 3)
      (t 4))
```

**Рис. 8.** Результат макроподстановки оператора case

**Fig. 8.** Case operator macro expansion

Оператор let создает блок выражений с лексическими переменными, связывая эти переменными со значениями. Например выражение (рис. 9) будет иметь значение 30.

```
(let ((x 10)
      (y 20))
  (+ x y))
```

**Рис. 9.** Оператор let

**Fig. 9.** Let operator

Для реализации этого оператора можно воспользоваться абстрактной λ-функцией с параметрами, которые соответствуют переменным. Эта функция применяется к значениям, с которыми нужно связать переменные. Тогда выражение на рисунке 9 можно записать как применение lambda-функции (рис. 10).

```
((lambda (x y)
   (+ x y)) 10 20)
```

**Рис. 10.** Lambda-выражение, соответствующее оператору let

**Fig. 10.** Lambda-expression for let operator

Как и в случае с оператором выбора, для реализации оператора let определим две вспомогательные функции, которые на основе списка привязок переменных получают списки параметров и значений (рис. 11).

```
(defmacro let (vars &rest body)
  "Блок локальных переменных"
  `((lambda ,(get-vars vars) ,@body)
    ,@(get-vals vars)))

  (defun get-vars (v)
  "Получение списка переменных
для let"
  (if (null v) nil
      (cons (caar v) (get-vars (cdr v)))))

  (defun get-vals (v)
  "Получение списка значений для
let"
  (if (null v) nil
      (cons (cadar v) (get-vals (cdr v)))))
```

**Рис. 11.** Реализация оператора let

**Fig. 11.** Let operator macro

Подобный подход к связыванию лексических переменных позволяет определять только независимые

переменные. Если существует необходимость при связывании новой переменной использовать значение предыдущей, то необходимо определить другой оператор let* (рис. 12).

```
(let* ((x 0)
       (y (+ x 1))
       (z (* y 5)))
   (+ x y z))
```

**Рис. 12.** Оператор let*

**Fig. 12.** Let* operator

Чтобы последующая переменная могла обращаться к предыдущей, необходимо, чтобы предыдущие переменные были свободными. Это можно выразить как группу вложенных абстракций и применений (рис. 13).

```
((lambda (x)
  ((lambda (y)
    ((lambda (z)
      (+ x y z))
     (* y 5)))
    (+ x 1)))
  0)
```

**Рис. 13.** Требуемый результат макроподстановки для оператора let*

**Fig. 13.** Needed macroexpansion for let* operator

При макроподстановке мы можем получить список параметров и аргументов, используя те же вспомогательные функции для оператора let. Но также необходима дополнительный рекурсивный макрос, который бы формировал каскад из абстракций с применением, параллельно обходя два списка параметров и аргументов. В результате макрос для оператора имеет следующий вид (рис. 14).

```
(defmacro inner−let* (vars vals &rest
body)
  `((lambda (,(car vars)) (if ,(null (cdr
vars))
              (progn ,@body)
              (inner−let* ,(cdr vars) ,(cdr
vals) ,@body)))
            ,(car vals)))

(defmacro let* (vars &rest body)
  `(inner−let* ,(get−vars vars) ,(get−vals
vars) ,@body))
```

**Рис. 14.** Макроопределение для оператора let*

**Fig. 14.** Let* operator macrodefinition

## Выводы

В результате работы была построена модель с денотационной семантикой для интерпретатора функционального языка с возможностями метапрограммирования. Был реализован интерпретатор с поддержкой метапрограммирования. Созданы макросы для операторов ветвления, выбора, блока связанных переменных. Таким образом, интерпретатор с поддержкой метапрограммирования может содержать небольшое число встроенных форм, а любые другие формы могут быть реализованы как макросы, при необходимости совместно с функциями. Это позволяет существенно сократить объем программного кода, необходимого для написания интерпретатора.

## Список литературы

1. Сайбель П. Практическое использование Common Lisp. М.: ДМК Пресс, 2017. 488 с.

2. Душкин Р. В. Функциональное программирование на языке Haskell. М.: ДМК Пресс, 2016. 608 с.

3. Узких Г. Ю. Функциональное программирование и его влияние на качество кода и обслуживаемость // Вестник науки. 2023. Т. 4, № 8(65). С. 316–318. EDN TQONQF.

4. Телегин В. А. Влияние функционального программирования на современные языки программирования // Инновации и инвестиции. 2023. № 7. С. 189–192. EDN VQEXEK

5. Городняя Л. В. Перспективы функционального программирования параллельных вычислений // Электронные библиотеки. 2021. Т. 24, № 6. С. 1090–1116. https://doi.org/10.26907/1562-5419-2021-24-6-1090-1116. EDN LOGTRU

6. Билуха И. Н. Актуальность функционального программирования // Студенческий вестник. 2020. № 5-4 (103). С. 60–61. EDN VTDJGN

7. Krasnov M. M., Feodoritova O. B. Using the functional programming library for solving numerical problems on graphics accelerators with CUDA technology // Proceedings of the Institute for System Programming of the RAS. 2021. Vol. 33, No. 5. P. 167–180. https://doi.org/10.15514/ISPRAS-2021-33(5)-10. EDN WXDIFI

8. Domkin V. Programming Algorithms in Lisp. Berkeley: Apress, 2021. 377 p.

9. Грэй П. ANSI Common LISP. М.: Символ-Плюс, 2020. 448 с.

10. Малов А. В., Родионов С. В. Реализация упрощенного алгоритма Байеса в среде функционального программирования COMMON LISP // Известия СПбГЭТУ «ЛЭТИ». 2015. № 2. С. 32–37. EDN TKJVLJ

11. Вторников А. Lisp: маленький гигант // Системный администратор. 2016. № 6 (163). С. 64–69. EDN VZGXVD

12. Nystrom R. Crafting interpreters. M.: Nobel Press, 2024. 640 p.

13. Ульман Джеффри Д., Сети Рави. Компиляторы: принципы, технологии и инструментарий. М.: Диалектика-Вильямс, 2018. 1184 с.

14. Халилов Э. Р. Разработка интерпретатора для языка программирования видеоигр // Информационно-компьютерные технологии в экономике, образовании и социальной сфере. 2020. № 1 (27). С. 79–89. EDN KNIRUT

15. Абельсон Х., Сассман Д. Структура и интерпретация компьютерных программ. М.: КДУ, 2022. 608 с.

16. Григорьев Е. А., Климов Н. С. Использование Ahead-Of-Time компиляции в платформе.NET как альтернатива Just-In-Time компиляции // E-Scio. 2019. № 11 (38). С. 364–371. EDN EOMDHH

17. Харин И. А., Раскатова М. В. Анализ алгоритмов составляющих частей компилятора и его оптимизации // Computational Nanotechnology. 2023. Т. 10, № 2. С. 26–35. https://doi.org/10.33693/2313-223X-2023-10-2-26-35. EDN BDGKMA

18. Штейнберг Б. Я., Штейнберг О. Б. Преобразования программ – фундаментальная основа создания оптимизирующих распараллеливающих компиляторов // Программные системы: теория и приложения. 2021. Т. 12, № 1 (48). С. 21–113. https://doi.org/10.25209/2079-3316-2021-12-1-21-113. EDN FZFEPX

19. 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.

20. Krishnamurthi Sh. Programming Languages: Application and Interpretation. Providence: Brown University, 2022. 376 p.

## References

1. Seibel P. Practical use of Common Lisp. Moscow: DMK Press; 2017. 488 p. (In Russ.)

2. Dushkin R.V. Functional programming in Haskell. Moscow: DMK Press; 2016. 608 p. (In Russ.)

3. Uzkikh G.Y. Functional programming and its impact on code quality and maintainability. *Vestnik nauki = Bulletin of Science*. 2023;4(8):316–318. (In Russ.) EDN TQONQF

4. Telegin V.A. The influence of functional programming on modern programming languages. *Innovatsii i investitsii = Innovations and Investments*. 2023;(7):189–192. (In Russ.) EDN VQEXEK

5. Gorodnyaya L.V. Prospects for functional programming of parallel computing. *Elektronnye biblioteki = Electronic Libraries*. 2021;24(6):1090–1116. (In Russ.) https://doi.org/10.26907/1562-5419-2021-24-6-1090-1116. EDN LOGTRU

6. Bilukha I.N. Relevance of functional programming. *Student Bulletin*. 2020;(5-4):60–61. (In Russ.) EDN VTDJGN

7. Krasnov M.M., Feodoritova O.B. Using the functional programming library for solving numerical problems on graphics accelerators with CUDA technology. *Proceedings of the Institute for System Programming of the RAS*. 2021;33(5):167–180. https://doi.org/10.15514/ISPRAS-2021-33(5)-10. EDN WXDIFI

8. Domkin V. Programming Algorithms in Lisp. Berkeley: Apress; 2021. 377 p.

9. Graham P. ANSI Common LISP. Moscow: Symvol-Plus; 2020. 448 p. (In Russ.)

10. Malov A.V., Rodionov S.V. Implementation of the simplified Bayes algorithm in the COMMON LISP functional programming environment. *Proceedings of Saint Petersburg Electrotechnical University = Proceedings of St. Petersburg Electrotechnical University "LETI"*. 2015;(2):32–37. (In Russ.) EDN TKJVLJ

11. Vtornikov A. Lisp: a little giant. *Sistemnyi administrator = System Administrator*. 2016;(6):64–69. (In Russ.) EDN VZGXVD

12. Nystrom R. Crafting interpreters. Moscow: Nobel Press; 2024. 640 p.

13. Ullman Jeffrey D., Ravi Networks. Compilers: principles, technologies and tools. Moscow: Dialectics-Williams; 2018. 1184 p. (In Russ.)

14. Khalilov E.R. Development of an interpreter for a video game programming language. *Informatsionno-komp'yuternye tekhnologii v ekonomike, obrazovanii i sotsial'noi sfere = Information and Computer Technologies in Economics, Education and Social Sphere*. 2020;(1):79–89. (In Russ.) EDN KNIRUT

15. Abelson H., Sussman D. Structure and interpretation of computer programs. Moscow: KDU; 2022. 608 p. (In Russ.)

16. Grigoriev E.A., Klimov N.S. Using Ahead-Of-Time compilation in the .NET platform as an alternative to Just-In-Time compilation. *E-Scio*. 2019;11:364–371. (In Russ.) EDN EOMDHH

17. Kharin I.A., Raskatova M.V. Analysis of algorithms for compiler components and its optimization. *Computational Nanotechnology*. 2023;10(2):26–35. (In Russ.) https://doi.org/10.33693/2313-223X-2023-10-2-26-35. EDN BDGKMA

18. Steinberg B.Y., Steinberg O.B. Program transformations are the fundamental basis for creating optimizing parallelizing compilers. *Programmnye sistemy: teoriya i prilozheniya = Software Systems: Theory and Applications*. 2021;12(1):21–113. (In Russ.) https://doi.org/10.25209/2079-3316-2021-12-1-21-113. EDN FZFEPX

19. Chaplygin A.A. Using metaprogramming tools of the Common Lisp language to develop emulator systems. *Izvestiya Yugo-Zapadnogo gosudarstvennogo universiteta. Seriya: Upravlenie, vy-chislitel'naya 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.)

20. Krishnamurthi Sh. Programming Languages: Application and Interpretation. Providence: Brown University; 2022. 376 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**, Candidate of Sciences (Engeneering), 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