Оригинальная статья / Original article

УДК 004.451

https://doi.org/10.21869/2223-1560-2025-29-3-99-112

Модель и реализация компилятора функционального языка Common LISP

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 создании модели компилятора функционального языка Common Lisp, реализации этой модели и тестировании модели компилятора с помощью целевой виртуальной машины, чтобы увеличить скорость выполнения программ.*

Методы. *С помощью денотационной семантики была построена формальная модель компилятора функционального языка Common Lisp. Компиляция происходит в несколько этапов. На первом этапе исходный язык преобразуется в промежуточный lambda-язык, в котором все макросы раскрываются, встроенные формы преобразуются в аналогичные выражения, а имена переменных заменяются на локальные, глобальные и глубокие ссылки. На втором этапе выражение на промежуточном языке преобразуется из древовидной структуры в линейный список из примитивных инструкций целевой виртуальной машины.*

Результаты. *Полученные в результате компиляции примитивные инструкции кодируются с помощью специального ассемблера в числовой код для выполнения на целевой виртуальной машине. Также в результате компиляции получаются список констант и размер памяти необходимый для работы скомпилированной программы. Целевая виртуальная машина состоит из отделов памяти для кодированной программы, констант, глобальных переменных, стека, списка кадров активации, регистров (аккумулятор, указатель стека, указатель команд, текущий кадр активации). Кадры активации представляют собой объекты-массивы, которые хранят указатель на предыдущий кадр, номер уровня глубины вызовов и локальные аргументы. Сборка мусора происходит с помощью метода пометки и очистки.*

Заключение. *В результате работы была построена и реализована модель компилятора функционального языка Common Lisp. По сравнению с интерпретатором скорость работы программы возросла в среднем в 20 раз. Дальнейшее увеличение скорости можно достичь с помощью различных оптимизаций компилятора на разных стадиях. Из простых оптимизаций можно отметить: оптимизацию арифметических выражений, устранение лишних команд, упрощение выражений.*

Ключевые слова: *компилятор; байт-код; виртуальная машина; функциональный язык; Common Lisp; денотационная семантика.*

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

Для цитирования: Чаплыгин А.А. Модель и реализация компилятора функционального языка Common LISP // Известия Юго-Западного государственного университета. 2025; 29(3): 99-112. https://doi.org/10.21869/-2223-1560-2025-29-3-99-112.

Поступила в редакцию 03.06.2025 *Подписана в печать 21.08.2025* *Опубликована 30.09.2025*

Modeling and implementation of Common LISP functional language compiler

Aleksandr A. Chaplygin [1] ✉

[2] Southwest State University
50 Let Oktyabrya str. 94, Kursk 305040, Russian Federation

✉ e-mail: alex_chaplygin@mail.ru

Abstract

Purpose of research *is to create a compiler model for the functional language Common Lisp, implement this model, and test the compiler model using a target virtual machine to increase the execution speed of programs.*
Methods. *A formal compiler model of the functional language Common Lisp was built using denotational semantics. Compilation takes place in several stages. At the first stage, the source language is transformed into an intermediate lambda language in which all macros are expanded, embedded forms are transformed into similar expressions, and variable names are replaced with local, global, and deep references. At the second stage, the expression in the intermediate language is transformed from a tree structure into a linear list of primitive instructions of the target virtual machine.*
Results. *The resulting primitive instructions are encoded using a special assembler into numeric code for execution on the target virtual machine. The compilation also results in a list of constants and the amount of memory required for the compiled program to run. The target virtual machine consists of memory sections for the encoded program, constants, global variables, stack, list of activation frames, registers (accumulator, stack pointer, instruction pointer, current activation frame). Activation frames are array objects that store a pointer to the previous frame, the call depth level number, and local arguments. Garbage collection takes place using the tagging and cleaning method.*
Conclusion. *As a result, a Common Lisp functional language compiler model was built and implemented. Compared to the interpreter, the speed of the program has increased by an average of 20 times. Further speed increases can be achieved by using various compiler optimizations at different stages. Of the simple optimizations, it can be noted: optimization of arithmetic expressions, elimination of unnecessary commands, simplification of expressions.*

Keywords: *compiler; bytecode; virtual machine; functional language; Common Lisp; denotational semantics.*

Conflict of interest. *The Author declare the absence of obvious and potential conflicts of interest related to the publication of this article.*

For citation: Chaplygin A. A. Modeling and implementation of Common LISP functional language compiler. *Izvestiya Yugo-Zapadnogo gosudarstvennogo universiteta = Proceedings of the Southwest State University*. 2025; 29(3): 99-112 (In Russ.). https:// doi.org/10.21869/2223-1560-2025-29-3-99-112.

Received 03.06.2025 *Accepted 21.08.2025* *Published 30.09.2025*

***

Введение

Язык Лисп – один из старейших языков программирования, которые используются сейчас. Разработанный в 50-х годах Джоном Маккарти, он активно развивался до настоящего времени. Существует много диалектов Лиспа, таких как Common Lisp, Scheme [1], Racket [2], Closure и другие [3].

Первый компилятор Лиспа был написан в 1962 году в MIT Тимом Хартом и Майком Леви [4]. Он был написан на самом Лиспе и мог компилировать себя с помощью интерпретатора [5]. Машинный код, генерируемый этим компилятором, работал в 40 раз быстрее, чем исходная программа в интерпретаторе. Функции могли быть как интерпретируемыми, так и компилируемыми, и свободно работать вместе. Такая модель называется инкрементальной компиляцией. Свободные (глобальные) переменные могли передаваться между скомпилированными функциями и интерпретатором. Также можно было скомпилировать специальные формы. Для компиляции использовались свойства EXPR или FEXPR у переменной (символа) функции. Свободные (глобальные) переменные нужно было объявлять перед компиляцией. Порядок компиляции функций не имел значения, таким образом можно было компилировать не все, а только необходимые функции, которые работали очень медленно. В первой фазе компиляции S-выражения функций переписывались, чтобы более эффективно использовать встроенные и пользовательские особые формы (FSUBR и FEXPR), рекурсивные функции переписывались через итерацию с помощью формы PROG. Во второй фазе на основе S-выражений генерировался ассемблерный код: определялся порядок вычислений, места расположения аргументов и переменных, переходы для условных выражений.

Язык Common Lisp стал ANSI стандартом в 1994. Он объединил в себе достижения многих диалектов Лиспа, таких как Mac Lisp, Zeta Lisp, Spice Lisp, S-1 Lisp и других. Это мультипарадигменный язык общего назначения, включает в себя процедурное [6], функциональное [7] и объектно-ориентированное программирование [8]. Как и многие другие Lisp системы он является динамическим языком, который поощряет эволюционную и инкрементальную разработку ПО с итеративной компиляцией в эффективные программы, причем инкрементальная разработка часто происходит интерактивно, не прерывая работу программы.

В современных Common Lisp системах компилятор генерирует машинный код [9] или байт-код [10], который может быть сохранен [11]. Отдельные функции могут быть скомпилированы в памяти [12]. Модуль программы из файла может быть скомпилирован в байт-код, программу на языке C, в машинный код или в любую комбинацию из вышеперечисленного. Стандарт Common Lisp спроектирован для инкрементальной компиляции [13], а также включает объявления для оптимизаций компиляции[1] такие

[1] Свидетельство о государственной регистрации программы для ЭВМ № 2024668438 Российская Федерация. Оптимизирующий компилятор программ на языках высокого уровня C, C++, Фортран: № 2024667303: заявл. 26.07.2024: опубл. 07.08.2024; заявитель Акционерное общество «МЦСТ». EDN SNOHFN.

как спецификация типов и подстановка функций [14].

Наиболее известные современные реализации Common Lisp: Allegro Common Lisp, LispWorks, CLISP, CMUCL, Steel Bank Common Lisp и другие [15].

В этой работе была разработана и реализована модель компилятора для подмножества языка Common Lisp [16], чтобы ускорить выполнение программ интерпретатором [17].

Материалы и методы

Компиляция исходной программы будет проходить в несколько этапов. Для простоты здесь не рассматривается этап лексического и синтаксического анализа, а также раскрытие макросов. Будем считать, что на вход компилятора подается S-выражение как объект.

На первом этапе компиляции исходная программа анализируется и преобразуется в промежуточную форму: расширенный lambda-язык, где все макросы раскрываются, а встроенные формы имеют свои аналоги. Главное преобразование касается переменных. Компилятор преобразует имена переменных в такой вид, чтобы можно было сразу обратиться к нужному месту в памяти, без поиска по имени.

Воспользуемся денотационной семантикой [5], чтобы построить модель компилятора. Основу вычисления первой фазы компиляции составляет функция $\mathscr{E}$, которая будет иметь следующий тип:

$$\mathscr{E}:\varepsilon\times\rho\times\gamma\times\phi\times\pi \rightarrow \varepsilon\times\rho\times\gamma\times\phi\times\pi, \quad (1)$$

где ε – *S*-выражение языка; ρ – вычисляемое на момент компиляции окружение (в начале пустое – []); γ – список глобальных переменных; ϕ – список глобальных функций; π – список примитивов.

Правила преобразования программы в первой фазе компиляции представлены формулами:

$$\mathscr{E}[[c]]\rho\gamma\phi\pi = [\text{CONST } c]\rho\gamma\phi\pi \quad (2)$$

$$\mathscr{E}[[\text{quote } \varepsilon]]\rho\gamma\phi\pi = [\text{CONST } \varepsilon]\rho\gamma\phi\pi \quad (3)$$

$$\mathscr{E}[[\nu]]\rho\gamma\phi\pi = [\text{LOCAL\_REF i}]\rho\gamma\phi\pi,$$
если $fv(\nu,\rho,\gamma) = (\text{local,i})$
$$[\text{GLOBAL\_REFi }]\rho\gamma\phi\pi,$$
если $fv(\nu,\rho,\gamma) = (\text{global,i})$
$$[\text{DEEP\_REF i j}]\rho\gamma\phi\pi,$$
если $fv(\nu,\rho,\gamma) = (\text{deep,i,j})$ (4)

$$\mathscr{E}[[\text{progn } \varepsilon+]]\rho\gamma\phi\pi = \mathscr{E}+[\varepsilon+]\rho\gamma\phi\pi$$
$$\mathscr{E}+[[\varepsilon]]\rho\gamma\phi\pi = \mathscr{E}[\varepsilon]\rho\gamma\phi\pi$$
$$\mathscr{E}+[[\varepsilon\ \varepsilon+]]\rho\gamma\phi\pi=[\text{SEQ } \mathscr{E}[\varepsilon]\ \mathscr{E}+[\varepsilon+]]\rho\gamma\phi\pi \quad (5)$$

$$\mathscr{E}[[\text{if } \varepsilon_c\ \varepsilon_t\varepsilon f]]\rho\gamma\phi\pi =$$
$$=[\text{ALTER } \mathscr{E}[\varepsilon_c]\ \mathscr{E}[\varepsilon_t]\ \mathscr{E}[\varepsilon_f]]\rho\gamma\phi\pi \quad (6)$$

$$\mathscr{E}[[\text{setq } \nu\ \varepsilon]]\rho\gamma\phi\pi = sq(\nu\varepsilon\rho\gamma\phi\pi)$$
$$sq(\nu\varepsilon\rho\gamma\phi\pi) = [\text{LOCAL\_SET i } \mathscr{E}[\varepsilon]],$$
если $fv(\nu,\rho,\gamma) = (\text{local,i})$
$$[\text{GLOBAL\_SET i } \mathscr{E}[\varepsilon]],$$
если $fv(\nu,\rho,\gamma) = (\text{global,i})$
$$[\text{DEEP\_SET i j } \mathscr{E}[\varepsilon]],$$
если $fv(\nu,\rho,\gamma) = (\text{deep,i,j})$ (7)

$$\mathscr{E}[[\text{defun } \nu\ \sigma\ \varepsilon]]\rho\gamma\phi\pi =[\text{LABEL}\nu$$
$$[\text{SEQ } \mathscr{E}[\varepsilon](\sigma.\rho)\gamma\phi\pi\ \text{RETURN}]]\rho\gamma(\nu.\phi)\pi \quad (8)$$

$$\mathscr{E}[[\lambda\ \sigma\ \varepsilon]]\rho\gamma\phi\pi = [\text{FIX\_CLOSURE}$$
$$\text{gen}()[\text{SEQ } \mathscr{E}[\varepsilon](\sigma.\rho)\gamma\phi\pi]]\rho\gamma\phi\pi \quad (9)$$

$$\mathscr{E}[[\text{tagbody } \varepsilon+]]\rho\gamma\phi\pi = \mathscr{E}*[\varepsilon+]\rho\gamma\phi\pi$$
$$\mathscr{E}*[[\nu]]\rho\gamma\phi\pi = [\text{LABEL } \nu]\rho\gamma\phi\pi$$
$$\mathscr{E}*[[\varepsilon]]\rho\gamma\phi\pi = \mathscr{E}[\varepsilon]\rho\gamma\phi\pi$$

$\mathscr{E}*[[\varepsilon\ \varepsilon+]]\rho\gamma\phi\pi$ =[SEQ $\mathscr{E}[\varepsilon]$ $\mathscr{E}*[\varepsilon+]]\rho\gamma\phi\pi$ (10)

$\mathscr{E}[[\text{go}\ \nu]]\rho\gamma\phi\pi$ = [GOTO ν]$\rho\gamma\phi\pi$ (11)

$\mathscr{E}[[(\lambda\ \sigma\ \varepsilon_1)\varepsilon+]]\rho\gamma\phi\pi$ =

=[FIX_LET #ν ν $\mathscr{E}[\varepsilon_1](\sigma.\rho)\gamma\phi\pi]\rho\gamma\phi\pi$

$\nu = \mathscr{E}^{\dagger}[\varepsilon+]$

$\mathscr{E}^{\dagger}[[\varepsilon]]\rho\gamma\phi\pi = \mathscr{E}[\varepsilon]\rho\gamma\phi\pi$

$\mathscr{E}^{\dagger}[[\varepsilon\varepsilon+]]\rho\gamma\phi\pi = [\mathscr{E}[\varepsilon]\mathscr{E}^{\dagger}[\varepsilon+]]\rho\gamma\phi\pi$ (12)

$\mathscr{E}[[\sigma\varepsilon+]]\rho\gamma\phi\pi$ = [FIX_CALL σ #ν ν]$\rho\gamma\phi\pi$

$\nu = \mathscr{E}^{\dagger}[\varepsilon+]$ (13)

$\mathscr{E}[[\text{prim}\ \varepsilon+]]\rho\gamma\phi\pi$ = [FIX_PRIM prim ν]$\rho\gamma\phi\pi$

$\nu = \mathscr{E}^{\dagger}[\varepsilon+]$, $\forall \text{prim} \in \pi$ (14)

Преобразование констант – тривиально (формула 2).

Цитирование аналогично преобразованию констант (формула 3).

В формуле (4) показано преобразование переменной. Для этого определяется тип переменной: локальная, глобальная или свободная. Функция поиска переменной имеет вид:

$fv(\nu,\rho,\gamma)$ = (local $\rho0 \uparrow \nu$), если $\nu \in \rho0$

$fv(\nu,\rho,\gamma)$ = (global $\gamma \uparrow \nu$), если $\nu \in \gamma$

$fv(\nu,\rho,\gamma)$=(deep($\rho n \uparrow \nu$)n), если $\nu \in \rho n$ (15)

Локальная переменная ν присутствует в первом кадре локального ок-ружения ρ, глобальные переменные на-ходятся в глобальном окружении γ, свободные переменные находятся в любом кадре ρ, кроме первого. Выражение $\rho_n \uparrow \nu$ обозначает позицию переменной ν в кадре ρ_n.

Для денотации компиляции последовательности вычислений используется вспомогательная функция $\mathscr{E}+$. Выражение $\varepsilon+$ означает последовательность из одного или более выражений ε.

При компиляции формы *if* 6 компилируются условие ε_c, выражение по истине ε_t, выражение по лжи ε_f.

При компиляции присваивания (формула 7) используется та же функция поиска переменной (формула 15), что и при обращении к переменной.

Компиляция пользовательской функции (формула 8) включает в себя компиляцию тела функции ε и расширение окружения функций ϕ. Выражение $(\nu.\phi)$ означает добавление в начало окружения ϕ имени функции ν.

В формуле (9) показана компиляция замыкания λ-функции, при этом необходимо генерировать уникальный символ (имя замыкания) с помощью функции *gen()*.

При компиляции формы *tagbody* (формула 10) используется вспомогательная функция $\mathscr{E}*$. В этой функции символы компилируются как метки, а остальные формы как при компиляции последовательности (формула 5). Форма *go* компилируется как переход на метку (формула 11).

Компиляция применения функции (формулы 12 и 13) заключается в компиляции выражений для аргументов $\varepsilon+$ и выбора необходимой команды для вызова функции. Если функция представляет собой λ-абстракцию, то используется упрощенный вызов *FIX_LET* (формула 12), который подразумевает создание кадра активации (расширение окружения переменных) без вызова функции. При этом тело функции ε_1 компилируется в расширенном окружении. Если идет вызов пользовательской функции из окружения ϕ, то используется обычный вызов *FIX_CALL* (формула 13).

Компиляция применения встроенного примитива (формула 14) аналогична компиляции применения функции за исключением того, что нет необходимости передавать число аргументов, для каждой примитивной функции это число фиксировано.

Во время второй фазы компиляции полученное выражение линеаризируется (из древовидного выражения получается линейный список) и преобразуется в примитивные инструкции (фаза оптимизации не рассматривается). Функция G этого преобразования будет иметь следующий тип:

$$G : \varepsilon \times (\nu \to \nu) \to \nu, \quad (16)$$

где ε – выражение после первой фазы; $\nu \to \nu$ – функция продолжения, которое используется, чтобы задать порядок преобразований (начальное продолжение возвращает свой аргумент). В результате получается пустое значение ν, так как сама генерация будет представлена функцией ω, которая должна добавлять свой аргумент в линейный список.

Правила преобразования второй фазы представлены формулами:

$$G[[CONST\ \varepsilon]]\kappa = \kappa\omega(CONST\ \varepsilon) \quad (17)$$

$$G[[GLOBAL\_REF\ i]]\kappa = \kappa\omega(GLOBAL\_REF\ i) \quad (18)$$

$$G[[LOCAL\_REF\ i]]\kappa = \kappa\omega(LOCAL\_REF i) \quad (19)$$

$$G[[DEEP\_REF\ i\ j]]\kappa = \kappa\omega(DEEP\_REF\ i\ j) \quad (20)$$

$$G[[RETURN]]\kappa = \kappa\omega(RETURN) \quad (21)$$

$$G[[GLOBAL\_SET\ i\ \varepsilon]]\kappa = G[\varepsilon]\lambda().\kappa\omega(GLOBAL\_SET\ i) \quad (22)$$

$$G[[LOCAL\_SET\ i\ \varepsilon]]\kappa = G[\varepsilon]\lambda().\kappa\omega(LOCAL\_SET\ i) \quad (23)$$

$$G[[DEEP\_SET\ i\ j\ \varepsilon]]\kappa = G[\varepsilon]\lambda().\kappa\omega(DEEP\_SET\ i\ j) \quad (24)$$

$$G[[LABEL\ \nu\ \varepsilon]]\kappa = \kappa\omega(LABEL\ \nu),$$

если $\varepsilon = \{\}$

$$G^{\updownarrow}[\omega(JMP\ l)]\lambda().G^{\updownarrow}[\omega(LABEL\nu)]\lambda().G[\varepsilon]\lambda().\kappa\omega(LABELl),$$

в противном случае $G^{\updownarrow}[\varepsilon]\kappa = \kappa\varepsilon$

$$l = gen() \quad (25)$$

$$G[[SEQ\ \varepsilon+]]\kappa = G+[\varepsilon+]\kappa$$

$$G+[[\varepsilon]]\kappa = G[\varepsilon]\kappa$$

$$G+[[\varepsilon\ \varepsilon+]]\kappa = G[\varepsilon]\lambda().G+[\varepsilon+]\kappa \quad (26)$$

$$G[[ALTER\ \varepsilon_c\ \varepsilon_t\ \varepsilon_f]]\kappa = G[\varepsilon_c]\lambda().G^{\updownarrow}[\omega(JNT\ l_f)]\lambda().G[\varepsilon_t]\lambda().G^{\updownarrow}[\omega(JMP\ l_a)]\lambda().G^{\updownarrow}[\omega(LABEL\ l_f)]\lambda().G[\varepsilon_f]\lambda().\kappa\omega(LABEL\ l_a)$$

$$l_f = gen()$$

$$l_a = gen() \quad (27)$$

$$G[[FIX\_CLOSURE\ \nu\ \varepsilon]]\kappa = G^{\updownarrow}[\omega(FIX\_CLOSURE\ \nu)]\lambda().\ G[\varepsilon]\kappa \quad (28)$$

$$G[[FIX\_CALL\ \nu\ \rho\ \varepsilon+]]\kappa = A[\varepsilon+]\lambda().\omega([SAVE\_ENV,\ SET\_ENV\ \rho,\ ALLOC\ \#\varepsilon+,\ REG\_CALL\ \nu,\ RESTORE\_ENV]),$$

где $A[[\varepsilon]]\kappa = G[\varepsilon]\lambda().\kappa\omega(PUSH)$

$$A[[\varepsilon\ \varepsilon+]]\kappa = A[\varepsilon]\lambda().A[\varepsilon+]\kappa \quad (29)$$

$$G[[FIX\_PRIM\ \nu\ \varepsilon+]]\kappa = A[\varepsilon+]\lambda().\kappa\omega(PRIM\ \nu) \quad (30)$$

$$G[[GOTO\ \nu]]\kappa = \kappa\omega(JMP\ \nu) \quad (31)$$

Преобразование констант, обращений к переменным, возврат из функции остаются в неизменном виде (формулы 17 – 21).

В случае присваивания переменной (формулы 22 – 24) сначала генерируется код для выражения, а затем результат записывается в переменную.

Команда *LABEL* (формула 25) используется для генерации кода функции и как метка внутри формы *TAGBODY*. В первом случае генерируется дополнительная метка после кода функции и команда перехода на эту метку, чтобы код функции нельзя было выполнить кроме как через вызов или замыкание, при последовательном выполнении программы код функции будет пропущен. Вспомогательная функция $G^{\updownarrow}$ используется для денотации последовательной генерации команд.

Последовательность выражений (формула 26) разворачивается также как при компиляции формы *PROGN*, используя аналогичную вспомогательную функцию $G+$.

Линеаризация условий (формула 27) происходит с помощью сгенерированных двух меток l_f и l_a. Сначала генерируется код условия. Затем идет команда перехода на метку l_f, если условие ложно. После этого идет генерация кода, выполняющегося по истинности условия. Затем вместе с меткой l_f генерируется код, выполняющегося по лжи. В конце добавляется метка l_a.

При генерации замыкания функции (формула 28) команда замыкания просто вставляется перед кодом функции. Этот код будет включать метку и команду перехода на точку после функции.

При генерации вызова функции (формула 29) создается последовательность разных команд. В начале генерируется код для вычисления аргументов. Для этого используется вспомогательная функция A, которая похожа на функцию $G+$ за исключением того, что после каждого выражения генерируется команда *PUSH*, которая будет сохранять результат вычисления аргумента в стеке. После этого генерируются команды сохранения текущего кадра активации, установка необходимого кадра (окружение, где была объявлена функция), создания нового кадра активации, собственно вызова функции и восстановление сохраненного кадра.

Генерация вызова примитива (формула 30) похожа на вызов функции, также генерируется код вычисления аргументов, а затем идет команда вызова примитива по номеру.

Команда *GOTO* преобразовывается в команду *JMP* (формула 31).

Результаты и их обсуждение

Полученные в результате компиляции примитивные инструкции кодируются с помощью специального ассемблера в числовой код. Этот код предназначен для выполнения с помощью целевой виртуальной машины. Константы, полученные в результате компиляции, сохраняются в память констант. Глобальные переменные будут находиться в своей памяти. Число глобальных переменных (размер памяти) также генерируется компилятором. Метки, которые были получены в результате компиляции, преобразуются в адреса.

В табл. 1 представлены коды и значения инструкций целевой машины.

Таблица 1. Инструкции целевой машины

Table 1. Target machine instructions

0	CONST num	поместить константу с номером num в регистр ACC.
1	JMP ofs	безусловный переход на смещение ofs относительно PC.
2	JNT ofs	если ACC == NIL, то относительный переход на смещение ofs.
3	ALLOC n	создать новый кадр активации с числом аргументов n.
4	GLOBAL-REF i	устанавливает регистру ACC значение глобальной переменной с индексом i.
5	GLOBAL-SET i	устанавливает глобальной переменной с индексом i значение регистра ACC.
6	LOCAL-REF i	загружает в ACC значение i локальной переменной (текущего кадра активации).
7	LOCAL-SET i	присваивает локальной переменной i (текущего кадра активации) значение регистра ACC.
8	DEEP-REF i j	загружает в ACC значение локальной переменной с индексом j в кадре i (начиная от текущего).
9	DEEP-SET i j	присваивает локальной переменной j в кадре i значение регистра ACC.
10	PUSH	добавляет значение регистра ACC в стек
11	PACK n	собирает последние n элементов из стека в список и добавляет его в стек.
12	REG-CALL ofs	добавляет адрес следующей инструкции в стек и производит переход по смещению ofs.
13	RETURN	производит переход на адрес из верхушки стека, при этом удаляет этот адрес из стека
14	FIX-CLOSURE ofs	в регистр ACC добавляется объект замыкание с текущим кадром активации и смещением на код функции относительно текущего адреса ofs.
15	SAVE-FRAME	сохраняет кадр активации в стеке
16	SET-FRAME num	устанавливает кадр активации с номером num относительно начала глубины вызовов.
17	RESTORE-FRAME	восстанавливает кадр активации из стека.
18	PRIM n	вызывает примитив с номером n из таблицы примитивов с фиксированным числом аргументов.
19	NPRIM n	вызывает примитив с номером n из таблицы примитивов с переменным числом аргументов.
20	HALT	останов машины
21	PRIM-CLOSURE n	в регистр ACC добавляется объект замыкание с текущим кадром активации и адресом кода примитива с фиксированным числом аргументов и номером n.
22	NPRIM-CLOSURE n	в регистр ACC добавляется объект замыкание с текущим кадром активации и адресом кода примитива с переменным числом аргументов и номером n.
23	CATCH ofs	добавляет запись в стек catch, сохраняется имя метки в acc, абсолютный адрес по смещению ofs, текущий кадр активации, укзатель стека.
24	THROW	извлекает из стека имя метки блока catch, ищет в стеке catch запись с этим именем, выполяет переход на сохраненный адрес конца блока catch

Архитектура целевой виртуальной машины представлена на рис. 1.

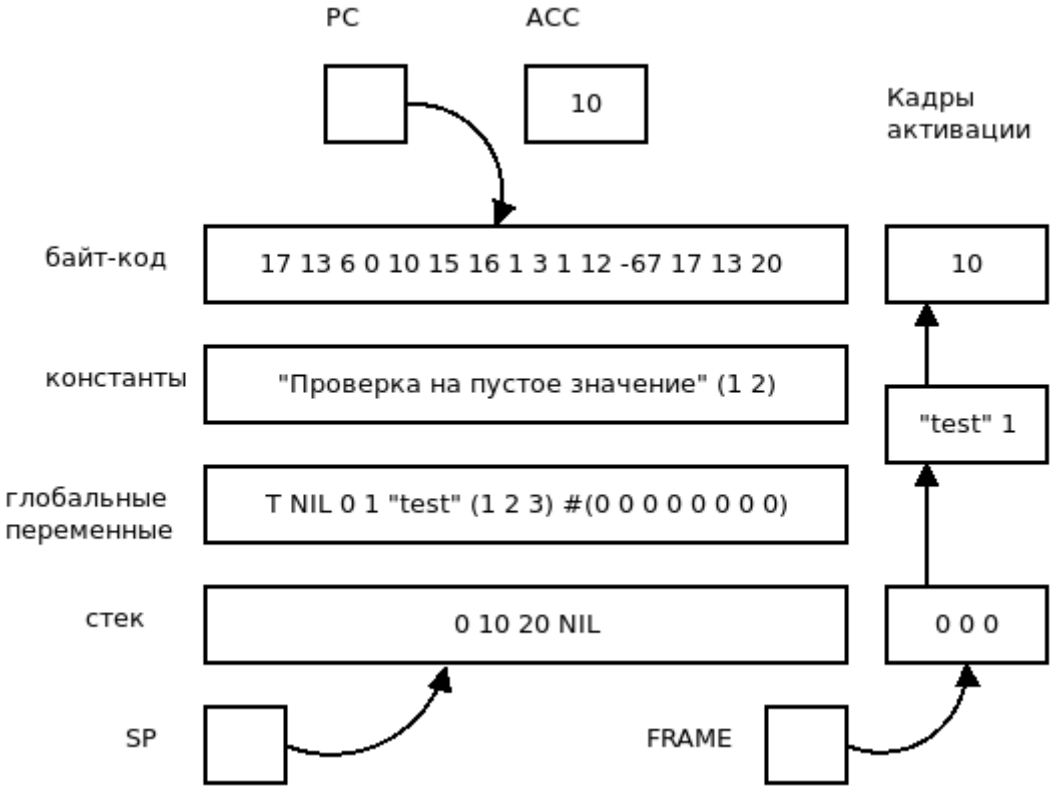

Рис. 1. Архитектура целевой виртуальной машины

Fig. 1. Target virtual machine architechture

Машина включает в себя память программы (где хранится байт код программы), память констант, память глобальных переменных, стек, список кадров активации и регистры.

В памяти программы хранятся инструкции программы Каждая инструкция включает код операции и возможные параметров.

Кадры активации представляют собой объекты-массивы (рис. 2).

Первый элемент массива – это ссылка на предыдущий кадр. Следующий элемент служит для ускорения поиска кадра по глубине вызова, здесь хранится номер кадра (глубина вызовов).

Рис. 2. Структура кадров активации

Fig. 2. Activation frames structure

Остальные элементы массива – это локальные аргументы. Кадры могут иметь не только линейную, но и древовидную структуру, поэтому необходима ссылка на предыдущий кадр. Такие объекты удобно записывать в стек, вос-

станавливать из стека. Кадры активации создаются динамически как и другие объекты программы.

Регистры машины:

PC – хранит адрес текущей выполняемой инструкции из памяти программы.

ACC – хранит результат последней операции, может быть любым объектом.

FRAME – текущий кадр активации.

SP – указатель стека.

Сборка мусора включает в себя фазу пометки и фазу очистки. В фазе пометки помечаются объекты, до которых можно дойти с помощью обхода, начиная с корневых объектов в регистрах. Это регистр аккумулятора ACC, указатель на текущий кадр активации FRAME, все объекты, находящиеся в стеке, начиная с текущей позиции SP, а также все объекты в памяти констант и глобальных переменных. Сборка мусора выполняется, когда число созданных массивов (кадров-активации) превысит заданный порог (половина максимального числа массивов).

В табл. 2 показано сравнение скорости работы полученного байт-кода по сравнению с интерпретатором на различных тестах.

Таблица 2. Время (в сек.) работы тестов на интерпретаторе и байт-кода в виртуальной машине

Table 2. Time (in sec.) to run of tests on interpreter and byte-code with virtual machine

Тест / Test	Интерпретатор / Interpreter	Скомпилированный байт-код / Compiled Bytecode	Отношение / Relation
Списки	0.034	0.011	3.1
Массивы	0.026	0.011	2.3
Хеш-таблицы	0.024	0.008	3
Строки	0.032	0.010	3.2
Объекты и классы	0.026	0.009	2.9
Векторы на плоскости	0.04	0.009	4.4
Шифрование AES	2.503	0.111	22.5
Регулярные выражения	1.923	0.116	16.6
Собственная компиляция	44.231	1.987	22.2

Из табл. 2 видно, что отношение скорости возрастает при увеличении объема программы.

Выводы

В результате работы была построена и реализована модель компилятора

для функционального языка Common Lisp. Компилятор формирует байт-код, который выполняется с помощью виртуальной машины. Выигрыш в скорости по сравнению с интерпретатором на реалистичных тестах (шифрование, регулярные выражения, компиляция самого себя) составляет 16-25 раз. Дальнейшее увеличение скорости можно достичь с помощью различных оптимизаций компилятора на разных стадиях [18]. Из очевидных оптимизаций можно отметить: оптимизацию арифметических выражений (преобразование операций с переменным числом параметров в операции с двумя параметрами), устранение лишних команд (множественная загрузка в аккумулятор, установка / чтение одной переменной), упрощение выражений [19].

Список литературы

1. Sperber Michael, Dybvig R. Kent, Flatt Matthew, Van Straaten Anton; et al. (August 2007). Revised6 Report on the Algorithmic Language Scheme (R6RS). Scheme Steering Committee. Retrieved 2011-09-13.

2. The Racket Manifesto (PDF) / M. Felleisen, R.B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, S. Tobin-Hochstadt // Proceedings of the First Summit on Advances in Programming Languages; 2015. P. 113–128.

3. Абельсон Х., Сассман Д. Структура и интерпретация компьютерных программ, КДУ, 2022. 608 с.

4. Hart Tim, Levin Mike. AI Memo 39-The new compiler (PDF). Archived from the original (PDF) on 2020-12-13. Retrieved 2019-03-18.

5. Krishnamurthi Shriram. Programming Languages: Application and Interpretation. 3-thd edition. - Shriram Krishnamurthi, 2022. 376 p.

6. Ершов А. П., Покровский С. Б. Эволюция языков программирования // Проблемы информатики. 2017. № 2(35). С. 70-79. EDN ZOLFPJ.

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

8. Романов С. С. Ключевые понятия и особенности объектно-ориентированного программирования // Таврический научный обозреватель. 2016. № 12-2(17). С. 141-146. EDN YFXCCN.

9. Гонцова А. В. Опыт разработки транслятора с языка Ассемблер RISC-V в машинный код // Обработка информации и математическое моделирование: материалы Всероссийской научно-технической конференции, Новосибирск, 24–25 апреля 2024 года. Новосибирск: Сибирский государственный университет телекоммуникаций и информатики, 2024. С. 85-88. DOI 10.55648/OIMM-2024-1-85. EDN DUZEDV.

10. Наркевич А. С. Структура байт-кода виртуальной машины Java // Информационные технологии: материалы 85-й научно-технической конференции профессорско-преподавательского состава, научных сотрудников и аспирантов (с международным участием), Минск, 01–13 февраля 2021 года. Минск: Белорусский государственный технологический университет, 2021. С. 80-82. EDN VLEPYZ.

11. Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри. Компиляторы: принципы, технологии и инструментарий. 2-е изд. М.: Диалектика-Вильямс, 2020. 1184 с.

12. Намаконов Е. С., Подкопаев А. В. Компиляция модели памяти OCaml в Power // Труды Института системного программирования РАН. 2019. Т. 31, № 5. С. 63-78. DOI 10.15514/ISPRAS-2019-31(5)-4. EDN GPHUIK.

13. Джереми Сик. Основы компиляции: инкрементный подход. СПб.: Питер, 2024. 256 с.

14. Компиляторы: принципы, технологии и инструментарий / Ахо Альфред В., Лам Миника С., Сети Рави, Ульман Джеффри Д. М.: Диалектика-Вильямс, 2018. 1184 с.

15. Сайбель Питер. Практическое использование Common Lisp. М.: ДМК Пресс, 2017. - 488 с.

16. Грэй Пол. ANSI Common LISP. М.: Символ-Плюс, 2020. 448 с.

17. 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

18. Владимиров К. И. Оптимизирующие компиляторы. Структура и алгоритмы. М.: АСТ, 2024. 272с.

19. Лопес Бруно Кардос, Аулер Рафаэль. LLVM: инфраструктура для разработки компиляторов. М.: ДМК Пресс, 2015. 342 с.

References

1. Sperber Michael, Dybvig R. Kent, Flatt Matthew, Van Straaten Anton; et al. (August 2007). Revised6 Report on the Algorithmic Language Scheme (R6RS). Scheme Steering Committee. Retrieved 2011-09-13.

2. Felleisen M., Findler R.B., Flatt M., Krishnamurthi S., Barzilay E., McCarthy J., Tobin-Hochstadt S. The Racket Manifesto (PDF). Proceedings of the First Summit on Advances in Programming Languages: 2015. 113–128.

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

4. Hart Tim; Levin Mike. AI Memo 39-The new compiler (PDF). Archived from the original (PDF) on 2020-12-13. Retrieved 2019-03-18.

5. Krishnamurthi Shriram. Programming Languages: Application and Interpretation. 3rd ed. Shriram Krishnamurthi, 2022. 376 p.

6. Ershov A. P., Pokrovsky S. B. Evolution of programming languages. *Problemy informatiki = Problems of informatics*. 2017; (2): 70-79. (In Russ.). EDN ZOLFPJ.

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

8. Romanov S. S. Key concepts and features of object-oriented programming. *Tavricheskii nauchnyi obozrevatel' = Tavricheskiy scientific observer*. 2016; (12-2): 141-146. (In Russ.). EDN YFXCCN.

9. Gontsova A. V. Experience in Developing a Translator from RISC-V Assembly Language to Machine Code. In: *Obrabotka informatsii i matematicheskoe modelirovanie : materialy Vserossiiskoi nauchno-tekhnicheskoi konferentsii = Information Processing and Mathematical Modeling. Proceedings of the All-Russian Scientific and Technical Conference.* 2024. Novosibirsk: Siberian State University of Telecommunications and Informatics; 2024. P. 85–88. (In Russ.). DOI 10.55648/OIMM-2024-1-85. EDN DUZEDV.

10. Narkevich A. S. The structure of the bytecode of the Java virtual machine. In: *Informatsionnye tekhnologii: materialy 85-i nauchno-tekhnicheskoi konferentsii professorsko-prepodavatel'skogo sostava, nauchnykh sotrudnikov i aspirantov (s mezhdunarodnym uchastiem) = Information technologies. Materials of the 85th scientific and technical conference of faculty, researchers and graduate students (with international participation.* Minsk: Belarusian State Technological University; 2021. P. 80-82. (In Russ.). EDN VLEPYZ.

11. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey. Compilers:Principles, Technologies and Tools. 2nd ed. Moscow: Dialectics-Williams; 2020. 1184 p. (In Russ.)

12. Namakonov E. S., A. V. Podkopaev Compilation of the OCaml memory model to Power. *Trudy Instituta sistemnogo programmirovaniya RAN = Proceedings of the Institute for System Programming of the Russian Academy of Sciences*. 2019; 31(5): 63-78. (In Russ.) DOI 10.15514/ISPRAS-2019-31(5)-4. EDN GPHUIK.

13. Jeremy Sick. Compilation Basics: An Incremental Approach. St.Peterburg: Peter; 2024. 256 p. (In Russ.)

14. Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Networks. Compilers: principles, technologies and tools. Moscow: Dialectics-Williams; 2018. 1184 p. (In Russ.)

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

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

17. Chaplygin A.A. Modeling of a functional programming language interpreter with metaprogramming capabilities. *Izvestiya Yugo-Zapadnogo gosudarstvennogo universiteta. Seriya: Upravlenie, vychislitel'naya tekhnika, informatika. Meditsinskoe priborostroenie = Proceedings of the Southwest State University. Series: Control, Computing 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

18. Vladimirov K. I. Optimizing compilers. Structure and algorithms. Moscow: AST; 2024. 272 p. (In Russ.)

19. Lopez Bruno Cardos, Auler Rafael. LLVM: An Infrastructure for Compiler Development. Moscow: DMK Press; 2015. 342 p. (In Russ.)

Информация об авторе / Information about the Author

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

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