Минобрнауки России

Юго-Западный государственный университет

Кафедра программной инженерии

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

ПО ПРОГРАММЕ БАКАЛАВРИАТА

09.03.04 Программная инженерия

(код, наименование ОПОП ВО: направление подготовки, направленность (профиль))

«Разработка программно-информационных систем»

Интерпретатор функционального языка программирования

с поддержкой метапрограммирования

(название темы)

Дипломный проект

(вид ВКР: дипломная работа или дипломный проект)

Автор ВКР _____ Д. А. Антипов

(подпись, дата) (инициалы, фамилия)

Группа ПО-02б

Руководитель ВКР _____ А. А. Чаплыгин

(подпись, дата) (инициалы, фамилия)

Нормоконтроль _____ А. А. Чаплыгин

(подпись, дата) (инициалы, фамилия)

ВКР допущена к защите:

Заведующий кафедрой _____ А. В. Малышев

(подпись, дата) (инициалы, фамилия)

Курск 2024 г.

Минобрнауки России

Юго-Западный государственный университет

Кафедра программной инженерии

УТВЕРЖДАЮ:

Заведующий кафедрой

_____

(подпись, инициалы, фамилия)

«____» _____ 20____ г.

ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ ПО ПРОГРАММЕ БАКАЛАВРИАТА

Студента Антипова Д. А., шифр 20-06-0118, группа ПО-02б

1. Тема «Интерпретатор функционального языка программирования с поддержкой метапрограммирования» утверждена приказом ректора ЮЗГУ от «04» апреля 2024 г. № 1616-с.

2. Срок предоставления работы к защите «11» июня 2024 г.

3. Исходные данные для создания программной системы:

3.1. Перечень решаемых задач:

- провести анализ предметной области;
- спроектировать функциональный язык программирования с поддержкой метапрограммирования;
- спроектировать интерпретатор этого языка;
- выбрать технологии и методики для реализации интерпретатора;
- реализовать интерпретатор средствами языка программирования “C” и ОС “GNU/Linux”.

3.2. Входные данные и требуемые результаты для программы:

1) Входными данными для программной системы является файл с кодировкой “UTF-8”, содержащий программный код на разработанном в рамках этой работы языке программирования.

2) Выходными данными для программной системы является результат выполнения переданного в программную систему программного кода.

4. Содержание работы (по разделам):

4.1. Введение

4.1. Анализ предметной области

4.2. Техническое задание: основание для разработки, назначение разработки, требования к программной системе, требования к оформлению документации.

4.3. Технический проект: общие сведения о программной системе, обоснование выбора технологии, проектирование архитектуры программной системы.

4.4. Рабочий проект: перечень разработанных модулей, спецификация компонентов и модулей программной системы, тестирование программной системы, сборка компонентов программной системы.

4.5. Заключение

4.6. Список использованных источников

5. Перечень графического материала:

Руководитель ВКР	_____	А. А. Чаплыгин
	(подпись, дата)	(инициалы, фамилия)
Задание принял к исполнению	_____	Д. А. Антипов
	(подпись, дата)	(инициалы, фамилия)

РЕФЕРАТ

Объем работы равен 98 страницам. Работа содержит 6 иллюстраций, 18 таблиц, 30 библиографических источников и 0 листов графического материала. Количество приложений – 2. Графический материал представлен в приложении А. Фрагменты исходного кода представлены в приложении Б.

Перечень ключевых слов: интерпретатор, функциональное программирование, метапрограммирование, Lisp, Common Lisp, C, системное программирование, информационные технологии, сокращение исходного кода.

Объектом разработки является интерпретатор функционального языка программирования с поддержкой метапрограммирования.

Целью выпускной квалификационной работы является разработка программной системы, позволяющей сокращение размера исходного кода программ за счёт метапрограммирования.

В процессе создания интерпретатора были выделены основные сущности путем создания информационных блоков, использованы поля и методы модулей, обеспечивающие работу с сущностями предметной области, а также корректную работу интерпретатора.

Разработанный интерпретатор был успешно использован для сокращения исходного кода существующей программы путём переписывания на разработанный язык с применением возможностей метапрограммирования.

ABSTRACT

The volume of work is 98 pages. The work contains 6 illustrations, 18 tables, 30 bibliographic sources and 0 sheets of graphic material. The number of applications is 2. The graphic material is presented in annex A. The layout of the site, including the connection of components, is presented in annex B.

List of keywords: interpreter, functional programming, metaprogramming, Lisp, Common Lisp, C, system programming, information technology, source code reduction.

The object of development is a functional programming language interpreter with metaprogramming support.

The purpose of the final qualification work is to develop a program system that allows reducing the size of the source code of programs due to metaprogramming.

In the process of creating the interpreter the main entities were identified by creating information blocks, the fields and methods of modules were used to ensure the work with the entities of the subject area, as well as the correct operation of the interpreter.

The developed interpreter was successfully used to reduce the source code of an existing program by rewriting it into the developed language with the use of metaprogramming capabilities.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 9
1 Анализ предметной области 12
1.1 Понятие интерпретатора 12
1.2 Строение интерпретатора 12
1.2.1 Лексический анализатор и лексема 12
1.2.2 Синтаксический анализатор, синтаксис языка программирования и внутреннее представление 13
1.2.3 Исполнитель 14
1.2.4 Сборщик мусора 15
1.3 Функциональное программирование 15
1.4 Метапрограммирование 16
1.5 Язык программирования Lisp 17
2 Техническое задание 19
2.1 Основание для разработки 19
2.2 Цель и назначение разработки 19
2.3 Описание разрабатываемого языка 20
2.3.1 Алфавит языка 20
2.3.2 Лексемы, распознаваемые лексическим анализатором 21
2.3.3 Типы данных 21
2.3.4 Функции и лямбда-выражения 23
2.3.5 Макросы 24
2.4 Компоненты интерпретатора 26
2.5 Требования к программной системе 27
2.5.1 Требования к данным программной системы 27
2.5.2 Требования к программному обеспечению 27
2.5.3 Требования к аппаратному обеспечению 28
2.6 Требования к оформлению документации 28
3 Технический проект 29
3.1 Общая характеристика организации решения задачи 29

3.2 Обоснование выбора технологии проектирования 29
3.2.1 Описание используемых технологий и языков программирования 30
3.2.2 Язык программирования C 30
3.3 Компоненты интерпретатора 31
3.3.1 Алгоритм взаимодействия компонентов 31
3.3.2 Лексический анализатор 33
3.3.3 Синтаксический анализатор 34
3.3.4 Объекты внутреннего представления 34
3.3.5 Исполнитель 39
3.3.6 Сборщик мусора 41
3.3.7 Примитивные функции 42
3.3.7.1 Арифметические функции 42
3.3.7.2 Функции исполнителя 44
3.3.7.3 Функции модуля работы с массивами 47
3.3.7.4 Функции модуля работы с точечными парами 48
3.3.7.5 Функции модуля работы со строками 49
3.4 Символы 51
3.5 Регионы 53
3.6 Пространства имён и область видимости 54
4 Рабочий проект 57
4.1 Модули, разработанные для реализации интерпретатора 57
4.2 Спецификация модулей лексического и синтаксического анализаторов 59
4.3 Модульное тестирование разработанного интерпретатора 64
4.4 Системное тестирование разработанного интерпретатора 67
4.5 Сборка программной системы 70
ЗАКЛЮЧЕНИЕ 72
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 72
ПРИЛОЖЕНИЕ А Фрагменты исходного кода программы 77
На отдельных листах (CD-RW в прикрепленном конверте) 98

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

ИТ – информационные технологии.

ПО – программное обеспечение.

ПС – программная система.

ЯП – язык программирования.

ДЯП – демонстрационный язык программирования.

ЛА – лексический анализатор.

СА – синтаксический анализатор.

UML (Unified Modelling Language) – язык графического описания для объектного моделирования в области разработки программного обеспечения.

ВВЕДЕНИЕ

Парадигма метапрограммирования уже многие годы активно используется при разработке гибкого и адаптивного программного обеспечения, обеспечивая одни программы возможностью генерировать и трансформировать другие или самих себя, тем самым позволяя сокращать исходный код и одновременно с тем повышая его конфигурируемость и масштабируемость.

Со временем сложность программ возрастает, их функциональные возможности становятся более обширными, а требования к мобильности растут. Метапрограммирование может выступать как эффективный инструмент по снижению влияния таких требований на скорость и сложность разработки. В результате программисты могут писать ПО, формирующее программный код с учётом внешних условий и на основе предъявляемых требований.

Ещё одним популярным подходом, эффективным для решения всё тех же проблем и хорошо сочетаемым с генерацией кода, выступает функциональная парадигма программирования. Языки, берущие её за основу, приобрели значительную востребованность в последние годы благодаря своей способности обеспечивать более предсказуемый и прозрачный, легко анализируемый, модульный код. Поддержка функций высшего порядка и следование принципу неизменяемости данных также выделяют их как разумный вариант для разработок с такими требованиями.

Включая метапрограммирование в функциональный язык, производится заметное повышение потенциала языка к достижению поставленных разработкой целей [29]. Удачность симбиоза этих идей, наряду с другими успешными решениями, привела к тому, что языки, такие как Haskell, F#, Erlang, OCaml, Lisp и Clojure стали превосходящими в индустрии, где требования к надежности и адаптивности систем высоки, а многие языки, изначально не имевшие таких возможностей, получают их частичную или полную поддержку [28].

В этой выпускной квалификационной работе будет спроектирован функциональный язык программирования с поддержкой метапрограммиро-

вания и программная система для выполнения кода на этом языке - транслятор типа интерпретатор. Интерпретатор читает исходный код программы, анализирует и исполняет его инструкции, выводя результаты на экран или в файл. В отличие от компилятора [2], который переводит весь код программы в машинный, интерпретатор обрабатывает его по одной инструкции, тем самым выполняя программы без предварительной компиляции. Язык, разработанный для этого интерпретатора, ставит минимализм и выразительность синтаксиса в приоритет, гарантируя, что основные концепции функционального программирования и метапрограммирования доступны и удобны в использовании.

Цель настоящей работы – разработка программной системы, позволяющей сокращение размера исходного кода программ за счёт метапрограммирования. Для достижения поставленной цели необходимо решить следующие задачи:

- провести анализ предметной области;
- спроектировать функциональный язык программирования с поддержкой метапрограммирования;
- спроектировать интерпретатор этого языка;
- выбрать технологии и методики для реализации интерпретатора;
- реализовать интерпретатор средствами языка программирования “C” и ОС “GNU/Linux”.

Структура и объем работы. Отчет состоит из введения, 4 разделов основной части, заключения, списка использованных источников, 2 приложений. Текст выпускной квалификационной работы равен 10 страницам.

Во введении сформулирована цель работы, поставлены задачи разработки, описана структура работы, приведено краткое содержание каждого из разделов.

В первом разделе на стадии описания технической характеристики предметной области приводится сбор информации о технологиях и методиках, необходимых для реализации программной системы.

Во втором разделе на стадии технического задания приводятся требования к разрабатываемому интерпретатору и языку.

В третьем разделе на стадии технического проектирования представлены проектные решения для интерпретатора.

В четвертом разделе приводится список модулей и их полей и методов, созданных при разработке, а также производится тестирование разработанного интерпретатора.

В заключении излагаются основные результаты работы, полученные в ходе разработки.

В приложении А представлен графический материал. В приложении Б представлены фрагменты исходного кода.

1 Анализ предметной области

1.1 Понятие интерпретатора

Интерпретатор языка программирования – это программа типа транслятор, считывающая программный код, написанный на определённом языке программирования, по одной инструкции или группе инструкций и немедленно исполняющая их в соответствии с синтаксисом и семантикой языка.

В сравнении с компилятором, анализирующим и единоразово преобразующим весь исходный код программы в машинный код перед её выполнением, и JIT-компилятором, выполняющим компиляцию во время исполнения, интерпретатор в реальном времени анализирует и выполняет код по одной инструкции, сначала преобразуя её во внутреннее представление и затем в машинный код [26].

1.2 Строение интерпретатора

1.2.1 Лексический анализатор и лексема

Всё начинается с лексемы [20] — символа или набора символов, минимально значимой единицы исходного кода, которую интерпретатор может распознать и обработать. Лексемы включают в себя ключевые слова, идентификаторы, константы, символы и операторы. Они служат для разделения кода на части, которые затем будут преобразованы во внутреннее представление интерпретатора и выполнены им.

С применением лексического анализатора лексемы постепенно перерабатываются в токены [20].

Токен – та же лексема, но представленная специальным значением или имеющая дополнительный атрибут, что позволит синтаксическому анализатору идентифицировать её или узнать о ней дополнительную информацию. Например, тип лексемы – число, строка и так далее.

Лексический анализ (токенизация) – формирование токенов на основе исходного кода программы, таких как ключевые слова, идентификаторы, операторы, литералы и так далее.

Лексический анализатор (ЛА) распознаёт лексемы в исходном коде программы, пропуская пустоты (пробелы, переводы строк) [19], определяет особенности лексемы и из полученных данных формирует токен, который будет передан следующим компонентам интерпретатора. Считывание текста программы из потока ввода производится до тех пор, пока не встретится зарезервированный символ, после чего начинается формирование токена. Либо, если в синтаксисе языка предусмотрены односимвольные лексемы, то формирование происходит сразу. Также лексический анализатор берёт на себя задачу по выявлению символов, не являющихся частью синтаксиса языка в анализируемом контексте, и при обнаружении таковых выводит на экран соответствующую ошибку.

1.2.2 Синтаксический анализатор, синтаксис языка программирования и внутреннее представление

Следующим этапом работы интерпретатора является анализ полученных от лексического анализатора токенов и преобразование их во внутреннее представление интерпретатора для передачи следующим его компонентам.

Внутреннее представление – это проекция выражений и конструкций языка программирования внутри интерпретатора. Им определяется как интерпретатор хранит данные и оперирует ими во время выполнения программы [18].

Синтаксис языка программирования представляет собой набор правил для написания языковых выражений и конструкций, которые будут интерпретированы и выполнены компилятором или интерпретатором соответствующего языка ровно таким образом, каким это описано в спецификации языка.

Синтаксис определяет как символы образуют лексемы и как они комбинируются для формирования языковых конструкций и выражений, которые затем интерпретируются компьютером. Он определяет порядок опера-

ций, приоритет операторов, структуру программы, а также способы объявления переменных, функций, ветвлений и других конструкций, составляющих структуру программного кода. При отсутствии ошибок в реализации интерпретатора, несоблюдение или неполное понимание синтаксических правил может привести как к ошибкам, так и к неожиданному с точки зрения разработчика поведению.

Синтаксический анализатор (СА) отвечает за анализ синтаксиса программного кода, проверку его на соответствие правилам и построение в соответствии с синтаксисом языка программирования объектов внутреннего представления интерпретатора на основе полученного от лексического анализатора токена.

Если обнаруживается ошибка в синтаксисе программы, то этот анализатор выводит на экран соответствующее сообщение об ошибке и прекращает дальнейшую обработку программы. Такой ошибкой может быть, например, отсутствующая закрывающая скобка или ключевое слово, которое отсутствует в языке программирования.

1.2.3 Исполнитель

После успешного завершения синтаксического анализа и получения сформированных СА объектов, они передаются на исполнение.

Исполнение – выполнение кода, представленного во внутреннем формате, пошаговое выполнение инструкций программы, обрабатывая операторы и вычисляя значения выражений.

В процессе исполнения интерпретатор манипулирует теми объектами, которые ранее были сформированы синтаксическим анализатором из токенов. Именно на этапе исполнения происходят все вычисления и работа с данными программы – объявление переменных и задание им значений, вызов функций и так далее.

Таким образом, исполнитель, выполняющий исполнение внутреннего представления, можно назвать ключевым компонентом интерпретатора.

1.2.4 Сборщик мусора

Сборщик мусора — это компонент интерпретатора, который отслеживает неиспользуемые объекты внутреннего представления и освобождает занятую ими память [16].

Неиспользуемыми объектами считаются те, которых нельзя достичь перемещаясь по дереву ссылок от активных объектов.

В зависимости от целей, поставленных разработчиком, наличие сборщика мусора может быть как положительной, так и отрицательной чертой языка программирования. Если вопрос занимаемой программой ОЗУ не является особенно важным, его наличие освобождает программиста от необходимости в ручном режиме выделять и освобождать память, что значительно упрощает процесс разработки, сводит к минимуму потенциальные уязвимости и проблематику утечек памяти в разрабатываемом ПО.

1.3 Функциональное программирование

Функциональное программирование – это парадигма разработки ПО, где функции выступают в роли основного элемента конструкции программ и могут в качестве аргументов принимать другие функции [8]. Тем самым выстраивается структура, основанная на функциях, их взаимодействии и композиции.

Также в этой парадигме принята идея о том, что по возможности и если то будет разумным, стоит придерживаться написания “чистых” функций – функций, не имеющих побочных эффектов и при вызове с одними и теми же аргументами всегда возвращающих одинаковый результат, без изменения состояния программы или лексического окружения.

Но, для поддержания чистоты функций, стоит по возможности следовать концепции неизменности данных, суть которой заключается в отказе от изменения каких-либо уже сформированных данных. Таким образом, предпочтение отдаётся написанию функций, которые не изменяют исходные данные, а формируют и возвращают новые на основе исходных. Оба эти подхода

способствуют формированию модульного, предсказуемого и надёжного кода, что в последствии упростит отладку и тестирование программы.

Основу для реализации функциональной парадигмы составляют функции высшего порядка и лямбда-функции.

Функции, имеющие возможность принимать в качестве аргументов другие функции, тем самым формируя цепи функциональных преобразований, где одни функции могут быть переданы и манипулируемы подобно другим объектам языка вроде списков или чисел, называются функциями высшего порядка [11]. Умелое использование таких возможностей для создания обобщённых функций, применяемых для выполнения более широкого спектра условий, приводит к повышению модульности, переиспользуемости и выразительности кода.

Лямбда-функции (анонимные функции) – это безымянные функции высшего порядка, которые используют в качестве аргумента для передачи другим функциям, возврата функции из функции или одноразового применения, что позволяет без необходимости не занимать пространство имён.

1.4 Метапрограммирование

Метапрограммирование — это вид программирования, который связан с созданием программ, генерирующих другие программы как результат своей работы, или программ, изменяющих себя во время выполнения. В функциональном программировании такой подход используется часто, потому как функции сами являются данными и могут быть переданы как аргументы другим функциям, создавать новые функции и изменять собственные тела.

Метапрограммирование реализуется системой макросов [5], позволяющей разработчику создавать новые языковые конструкции, генерируя, изменяя и делая динамическим код программы .

Макросы – это функции, генерирующие код, который в последствии заменит код генерации. Они работают на этапе компиляции или интерпретации, позволяя трансформировать исходный код перед его выполнением. Макросы позволяют создавать собственные синтаксические конструкции и

расширять язык, что открывает для разработчика почти безграничные возможности по адаптации языка под свои нужды и создания адаптивного ПО.

Например, макросы можно применить для включения или исключения частей кода в зависимости от условий запуска и особенностей компьютера, на котором происходит запуск. Помимо прочего, использование макросов может ускорить работу программ за счёт возможности один раз сгенерировать функцию с определёнными аргументами, а после переиспользовать её, без необходимости каждый раз заново вызывать функцию с одними и теми же аргументами.

1.5 Язык программирования Lisp

Функциональный язык программирования с уклоном в сферу разработки искусственного интеллекта, один из самых старых используемых и теперь языков — Lisp, появившийся в 1958 трудами учёного Джона Маккарти.

Инновационность языка состояла в том, что его автор спроектировал удобный инструментарий для работы со списками и символами, что было очень востребовано при решении задач обработки естественного языка и символьной логики [1]. Список в Lisp – главный элемент, потому как весь программный код на нём в конечном итоге состоит из множества списков. Хотя в первое время Lisp использовался только для решения неширокого перечня задач в сфере искусственного интеллекта, спустя чуть более чем десять лет с момента создания он всё же получил широкую известность и на долгое время стал центральным в этой сфере.

Кроме того, наличие успешной реализации системы макросов, составляющей в нём основу парадигмы метапрограммирования, в сумме с другими преимуществами сделала его востребованным для разработки предметно-ориентированных языков [10]. Суть языков такого типа заключается в их адаптированности под конкретные задачи и способы применения, способствуя таким образом удобству написания программного кода.

Также примечательным является, что сборщик мусора и возможность использовать функции подобно данным впервые были введены именно в этом языке.

Постепенно оригинальный Lisp отходил на второй план и известность перенимали его диалекты. На данный момент одним из наиболее используемых является Common Lisp — диалект, ставящий своей целью объединение удачных решений других разновидностей оригинального языка, чтобы сформировать мультипарадигменную, очень гибкую и достаточно широкую в плане способов применения и базовой функциональности вариацию.

2 Техническое задание

2.1 Основание для разработки

Полное наименование системы: “Интерпретатор функционального языка программирования с поддержкой метапрограммирования”.

Основанием для разработки программы является приказ ректора ЮЗГУ от «15» апреля 2024 г. №1779-с «Об утверждении тем выпускных квалификационных работ».

2.2 Цель и назначение разработки

Цель этой работы – разработка программной системы, позволяющей сокращение размера исходного кода программ за счёт метапрограммирования.

Для достижения этой цели было принято решение разработать интерпретатор функционального языка программирования с поддержкой метапрограммирования. Основной задачей разработки является разработка программного обеспечения, способного анализировать и исполнять программы, написанные на функциональном языке программирования, а также обеспечивать возможности генерации и изменения кода на этом языке с использованием инструментов метапрограммирования.

Интерпретатор, созданный в рамках данной работы, должен иметь все ключевые функции, обеспечивающие поддержку парадигм метапрограммирования и функционального программирования. Для их реализации будет разработан простой и минималистичный функциональный язык программирования, называемый “демонстрационный язык программирования” (ДЯП), поддерживающий метапрограммирование.

Таким образом, интерпретатор сможет работать с числами, строками, переменными, функциями, лямбда-выражениями, макросами и другими необходимыми конструкциями, обеспечивающими разработчику возможность использовать метапрограммирование для создания адаптивных и реплицируемых приложений.

Задачами данной разработки являются:

- разработка синтаксиса ДЯП, достаточного для реализации функционального программирования и метапрограммирования;
- разработка объектов, используемых для представления интерпретируемого исходного кода внутри интерпретатора;
- разработка сборщика мусора;
- разработка лексического анализатора для созданного языка;
- разработка синтаксического анализатор для созданного языка;
- разработка исполнителя инструкций;
- реализация примитивных функций созданного языка.

2.3 Описание разрабатываемого языка

Разрабатываемый язык является подмножеством языка программирования “Common Lisp” и сосредотачивается на реализации его базовых возможностей по работе с данными и метапрограммирования. Потому, он будет иметь функциональность для работы с переменными, функциями, лямбда-выражениями, функциями высшего порядка, числами, строками, символами, списками и массивами. Введутся базовые функции для обработки данных, включая операции сложения чисел, выделения подстрок из строк, определения имени символа по символьному объекту и другие, реализующие минимально необходимые возможности для манипуляции данными. Также будет включена система макросов как основной элемент реализации метапрограммирования.

2.3.1 Алфавит языка

Алфавит языка программирования – это перечень символов, допустимых к использованию для записи синтаксических конструкций этого языка [3]. Символом может быть как буква, цифра или знак препинания, так и любой другой знак, рассматриваемый как неделимый элемент языка.

Алфавит разработанного языка включает:

- латинские символы верхнего и нижнего регистра;

- римские цифры;
- символы, зарезервированные под описание конструкций языка, перечисленные через пробел: ’ ‘ , . ” #;
- другие символы, перечисленные через пробел: + - * / = _ & | < >.

2.3.2 Лексемы, распознаваемые лексическим анализатором

Список наименований лексем, распознаваемых лексическим анализатором, а также их символьное представление или пример:

- десятичное и шестнадцатеричное целое число: 10, 0xFFAA;
- вещественное число: 1.34;
- символ: A;
- цитата: ’;
- квазицитата: ‘;
- запятая: ,;
- запятая-at: ,@;
- решётка: #;
- левая скобка: (;
- правая скобка:);
- точка: .;
- строка: “a b c v ddd”;
- неизвестный объект – объект, который ЛА не смог определить;
- конец потока.

Помимо этого, для удобства разработчика, синтаксисом языка предусмотрена возможность добавлять в код комментарии. Комментарий начинается со знака “;” и заканчивается переносом строки.

Символ “\” используется для экранирования.

2.3.3 Типы данных

Для языка программирования были определены шесть фактических типов данных и два псевдотипа:

- Число – десятичные и шестнадцатеричные числа, размер которых ограничен 28 битами (от -134217728 до 134217727). Например: 10, 0xFFAA;

- Большое число – число, для хранения которого необходимо более 28 бит. Например: 134217728;

- Строка – произвольный набор алфавитных символов, задающийся в двойных кавычках. Например: “ab 12 /”;

- Символ – именованный нечувствительно к регистру объект, который может указывать на некоторое значение – число, лямбда-выражение, макрос, строку, массив, список или функцию. Имя символа должно начинаться с буквы или разрешенного символа и может содержать буквы, цифры, символы. Таким образом, переменные, функции и другие объекты языка, к которым можно обращаться по имени, являются символами, содержащими указатель на объект с данными, заданными для этого символа. Пример символа: A;

- Атом – псевдотип, специальное название для обозначения примитивных объектов данных, которые не разбиваются составляющие: символы, числа и строки;

- Точечная пара – это хранилище, содержащее только два элемента, называемые левым и правым. Пара носит название точечной, так как в синтаксисе эта конструкция представляет собой два элемента, разделённые точкой, обрамлённые в круглые скобки. Например: (’a . 2);

- Список – хранилище с последовательным доступом к элементам, содержащее ноль или более атомов, разделенных пустотами (пробелы или переводы строк) и заключённых в круглые скобки. С точки зрения внутреннего представления списки являются синтаксическим упрощением, реализованным за счёт точечных пар, где левый элемент пары – значение, а правый – указатель на следующую точечную пару. Таким образом выстраивается цепь точечных пар. Правый элемент последнего элемента такой цепи указывает на специальное значение nil. Списки могут содержать в себе другие списки. Пример: (x 2 ’p);

- Массив – хранилище с прямым доступом к элементам (можно обращаться по индексу), содержащее ноль или более атомов, разделенных пусто-

тами и заключённых в круглые скобки, перед открывающей ставится #. Массивы могут содержать в себе другие массивы. Пример: #(3 6 9).

Для идентификации типов в интерпретаторе будет использоваться перечисление, содержащее следующие значения:

- NUMBER: число;
- BIGNUMBER: большое число;
- SYMBOL: символ;
- PAIR: точечная пара (список);
- STRING: строка;
- ARRAY: массив.

S-выражение – это основной элемент синтаксиса языка, который может быть атомом или списком. S-выражения нечувствительны к регистру. Все инструкции в ДЯП являются s-выражениями, из чего следует, что программный код представляет собой множество s-выражений [9].

Для обозначения “истинного” и “ложного” используются зарезервированные объекты-символы “T” и “NIL” соответственно.

2.3.4 Функции и лямбда-выражения

В языке программировании применяются функции и лямбда-выражения.

Функция представлена в виде списка, содержащего символ, имя которого соответствует имени функции, список аргументов и тело функции.

Для объявления новой функции используется функция defun, имеющая следующий синтаксис: (defun name (p_1 ... p_n) e), где name – имя объявляемой функции, p_1 ... p_n – параметры функции, e – тело функции.

Вызов функции – список, где первый элемент это имя функции, а последующие являются её аргументами.

Синтаксис вызова функции: (name a_1 ... a_n), где name – имя вызываемой функции, а a_1 ... a_n – передаваемые функции аргументы.

Лямбда-выражения объявляются идентично функциям, но вместо функции defun используется lambda и, так как лямбда-выражения безымянны, имя не задаётся.

Для вызова лямбда-выражения необходимо создать список, где первым элементом будет само лямбда-выражение, а $a_1 ... a_n$ – передаваемые выражению аргументы: ((lambda ($p_1 ... p_n$) e) $a_1 ... a_n$).

При вызове, сначала вычисляются все аргументы $a_1 ... a_n$. Затем каждому параметру $p_1 ... p_n$ ставится в соответствие вычисленное значение аргументов $a_1 ... a_n$. После этого вычисляется выражение e, содержащее параметры, вместо которых будут подставлены их значения.

Например:

< ((lambda (x y) (cons x (cdr y))) 'z '(a b c))

> (Z B C).

Это лямбда-выражение с помощью функции cons создаёт список, состоящий из значения аргумента x и обрезанного со второго элемента с помощью cdr списка y. Результатом выполнения этого кода будет новый список – "(Z B C)".

Функции и лямбда-выражения могут быть получены и переданы в качестве аргументов или возвращены из функции и лямбда-выражения, как и необходимо в функциональной парадигме программирования.

2.3.5 Макросы

В ДЯП для создания макросов применяется функция defmacro, а также операторы quote, backquote и comma.

Функция defmacro имеет следующий синтаксис: (defmacro name ($a_1 ... a_n$) e), где name – имя макроса, $a_1 ... a_n$ – параметры макроса, e – тело макроса.

Пример использования:

1. Создам макрос, задающий шаблон для генерации выражения:

(defmacro test (var val) (list 'defvar var val)).

При вычислении этот макрос заменится списком, первым элементом которого будет символ defvar для объявления переменной, а последующими – переданные при вызове макроса аргументы.

2. Вызову макрос с символом “abc” и числом 100 в качестве аргумента. При вызове происходит вычисление тела макроса (развертывание макроса):

(test abc 100) преобразуется в (defvar abc 100).

3. Получившееся выражение вычисляется. Список расценивается как код, который определяет переменную и присваивает ей значение:

< (defvar abc 100)

> ABC.

По итогу был создан макрос test, объявляющий переменную с именем, переданным ему в качестве первого аргумента, и значением в качестве второго.

Кавычки (’) – символ, используемый для реализации цитирования – предотвращения вычисления выражения. Например, выражение (* 2 2) будет автоматически вычислено и даст 4, в то время как ’(* 2 2) будет восприниматься как список символов.

Но для предоставления по-настоящему широкого спектра возможностей для разработчика, необходимо реализовать инструментарий, позволяющий выполнять частичные вычисления – квазицитирование.

Для того будут использоваться символы обратной кавычки “‘” и запятой “,”. Обратная кавычка будет указывать на то, что выражение содержит вычисляемые элементы, а запятая укажет на них. Кавычка, обратная кавычка и запятая действуют на вычисление только того выражения, перед которым стоят.

Пример использования:

< (defvar b 12) (print ‘(+ a ,b))

> (+ A 12).

Таким образом, “+” и “a” были восприняты интерпретатором как символы и остались невычисленными, а “b” заменён значением соответствующей переменной.

2.4 Компоненты интерпретатора

На рисунке 2.1 в виде UML-диаграммы показаны компоненты, составляющие интерпретатор [25].

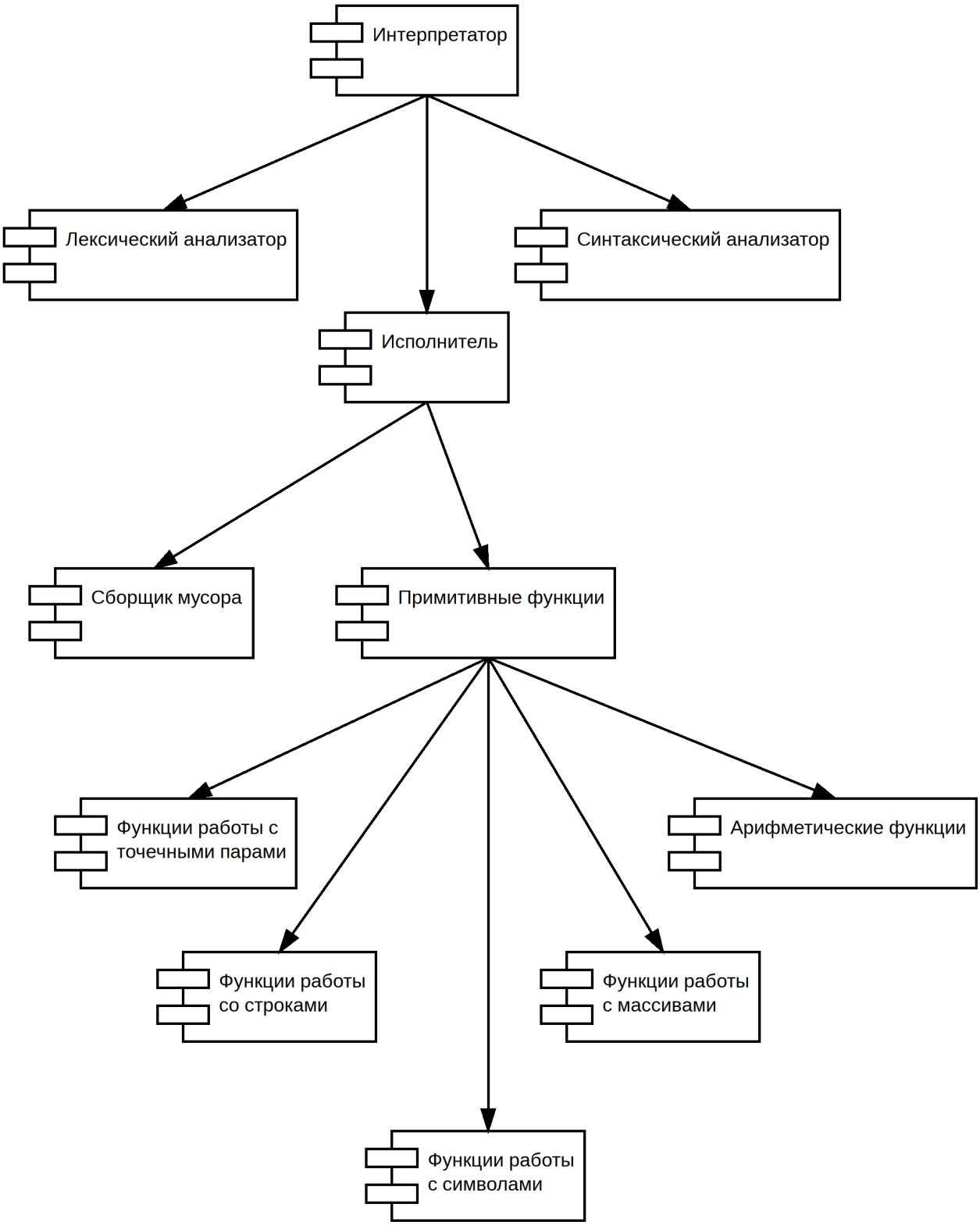

Рисунок 2.1 – Диаграмма компонентов интерпретатора

Таким образом, разработанный интерпретатор должен реализовывать следующие компоненты:

- Лексический анализатор - для формирования токенов на основе текстового представления программы и выявления ошибок, связанных с использованием отсуствующих в алфавите языка символов или недопустимых в к использованию в некотором контексте символов (например, буква внутри числа: 124a6);

- Синтаксический анализатор - для формирования на основе токенов представления программы внутри интерпретатора и выявления синтаксических ошибок: отсутствие закрывающей скобки, отсутствие аргументов и тому подобные;

- Исполнитель - для выполнения инструкций, описанных в интерпретируемой программе;

- Сборщик мусора - для выявления и освобождения памяти, хранящей элементы, более не использующиеся интерпретатором;

- Примитивные функции - для реализации встроенных в язык функций, инструментов и конструкций, позволяющих производить манипуляции данными.

2.5 Требования к программной системе

2.5.1 Требования к данным программной системы

Программная система должна принимать на вход файл с кодировкой “UTF-8”, содержащий программный код на ДЯП.

2.5.2 Требования к программному обеспечению

Для реализации программной системы должны быть использованы: язык программирования “C”, реализация стандартной библиотеки языка - “glibc”.

Для сборки ПС требуется ОС семейства “GNU/Linux”, поддерживающая компилятор GCC версии “10.1” или новее.

2.5.3 Требования к аппаратному обеспечению

Для работы ПС необходим компьютер с архитектурой центрального процессора “x86” и свободной для использования пользовательскими процессами оперативной памятью от 64 Мб.

2.6 Требования к оформлению документации

Разработка программной документации и программного изделия должна производиться согласно ГОСТ 19.102-77 и ГОСТ 34.601-90. Единая система программной документации.

3 Технический проект

3.1 Общая характеристика организации решения задачи

Необходимо спроектировать и разработать программную систему, которая должна способствовать сокращению исходного кода программ без ущерба их функциональности за счёт возможностей метапрограммирования.

Для достижения этой цели было принято решение спроектировать язык программирования и создать программу для его интерпретации, удовлетворяющие описанным выше требованиям.

Главной задачей разработки интерпретатора языка программирования является создание программного обеспечения, которое способно интерпретировать и выполнить исходный программный код, написанный на определенном языке программирования, так, что результат выполнения соответствует правилам, описанным в спецификации интерпретируемого языка.

Для обеспечения конкурентоспособности интерпретатора, эта задача должна выполняться как можно более эффективно и быстро. Кроме того, он должен выполнять свою задачу в соответствии с главным принципом интерпретации – код программы обрабатывается по одной инструкции или группе инструкций, выполняющихся сразу после анализа и обработки.

Для реализации интерпретатора необходимо разработать следующие компоненты: лексический анализатор, объекты внутреннего представления, синтаксический анализатор, инструментарий выполнения команд языка и сборщик мусора.

3.2 Обоснование выбора технологии проектирования

Уже многие годы сфера ИТ предоставляет массу инструментов для разработки системного ПО, коим и является разработка интерпретатора.

3.2.1 Описание используемых технологий и языков программирования

В процессе разработки интерпретатора ДЯП используются язык программирования “C” и программные средства операционной системы семейства “GNU/Linux”. Используемые для создания программно-информационной системы средства отвечают современным практикам разработки и являются подходящими и достаточными для решения задач, выявленных при анализе предметной области.

3.2.2 Язык программирования C

Низкоуровневый язык программирования “C” (Си) – один из первых языков программирования и, одновременно с этим, один из самых используемых до сих пор. Его появлению в начале 1970-х годов мир обязан инженеру Деннису Ритчи из американской компании “Bell Labs”, разрабатывавшим его как развитие языка “Би” для написания операционной системы “Unix” [6]. С тех пор Си стал одним из самых популярных языков для системного программирования.

Об успешности решений, принятых при его разработке, говорит впечатлительный список узнаваемых последователей, перенявших многие его идеи – C++, C#, Objective-C, Java, Python, PHP и другие обязаны Си своей структурой кода и базовым синтаксисом.

Узнаваемость и простота его синтаксиса, близость к аппаратной части ЭВМ, наличие компилятора почти для всех вычислительных устройств и операционных систем, обширная стандартная библиотека, а также ручное управление памятью убеждают в выборе языка C для системного программирования, коей и является разработка интерпретатора.

Ввиду того, что реализация стандартной библиотеки этого языка, libc, отличается для различных операционных систем, было принято решение выбрать целевой платформой для разработки интерпретатора одно семей-

ство ОС – “GNU/Linux”. В этих системах применяется реализация “GNU C Library” (glibc) [12].

3.3 Компоненты интерпретатора

3.3.1 Алгоритм взаимодействия компонентов

Пошаговый алгоритм взаимодействия компонентов, составляющих интерпретатор:

Шаг 1. Инициализируем исполнитель и регистрируем все примитивные функции ДЯП;

Шаг 2. Сохраняем текущее состояние программы, сохранив регистры процессора и стек с помощью setjmp. Если во время работы интерпретатора произойдёт ошибка – вывести на экран сообщение об ошибке и перейти к шагу 8;

Шаг 3. Запускаем синтаксический анализатор;

Шаг 4. Синтаксический анализатор вызывает лексический анализатор;

Шаг 5. Лексический анализатор на основе символов из входного потока формирует лексему и возвращает её;

Шаг 6. Синтаксический анализатор на основе полученной лексемы формирует объект для внутреннего представления и возвращает его;

Шаг 7. Если был достигнут конец входного потока – переходим к следующему шагу, иначе к шагу 12;

Шаг 8. Восстановить состояние программы к тому, что было сохранено на шаге 2, и перейти к шагу 13;

Шаг 9. Исполнитель производит вычисления с объектом, сформированным СА, в качестве аргумента и возвращает результат вычислений в виде объекта;

Шаг 10. Вывести возвращённый исполнителем объект на экран;

Шаг 11. Перейти к шагу 2;

Шаг 12. Сборщик мусора освобождает неиспользуемые объекты внутреннего представления. Перейти к шагу 2;

Шаг 13. Завершить работу интерпретатора.

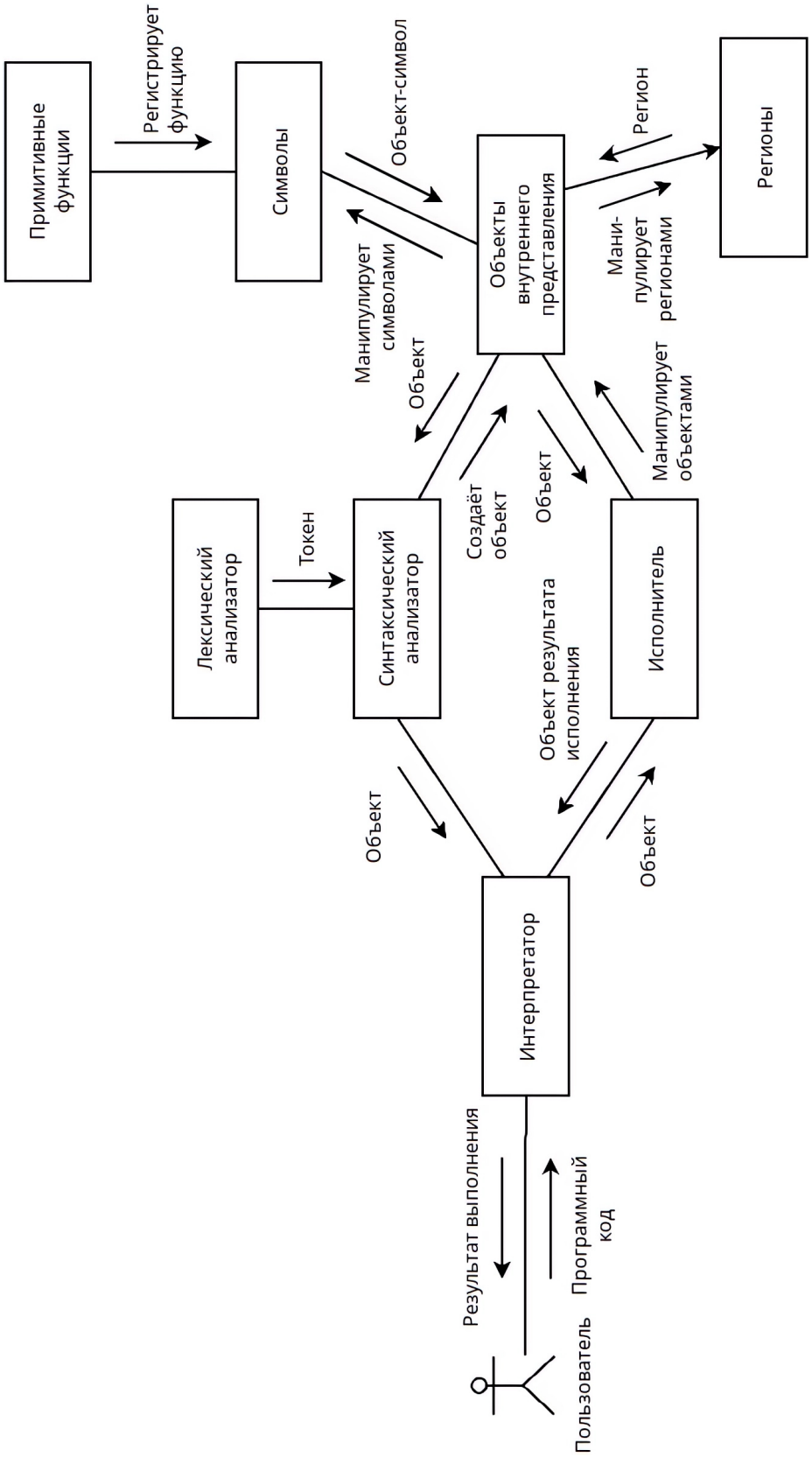

Рисунок 3.1 – Диаграмма взаимодействия компонентов интерпретатора

На рисунке 3.1 в виде UML-диаграммы взаимодействия показано как компоненты интерпретатора обмениваются данными.

3.3.2 Лексический анализатор

В разработанной программной системе лексический анализатор представляет собой компонент, в задачи которого входит считывание лексемы, проверка её на соответствие алфавиту языка и формирование токена.

Как только лексема была считана, она помещается в буфер размерностью в восемь символов.

Считанная лексема сравнивается со множеством зарезервированных под конструкции языка символов. При совпадении с каким-либо, формируется токен и ЛА возвращает его.

Сформированный токен представляет собой структуру с такими полями:

- type – тип токена;
- value – поле для токенов числового типа, содержит значение числа;
- str – поле для токенов строкового типа, значение строки.

Особым случаем является считывание строки. Как только обнаруживается символ кавычки, запускается функция, собирающая символы до тех пор, пока:

- не встретит второй символ кавычки, чем будет закончено считывание строки, после чего ЛА вернёт сформированный токен;
- количество символов не превысит допустимую длину строки, что приведёт к ошибке;
- не будет достигнут конец входного потока, что также приведёт к ошибке.

Если лексема не является одной из зарезервированных, производится проверка на то, является она числом или символом.

Ввиду того, что имя символа, как и отрицательное число, может начинаться со знака минус, решается неоднозначность, связанная с восприятием ЛА следующих символов. Для этого считывается ещё один символ и, если

он является числовым, дальнейшая запись определяется как число, иначе как символ.

Если имя символа состоит из допустимых знаков, то бишь из алфавита языка, формируется токен типа символ и возвращается как результат работы ЛА. В противном случае, выдаётся ошибка о некорректности символа.

3.3.3 Синтаксический анализатор

Синтаксический анализатор запрашивает у ЛА по одному токену, строит для них внутреннее представление и повторяет процесс, пока не будет сформировано одно s-выражение, затем происходит возврат выражения.

СА, как и ЛА, выполняет проверку на ошибки, но уже синтаксические. Он выявляет отсутствие аргументов у операторов цитирования и квазицитирования, неоконченость списков (отсутствие закрывающей скобки) и отсутствие списка после символа “#” для создания массива.

3.3.4 Объекты внутреннего представления

Как уже было рассмотрено ранее, весь программный код, считываемый из потока ввода, ещё на этапе обработки лексическим анализатором преобразуется в особые структуры, а не хранится в виде текста, ввиду необходимости манипулирования предоставленной им информацией, что было бы проблематичным и неэффективным при текстовом хранении. После обработки синтаксическим анализатором, все данные, полученные из программного кода, принимают своё окончательное представление и при исполнении инструкций меняется уже не их представление, а содержимое.

В разрабатываемом интерпретаторе данные объектов хранятся в виде структур языка C, доступ к которым осуществляется через специальный указатель, называемый оболочкой объекта (object_t, псевдоним типа long long размером 8 байт), содержащий в себе:

- Тип объекта (биты с 0 по 2): позволяет функциям узнать тип хранимых данных и верно их обработать;

- Бит пометки (3 бит), необходимый для функционирования сборщика мусора. Используется только в точечных парах;

- Данные объекта (биты с 4 по 64): малое число при числовом типе или указатель на экземпляр структуры данных, соответствующей одному из остальных типов.

Выделение и последующее чтение или изменение этих битов из оболочки происходит за счёт функций побитового сдвига [4].

Тип объекта извлекается из оболочки с использованием макроса “TYPE”. Он возвращает числовое значение, соответствующее позиции типа в перечислении (enum) с индексами от нуля до пяти: NUMBER, BIGNUMBER, SYMBOL, PAIR, STRING, ARRAY.

Макросы “GET_PAIR”, “GET_BIGNUMBER”, “GET_STRING”, “GET_SYMBOL”, “GET_ARRAY” позволяют получать из оболочки указатель на структуру, интерпретируемую в соответствии с выбранной функцией. Их работа основана на другом макросе – “GET_ADDR”, выделяющем адрес структуры в памяти из оболочки.

Оболочка объекта является результатом выполнения любого s-выражения и доступ из примитивных функций к структурам данных осуществляется только через неё. Всего было разработано пять структур, реализующих все доступные в языке типы, за исключением числового: большое число, пара, символ, строка, массив. Числовой же тип автоматически освобождается средствами языка “C”.

Далее под числовым типом будут подразумеваться как объекты типа “число”, так и “большое число”. При необходимости явно выделить последний, его наименование будет писаться полностью.

Структуры, представляющие данные объекта, по мере надобности распределяются из соответствующего для их типа пула – массива фиксированного размера.

При необходимости создать экземпляр структуры, если список свободных структур этого типа, сформированный сборщиком мусора, пуст – созда-

ётся новый, иначе берётся из головы соответствующего типу списка свободных экземпляров.

Структуры этих пяти типов объектов представлены в таблицах 3.1 – 3.5.

Таблица 3.1 – Структура “bignumber_t” для объекта типа “большое число”

Имя	Тип	Описание	Возможные значения
value	int	Числовое значение	Нет ограничений
next	object_t*	Указатель для сборщика мусора на следующий свободный объект-пару	Если NULL — данная пара является последней в списке свободных или список пуст, иначе содержит указатель на следующий свободный объект-пару.
free	int	Свободна ли пара для перезаписи	Если 1 — пара в списке свободных пар, иначе занята

Таблица 3.2 – Структура “pair_t” для объекта-пары

Имя	Тип	Описание	Возможные значения
1	2	3	4
left	object_t*	Указатель на левый элемент пары	Указатель на объект или NULL
right	object_t*	Указатель на правый элемент пары	Указатель на объект или NULL
next	object_t*	Указатель для сборщика мусора на следующий свободный объект-пару	Если NULL — данная пара является последней в списке свободных или список пуст, иначе содержит указатель на следующий свободный объект-пару.

Продолжение таблицы 3.2

1	2	3	4
free	int	Свободна ли пара для перезаписи	Если 1 — пара в списке свободных пар, иначе занята

Таблица 3.3 – Структура “string_t” для объекта-строки

Имя	Тип	Описание	Возможные значения
data	char*	Данные строки	Нет ограничений
length	int	Длина строки	Натуральное число, соответствующее количеству символов в строке
next	string_t*	Указатель для сборщика мусора на следующую свободную строковый объект	Если NULL — данная строка является последней в списке свободных или список пуст, иначе содержит указатель на следующую свободную объект-пару.
free	int	Свободна ли строка для перезаписи	Если 1 — строка в списке свободных строк, иначе занята

Таблица 3.4 – Структура “array_t” для объекта-массива

Имя	Тип	Описание	Возможные значения
1	2	3	4
data	object_t**	Данные массива	Нет ограничений
length	int	Количество элементов массива	Натуральное число, соответствующее количеству элементов массива

Продолжение таблицы 3.4

1	2	3	4
next	array_t*	Указатель для сборщика мусора на следующий свободный массив	Если NULL — данный объект-массив является последним в списке свободных или список пуст, иначе содержит указатель на следующий свободный объект-массив.
free	int	Свободен ли массив для перезаписи	Если 1 — массив в списке свободных массивов, иначе занят

Таблица 3.5 – Структура “symbol_t” для объекта-символа

Имя	Тип	Описание	Возможные значения
1	2	3	4
str	char[]	Имя символа	NUMBER
next	symbol_t*	Указатель на следующий за данным символ в хеш-таблице	Нет ограничений
value	object_t *	Указатель на объект, связанный с символом	Если NULL — данный объект является последним в списке свободных или список пуст, иначе содержит указатель на следующий свободный объект.

Продолжение таблицы 3.5

1	2	3	4
lambda	object_t*	Указатель на объект лямбда-выражения, связанный с символом	Если 1 — связан, 0 — не связан.
macro	object_t	Указатель на объект макроса, связанный с символом	Если 1 — связан, 0 — не связан.
func	func_t	Указатель на примитивную функцию, связанную с символом.	Нет ограничений

“func_t” – это указатель на функцию, которая принимает “object_t” в качестве аргумента и возвращает “object_t”.

3.3.5 Исполнитель

Исполнитель получает на вход два значения:

- obj – объект, представляющий s-выражение, которое необходимо выполнить;
- env – окружение, в котором s-выражение obj будет выполняться.

Ниже приведено пошаговое представление алгоритма работы исполнителя, где некоторые шаги имеют подшаги, которые также надо пройти, если условие шага верхнего уровня выполняется:

Шаг 1. Если объект obj равен NULLOBJ, вернуть NULLOBJ, окончив этим выполнение алгоритма;

Шаг 2. Иначе, если тип obj равен NUMBER, BIGNUMBER, STRING или ARRAY – вернуть obj, окончив этим выполнение алгоритма;

Шаг 3. Иначе, если тип obj равен SYMBOL;

Подшаг 3.1. Если в env есть объект, содержащий указатель на искомый символ, вернуть этот объект, окончив этим выполнение алгоритма;

Подшаг 3.2. Иначе проверить наличие символа, соответствующего имени искомого, в хеш-таблице. Если символ найден и ссылается на объект, содержащий его, вернуть данный объект, окончив этим выполнение алгоритма;

Подшаг 3.3. Иначе вызвать функцию error с описанием ошибки об отсутствии символа, которое будет выведено на экран, окончив этим выполнение алгоритма;

Шаг 4. Иначе, если тип obj равен PAIR;

Подшаг 4.1. Если первый элемент цепи obj является цепью;

Подшаг 4.1.1. Если первый элемент цепи obj имеет особенности, указывающие на то, что он является лямбда-выражением – выполнить это лямбда-выражение в окружении env и вернуть результат, окончив этим выполнение алгоритма;

Подшаг 4.1.2. Иначе вызвать функцию error с описанием ошибки в структуре лямбда-выражения, которое будет выведено на экран, окончив этим выполнение алгоритма;

Подшаг 4.2. Ввести переменную s. Найти (или создать при отсутствии), символ в хеш-таблице, имя которого соответствует имени искомого, и задать значением для s этот символ; Ввести переменную args;

Подшаг 4.3. Если первый элемент цепи obj имеет особенности, позволяющие определить его как специальную форму, задать для args значение хвоста цепи obj;

Подшаг 4.4. Иначе рекурсивно вычислить в окружении env список аргументов из хвоста цепи obj;

Подшаг 4.5. Если символ s содержит указатель на функцию – выполнить её с аргументами args в окружении env и вернуть вычисленное значение, окончив этим выполнение алгоритма;

Подшаг 4.6. Иначе, если символ s содержит указатель на функцию примитивного типа – выполнить её с аргументами args и вернуть вычисленное значение, окончив этим выполнение алгоритма;

Подшаг 4.7. Иначе, если символ s содержит указатель на макрос – вычислить макро-подстановку с аргументами args в окружении env и вернуть вычисленное значение, окончив этим выполнение алгоритма;

Подшаг 4.8. Иначе вызвать функцию error с описанием ошибки о том, что функцию не удалось найти, окончив этим выполнение алгоритма;

Шаг 5. Иначе вызвать функцию error с описанием ошибки о том, что исполнитель не может определить тип переданного ему объекта. Конец алгоритма.

3.3.6 Сборщик мусора

Спроектированный в рамках этой работы сборщик мусора работает в две фазы по алгоритму пометки и очистки и осуществляет сборку по следующему принципу:

1. Фаза пометки. Обходим все символы в таблице символов и выполняем пометку объектов, на которые они указывают. Пометка реализуется через третий бит оболочки объекта. Если помечается объект-пара, то левый и правый объекты этой пары пометятся рекурсивно;

2. Фаза очистки. Обходим все выделенные объекты и пары. Если есть пометка – снимаем её, иначе рекурсивно освобождаем объект, задавая значением для поля “free” число 1. Для “next” указываем следующий свободный объект, а только что освобождённый добавляется в начало соответствующего списка свободных объектов.

Объекты освобождаются в конце вычисления выражения верхнего уровня. Объекты типа “символ” и “число (маленькое)” сборщиком не затрагиваются.

Пометка производится в бит пометки, под который выделена часть в одном из полей экземпляра структуры. Для разных структур оно отличается:

- Большие числа – старший бит поля free (int, 4 байта);
- Точечные пары – четвёртый бит поля left (long long, 8 байт);
- Строки и массивы – старший бит поля length (int, 4 байта).

Таким образом, в случае больших чисел, массивов и строк запись производится непосредственно в один их элементов структуры объекта, а в случае пар – в оболочку объекта “object_t”.

Пометка реализуется функцией “mark_object”, принимающей в качестве аргумента оболочку объекта и производит пометку данных, на которые она ссылается. В случае с парами, пометка производится рекурсивно.

Освобождение непомеченных объектов выполняет функция “sweep”.

Сборщиком мусора формируются списки объектов разных типов, которые были освобождены для дальнейшей перезаписи: “free_bignumbers”, “free_pairs”, “free_strings” и “free_arrays”. В программной реализации они представляют собой указатели на некоторый объект, освободившийся последним, через поле “next” которого можно продвигаться по списку дальше, к следующим свободным объектам, пока не будет достигнуто значение NULL, означающее конец списка.

Если такой список не пуст, то есть не равен “NULL”, при необходимости создать объект, вместо этого он будет взят из головы списка и его поля будут перезаписаны, после чего голова будет сдвинута на значение вперёд по полю “next”.

3.3.7 Примитивные функции

Было реализовано множество функций, встроенных в разрабатываемый язык.

3.3.7.1 Арифметические функции

Разработанные арифметические функции позволяют выполнять базовые арифметические операции вроде суммирования и деления, а также побитовые, эквивалентности и сравнения. Перечень таких функций, их описания и примеры использования представлены в таблице 3.6.

Также там содержатся перечни типов обрабатываемых функцией аргументов. Перечисление происходит через запятую, если функция принимает сразу несколько аргументов. Если же необходимо указать что для одного ар-

гумента функция может принимать объекты определённых нескольких типов, они записываются через ”или”.

Таблица 3.6 – Перечень функций арифметического модуля

Имя	Аргументы	Описание	Пример
1	2	3	4
Суммирование	Числа	Возвращает сумму чисел списка	< (+ 1 2 3) > 6
Вычитание	Числа	Возвращает разность чисел списка	< (- 5 2 1) > 2
Произведение	Числа	Возвращает произведение чисел списка	< (* 2 1 2) 4
Деление	Числа	Возвращает результат от деления чисел списка	< (/ 8 2) 4
Больше чем	Числа	Возвращает результат сравнения на большее из двух чисел списка. Если левое больше правого – T, иначе NIL	< (> 2 1) > T
Меньше чем	Числа	Возвращает результат сравнения на меньшее из двух чисел списка. Если левое меньше правого – T, иначе NIL	< (< 2 1) > NIL
Разность чисел	Числа	Возвращает результат сравнения на меньшее из двух чисел списка. Если числа равны – T, иначе NIL	< (= 2 1) > NIL

Продолжение таблицы 3.6

1	2	3	4
Эквивалентность объектов по значению	Любые	Возвращает результат сравнения значений двух объектов списка. Если значения идентичны – T, иначе NIL	< (equal 2 2) > T
Побитовое И	Числа	Возвращает результат побитового умножения чисел списка	< (& 1 1 0) > 0
Побитовое ИЛИ	Числа	Возвращает результат побитового сложения чисел списка	< (bitor 1 1 0) > 0
Побитовый сдвиг влево	Числа	Первый аргумент – число, на которое будет применён сдвиг, второй – число бит сдвига. Возвращает результат побитового сдвига влево числа	< (« 1 2) > 8
Побитовый сдвиг вправо	Числа	Первый аргумент – число, на которое будет применён сдвиг, второй – число бит сдвига. Возвращает результат побитового сдвига вправо числа	< (» 0xF0 8) > 15

3.3.7.2 Функции исполнителя

Исполнитель содержит в своём модуле все функции, отвечающие за:

- объявление функций, переменных, макросов и их вычисление нестандартными способами;

- логические операторы;

- создание списков;
- цитирование и квазицитирование;
- проверка объектов на атомарность и эквивалентность атомов.

Полный перечень функций представлен в таблице 3.7.

Таблица 3.7 – Перечень функций исполнительного модуля

Имя	Аргу-менты	Описание	Пример
1	2	3	4
Проверка на атом	Любой	Если объект является атомом – возвращает T, иначе NIL	< (atom ’a) > T
Эквивалентность атомов	Любые	Если два атома равны – возвращает T, иначе NIL	< (eq ’a ’a) > T
Цитирование	Любой	Возвращает аргумент без вычисления	< (quote (+ 1 2)) > (+ 1 2)
Квазицитирование	Любой	Возвращает аргумент с частичными вычислениями	< (setq b 1) < (backquote (+ ,b 2 3)) > (+ 1 2 3)
Условие	Списки	Аргументы вычисляются до тех пор, пока не будет достигнут результат вычисления T. Каждый аргумент – список, где первый элемент – проверяемое выражение, а второй – результат, который будет возвращен при истинности выражения	< (cond ((eq ’a ’b) 1) (t 2)) > 2

Продолжение таблицы 3.7

1	2	3	4
Объявление функции	Символ, списки	Объявляет новую функцию с именем, соответствующим первому аргументу, списку параметров – второму аргументу и телу функции – третьему	< (defun pl (x) (+ 1 x)) < (pl 2) > 3
Применение функции к аргументам	Символ или лямбда-выражение, любые	Эта функция применяет значение первого аргумента как функцию к остальным аргументам и возвращает результат применения	< (funcall '+ 1 2) > 3
Объявление макроса	Символ, списки	Объявляет новый макрос с именем, соответствующим первому аргументу, списку параметров – второму аргументу и телу макроса – третьему	< (defmacro db (x) `(* 2 ,x)) < (db 3) > 6
Макроподстановка	Список	Возвращает результат макроподстановки для переданного в качестве аргумента квотированного вызова макроса	< (defmacro db (x) `(* 2 ,x)) < (macroexpand '(db 3)) > (* 2 3)
Последовательное выполнение	Любые	Последовательно вычисляет все s-выражения, переданные в качестве аргументов, и возвращает результат последнего вычисленного	< (progn (+ 1 2) 5) > 5

Продолжение таблицы 3.7

1	2	3	4
Объявление переменной	Символ, любой	Объявляет переменную с именем, переданным в качестве первого аргумента, и значением – второго.	< (setq val 3) > 3
Логическое ИЛИ	Списки	Возвращает T после нахождения первого истинного условия. Если истинных нет – вернёт NIL. Должно быть хотя бы одно условие	< (or (= 1 2)) > NIL
Логическое И	Списки	Возвращает NIL после нахождения первого ложного условия. Если ложных нет – вернёт T. Должно быть хотя бы одно условие	< (and (> 2 1) (< 1 2)) > T
Создание списка	Любые	Возвращает список, сформированный из переданных аргументов	< (list 1 ’x ’(12 3)) > (1 X (12 3))
Вычисление s-выражения	Любой	Вычисляет s-выражение и возвращает результат его вычисления	< (eval ’(/ 4 2)) > 2

3.3.7.3 Функции модуля работы с массивами

Этот модуль функций включает инструменты для создания массива, получения и установки значения.

Полный перечень функций представлен в таблице 3.8.

Таблица 3.8 – Перечень функций модуля работы с массивами

Имя	Аргументы	Описание	Пример
Создание пустого массива	Число	Создает пустой массив заданной аргументом длины и возвращает его	< (make-array 3) > #(NIL NIL NIL)
Размер массива	Массив	Возвращает размер массива	< (array-size #(1 2)) > 2
Задать значение элементу	Массив, число, любой	Задает значение элементу массива с некоторым индексом и возвращает массив. Аргументы: массив, индекс, значение	< (seta #(1 2) 0 10)) > #(10 2)
Чтение элемента	Массив, число	Возвращает значение элемента массива по некоторому индексу. Аргументы: массив, индекс	< (aref #(4 2) 1) > 2

3.3.7.4 Функции модуля работы с точечными парами

Этот модуль реализует инструменты для работы со списками и точечными парами.

Полный перечень функций представлен в таблице 3.9.

Таблица 3.9 – Перечень функций модуля работы с точечными парами

Имя	Аргументы	Описание	Пример
Первый элемент	Список	Возвращает первый элемент переданного аргументом списка	< (car ’(a b)) > A
Исключение первого элемента	Список	Возвращает переданный аргументом список без первого элемента	< (cdr ’(a b c)) > B C
Создание пары	Список	Создаёт точечную пару, где левая часть – первый элемент переданного аргументом списка, правая – второй.	< (cons ’a ’b) > (A . B)
Заменить левую часть пары	Пара, любой	Заменяет левую часть пары, переданной первым аргументом, значением второго аргумента и возвращает получившуюся пару	< (rplaca ’(a . b) ’d) > (D . B)
Заменить правую часть пары	Пара, любой	Заменяет правую часть пары, переданной первым аргументом, значением второго аргумента и возвращает получившуюся пару	< (rplacd ’(a . b) ’d) > (A . D)

3.3.7.5 Функции модуля работы со строками

Этот модуль реализует инструменты для работы непосредственно со строками и строковыми преобразованиями, а также с именами символов

Полный перечень функций представлен в таблице 3.10.

Таблица 3.10 – Перечень функций модуля работы со строками

Имя	Аргументы	Описание	Пример
1	2	3	4
Создание символа	Строка	Создаёт символ с именем, соответствующим первому аргументу, и возвращает созданный символ	< (intern ”A”) > A
Объединение двух строк	Строка, строка	Возвращает объединение двух строк, переданных аргументами	< (concat ”a_”) > ”a_”
Получение имени символа	Символ	Возвращает имя символа, переданного в качестве аргумента	< (symbol-name ’sym) > SYM
Получение длины строки	Строка	Возвращает длину строки, переданной в качестве аргумента	< (string-size ”123”) > 3
Получение символа из строки	Строка, число	Возвращает код символа из строки по некоторому индексу	< (char ”123”2) > 51
Получение подстроки	Строка, число, число	Возвращает подстроку из строки, начиная с начального индекса и по конечный индекс, не включая последний. Аргументы: строка начальный_индекс конечный_индекс	< (subseq ”123”0 2) > ”12”
Число в строку	Число	Возвращает строку, содержащую число, переданное в качестве аргумента	< (inttostr 12) > ”12”

Продолжение таблицы 3.10

1	2	3	4
Код символа в строку	Число	Возвращает символ в строковом представлении на основе кода символа, переданного в качестве аргумента	< (code-char 51) > 3

Также была разработана функция для проверки объекта на принадлежность к символам – “symbolp”. Если переданный ей объект является символом, она возвращает T, иначе NIL. Пример использования:

< (symbolp ‘a)

> T.

3.4 Символы

Ввиду того, что объекты-символы необходимо получать по имени и делать это часто, для их хранения был выбран способ хеш-таблицы. Хранение в массиве и поиск символов в нём с помощью перебора будет занимать всё больше времени по мере роста количества элементов. С хеш-таблицей [15] же перебор не требуется, потому время не возрастает. Этот способ основан на вычислении специального целочисленного значения, хеша, на основе имени символа. Хеш для каждого уникального имени так же будет уникальным. Таким образом, можно реализовать систему, где значение хеша будет являться индексом элемента в массиве и выполнять запись и чтение с помощью него.

Для реализации этого мезанизма был создан массив со значениями типа symbol_t. Разработанная хеш-функция позволяет генерировать 257659 индексов. Для отслеживания количества созданных символов имеется переменная last_symbol с изначальным значением “0”, которое повышается на единицу при создании нового символа. Такое отслеживание необходимо для контроля за тем, когда максимально возможное количество символов в массиве будет

достигнуто, а также получения сборщиком мусора информации о том, сколько символов ему необходимо обходить на фазе пометки.

Поля и функции, разработанные для реализации таблицы символов, добавления в неё элементов и поиска представлены в таблицах 3.11 и 3.12 соответственно.

Таблица 3.11 – Спецификация полей модуля “symbols.c”

Имя	Тип	Описание
HASH_SIZE	Символическая константа	Задаёт размер хеш-таблицы
MAX_SYMBOL_SIZE	Символическая константа	Задаёт максимальную длину имени символа
hash_table	symbol_t* []	Хеш-таблица размером “HASH_SIZE” для хранения символов

Таблица 3.12 – Спецификация методов модуля “symbols.c”

Имя	Тип	Описание
1	2	3
hash	< char *str > unsigned int	Генерирует хеш-код для строки str
compare_str	< char *str1, char *str2 > int	Посимвольно сравнивает строки str1 и str2. Если одинаковы – возвращает 1, иначе 0
check_symbol	< char *str > symbol_t*	Ищет символ в хеш-таблице по имени str. Если найден, возвращает указатель на него, иначе NULL
find_symbol	< char *str > symbol_t*	Ищет символ в хеш-таблице по имени str. Если не найден – создаёт. Возвращает указатель на полученный символ

Продолжение таблицы 3.12

1	2	3
register_func	< char *name, func_t func_ptr	Регистрирует символ с именем name, который будет указывать на функцию func_ptr

3.5 Регионы

Хранение точечных пар, строк, массивов и больших чисел организовано на основе специально разработанного для этой цели инструмента – регионов.

Регионы организованы в двунаправленный список. Каждый регион может быть свободным или занятым, однако последовательно идущих свободных регионов быть не может, потому как они объединяются при освобождении. Изначально существует один свободный регион, из которого выделяются фрагменты – области памяти изначального региона.

Такая организация хранения объектов предполагает, что механизм управления памятью берёт под регионы достаточно большие фрагменты памяти и самостоятельно реализует только необходимые возможности для управления фрагментами. Данный подход позволит снизить производительные затраты, поскольку при такой организации возможно выделять и освобождать память большими блоками, избегая фрагментации и убирая из процесса нагрузку, связанную с освобождением каждого отдельного фрагмента, как это делалось бы при реализации на массивах.

Для хранение точечных пар, строк, массивов и больших чисел производится выделение региона под каждый из типов. Их размеры соответствуют максимальному количеству объектов типа, умноженному на занимаемое соответствующей структурой объёмом памяти. Аналогично объектам-символам, для каждого присутствует поле для отслеживания количества созданных объектов.

Функции управления регионами памяти:

- Выполняющаяся первой функция init_regions инициализирует изначальный свободный регион, из которого далее будут выделяться области памяти для новых регионов. Здесь же создаётся глобальная переменная regions, представляющая список регионов. Первый регион помечается как свободный, инициализируются указатели на следующий и предыдущий регионы (изначально оба равны NULL), а также размер региона – разница между выделенным под регионы количесту байт и памяти, необходимой для хранения структуры этого региона;

- Выделение нового региона заданного в байтах размера происходит с использованием alloc_region. Ей производится поиск свободного региона, размер которого достаточен. Если такой найден, внутри него выделяется память, обновляются указатели на следующий и предыдущий регионы, и регион помечается как занятый. В противном случае выведется ошибка о нехватке памяти. В результате работы функция возвращает указатель на данные внутри выделенного региона;

- Освобождение памяти региона производится с функцией free_region, куда передаётся адрес данных освобождаемого региона. Сначала проверяется поле MAGIC структуры переданного региона с целью убедиться в том, что переданный указатель не указывает на изначальный регион. После происходит освобождение памяти региона, соответствующего переданному. Он помечается как свободный. Также, если следующий или предыдущий регион свободен, происходит замена указателей на них;

- Для удобства отладки была разработана функция, вычисляющая объём занятой регионами памяти – regions_mem. Она проходит по всем регионам, суммирует размеры занятых регионов в байтах и возвращает полученное значение.

3.6 Пространства имён и область видимости

Пространство имён – это логическое хранилище, связывающее уникальные имена с данными, а также поддерживающее эту уникальность и формирующееся в соответствии с правилами языка программирования [17].

Чаще всего в языках программирования структуры разных типов могут иметь одинаковые имена, но они всё так же должны быть уникальны в рамках своего типа. Это достигается за счёт логической группировки имён, реализуемой на основе использования множества пространств имён [21].

В разрабатываемом языке таких пространств два – для переменных и функций. В итоге, одновременно могут существовать одна переменная и одна функция с одинаковыми именами. Повторное же определение в рамках пространства, где имя уже занято, приведёт к ошибке.

Ввиду того, что при разработке хоть сколь-нибудь больших программ создаётся большое количество переменных и занимается пространство имён соответственно, появляется необходимость в механизме, который позволял бы управлять доступностью и принципом создания переменных в некоторой точке выполнения программы. Иначе говоря, формировал контекст. Это обеспечит большее удобство именования переменных и позволит решить проблему конфликта имён. Такой механизм – область видимости [17].

Для разных областей видимости, глобальной и локальной, также имеются два пространства имён.

К переменным и функциям, заданным в локальной области, можно обратиться исключительно из той области, в которой они были заданы. Такими переменными являются параметры функций и доступны они только до тех пор, пока функция выполяется. По окончании выполнения первой функции из цепочки вызовов, пространство удаляется.

В случае с глобальной областью, объявленные в ней переменные доступны для чтения и изменения из любой части программы.

Локальная область видимости имеет приоритет над глобальной. Если выполняется блок кода, для которого была сформирована локальная область, поиск и изменение существующих переменных будет происходить в первую очередь в ней. Если же переменная там отсутствует, произведётся поиск в глобальной области. Иначе выведется ошибка.

Таблица символов представляет собой глобальное окружение. Функция для задания значения переменным, setq, создаёт только глобальные переменные.

Текущее же окружение варьируется и хранится в глобальной переменной в модуле исполнителя. Локальными переменными являются параметры функций. Применение setq на такую переменную приведёт к изменению её значения.

4 Рабочий проект

4.1 Модули, разработанные для реализации интерпретатора

С целью обеспечить логическое разделение кода, во время его написания были разработаны модули, представляющие собой связанные друг с другом исходные файлы на языке C, составляющие функциональность интерпретатора.

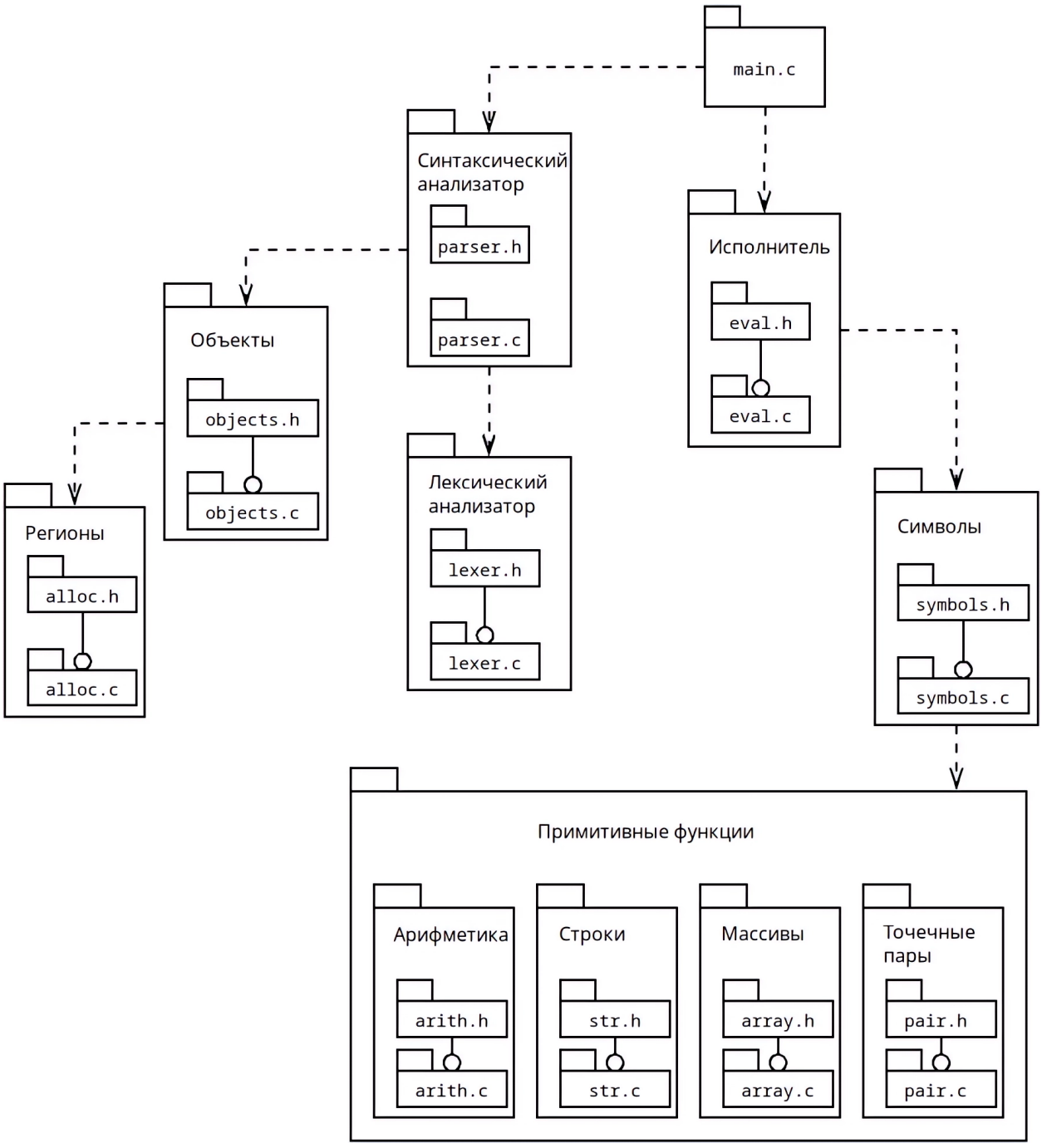

Рисунок 4.1 – Диаграмма пакетов интерпретатора

Эти файлы и их зависимость друг от друга продемонстрированы на рисунке 4.1 в виде UML-диаграммы пакетов [30].

Перечень модулей представлен в таблице 4.1, куда входят их имена, роль в структуре разрабатываемой ПС, а также описания выполняемых ими задач.

Таблица 4.1 – Модули интерпретатора

Имя	Роль	Задача
1	2	3
main.c	Главный модуль	Выполняет инициализацию всех остальных модулей, организует работу между ЛА и исполнителем, запускает сборщик мусора, выполняет возвращение в точку до ошибки при её возникновении
alloc.c	Модуль работы с регионами	Хранение массивов и строк реализовано через “регионы”, инструменты для управления которыми помещены в этот модуль
objects.c	Модуль внутреннего объектного представления	Содержит функции, позволяющие создавать объекты и манипулировать ими
arith.c	Модуль арифметических функций	Реализует примитивные арифметические функции
str.c	Модуль работы со строками	Реализует инструменты для создания объектов-строк и манипуляции ими
array.c	Модуль работы с массивами	Реализует инструменты для создания объектов-массивов и манипуляции ими
symbols.c	Модуль работы с символами	Реализует инструменты для создания объектов-символов и манипуляции ими
pair.c	Модуль работы с парами	Реализует инструменты для создания точечных пар и списков и манипуляции ими

Продолжение таблицы 4.1

1	2	3
lexer.c	Лексический анализатор	Реализует все возможности, необходимые для анализа лексем и создания токенов на их основе
parser.c	Синтаксический анализатор	Реализует все возможности, необходимые для формирования объектов внутреннего представления на основе токенов
eval.c	Модуль исполнения	Исполняет s-выражения, связывая тем самым примитивные функции языка и объекты внутреннего представления

4.2 Спецификация модулей лексического и синтаксического анализаторов

Для более глубокого раскрытия задач ЛА (lexer.c) и СА (parser.c) в таблицах 4.2 - 4.5 приведены поля и методы этих модулей с их именами, описаниями и типами входных и выходных данных.

Таблица 4.2 – Спецификация полей модуля “lexer.c”

Имя	Тип	Описание
1	2	3
cur_symbol	char	Символ, считанный последним (текущий символ)
token	token_t	Токен, генерируемый в момент лексического анализа (текущий токен)

Продолжение таблицы 4.2

1	2	3
token_error	int	Флаг, оповещающий парсер о возникновении ошибки при лексическом разборе. Если произошла ошибка, его значение – 1, иначе 0
SYMBOL_BUFFER_SIZE	Целочисленная символическая константа	Задаёт размер буфера символов
symbol_buffer	char[]	Буфер символов, считанных из стандартного потока ввода. Имеет обратный порядок элементов. Размер буфера задаётся значением “SYMBOL_BUFFER_SIZE”
buffer_write_pos	int	Текущая позиция записи в буфере символов
buffer_read_pos	int	Текущая позиция чтения из буфера символов

Таблица 4.3 – Спецификация методов модуля “lexer.c”

Имя	Ввод / вывод	Описание
1	2	3
get_cur_char	> void	Считывает символ из потока ввода и помещает его в буфер. Когда буфер заполняется, производится сдвиг позиций чтения и записи

Продолжение таблицы 4.3

1	2	3
unget_cur_char	> void	Возвращает позицию чтения назад на единицу и переприсваивает текущий символ
is_whitespace	< char c > int	Проверяет является ли символ пробельным (пробел, перенос строки, табуляция)
skip_comment	> void	Пропускает все символы, пока не достигнет переноса строки или конца входного потока
skip_white_space	> void	Выполняет пропуск при обнаружении пробельного символа или знака комментария, оперируя для этого функциями “is_whitespace” “skip_comment”
is_digit	< char c > int	Проверяет является ли символ “c” цифрой (от 0 до 9). Если да – возвращает 1, иначе 0
is_alpha	< char c > int	Проверяет является ли символ “c” буквой латинского алфавита в верхнем или нижнем регистре. Если да – возвращает 1, иначе 0
is_symbol	< char c > int	Проверяет является ли символ “c” разрешённым (+-*/=_&|<>). Если да – возвращает 1, иначе 0
is_hex_symbol	< char c > int	Проверяет является ли символ “c” одним из шестнадцатеричных символов от “a” до “f” и от “A” до “F”

Продолжение таблицы 4.3

1	2	3
is_delimeter	< char c > int	Проверяет является ли символ “c” разделителем: открывающая и закрывающая скобки, обратная косая черта, двойная кавычка, пробельный символ, конец потока
hex_num	> int	Преобразует шестнадцатеричное число из потока ввода в десятичное и возвращает его
get_float_num	> void	Принимает целую часть “int_num” от вещественного числа и считывает оставшиеся после точки числа. Преобразует имеющиеся данные в число с плавающей точкой в формате целочисленного (int) и возвращает его
get_num	> int	Считывает из потока ввода число в десятичной или шестнадцатеричной системе счисления и приводит его к десятичной. Записывает его в переменную “cur_num”, после чего возвращает
get_symbol	< char *cur_str > void	Считывает имя символа из “cur_str” и проверяет его на корректность. Если не корректное – задаёт для “token_error” значение 1 и выводит ошибку
get_string	< char *cur_str > void	Считывает строку, обрамлённую двойными кавычками, из “cur_str”. Если отсутствует закрывающая кавычка, выводит ошибку и устанавливает значение “token_error” в 1

Продолжение таблицы 4.3

1	2	3
get_comma	> token_t	Обрабатывает лексему “,” или “,@” из входного потока, формирует для неё токен и возвращает его
get_sharp	> token_t	Обрабатывает лексему “#” или “# \” из входного потока, формирует для неё токен и возвращает его
get_token	> token_t*	Считывает лексему из потока ввода, формирует на её основе токен и возвращает его, используя для этого все вышеперечисленные функции

Таблица 4.4 – Спецификация полей модуля “parser.c”

Имя	Тип	Описание
token_error	extern int	Флаг, устанавливаемый лексером для оповещения парсера о возникновении ошибки при лексическом разборе. Если произошла ошибка, его значение – 1, иначе 0
cur_token	token_t	Последний полученный токен (текущий токен)

Таблица 4.5 – Спецификация методов модуля “parser.c”

Имя	Ввод / вывод	Описание
1	2	3
strupr	< char *str > char*	Преобразует строку “str” в верхний регистр и возвращает её

Продолжение таблицы 4.5

1	2	3
parse_quote	< char *quote_sym > object_t	Считывает s-выражение из потока ввода и помещает его как аргумент в вызов функции цитирования или квазицитирования, после чего возвращает полученный объект-список
parse_element	< type_t type, void *data, tokentype_t t_type > object_t	Обрабатывает элемент списка, формирует на его основе объект и возвращает
parse_list	> object_t	Обрабатывает список, формируя объекты на основе входящих в него токенов, пока не будет достигнута закрывающая скобка. По окончании возвращает указатель на сформированный объект-список
parse_array	> object_t	Формирует объект-массив на основе входящих в него токенов
parse	> object_t	Считывает токены, составляющие s-выражение, формирует на их основе объект-список и возвращает его

4.3 Модульное тестирование разработанного интерпретатора

Для тестирования разработанной программной системы были созданы пакеты модульных и системного тестов.

Модульные тесты вызывают функции компонентов интерпретатора с некоторыми параметрами и проверяют полученные на их выходе результаты с ожидаемыми. Этот тип тестов позволяет достаточно детально проверить

работу не только какого-либо механизма ПС, но и функций, которые этот механизм формируют [13].

На каждый модуль системы был разработан собственный пакет тестов, где почти каждая функция проверяется несколькими способами, покрывая проверками все их сценарии работы.

Один из них, тест модуля символов (test_symbols.c), продемонстрирован в виде таблицы 4.6, содержащей наименование тестируемой функции модуля, тестовый случай и ожидаемый результат, а также краткое описание принципа работы теста.

Таблица 4.6 – Тестовые случаи для модуля “symbols.c”

Имя	Ввод/вывод	Цель проверки
1	2	3
Сравнение двух строк (compare_str)	< “abc”, “abc” > 1 < “abc”, “abc1” > 0	Тестовый случай охватывает варианты с совпадающими и несовпадающими строками, когда на выходе единица и ноль соответственно
Создание символа (find_symbol)	< “ab” > Объект-символ с именем “ab” < “a” > Объект-символ с именем “a”	Символ с заданным именем отсутствует в таблице символов, потому будет создан. Тестовый случай проверяет, что имя созданного символа соответсвует заданному
Тест на поиск символа по пустой строке (поочерёдно для find_symbol и check_symbol)	< “” > NULL	Тестовый случай проверяет, что при получении пустой строки, она будет обработана особым образом и будет возвращено значение NULL

Продолжение таблицы 4.6

1	2	3
Строка для поиска символа имеет недопустимую длину (поочерёдно для find_symbol и check_symbol)	< Строка длиной 82 символа > NULL	Тестовый случай проверяет, что при получении функцией строки, длиной превосходящей допустимую – более 81 символа, она будет обработана особенным образом и из неё вернётся значение NULL
Создание символа по строке максимальной длины (find_symbol)	< Строка длиной 81 символ > Новый символ с заданным именем	Тестовый случай проверяет, что при обработке функцией строки максимальной длины не происходит ошибка неучтённой единицы
Поиск символ по строке максимальной длины (check_symbol)	< Строка длиной 81 символ > Найденный по заданному имени символ	Тестовый случай проверяет, что при обработке функцией строки максимальной длины не происходит ошибка неучтённой единицы
Регистрация функции (register_func)	< ”TEST test >	Регистрация функции прошла успешно и можно получить её по имени ”TEST”и указатель на её тело соответствует переданному – test.

Продолжение таблицы 4.6

1	2	3
Создание и получение символа (check_symbol и find_symbol)	< ”f” > NULL > Новый символ с именем ”f” > Созданный ранее символ с именем ”f”	Проверив с помощью check_symbol отсутствие символа, он будет создан с применением find_symbol и проверка проведётся повторно. Тем самым производится контроль за корректностью работы двух функций вместе
Разные символы с одинаковым хеш-значением (hash и find_symbol)	< “PJ” “452” ”\xe4\x44\x8a” > Символы не равны	Хеш-значения, вычисленные на основе строк из параметров, одинаковы, но объекты-символы, сформированные по ним, указывают на разные области памяти

Вывод в консоль результатов выполнения этих тестов также показан на рисунке 4.2.

4.4 Системное тестирование разработанного интерпретатора

Системные же тесты проверяют работу ПС по принципу чёрного ящика, с теми же возможностями, что есть у пользователя. Этот метод подходит для проверки корректности работы интерпретатора в целом [14]. Суть подхода заключается в запуске передаваемого в виде строки программного кода и сравнения выведенных в консоль результатов его выполнения с эталонными.

Для тестирования используется bash-скрипт “sys_test”. При его вызове программный код подаётся в двойных кавычках первым аргументом, а эталонный (ожидаемый) результат аналогичным образом вторым аргументом. При совпадении фактического и ожидаемого результата, в поток вывода [27]

```
--------------test symbols----------------------
test_compare_str:  1 OK
test_compare_str:  0 OK
test_find_symbol:  0 OK
test_find_symbol:  0 OK
test_find_symbol_empty_string:  0 OK
test_find_symbol_invalid_string_length:  0 OK
test_find_symbol_max_string_length:  0 OK
test_check_symbol_empty_string:  0 OK
test_check_symbol_invalid_string_length:  0 OK
test_check_symbol_max_string_length:  0 OK
test_register_func:  -1813093642 OK
test_check_symbol:  0 OK
 -1804290000 OK
test_register_func:  -1813093642 OK
test_same_hash:88900
88900
 1 OK
test_same_hash_three_symbols:88900
88900
88900
 1 OK
```

Рисунок 4.2 – Вывод в консоль результатов выполнения модульного теста test_symbols.c

попадёт “OK”, иначе “FAIL”. Запуск тестового пакета необходимо производить в консольном интерфейсе, запись результатов при этом будет происходить в стандартный поток вывода консоли.

Несколько тестов из пакета системного тестирования представлены на рисунке 4.3.

```
sh sys_test "(< 1 2)" "T"
sh sys_test "(< 3 2)" "NIL"
sh sys_test "(< 3 3)" "NIL"
sh sys_test "(progn (+ 2 3) (* 3 4))" "12"
sh sys_test "(defun test (x) (+ x 10) (* x 20)) (test 5)" "TEST
100" # 18
sh sys_test "(setq a 10) (A)A" "10
Unknown func: A
10" # 23 24
sh sys_test "(setq  a 10) A" "10
10" # 23 24
sh sys_test "T NIL" "T
NIL"
sh sys_test "(setq +bgr-index+ 0x1ce) \
(setq +bgr-data+ 0x1cf) +BGR-INDEX+ +BGR-DATA+ X ABC" "462
463
462
463
Unknown SYMBOL: X
Unknown SYMBOL: ABC" # 23
```

Рисунок 4.3 – Часть тестов из пакета системного тестирования

Вывод в консоль результатов выполнения этих тестов так же показан на рисунке 4.4.

Алгоритм работы тестового пакета:

1. Определяем переменные “IN”, “OUT”, и “OUT_EXP”, используемые для хранения путей к временным файлам для хранения кода, фактических результатов и ожидаемых результатов соответственно. В дальнейшем ”IN”будет использоваться для передачи программного кода на выполнение, а “OUT””и “OUT_EXP” для сравнения результатов;
2. Выводим строку “TEST: $1”, где “$1” — выполняемый программный код;
3. Записываем строку выполняемого программного кода в файл, путь к которому задан в переменной “IN”. Аналогично поступаем с ожидаемым результатом, но берём путь из переменной “OUT_EXP”;
4. Запускаем интерпретатор, перенаправляя в его поток вввода данные из файла “IN”, а также перенаправляем поток вывода в “OUT”;

```
TEST: (< 1 2)
OK

TEST: (< 3 2)
OK

TEST: (< 3 3)
OK

TEST: (progn (+ 2 3) (* 3 4))
OK

TEST: (defun test (x) (+ x 10) (* x 20)) (test 5)
OK

TEST: (setq a 10) (A)A
OK

TEST: (setq  a 10) A
OK

TEST: T NIL
OK

TEST: (setq +bgr-index+ 0x1ce) (setq +bgr-data+ 0x1cf) +BGR-INDEX+ +BGR-DATA+ X ABC
OK
```

Рисунок 4.4 – Вывод в консоль результатов выполнения системных тестов

5. Сравниваем данные из файлов “OUT” и “OUT_EXP”, используя встроенную в систему утилиту “diff” [23]. Если файлы идентичны, выводим “OK” – успешное завершение теста, иначе “FAIL” – несоответствие ожидаемого вывода фактическому.

6. Выводим пустую строку, создав тем самым перенос строки для визуального разделения результатов тестов.

4.5 Сборка программной системы

Для компиляции созданной ПС были разработаны два файла сборки “makefile” [7], реализующие разные варианты сборки интерпретатора.

Первый, расположенный в корневой папке разработанного интерпретатора, компилирует все модули интерпретатора в один готовый к использованию исполняемый файл. Для запуска сборки используется команда “make build”. В результате в той же папке будет сформирован файл “cl-inter”. Теперь

можно запустить интерпретатор с нужным файлом исходного кода программы, передав путь до него в качестве параметра. Например: “cl-inter ./script”.

Второй, “test”, предназначен для запуска модульных и системных тестов, при этом, в случае модульного тестирования, в скомпилированный файл входят только необходимые для тестируемого модуля компоненты. Расположен в папке test, где также находятся все файлы тестов. При вызове “make test” поочерёдно будут собираться и выполняться все разработанные модульные и системный тесты. Исполняемые файлы при этом будут помещены в директорию ОС для временных файлов - “/tmp” [24].

ЗАКЛЮЧЕНИЕ

В процессе выполнения данной работы была создана программная система, позволяющая сокращение размера исходного кода программ за счёт метапрограммирования.

Программная система предлагает к использованию возможности функциональной парадигмы и метапрограммирования для повышения скорости разработки, уменьшения объема кода, упрощения масштабирования и обеспечения большей мобильности разрабатываемого ПО.

Основные результаты работы:

1. Проведён анализ предметной области;
2. Спроектирован функциональный язык программирования с поддержкой метапрограммирования;
3. Спроектирован интерпретатор этого языка;
4. Выбраны технологии и методики для реализации интерпретатора;
5. Интерпретатор реализован средствами языка программирования “C” и ОС “GNU/Linux”.

Все требования, объявленные в техническом задании, были полностью реализованы, все задачи, поставленные в начале разработки проекта, были также решены.

Разработанная программная система была успешно использована для сокращения исходного кода существующей программы путём её переписывания на разработанный язык с применением возможностей метапрограммирования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Пратт Т., Зелковиц М. Языки программирования: разработка и реализация / Под общей ред. А. Матросова. – СПб.: Питер, 2002. – 688 с.: ил. – ISBN 5–318–00189–0. – Текст : непосредственный.

2. Ахо, Альфред В., Лам, Моника С., Сети, Рави, Ульман, Джеффри Д. Компиляторы: принципы, технологии и инструментарий, 2 е изд . : Пер . с англ. – М. : ООО “И.Д. Вильямс”, 2018 – 1184 с. : ил. – ISBN 978–5–8459–1932–8 – Текст : непосредственный.

3. Свердлов С. З. Конструирование компиляторов. Учебное пособие // LAP Lambert Academic Publishing, 2015 – 571 стр., ил. – ISBN 978–3–659–71665–2. – Текст : непосредственный.

4. Хэзфилд Ричард, Кирби Лоуренс и др. Искусство программирования на С. Фундаментальные алгоритмы, структуры данных и примеры приложений. Энциклопедия программиста: Пер. с англ./Ричард Хэзфилд, Лоуренс Кирби и др. –К.: Издательство «ДиаСофт», 2001. – 736 с. – ISBN 966–7393–82–8. – Текст : непосредственный.

5. Эмерик Ч., Карпер Б., Гранд К. Программирование на Clojure: Пер. с англ. Киселева А. Н. – М.: ДМК Пресс, 2015. – 816 с.: ил. – ISBN 978-5-97060-299-7. – Текст : непосредственный.

6. Керниган Б., Ритчи Д. Язык программирования Си. \Пер. с англ., 3-е изд., испр. – СПб.: ”Невский Диалект 2001 – 352 с: ил. – ISBN5-7940-0045-7. – Текст : непосредственный.

7. Клеменс, Бен. Язык С в XXI веке: Бен Клеменс; пер. с англ. А. А. Слинкина. – Москва : ДМК Пресс, 2015. - 376 с. : ил. – ISBN 978-5-97060-101-3. – Текст : непосредственный.

8. Норманд Эрик. Грокаем функциональное мышление. – СПб.: Питер, 2023. – 608 с.: ил. – ISBN 978-5-4461-1887-8. – Текст : непосредственный.

9. Сайбель П. Практическое использование Common Lisp / пер. с англ. А.Я. Отта. – М.:ДМК Пресс, 2015. – 488 с.: ил. – ISBN 978–5–94074–627–0. – Текст : непосредственный.

10. Фаулер, Мартин. Предметно-ориентированные языки программирования. : Пер. с англ. – М. : ООО ”И.Д. Вильямс 2011. – 576 с. : ил. – ISBN 978-5-8459-1738-6. – Текст : непосредственный.

11. Структура и интерпретация компьютерных программ / Харольд Абельсон, Джеральд Джей Сассман, при участии Джули Сассман ; [пер. Г. К. Бронникова]. – 2–е изд. – Москва : Добросвет : КДУ, 2012. – 608 с. : ил. – ISBN 978–5–98227–829–6. – Текст : непосредственный.

12. Лав Р. Linux. Системное программирование. 2–е изд. – СПб.: Питер, 2014. – 448 с.: ил. – ISBN 978–5–496–00747–4. – Текст : непосредственный.

13. Хориков В. Принципы юнит-тестирования. – СПб.: Питер, 2021. – 320 с.: ил. – ISBN 978-5-4461-1683-6. – Текст : непосредственный.

14. Пероцкая, В. Н. Основы тестирования программного обеспечения: учеб. пособие. – Владимир: Изд-во ВлГУ, 2017. – 100 с. – ISBN 978-5-9984-0777-2. – Текст : непосредственный.

15. Меджедович Д., Тахирович Э. Алгоритмы и структуры для массивных наборов данных / пер. с англ. А. В. Логунова. – М.: ДМК Пресс, 2024. – 340 с.: ил. – ISBN 978-5-93700-250-1. – Текст : непосредственный.

16. Искусство программирования, том 1. Основные алгоритмы, 3-е изд. : Пер. с англ. – М. : ООО ”И.Д. Вильямс 2018. - 720с. : ил. – ISBN 978-5-8459-1984-7. – Текст : непосредственный.

17. Прата С. Язык программирования C++. Лекции и упражнения, 6-е изд. : Пер. с англ. – М. : ООО ”И.Д. Вильямс 2012. – 1248 с. : ил. – ISBN 978-5-8459-1 778-2. – Текст : непосредственный.

18. Карпов В.Э. Теория компиляторов. Учебное пособие. 2-е изд., испр. и дополн. М., 2018. – 92 с. – ISBN 5–230–16344–5. – Текст : непосредственный.

19. Столяров А. В. Оформление программного кода: методическое пособие. – М.: МАКС Пресс, 2012. – 100 с. – ISBN 978-5-317-04282-0. – Текст : непосредственный.

20. Бурмашева, Н. В. Лингвистические основы языка программирования С : учебное пособие / Н. В. Бурмашева ; Министерство науки и высшего

образования Российской Федерации, Уральский федеральный университет. – Екатеринбург : Изд-во Урал. ун-та, 2023. – 86 с. : ил. – 30 экз. – ISBN 978-5-7996-3680-7. – Текст : непосредственный.

21. Грэм П. ANSI Common Lisp. – Пер. с англ. – СПб.: Символ-Плюс, 2012. – 448 с., ил. – ISBN 978-5-93286-206-3. – Текст : непосредственный.

22. Свердлов С. З. Языки программирования и методы трансляции: Учебное пособие. – 2-е изд., испр. — СПб.: Издательство «Лань», 2019. — 564 с.: ил. – ISBN 978-5-8114-3457-2. – Текст : непосредственный.

23. Bash. Карманный справочник системного администратора, 2-е и зд.: Пер. с англ. – СпБ .: ООО “Альфа-книга”, 2017. — 152 с . : ил. – ISBN 978-5-9909445-4-1. – Текст : непосредственный.

24. Командная строка Linux и автоматизация рутинных задач. – СПб.: БХВ-Петербург, 2012. – 352 с.: ил. – ISBN 978-5-9775-0850-6. – Текст : непосредственный.

25. Буч Г., Рамбо Д., Якобсон И. Язык UML. Руководство пользователя. 2-е изд.: Пер. с англ. Мухин Н. – М.: ДМК Пресс. – 496 с.: ил. – ISBN 5-94074-334-X. – Текст : непосредственный.

26. Языки и методы программирования: учебник для студентов учреждений высшего образования, обучающихся по направлениям подготовки ”Прикладная математика и информатика ”Фундаментальная информатика и информационные технологии”/ И. Г. Головин, И. А. Волкова. - 3-е изд., стер. – Москва : Академия, 2018. - 303 с. : ил. – ISBN 978-5-4468-6833-9. – Текст : непосредственный.

27. Гунько А.В. Системное программирование в среде Linux: учебное пособие / А.В. Гунько. – Новосибирск: Изд-во НГТУ, 2020 – 235 с. – ISBN 978-5-7782-4160-2. – Текст : непосредственный.

28. Функциональное программирование на F#: учебное пособие для студентов технических вузов / Сошников Д. В. – Москва : ДМК Пресс, 2011. – 189 с. : ил. – ISBN 978-5-94074-689-8. – Текст : непосредственный.

29. Сенлорен С., Эйзенберг Д. Введение в Elixir: введение в функциональное программирование / пер. с анг. А. Н. Киселева – М.: ДМК Пресс, 2017. – 262 с.: ил. – ISBN 978-5-97060-518-9. – Текст : непосредственный.

30. Леоненков А. В. Самоучитель UML 2. – СПб.: БХВ-Петербург, 2007. – 576 с.: ил. – ISBN 978-5-94157-878-8. – Текст : непосредственный.

ПРИЛОЖЕНИЕ А

Фрагменты исходного кода программы

lexer.c

```
#include <stdio.h>
#include <ctype.h>
#include <limits.h>
#include "lexer.h"

char cur_symbol;
token_t token;
int token_error;
int boot_load = 0;
char *boot_code;
#define SYMBOL_BUFFER_SIZE 8
char symbol_buffer[SYMBOL_BUFFER_SIZE];
int buffer_write_pos = 0;
int buffer_read_pos = 0;

void error(char *str, ...);

void get_cur_char()
{
    if (buffer_write_pos == buffer_read_pos) {
  if (!boot_load)
      cur_symbol = getchar();
  else
      cur_symbol = *boot_code++;
  symbol_buffer[buffer_write_pos++] = cur_symbol;
  buffer_read_pos = buffer_write_pos &= SYMBOL_BUFFER_SIZE - 1;
    } else {
  cur_symbol = symbol_buffer[buffer_read_pos++];
  buffer_read_pos &= SYMBOL_BUFFER_SIZE - 1;
    }
}

void unget_cur_char()
{
    cur_symbol = symbol_buffer[(buffer_read_pos - 2) & SYMBOL_BUFFER_SIZE -
        1];
    --buffer_read_pos;
    buffer_read_pos &= SYMBOL_BUFFER_SIZE - 1;
}

int is_whitespace(char c)
{
    return c == ' ' || c == '\n' || c == '\r' || c == '\t';
}

void skip_comment()
{
    while (cur_symbol != '\n' && cur_symbol != EOF)
  get_cur_char();
```

```
49 }
50
51 void skip_white_space()
52 {
53     while (is_whitespace(cur_symbol) || cur_symbol == ';') {
54   if (cur_symbol == ';')
55       skip_comment();
56   get_cur_char();
57     }
58     unget_cur_char();
59 }
60
61 int is_digit(char c)
62 {
63     return c >= '0' && c <= '9';
64 }
65
66 int is_alpha(char c)
67 {
68     return c >= 'a' && c <= 'z' || c>= 'A' && c <= 'Z';
69 }
70
71 int is_symbol(char c)
72 {
73     char str[] = "+-*/=_&|<>";
74     for (int i = 0; i < sizeof(str); i++)
75   if (str[i] == c)
76       return 1;
77     return 0;
78 }
79
80 int is_hex_symbol(char c)
81 {
82     return c >= 'a' && c <= 'f' ||
83            c >= 'A' && c <= 'F';
84 }
85
86 int is_delimeter(char c)
87 {
88     return c == ')' || c == '(' || is_whitespace(c) || c == EOF || c == '"'
        || c == '\\';
89 }
90
91 int hex_num()
92 {
93     long long cur_num =  0;
94     int hex_value;
95     const int msb_shr = CHAR_BIT * sizeof(int);
96
97     do {
98     get_cur_char();
99     if (is_delimeter(cur_symbol))
100       break;
101     if (is_digit(cur_symbol)) {
```

```
102       cur_num = cur_num * 16 + cur_symbol - '0';
103     } else if (is_hex_symbol(cur_symbol)) {
104       cur_symbol = toupper(cur_symbol);
105       cur_num = cur_num * 16 + cur_symbol - 'A' + 10;
106     } else {
107       token_error = 1;
108       printf("invalid hex num\n");
109       return 0;
110     }
111     if ((cur_num >> msb_shr) & 1) {
112       token_error = 1;
113       printf("hex number overflow\n");
114       return 0;
115     }
116     } while (is_digit(cur_symbol) || is_hex_symbol(cur_symbol));
117 
118     unget_cur_char();
119     return cur_num;
120 }
121 
122 int get_float_num(int int_num)
123 {
124     int cnt = 0;
125     float float_num = 0;
126     int floatbits = 0;
127     int num = int_num;
128     int temp;
129     get_cur_char();
130     float count = 1;
131     while(is_digit(cur_symbol)) {
132   float_num = float_num * 10 + (cur_symbol - '0');
133   count *= 10;
134   get_cur_char();
135     }
136     float res_float = int_num > 0 ?  int_num + (float_num / count) : int_num
          - (float_num / count);
137     printf("res_float is equal %f\n", res_float);
138     return *(int *)&res_float;
139 }
140 
141 int get_num()
142 {
143     int fl = 0;
144     int cur_num = 0;
145     if (cur_symbol == '0') {
146   get_cur_char();
147   if (cur_symbol == 'x')
148       return hex_num();
149     } else if (cur_symbol == '-') {
150         fl = 1;
151         get_cur_char();
152     }
153     const int sgn_shr = CHAR_BIT * sizeof(int) - 1;
154     int sgn = fl ? -1 : 1;
```

```
155     int msb;
156     while (is_alpha(cur_symbol) || is_digit(cur_symbol) || is_symbol(
            cur_symbol) || cur_symbol == '.')  {
157   if (is_digit(cur_symbol)) {
158       cur_num = cur_num * 10 + sgn * (cur_symbol - '0');
159       msb = (cur_num >> sgn_shr) & 1;
160       if (msb != fl) {
161     token_error = 1;
162     printf("number overflow\n");
163     return 0;
164       }
165       get_cur_char();
166   } else if (cur_symbol == '.') {
167       token.type = T_FLOAT;
168       return get_float_num(cur_num);
169   } else {
170       token_error = 1;
171       printf("invalid num\n");
172       return 0;
173   }
174     }
175     unget_cur_char();
176     return cur_num;
177 }
178
179 void get_symbol(char *cur_str)
180 {
181     int c = 0;
182     while (!is_delimeter(cur_symbol))
183     {
184   if (cur_symbol == '\b') {
185       cur_str[--c] = ' ';
186       get_cur_char();
187       continue;
188   }
189   if (!(is_alpha(cur_symbol) || is_digit(cur_symbol) || is_symbol(cur_symbol)
        )){
190       printf("ERROR: lexer.c: Unsupported character in input: %c(#%x)",
              cur_symbol, cur_symbol);
191       token_error = 1;
192       return;
193   }
194         cur_str[c++] = cur_symbol;
195         get_cur_char();
196   if (c > MAX_SYMBOL) {
197       printf("ERROR: lexer.c: MAX_SYMBOL");
198       token_error = 1;
199       return;
200   }
201     }
202     unget_cur_char();
203     cur_str[c] = 0;
204 }
205
```

```
206 void get_string(char *cur_str)
207 {
208     int c = 0;
209 
210     get_cur_char();
211     while (cur_symbol != EOF) {
212   if (cur_symbol == '"')
213         break;
214   else if (c == MAX_STR) {
215     token_error = 1;
216     --c;
217     break;
218   }
219       else if (cur_symbol == '\\') {
220         get_cur_char();
221         if (cur_symbol == 'n')
222       cur_symbol = '\n';
223     else if (cur_symbol == 'x')
224     {
225         int code = hex_num();
226         if (code > 255)
227         {
228       printf("get_string: invalid symbol code\n");
229       token_error = 1;
230       return;
231         }
232         cur_str[c++] = code;
233         get_cur_char();
234         continue;
235     }
236   }
237   cur_str[c++] = cur_symbol;
238   get_cur_char();
239     }
240     if (cur_symbol != '"')
241   token_error = 1;
242     cur_str[c] = '\0';
243 }
244 
245 token_t get_comma()
246 {
247     char ctr[2];
248     ctr[0] = cur_symbol;
249     get_cur_char();
250     ctr[1] = cur_symbol;
251     if (ctr[0] == ',' && ctr[1] == '@') {
252         token.type = COMMA_AT;
253     }
254     else {
255         token.type = COMMA;
256         unget_cur_char();
257     }
258 }
259 
```

```
260 token_t get_sharp()
261 {
262     char ctr[2];
263     ctr[0] = cur_symbol;
264     get_cur_char();
265     ctr[1] = cur_symbol;
266     if (ctr[0] == '#' && ctr[1] == '\\') {
267   token.type = T_CHAR;
268   get_cur_char();
269   token.value = cur_symbol;
270     } else {
271   token.type = SHARP;
272   unget_cur_char();
273     }
274 }
275
276 token_t *get_token()
277 {
278     token_error  = 0;
279     get_cur_char();
280     skip_white_space();
281     get_cur_char();
282     switch (cur_symbol) {
283     case '(':
284   token.type = LPAREN;
285   break;
286     case ')':
287   token.type = RPAREN;
288   break;
289     case EOF:
290   token.type = END;
291   break;
292     case '\'':
293   token.type = QUOTE;
294   break;
295     case '`':
296   token.type = BACKQUOTE;
297   break;
298     case ',':
299   get_comma();
300   break;
301     case '"':
302   get_string(token.str);
303   token.type = T_STRING;
304   break;
305     case '#':
306   get_sharp();
307   break;
308     case '.':
309   token.type = DOT;
310   break;
311     default:
312   if (cur_symbol == '-' || is_digit(cur_symbol)) {
313       if (cur_symbol == '-') {
```

```
314             get_cur_char();
315       char c = cur_symbol;
316       unget_cur_char();
317             if (!is_digit(c))
318                 goto get_token_symbol;
319         }
320         token.type = T_NUMBER;
321         token.value = get_num();
322     } else if (is_alpha(cur_symbol) || is_symbol(cur_symbol)) {
323         get_token_symbol:
324             token.type =  T_SYMBOL;
325             get_symbol(token.str);
326     } else {
327         token.type = INVALID;
328         error("ERROR: lexer.c: INVALID SYMBOL");
329     }
330       }
331       return &token;
332 }
333
334 void print_token(token_t *token)
335 {
336       switch (token->type) {
337       case T_NUMBER:
338     printf("NUM %d\n", token->value);
339     break;
340       case T_SYMBOL:
341     printf("SYM %s\n", token->str);
342     break;
343       case LPAREN:
344     printf("LPAREN\n");
345     break;
346       case RPAREN:
347     printf("RPAREN\n");
348     break;
349       case END:
350     printf("END\n");
351     break;
352       case QUOTE:
353     printf("QUOTE\n");
354     break;
355       case BACKQUOTE:
356     printf("BACKQUOTE\n");
357     break;
358       case COMMA:
359     printf("COMMA\n");
360     break;
361       case COMMA_AT:
362     printf("COMMA_AT\n");
363     break;
364       case INVALID:
365     printf("INVALID\n");
366     break;
367       case T_STRING:
```

```
368   printf("STRING %s\n", token->str);
369   break;
370     case SHARP:
371   printf("SHARP\n");
372   break;
373     case DOT:
374   printf("DOT\n");
375     }
376 }
377
378 void reset_buffer()
379 {
380     buffer_read_pos = buffer_write_pos = 0;
381 }
```

parser.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "lexer.h"
5  #include "objects.h"
6  #include "symbols.h"
7  #include "parser.h"
8
9  extern int token_error;
10
11 token_t *cur_token;
12
13 char *strupr(char *str)
14 {
15     char *ptr = str;
16
17     while (*ptr != '\0') {
18         if (*ptr >= 'a' && *ptr <= 'z') {
19             *ptr = *ptr - 'a' + 'A';
20         }
21         ptr++;
22     }
23     return str;
24 }
25
26 object_t parse();
27 object_t parse_list();
28 object_t parse_array();
29
30 object_t parse_quote(char *quote_sym)
31 {
32     object_t o = parse();
33     if (o == NOVALUE)
34   error("quote: no args");
35     object_t p = new_pair(o, NULLOBJ);
36     return new_pair(NEW_SYMBOL(quote_sym), p);
37 }
38
```

```
object_t parse_element(type_t type, void *data, tokentype_t t_type)
{
    object_t obj;
    if (t_type == QUOTE)
  obj = parse_quote("QUOTE");
    else if (t_type == SHARP)
  obj = parse_array();
    else if (t_type == BACKQUOTE)
  obj = parse_quote("BACKQUOTE");
    else if (t_type == COMMA)
  obj = parse_quote("COMMA");
    else if (t_type == COMMA_AT)
  obj = parse_quote("COMMA-AT");
    else if (t_type == LPAREN)
  obj = parse_list();
    else if (t_type == T_NUMBER)
  obj = new_number(*(int *)data);
    else if (t_type == T_SYMBOL)
  obj = NEW_OBJECT(SYMBOL, find_symbol(data));
    else if (t_type == T_STRING)
  obj = NEW_STRING((char *)data);
    else
  obj = NEW_OBJECT(type, data);
    object_t tail = parse_list();
    return new_pair(obj, tail);
}

object_t parse_list()
{
    int val;
    char str[MAX_STR];
    token_t *cur_tok = get_token();
    if (token_error == 1)
        error("parse: token_error");
    if (cur_tok->type == END)
  error("expected )");
    if (cur_tok->type == RPAREN)
  return NULLOBJ;
    if (cur_tok->type == T_NUMBER) {
        val = cur_tok->value;
  return parse_element(NUMBER, &val, cur_tok->type);
    } else if (cur_tok->type == T_STRING) {
  strcpy(str, cur_tok->str);
  return parse_element(STRING, str, cur_tok->type);
    } else if (cur_tok->type == T_SYMBOL) {
        strcpy(str, cur_tok->str);
  return parse_element(SYMBOL, strupr(str), cur_tok->type);
    } else if (cur_tok->type == LPAREN || cur_tok->type == QUOTE
         || cur_tok->type == BACKQUOTE || cur_tok->type == COMMA
         || cur_tok->type == COMMA_AT || cur_tok->type == SHARP)
  return parse_element(SYMBOL, NULL, cur_tok->type);
    else if (cur_tok->type == DOT) {
  object_t res = parse();
  cur_tok = get_token();
```

```
93    if (cur_tok->type != RPAREN)
94        error("expected )");
95    return res;
96      } else if (cur_token->type == INVALID)
97          error("parse: invalid token");
98      else
99    error("invalid expression");
100 }
101
102 object_t parse_array()
103 {
104     object_t o = parse();
105     if (o != NULLOBJ && TYPE(o) != PAIR)
106   error("invalid array");
107     return NEW_ARRAY(o);
108 }
109
110 object_t parse()
111 {
112     object_t el;
113     token_t *cur_token = get_token();
114     if (token_error == 1)
115         error("parse: token_error");
116     if (cur_token->type == T_NUMBER)
117   return new_number(cur_token->value);
118     else if (cur_token->type == T_SYMBOL)
119   return NEW_OBJECT(SYMBOL, find_symbol(strupr(cur_token->str)));
120     else if (cur_token->type == LPAREN)
121   return parse_list();
122     else if (cur_token->type == QUOTE)
123   return parse_quote("QUOTE");
124     else if (cur_token->type == BACKQUOTE)
125   return parse_quote("BACKQUOTE");
126     else if (cur_token->type == COMMA)
127   return parse_quote("COMMA");
128     else if (cur_token->type == SHARP)
129   return parse_array();
130     else if (cur_token->type == T_STRING)
131   return NEW_STRING(cur_token->str);
132     else if (cur_token->type == END)
133         return NOVALUE;
134     else if (cur_token->type == INVALID)
135         error("parse: invalid token");
136     else
137   error("invalid expression");
138 }
```

eval.c

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <setjmp.h>
4 #include "objects.h"
5 #include "symbols.h"
6 #include "parser.h"
```

```
#include "eval.h"
#include "arith.h"

object_t t;
object_t nil;
symbol_t *quote_sym;
symbol_t *backquote_sym;
symbol_t *lambda_sym;
symbol_t *cond_sym;
symbol_t *defun_sym;
symbol_t *defmacro_sym;
symbol_t *setq_sym;
symbol_t *or_sym;
symbol_t *and_sym;
symbol_t *t_sym;
symbol_t *nil_sym;
symbol_t *rest_sym;
symbol_t *tagbody_sym;
symbol_t *block_sym;
symbol_t *return_from_sym;
symbol_t *labels_sym;
symbol_t *progn_sym;
object_t current_env = NULLOBJ;
jmp_buf repl_buf;
jmp_buf block_buf;

object_t eq(object_t list)
{
    if (list == NULLOBJ)
  error("eq: no args");
    if (TAIL(list) == NULLOBJ)
  error("eq: one arg");
    if (TAIL(TAIL(list)) != NULLOBJ)
  error("eq: too many args");
    object_t p1 = FIRST(list);
    object_t p2 = SECOND(list);
    if (p1 == p2)
  return t;
    else
  return nil;
}

object_t atom(object_t list)
{
    if (list == NULLOBJ)
  error("atom: no args");
    else if (TAIL(list) != NULLOBJ)
  error("atom: many args");
    object_t obj = FIRST(list);
    if (obj == NULLOBJ || TYPE(obj) != PAIR)
  return t;
    else
  return nil;
}
```

```
61
62 object_t quote(object_t list)
63 {
64     if (TAIL(list) != NULLOBJ)
65   error("quote: many args");
66     return FIRST(list);
67 }
68
69 void append_env(object_t l1, object_t l2);
70
71 object_t backquote_rec(object_t list)
72 {
73     if (list == NULLOBJ)
74   return NULLOBJ;
75     object_t o;
76     if (TYPE(list) == NUMBER || TYPE(list) == BIGNUMBER)
77   return new_number(get_value(list));
78     else if (TYPE(list) == SYMBOL)
79   return NEW_SYMBOL(GET_SYMBOL(list)->str);
80     else if (TYPE(list) == ARRAY) {
81   array_t *arr = new_empty_array(GET_ARRAY(list)->length);
82   int l = GET_ARRAY(list)->length;
83   for (int i = 0; i < l; i++)
84       arr->data[i] = backquote_rec(GET_ARRAY(list)->data[i]);
85   return NEW_OBJECT(ARRAY, arr);
86     } else if (TYPE(list) == STRING)
87   return NEW_STRING(GET_STRING(list)->data);
88     else if (TYPE(list) == PAIR) {
89   object_t el = FIRST(list);
90   if (el == NULLOBJ)
91       return new_pair(NULLOBJ, backquote_rec(TAIL(list)));
92   if (TYPE(el) == SYMBOL && !strcmp(GET_SYMBOL(el)->str, "BACKQUOTE"))
93       return list;
94   if (TYPE(el) == SYMBOL && !strcmp(GET_SYMBOL(el)->str, "COMMA"))
95       return eval(SECOND(list), current_env);
96   object_t first = backquote_rec(el);
97   if (first != NULLOBJ && TYPE(first) == PAIR) {
98       object_t comma_at = FIRST(first);
99       if (comma_at != NULLOBJ && TYPE(comma_at) == SYMBOL && !strcmp(
          GET_SYMBOL(comma_at)->str, "COMMA-AT")) {
100     object_t l = eval(SECOND(first), current_env);
101     if (l == NULLOBJ)
102         return backquote_rec(TAIL(list));
103     if (TYPE(l) != PAIR)
104         error("COMMA-AT: not list");
105     object_t new_comma = backquote_rec(l);
106     append_env(new_comma, backquote_rec(TAIL(list)));
107     return new_comma;
108       }
109   }
110   object_t tail = backquote_rec(TAIL(list));
111   return new_pair(first, tail);
112     }
113     error("backqoute: unknown type: %d\n", TYPE(list));
```

```
114 }
115
116 object_t backquote(object_t list)
117 {
118     if (list == NULLOBJ)
119   error("backquote: NULLOBJ");
120     return backquote_rec(FIRST(list));
121 }
122
123 object_t cond(object_t obj)
124 {
125     if (obj == NULLOBJ)
126   error("NULLOBJ in COND");
127     object_t pair = FIRST(obj);
128     if (TAIL(pair) == NULLOBJ)
129   error("cond: not enough params");
130     if (TAIL(TAIL(pair)) != NULLOBJ)
131   error("cond: too many params");
132     object_t p = FIRST(pair);
133     if (eval(p, current_env) == t)
134   return eval(SECOND(pair), current_env);
135     else
136   return cond(TAIL(obj));
137 }
138
139 object_t defun(object_t obj)
140 {
141     symbol_t *name = find_symbol(GET_SYMBOL(FIRST(obj))->str);
142     name->lambda = new_pair(NEW_SYMBOL("LAMBDA"), TAIL(obj));
143     return NEW_SYMBOL(name->str);
144 }
145
146 object_t defmacro(object_t obj)
147 {
148     symbol_t *name = find_symbol(GET_SYMBOL(FIRST(obj))->str);
149     name->macro = new_pair(NEW_SYMBOL("LAMBDA"), TAIL(obj));
150     return NEW_SYMBOL(name->str);
151 }
152
153 object_t progn(object_t params)
154 {
155     if (params == NULLOBJ)
156   error("progn: params = NULLOBJ");
157     object_t obj = eval(FIRST(params), current_env);
158     if (TAIL(params) == NULLOBJ)
159   return obj;
160     return progn(TAIL(params));
161 }
162
163 int check_params(object_t list)
164 {
165     if (list == NULLOBJ)
166   return 1;
167     if (TYPE(FIRST(list)) != SYMBOL)
```

```
168   return 0;
169     return check_params(TAIL(list));
170 }
171
172 int is_lambda(object_t list)
173 {
174     object_t lambda = FIRST(list);
175     if (TYPE(lambda) != SYMBOL || GET_SYMBOL(lambda) != lambda_sym)
176   error("Invalid lambda symbol");
177     if (TAIL(list) == NULLOBJ)
178   error("No params in lambda");
179     object_t params = SECOND(list);
180     if (params != NULLOBJ && TYPE(params) != PAIR)
181   error("Invalid params in lambda");
182     if (!check_params(params))
183   error("Not symbol in lambda attrs");
184     if (TAIL(TAIL(list)) == NULLOBJ)
185   error("No body in lambda");
186     return 1;
187 }
188
189 object_t make_env(object_t args, object_t values)
190 {
191     if (args != NULLOBJ && values == NULLOBJ && GET_SYMBOL(FIRST(args)) !=
          rest_sym)
192   error("Not enough values for params");
193     if (args == NULLOBJ && values != NULLOBJ)
194   error("Invalid number of arguments");
195     if (args == NULLOBJ)
196   return NULLOBJ;
197     object_t param = FIRST(args);
198     if (GET_SYMBOL(param) == rest_sym) {
199   if (TAIL(args) == NULLOBJ)
200       error("Missing parameter after &rest");
201   if (TAIL(TAIL(args)) != NULLOBJ)
202       error("Too many parameters after &rest");
203   return new_pair(new_pair(SECOND(args), values), nil);
204     }
205     object_t val = FIRST(values);
206     object_t pair = new_pair(param, val);
207     object_t new_env = make_env(TAIL(args), TAIL(values));
208     return new_pair(pair, new_env);
209 }
210
211 int find_in_env(object_t env, object_t sym, object_t *res)
212 {
213     if (env == NULLOBJ)
214   return 0;
215     object_t pair = FIRST(env);
216     object_t var = FIRST(pair);
217     if (GET_SYMBOL(var)  == GET_SYMBOL(sym)){
218   *res = TAIL(pair);
219   return 1;
220     } else
```

```
221   return find_in_env(TAIL(env), sym, res);
222 }
223
224 void append_env(object_t l1, object_t l2)
225 {
226     while (GET_PAIR(l1)->right != NULLOBJ)
227   l1 = GET_PAIR(l1)->right;
228     GET_PAIR(l1)->right = l2;
229 }
230
231 object_t eval_func(object_t lambda, object_t args, object_t env)
232 {
233     object_t new_env = make_env(SECOND(lambda), args);
234     object_t body;
235     if (GET_PAIR(TAIL(TAIL(lambda)))->right == NULLOBJ)
236   body = THIRD(lambda);
237     else
238   body = new_pair(NEW_SYMBOL("PROGN"), TAIL(TAIL(lambda)));
239     if (new_env == NULLOBJ)
240   new_env = env;
241     else
242   append_env(new_env, env);
243     return eval(body, new_env);
244 }
245
246 object_t macro_call(object_t macro, object_t args, object_t env)
247 {
248     object_t new_env = make_env(SECOND(macro), args);
249     object_t body;
250     object_t eval_res;
251     body = TAIL(TAIL(macro));
252     append_env(new_env, env);
253     while (body != NULLOBJ) {
254   eval_res = eval(FIRST(body), new_env);
255   eval_res = eval(eval_res, new_env);
256   body = TAIL(body);
257     }
258     return eval_res;
259 }
260
261 object_t eval_args(object_t args, object_t env)
262 {
263     if (args == NULLOBJ)
264   return NULLOBJ;
265     object_t f = FIRST(args);
266     object_t arg = eval(f, env);
267     object_t tail = eval_args(TAIL(args), env);
268     return new_pair(arg, tail);
269 }
270
271 int is_special_form(symbol_t *s)
272 {
273     return s == quote_sym || s == defun_sym || s == defmacro_sym
274   || s == setq_sym || s == backquote_sym || s == cond_sym
```

```
275   || s == or_sym || s == and_sym || s == return_from_sym
276   || s == labels_sym || s == tagbody_sym || s == progn_sym;
277 }
278
279 object_t eval_symbol(object_t obj)
280 {
281     object_t res;
282
283     if (find_in_env(current_env, obj, &res))
284   return res;
285     else {
286   symbol_t *res_sym = check_symbol(GET_SYMBOL(obj)->str);
287   if (res_sym != NULL && res_sym->value != NOVALUE)
288             return res_sym->value;
289   else
290             error("Unknown SYMBOL: %s", GET_SYMBOL(obj)->str);
291     }
292 }
293
294 object_t eval(object_t obj, object_t env)
295 {
296     current_env = env;
297     if (obj == NULLOBJ)
298         return NULLOBJ;
299     else if (TYPE(obj) == NUMBER || TYPE(obj) == BIGNUMBER || TYPE(obj) ==
        STRING || TYPE(obj) == ARRAY)
300   return obj;
301     else if (TYPE(obj) == SYMBOL)
302         return eval_symbol(obj);
303     else if (TYPE(obj) == PAIR) {
304   object_t first = FIRST(obj);
305   if (TYPE(first) == PAIR) {
306       is_lambda(first);
307       return eval_func(first, eval_args(TAIL(obj), env), env);
308   }
309   symbol_t *s = find_symbol(GET_SYMBOL(first)->str);
310   object_t args;
311   if (is_special_form(s) || s->macro != NULLOBJ)
312       args = TAIL(obj);
313   else
314       args = eval_args(TAIL(obj), env);
315   if (s->lambda != NULLOBJ)
316       return eval_func(s->lambda, args, env);
317   else if (s->func != NULL)
318       return s->func(args);
319   else if (s->macro != NULLOBJ)
320       return macro_call(s->macro, args, env);
321   else
322       error("Unknown func: %s", s->str);
323     } else
324         error("Unknown object_type");
325     current_env = env;
326 }
327
```

```
328 void set_in_env(object_t env, object_t sym, object_t val)
329 {
330     if (env == NULLOBJ)
331   error("ERROR: NULLOBJ as env in set_in_env");
332     object_t pair = FIRST(env);
333     object_t var = FIRST(pair);
334     if (GET_SYMBOL(var) == GET_SYMBOL(sym))
335   TAIL(pair) = val;
336     else
337   set_in_env(TAIL(env), sym, val);
338 }
339
340 object_t setq_rec(object_t params)
341 {
342     if (params == NULLOBJ)
343   return NULLOBJ;
344     symbol_t *sym = GET_SYMBOL(FIRST(params));
345     object_t res;
346     int find_res = find_in_env(current_env, FIRST(params), &res);
347     if (!find_res)
348   sym = find_symbol(GET_SYMBOL(FIRST(params))->str);
349     if (TAIL(params) == NULLOBJ)
350   error("setq: no value");
351     object_t obj = eval(SECOND(params), current_env);
352     if (find_res)
353   set_in_env(current_env, FIRST(params), obj);
354     else
355   sym->value = obj;
356     if (TAIL(TAIL(params)) == NULLOBJ)
357   return obj;
358     return setq_rec(TAIL(TAIL(params)));
359 }
360
361 object_t setq(object_t params)
362 {
363     if (params == NULLOBJ)
364   error("setq: params = NULLOBJ");
365     return setq_rec(params);
366 }
367
368 object_t and(object_t params)
369 {
370     if (params == NULLOBJ)
371   error("and: no params");
372     while (params != NULLOBJ) {
373   object_t first = FIRST(params);
374   object_t res = eval(first, current_env);
375   if (res == nil)
376       return nil;
377   else if (res == t)
378       params = TAIL(params);
379   else
380       error("and: invalid param");
381     }
```

```
382     return t;
383 }
384
385 object_t or(object_t params)
386 {
387     if (params == NULLOBJ)
388   error("or: no params");
389     while (params != NULLOBJ) {
390   object_t first = FIRST(params);
391   object_t res = eval(first, current_env);
392   if (res == t)
393       return t;
394   else if (res == nil)
395       params = TAIL(params);
396   else
397       error("or: invalid param");
398     }
399     return nil;
400 }
401
402 object_t macroexpand(object_t params)
403 {
404     if (params == NULLOBJ)
405   error("macroexpand: no params");
406     if (TAIL(params) != NULLOBJ)
407   error("macroexpand: many params");
408     object_t macro_c = FIRST(params);
409     if (TYPE(macro_c) != PAIR)
410   error("macroexpand: invalid macro call");
411     object_t macro_name = FIRST(macro_c);
412     object_t macro;
413     if (macro_name == NULLOBJ || TYPE(macro_name) != SYMBOL || GET_SYMBOL(
          macro_name)->macro == NULLOBJ)
414   error("macroexpand: invalid macro");
415     macro = GET_SYMBOL(macro_name)->macro;
416     object_t args = TAIL(macro_c);
417     object_t new_env = make_env(SECOND(macro), args);
418     append_env(new_env, current_env);
419     object_t body = TAIL(TAIL(macro));
420     object_t eval_res;
421     object_t res = NULLOBJ;
422     while (body != NULLOBJ) {
423   eval_res = eval(FIRST(body), new_env);
424   if (res == NULLOBJ)
425       res = eval_res;
426   else if (TYPE(res) == STRING)
427       res = NULLOBJ;
428   else
429       append_env(res, eval_res);
430   body = TAIL(body);
431     }
432     return res;
433 }
434
```

```
435 object_t funcall(object_t params)
436 {
437     if (params == NULLOBJ)
438   error("funcall: no arguments");
439     object_t func = FIRST(params);
440     if (func == NULLOBJ || TYPE(func) != SYMBOL &&
441   !(TYPE(func) == PAIR && is_lambda(func) == 1))
442   error("funcall: invalid func");
443     object_t args = TAIL(params);
444     if (TYPE(func) == PAIR)
445   return eval_func(func, args, current_env);
446     symbol_t *s = find_symbol(GET_SYMBOL(func)->str);
447     if (s->lambda != NULLOBJ)
448   return eval_func(s->lambda, args, current_env);
449     else if (s->func != NULL)
450   return s->func(args);
451     else
452   error("Unknown func: %s", s->str);
453 }
454
455 object_t list(object_t args)
456 {
457     return args;
458 }
459
460 object_t lisp_eval(object_t args)
461 {
462     return eval(FIRST(args), current_env);
463 }
464
465 object_t error_func(object_t args)
466 {
467     printf("ERROR: ");
468     PRINT(FIRST(args));
469     longjmp(repl_buf, 1);
470 }
471
472 object_t tagbody(object_t params)
473 {
474     object_t obj;
475     object_t form;
476     while (params != NULLOBJ) {
477         obj = FIRST(params);
478         if (TYPE(obj) != SYMBOL)
479             form = eval(obj, current_env);
480   params = TAIL(params);
481     }
482     return form;
483 }
484
485 object_t block(object_t list)
486 {
487     object_t obj;
488     if (list == NULLOBJ)
```

```
489   error("block: no arguments\n");
490     if (setjmp(block_buf) == 0)
491         return progn(list);
492 }
493
494 object_t return_from(object_t arg)
495 {
496     return arg;
497 }
498
499 object_t labels(object_t param)
500 {
501     return param;
502 }
503
504 void init_eval()
505 {
506     register_func("ATOM", atom);
507     register_func("EQ", eq);
508     register_func("QUOTE", quote);
509     register_func("BACKQUOTE",backquote);
510     register_func("COND", cond);
511     register_func("DEFUN", defun);
512     register_func("DEFMACRO", defmacro);
513     register_func("PROGN", progn);
514     register_func("SETQ", setq);
515     register_func("OR", or);
516     register_func("AND", and);
517     register_func("MACROEXPAND", macroexpand);
518     register_func("FUNCALL", funcall);
519     register_func("LIST", list);
520     register_func("EVAL", lisp_eval);
521     register_func("GC", print_gc_stat);
522     register_func("ERROR", error_func);
523     register_func("TAGBODY", tagbody);
524     register_func("BLOCK", block);
525     register_func("RETURN_FROM", return_from);
526     register_func("BLOCK", block);
527     register_func("LABELS", labels);
528     t = NEW_SYMBOL("T");
529     nil = NULLOBJ;
530     quote_sym = find_symbol("QUOTE");
531     backquote_sym = find_symbol("BACKQUOTE");
532     lambda_sym = find_symbol("LAMBDA");
533     cond_sym = find_symbol("COND");
534     defun_sym = find_symbol("DEFUN");
535     defmacro_sym = find_symbol("DEFMACRO");
536     setq_sym = find_symbol("SETQ");
537     or_sym = find_symbol("OR");
538     and_sym = find_symbol("AND");
539     t_sym = GET_SYMBOL(t);
540     t_sym->value = t;
541     nil_sym = find_symbol("NIL");
542     nil_sym->value = nil;
```

```
543     rest_sym = find_symbol("&REST");
544     tagbody_sym = find_symbol("TAGBODY");
545     block_sym = find_symbol("BLOCK");
546     labels_sym = find_symbol("LABEL");
547     progn_sym = find_symbol("PROGN");
548 }
```

Место для диска