Минобрнауки России

Юго-Западный государственный университет

Кафедра программной инженерии

ОТЧЕТ

о преддипломной (производственной) практике

наименование вида и типа практики

на (в) Юго-Западном государственном университете

наименование предприятия, организации, учреждения

Студента 4 курса, группы ПО-02б

курса, группы

Антипова Дмитрия Александровича

фамилия, имя, отчество

Руководитель практики от предприятия, организации, учреждения

Оценка _____

_____

должность, звание, степень

_____ _____

фамилия и. о. подпись, дата

Руководитель практики от университета

Оценка _____

к.т.н. доцент

должность, звание, степень

Чаплыгин А. А. _____

фамилия и. о. подпись, дата

Члены комиссии

_____ _____

подпись, дата фамилия и. о.

_____ _____

подпись, дата фамилия и. о.

_____ _____

подпись, дата фамилия и. о.

Курск 2024 г.

СОДЕРЖАНИЕ

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

ИТ – информационные технологии.

ПО – программное обеспечение.

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.

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.

1.2 Строение интерпретатора

1.2.1 Лексический анализатор и лексема

Всё начинается с лексемы — 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.

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), 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.

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.

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. 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. 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, 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. 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 искуственного 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. 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, который будет называться демонстрационным языком программирования (ДЯП) в рамках этой работы. Основной задачей этой работы 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 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, числами, строками, символами, списками и массивами. В результате 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. 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 им анализатором

Список наименований лексем, распознаваемых им 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: ,@;
- решетка: #;
- 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.

2.3.3 Типы данных

Для языка программирования были определены пять фактических типа данных и два псевдотипа:

- Число – десятичные и шестнадцатеричные числа. Например: 10, 0xFFAA;

- 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. Пример: (x 2 ’p);

- Массив – хранилище с прямым доступом к элементам (можно обращаться по индексу), содержащее ноль или более атомов, разделенных пустотами и заключённых в круглые скобки, перед открывающей ставится #. Пример: #(3 6 9).

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

- NUMBER: 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.

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 для объявления 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.

Пример использования:

< (defvar b 12)

< (print ‘(+ a ,b))

> (+ A 12)

Таким образом, ’+’ и ’a’ были восприняты компилятором как символы и остались невычисленными, а ’b’ заменён значением соответствующей переменной.

Стоит отметить, что кавычка, обратная кавычка и запятая действуют на вычисление только того выражения, перед которым стоят.

2.4 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 конкурентноспособности 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, 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 семей-

ства 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. 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 семейство ОС – GNU/Linux. В этих системах применяется реализация ”GNU C Library”(glibc).

3.3 Компоненты интерпретатора

На рисунке 3.1 в виде UML-диаграммы показаны компоненты, составляющие интерпретатор.

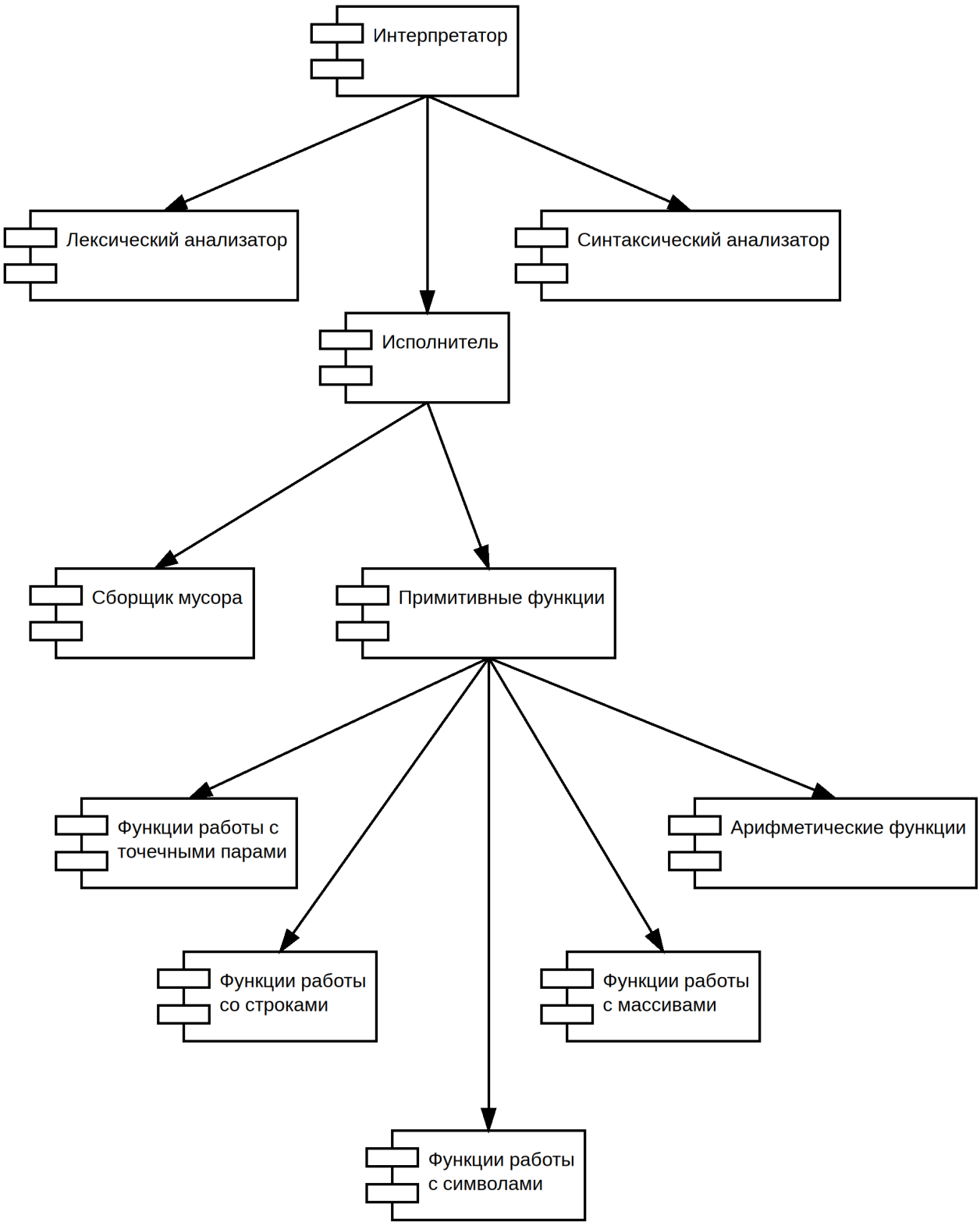

Рисунок 3.1 – Диаграмма компонентов интерпретатора

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.

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 символ не является одним из зарезервированных, производится проверка на то, является он числом или символом.

Ввиду того, что имя символа, как и отрицательное число, может начинаться со знака минус решается неоднозначность, связанная с восприятием лексическим анализатором 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, неоконченность 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 все данные хранятся в виде структур языка C. Всего было разработано пять структур. Структура верхнего уровня object_t, которая далее будет называться оболочкой объекта, является результатом выполнения любого s-выражения и доступ из примитивных функций к другим структурам осуществляется через поля этой структуры. Оставшиеся структуры реализуют четыре типа данных, за исключением числового: пара, символ, строка, массив.

Оболочка объекта позволяет узнать тип хранимых объектом данных, чтобы другие функции могли верно его обработать. Из поля объединения, соответствующего типу данных, функции будут получать число при числовом типе объекта или указатель на экземпляр другой структуры. Также обо-

лочка содержит поля, обеспечивающие работу сборщика мусора. Её полная структура представлена в таблице 3.1.

Таблица 3.1 – Структура object_t для оболочки объекта

Имя	Тип	Описание	Возможные значения
type	type_t	Тип объекта	NUMBER, SYMBOL, PAIR, STRING, ARRAY
u	union	Объединение, хранящее данные объекта в соответствии с его типом	Нет ограничений
next	object_t*	Указатель для сборщика мусора на следующий свободный объект	Если NULL — данный объект является последним в списке свободных или список пуст, иначе содержит указатель на следующий свободный объект.
mark	int	Указывает на то, связан ли объект с символом.	Если 1 — связан, 0 — не связан.
free	int	Свободен ли элемент для перезаписи.	Если 1 — элемент в списке свободных объектов, иначе занят

Объединение u — используется для хранения значения, соответствующего объекту, в структуру которого включено это объединение. В зависимости от типа объекта, одно из полей symbol, pair, str, arr этого объединения будет иметь значение, остальные не используются. Структура этого объединения представлена в таблице 3.2.

Для элемента value тип объекта – NUMBER, symbol – SYMBOL, pair – PAIR, str – STRING, arr – ARRAY.

Таблица 3.2 – Структура u для хранения значения объекта

Имя	Тип	Описание
value	int	Содержит числовое значение, если объект имеет тип NUMBER
symbol_s*	symbol	Указывает на объект-символ, если объект имеет тип SYMBOL
pair_s*	pair	Указывает на объект-пару, если объект имеет тип PAIR
string_s*	str	Указывает на объект-строку, если объект имеет тип STRING
array_s*	arr	Указывает на объект-массив, если объект имеет тип ARRAY

Структуры остальных четырёх типов объектов представлены в таблицах 3.3 – 3.6.

Таблица 3.3 – 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 3.3

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

Таблица 3.4 – 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 3.5 – 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 3.5

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 3.6 – 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 3.6

1	2	3	4
lambda	object_t*	Указатель на объект лямбда-выражения, связанный с символом	Если 1 — связан, 0 — не связан.
func	func_t	Указатель на примитив функции, связанный с символом.	Нет ограничений

3.3.5 Исполнитель

Алгоритм работы исполнителя представлен пошагово, где некоторые шаги имеют подшаги, которые также надо пройти, если условие 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

Разработанный в рамках этой работы сборщик мусора работает в две фазы по алгоритму пометки и очистки и осуществляет сборку по следующе-

му принципу. Объекты и пары освобождаются в конце вычисления выражения верхнего уровня. Символы сборщиком не затрагиваются.

1. Фаза пометки. Обходим все символы в таблице символов и выполняем пометку объектов, на которые они указывают. Пометка реализуется через поле mark в структуре объекта и пары. 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 и/или пару.

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 3.7.

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

Таблица 3.7 – Перечень функций арифметического модуля

Имя	Аргументы	Описание	Пример
1	2	3	4
Суммирование	Числа	Возвращает сумму чисел списка	< (+ 1 2 3) > 6

Продолжение таблицы 3.7

1	2	3	4
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
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

Продолжение таблицы 3.7

1	2	3	4
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 3.8.

Таблица 3.8 – Перечень функций исполнительного модуля

Имя	Аргументы	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
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

Продолжение таблицы 3.8

1	2	3	4
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
Объявление переменной	Символ, любой	Объявляет переменную с именем, переданным в качестве первого аргумента, и значением – второго.	< (setq val 3) > 3

Продолжение таблицы 3.8

1	2	3	4
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	< (or (= 1 2)) > NIL
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 3.9.

Таблица 3.9 – 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)
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 3.10.

Таблица 3.10 – Перечень функций модуля работы с точечными парами

Имя	Аргументы	Описание	Пример
1	2	3	4
Первый элемент	Список	Возвращает первый элемент переданного аргументом списка	< (car '(a b)) > A

Продолжение таблицы 3.10

1	2	3	4
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 3.11.

Таблица 3.11 – 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 3.11

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

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. Клинтон Л. Джеффери. Создайте свой собственный язык программирования. Руководство программиста по разработке компиляторов, интерпретаторов и доменно–ориентированных языков для решения современных вычислительных задач / пер. с англ. С. В. Минца. – М.: ДМК Пресс, 2023. – 408 с.: ил. – ISBN 978–5–93700–140–5. – Текст : непосредственный.

3. 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.

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

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

6. Костельцев А. В. Построение интерпретаторов и компиляторов : Использование программ BIZON, BYACC, ZUBR : [Учеб. пособие] / А. В. Костельцев. – СПб. : Наука и техника, 2001. – 218,[1] с. : ил. – ISBN 5–94387–033–4. – Текст : непосредственный.

7. Шостак, Е. В. Основы программирования трансляторов языков программирования : учеб. – метод. пособие / Е. В. Шостак, И. М. Марина, Д. Е. Оношко. – Минск : БГУИР, 2019. – 66 с. : ил. – ISBN 978–985–543–470–3. – Текст : непосредственный.

8. Коробова, И.Л. Основы разработки трансляторов в САПР : учебное пособие / И.Л. Коробова, И.А. Дьяков, Ю.В. Литовка. – Тамбов : Изд–во Тамб. гос. техн. ун–та, 2007 – 80 с. – ISBN 978–5–8265–0591–5. – Текст : непосредственный.

9. Ричард Бёрд. Жемчужины проектирования алгоритмов: функциональный подход / Пер. с англ. В. Н. Брагилевского и А. М . Пеленицына. – М .; Д М К Пресс, 2013. – 330 с.: ил. – ISBN 978–5–94074–867–0. – Текст : непосредственный.

10. 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.

11. Форд Н. Продуктивный программист. Как сделать сложное простым, а невозможное – возможным. – Пер. с англ. – СПб.: Символ–Плюс, 2009. – 256 с., ил. – ISBN 978–5–93286–156–1. – Текст : непосредственный.

12. Структура и интерпретация компьютерных программ [Текст] / 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.

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