Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ _____**ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ**_____

КАФЕДРА _____**КОМПЬЮТЕРНЫЕ СИСТЕМЫ И СЕТИ**_____

НАПРАВЛЕНИЕ ПОДГОТОВКИ __**09.03.01 Информатика и вычислительная техника** _____

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

Компилятор для языка программирования на основе обратной польской записи

Студент _ИУ6-53Б_____ _____ **_____В.К. Залыгин_**
(Группа) (Подпись, дата) (И.О. Фамилия)

Руководитель курсовой работы _____ **_____Б.И. Бычков___**
(Подпись, дата) (И.О. Фамилия)

2024 г.

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

УТВЕРЖДАЮ
Заведующий кафедрой ___ИУ6__
(Индекс)
_____ А.В. Пролетарский
(И.О. Фамилия)
« 11 » сентября 2024 г.

З А Д А Н И Е
на выполнение курсовой работы

по дисциплине ____Технология разработки программных систем_____

Студент группы ____ИУ6-53Б_____

_____Залыгин Вячеслав Константинович_____
(Фамилия, имя, отчество)

Тема курсовой работы ____Компилятор для языка программирования на основе обратной польской записи_____

Направленность КР (учебная, исследовательская, практическая, производственная, др.)

_учебная_____

Источник тематики (кафедра, предприятие, НИР) ____Кафедра_____

График выполнения КР: 25% к 4 нед., 50% к 7 нед., 75% к 11 нед., 100% к 14 нед.

Техническое задание ___см. техническое задание в приложении А_____

Оформление курсовой работы:

1. Расчетно-пояснительная записка (РПЗ) на 25-30 листах формата А4.
2. Техническое задание на 5-9 листах формата А4 – оформляется в качестве приложения А к РПЗ.
3. Руководство пользователя на 6-8 листах формата А4 – оформляется в качестве приложения Б к РПЗ.
4. Графический и иллюстративный материал оформляется в виде рисунков и помещается в РПЗ.
5. РПЗ закачать на страницу дисциплины на сайте кафедры не позднее, чем накануне защиты.
6. После защиты заменить титул РПЗ на скан титула РПЗ с оценкой комиссии и заново закачать РПЗ в формате .pdf на страницу дисциплины.

Дата выдачи задания «_2_ » _сентября_ 2024 г.

Руководитель курсовой работы 11.09.2024 Б.И. Бычков
(Подпись, дата) (И.О. Фамилия)

Студент 11.09.2024 В.К. Залыгин
(Подпись, дата) (И.О. Фамилия)

Примечание: Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедре.

РЕФЕРАТ

Расчетно-пояснительная записка состоит из 38 страниц, включающих в себя 15 рисунков, 8 таблиц, 8 источников и 3 приложения.

КОМПИЛЯТОР, СТЕКОВЫЙ ЯЗЫК, ОБРАТНАЯ ПОЛЬСКАЯ ЗАПИСЬ, LINUX, НАБОР КОМАНД X64

Объектом разработки является приложение-компилятор с исходного языка в машинный код архитектуры x64.

Цель работы – проектирование и реализация компилятора для стекого языка с синтаксисом на основе обратной польской записи, позволяющего создавать исполняемые файлы для целевой архитектуры x64.

Разрабатываемое программное обеспечение предназначенно для программистов, создающих программы на исходном языке. Область применения – создание программ алгоритмов обработки данных. Для разработки использовались:

– язык программирования rust,

– библиотеки clap, nom,

– программы nasm и ld,

– среда разработки visual studio code,

– git как система контроля версий,

– github для хранения кода и github actions для выполнения CI-процесса.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Обратная польская запись – форма записи выражений, в которой операнды расположены перед знаками операций

Стековый язык – язык программирования, в котором для передачи параметров используется машинная модель стека

Конкатенативный язык – язык программирования, основанный на том, что конкатенация двух фрагментов кода выражает их композицию

Грамматика языка – сущность, определяющая систему, по которой строится язык, в которую входят набор нетерминальных символов, набор терминальных символов, набор правил и аксиома языка

Парсер – программа или компонент, осуществляющий разбор текста программы в соответствии с грамматикой языка

Абстрактное синтаксическое дерево – результат разбора текста программы в машиночитаемом виде

Транслятор – программа или компонент, преобразующий программу на изначальном языке программирования в программу на целевом языке программирования

Компилятор – транслятор, целевым языком которого является язык ассемблера

Ассемблирование – процесс преобразования кода на языке ассемблера в машинные инструкции

Терминал – объект формальной грамматики, имеющий в нём конкретное неизменяемое значение

Нетерминал – объект, обозначающий какую-либо сущность языка и не имеющий конкретного символьного значения

ВВЕДЕНИЕ

В настоящее время существует ряд языков, синтаксис которых основан на обратной польской нотации (постфиксной нотации). Такие языки используют для описание программ для стековых машин – вычислительных устройств, которые оперируют при работе опериируют стеком, в противовес регистровым машинам, оперирующим регистрами. Языки, описывающие алгоритмы для стековых машин, называют стековыми. Одна из сфер применения стековых языков – описания алгоритмов обработки данных. Стековые языки позволяют более лаконично и кратко описывать алгоритмы благодаря иной парадигме работы с контейнерами данных – в программах переменные отсуствуют и все операции последовательно выполняются над одним контейнером, стеком.

Поскольку стековые машины, в отличие от регистровых, не получили широкого распространения, существует задача компиляции кода на стековом языке под целевую регистровую архитектуру.

Таким образом, предметная область, в рамках которой ведется работа, – компиляторы для стековых языков, служащих для описания алгоритмов обработки данных.

В рамках данной курсовой работы решается задача создания компилятора для стекового языка на основе обратной польской записи (далее – исходный язык) под целевую платформу Linux x64. Целевая платформа выбрана по причине своей широкой распространенности. К компилятору для соответствия предметной области предъявляются требования по грамматике распознаваемого языка, наличие операций ввода-вывода, полнота по Тьюрингу (иными словами – наличие условных переходов и циклов/рекурсии). Также к решению предъявляются функциональные требования:

– создание исполняемых файлов из программ на исходном языке;

– сборка объектный файлов из программ на исходном языке;

– составление ассемблерных листингов программ на исходном языке.

При сравнении с существующими решениями (forth, joy, cat) преимуществом данной разработки является использование современных

инструментов и парадигм, что позволяет значительно снизить количество ошибок в программном обеспечении.

1 Анализ требований и уточнение спецификаций

1.1 Анализ вариантов использования

Поскольку техническое задание предполагает реализацию различных вариантов использования программы, целесообразно показать их на диаграмме вариантов использования. Рисунок 1 показывает возможности использования программного обеспечения.

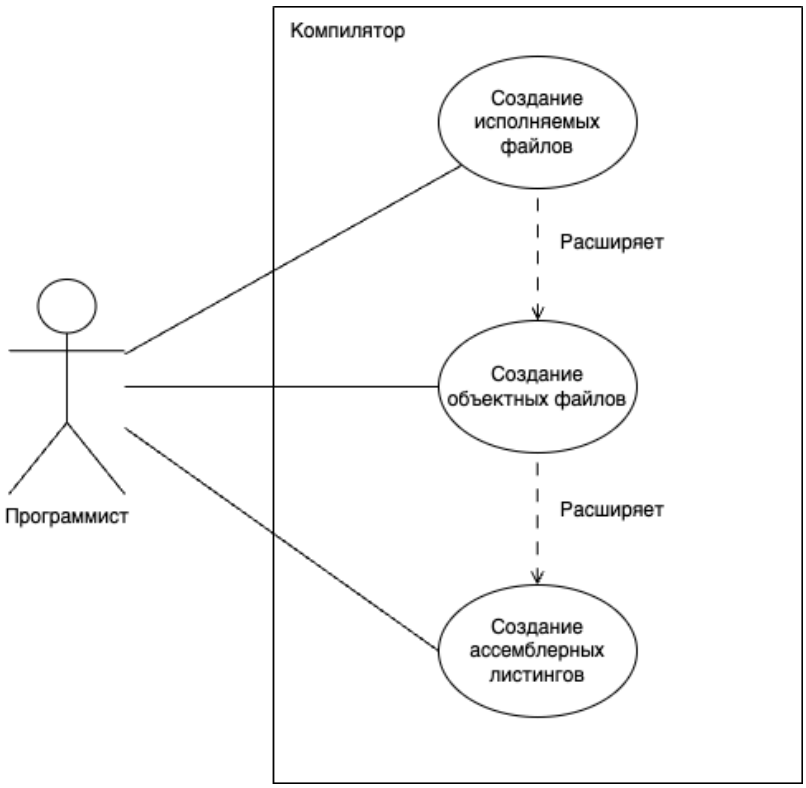

Рисунок 1 — Диаграмма вариантов использования

Отношения расширения определены из-за того, что расширяемый вариант использования является частью расширяющего варианта использования, но с дополнительными действиями. Так, создание объектного файла включает в себя создание ассемблерного листинга, который затем ассемблируется в объектный файл. Создание исполняемого файла включает в себя создание объектного файла, который затем компонуется с стандартной библиотекой в исполняемый файл. Количество дополнительных действий, которые выполняются при запуске программы, определяется при помощи выбранного режима работы. В зависимости от режима будет выполнен либо полный цикл операций над текстом программы, либо какая-то его часть.

На основе диаграммы вариантов использования можно составить ключевые варианты использования. В таблице 1 содержится информация о взаимодействии программиста с системой при генерации исполняемого файла с учетом ошибок, которые программист может допустить в программе.

Таблица 1 — Сценарий генерации исполняемого файла

Действия пользователя	Отклик системы
1) Программист создает файл с программой, составляет команду вызова компилятора и делает вызов. 3) Программист исправляет ошибки в файле с программой. 5) Программист запускает исполняемый файл или передает его на дальнейшее исполнение.	2) Компилятор возвращает сообщение с синтаксической ошибкой. 4) Компилятор выполняет генерацию и порождает исполняемый файл.

Одним из основных сценариев является генерация объектных файлов, которые за счет интеграции через систему сборки, могут быть скомпонованы с другими объектными файлами. Таблица 2 демонстрирует сценарий генерации объектного файла через использование системы сборки.

Таблица 2 — Сценарий генерации объектного файла

Действия пользователя	Отклик системы
1) Программист создает файл кодом, включает файл в список системы сборки, вызывает систему сборки, которая дает команду компилятору. 3) Программист вызывает сторонюю систему сборки, которая компонует полученный объектный файл с другими объектными файла-	2) Компилятор выполняет генерацию объектного файла в соответствии с переданными флагами.

Продолжение таблицы 2

ми в исполняемый файл. 4) Программист запускает исполняемый файл или передает его на дальнейшее исполнение.	

1.2 Анализ задания и выбор технологии, языка и среды разработки

В соответствии с требованиями технического задания необходимо разработать программу, которая может выполнять трансляцию кода на исходном языке. Компилятор должен обеспечивать поддержку ряда синтаксических конструкций, представляющих исходный язык и перечисленных в техническом задании. Исполняемые файлы, объектные файлы, ассемлерные листинги, являющиеся результатом работы компилятора, должны соответствовать набору команд x86-64 [1]. Программное обеспечение должно работать под управлением ОС Linux и иметь интерфейс командной строки.

Исторически к программам-компиляторам предъявляются требования по скорости работы, нативности, наличию интерфейса командной строки [2]. Иными словами, привычный компилятор – скомпилированное нативное CLI-приложение без сборщика мусора. При разработке решения также учитываются общие требования к программному обеспечению данной направленности, перечисленные раннее.

Вышеперечисленные требования сужают диапазон подходящих языков программирования до нескольких вариантов: C, C++, Rust, Zig. В результате по совокупности факторов был выбран язык Rust. Компилятор данного языка обеспечивает автоматический контроль за состоянием памяти без использования сбощика мусора, сам язык обладает наиболее строгой системой типов (среди предложенных) [3]. Указанные особенности Rust позволяют писать безопасное решение и недопускать ошибки в программном обеспечении. В таблице 3 показаны результаты сравнения языков программирования.

Для разработки на данном языке принято использовать Visual Studio Code, поэтому она выбрана в качестве среды разработки.

Таблица 3 — Сравнение свойств языков программирования

	C	C++	Zig	Rust
Работа с памятью	Ручная	Ручная	Ручная	Автоматическая
Компиляция в нативный код	Да	Да	Да	Да
Зрелость и стабильность	Да	Да	Нет	Скорее да
Современные методы разработки	Нет	Да	Да	Да

Поскольку процессы в рамках предметной области (создание исполняемых файлов, объектных файлов, ассемблерных листингов) удобно описывать как последовательность вызовов функций, поэтапно преобразующих код от исходного языка до исполняемого файла, рационально использовать структурный подход. Структурный подход также является идиоматичным при разработке на Rust.

Компилятор Rust автоматически контролирует состояние памяти, проверая корректность работы с памятью. Проверка осуществляется за счет концепций «владения» и «заимствования». При написании программы программист должен объяснить компилятору, как используется память в программе, а компилятор должен проверить, что такое использование безопасно. Преимуществом такого подхода по сравнению с другими автоматическими решениям является то, что все проверки осуществляются на этапе компиляции, что позволяет на этапе исполнения достичь производительность языков с ручной работой с памятью.

Под зрелостью и стабильностью подразумевается наличие стабильного канала выпуска новых версий программного обеспечения. Стабильным каналом принято называть канал, в рамках которого сохраняется обратная совместимость между версиями программного обеспечения, а также осуществляется долгосрочная поддержка версий.

Наконец критерий «современные методы разработки» учитывает поддержку языком принятых в профессиональном сообществе стандартов написания кода и программых продуктов. Так, например, язык C не поддерживает концепцию generic objects, в то время, как это делают все остальные указанные языки.

При создании программного обеспечения целесообразно проводить разработку нисходящим способом, как одним из рекомендуемых [4]. Версионирование программного обеспечения осуществляется при помощи инструмента git. Для проверки работоспособности используются автотесты, а в репозитории проекта настроен CI-процесс, который запускает автотесты с целью проверки изменений при попытке их фиксации.

Для создания интерфейса командной строки рационально использовать готовую библиотеку описания интерфейса – Clap [5]. Для разбора исходных кодов можно использовать комбинаторный подход и библиотеки, предоставляющие набор компонентов для построения генераторов комбинаторых парсеров. В данном случае используется библиотека Nom [6]. С целью ускорения разработки рационально использовать готовые решения для ассемблирования и компоновки. Под целевую платформу (Linux x64) одними из самых распрастраненных являются ассемблер nasm и компоновщик ld, они используются в рамках данной разработки.

1.3 Анализ процессов компиляции, ассемблирования, компоновки

В соответствии с техническим заданием (приложение Б) программное решение должно обеспечивать создание различных выходных файлов.

Для разработки решения необходимо разложить процесс создания выходных файлов на этапы.

Процесс создания ассемблерного листинга можно разбить на 2 этапа:

– разбор кода программы,

– трансляция в языка ассемблера (компиляция).

В случае, если необходимо собрать объектный файл, то к 2 этапам создания ассемблерного листинга добавляется еще один этап – «ассемблирование в объектный файл».

В случае, если необходимо сделать исполняемый файл, то к 3 этапам сборки объектного файла добавляется еще один этап – «компоновка исполняемого файла».

На рисунке 2 представлена функциональная диаграмма процесса трансляции программы при помощи программного решения.

Рисунок 3 уточняет блок A0, процесс трансляции исходного кода.

Рисунок 4 уточняет блок A3, процесс сборки.

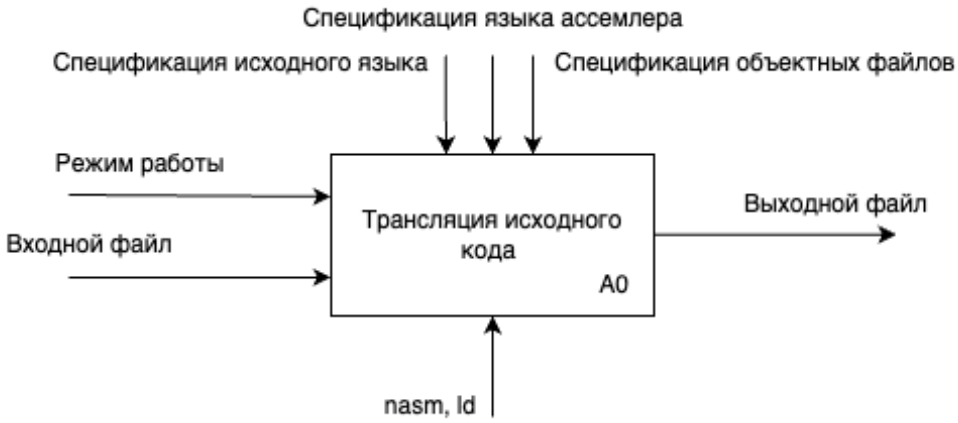

Рисунок 2 — Функциональная диаграмма

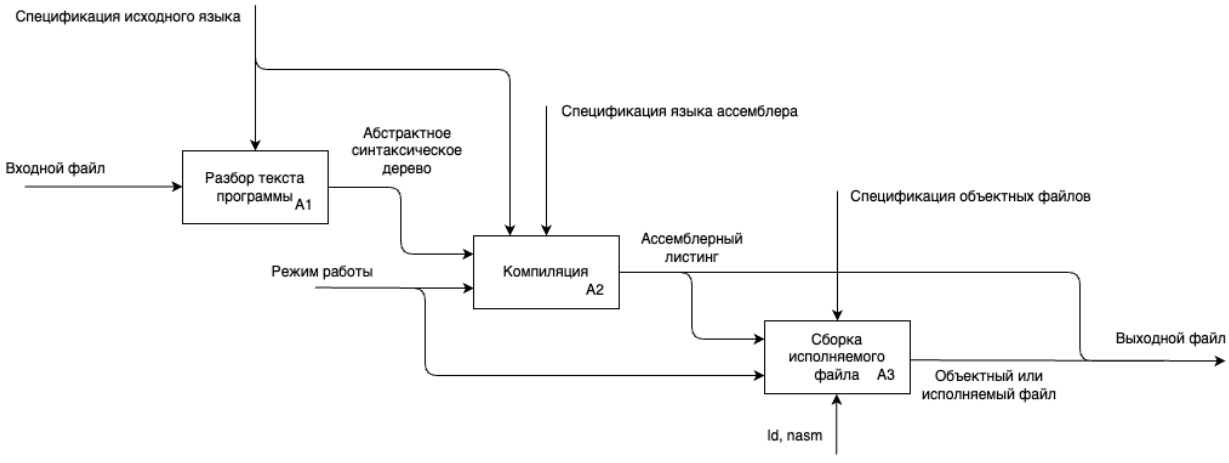

Рисунок 3 — Функциональная диаграмма, уточняющая процесс трансляции

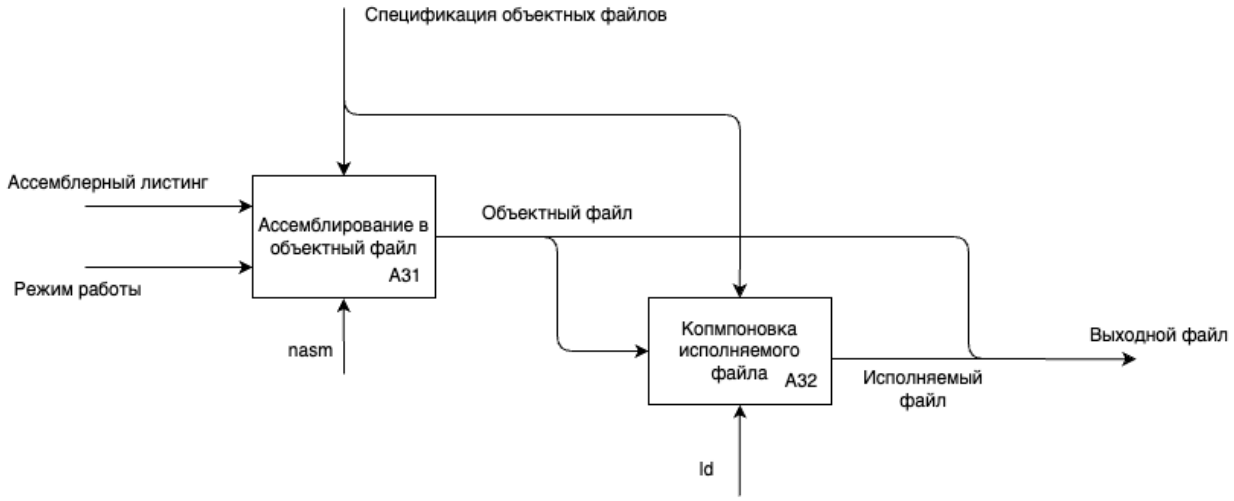

Рисунок 4 — Функциональная диаграмма, уточняющая процесс сборки

Спецификации объектных файлов, исходного языка, языка ассемблера, механизмы, слияния

Спецификация исходного языка описывает синтаксис и семантику языка. Спецификация языка ассемблера описывает синтаксис и семантику языке ассемблера, используемого в целевой архитектуре Linux x64. Спецификация объектных файлов в рамках данного проекта описывает возможности работы с объектными файлами.

Ассемблирование программы на языке ассемблера, как было указано в подразделе 1.2, осуществляется при помощи программы nasm. Блок A4, который обозначает процесс ассемблирования, использует программу nasm (показан механизмом на диаграмме), так как согласно подразделу 1.2 программа nasm выбрана в качестве ассемблера. Процесс компоновки использует программу ld, указанную как механизм, так как программа ld выбрана в качестве механизма осуществления компоновки объектных файлов.

Согласно диаграмме вариантов использования существует 3 варианта использования программы-компилятора. С целью передачи информации программе при вызове о требуемом варианте ее использования введено понятие режима работы, которое описывает, в рамках какого варианта использования программе необходимо работать.

При работе с программой в разных режимах используется унифицированный интерфейс обращения, который включает в себя 3 параметра: режим работы, входной файл, выходной файл. В связи с унификацией, тип выходного

файла зависит от выбранного режима работы, на диаграммах это показано слиянием стрелок, идущих из разных блоков, и в результате объединяющихся в стрелку «выходной файл».

2 Проектирование структуры и компонентов программного продукта

2.1 Проектирование структуры приложения

Согласно подразделу 1.2 в рамках разработки был выбран структурный подход. Для работы при таком подходе необходимо уточнить структурную схему программного решения. На рисунке 5 изображена структурная схема проекта, составленная по включению.

Рисунок 5 — Структурная схема

Описание частей структурной схемы приведено ниже:

– программа-компилятор, агрегирующая все модули программы, является точкой входая при запуске программы;

– интерфейс, часть, содержащая подпрограммы, ответственные за взаимодействие с пользователем;

– библиотека компонентов трансляции, агрегирует компоненты, участвующие в процессе трансляции;

– компонент разбора текстов, включает в себя подпрограммы для разрбора текстов на исходном языке;

– компонент компиляции, содержит подпрограммы, участвующие в процессе компиляции абстрактного синтаксического дерева в язык ассемблера

целевого набора команд;

– компонент сборки, агрегирует подпрограммы, ответственные за сборку и компоновку.

2.2 Проектирование интерфейса командной строки

При использовании нисходящего подхода, который был выбран в подразделе 1.2, необходимо начинать разработку от компонентов верхнего уровня, постепенно спускаясь вниз к компонентам нижних уровней. После уточнения структурной схемы в пункте 2.1 наиболее верхнеуровневым компонентом оказался компонент пользовательского интерфейса, в связи с чем с него начата разработка.

Согласно техническому заданию приложение должно иметь интерфейс командной строки. Для наглядности используется синтаксическая диаграмма грамматики интерфейса. Грамматика показа с использованием расширенной формы Бекуса-Наура (РБНФ). Форма изображена на рисунке 6. Аксиомой грамматики является нетерминал «plc».

```
plc          = "plc" [{run_options} file|info_options]
run_options  = "-"("S"|"c"|"o" file) |
               "--"("compile-only"|"assemble-only"|"output" file)
info_options = "-"("h"|"V") | "--"("help"|"version"|)
file         = спецсимвол|буква|цифра{спецсимвол|буква|цифра}
```

Рисунок 6 — Описание интерфейса командной строки в виду РБНФ

Интерфейс командной строки соответствует принятым идиомам проектирования интерфейсов для консольных приложений [7]. Интерфейс состоит из ключевого слова «plc», служащего именем приложения и началом команд для него, из набора флагов, определяющих поведение приложения, из имен файлов, которыми должна оперировать программа.

Флаги, определяющие поведение, деляется на 2 типа: опции трансляции (на диаграмме обозначены нетерминалом «run_options») и информацинные опции (нетерминал «info_options»). Перечисление поддерживаемых флагов и их семантика представлены в таблице 4.

В случае, если никакой флаг не был выставлен, то используется режим работы с созданием исполняемого файла по пути ./output.

Наконец, нетерминал «file» обозначает путь до файла. Программа принимает корректные пути операционной системы Linux.

Таблица 4 — Поддерживаемые флаги

Тип флага	Краткая форма флага	Длинная форма флага	Назначение
Информаци-онный	-h	--help	Вывод сообщения с информацией о приложении и доступных действиях
Информаи-ционный	-V	--version	Вывод версии приложения
Опция трансляции	-S	--compile-only	Выполнение только компиляции кода в ассемблерный листинг
Опция трансляции	-c	--assemble-only	Создание только объектного файла
Опция трансляции	-o	--output	Указание пути до выходного файла

2.3 Разработка основных алгоритмов

В подразделе 1.3 описаны функциональные диаграммы процессов, в рамках которых используется решение. Для переноса процессов в программное обеспечение необходимо описать алгоритмы, соответствующие процессам. В качестве представления алгоритмов целесообразно использовать схемы алгоритмов. На рисунке 7 представлен алгоритм для основной подпрограммы. В рамках основной подпрограммы происходит выбор режима, а также вызов агрегирующей подпрограммы «выполнить».

Схема алгоритма работы библиотечной подпрограммы «выполнить» представлена на рисунке 8. В зависимости от режима работы подпрограмма выполняет разное количество шагов для создания результирующего выходного

файла. Для выполнения промежуточных шагов используются временные файлы. После проведения необходимых операций и получения выходного файла временные файлы удаляются.

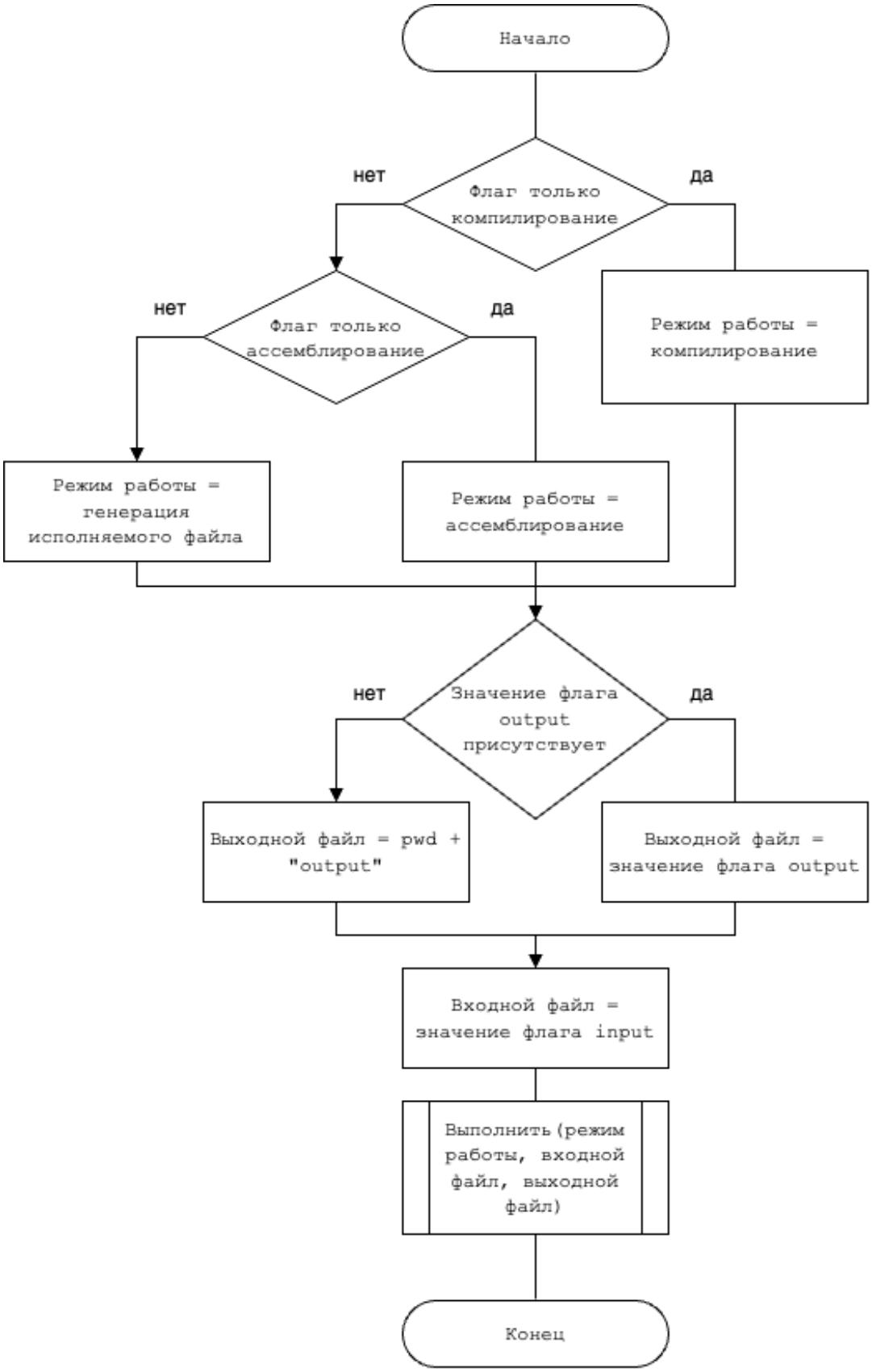

Рисунок 7 — Алгоритм работы основной подпрограммы

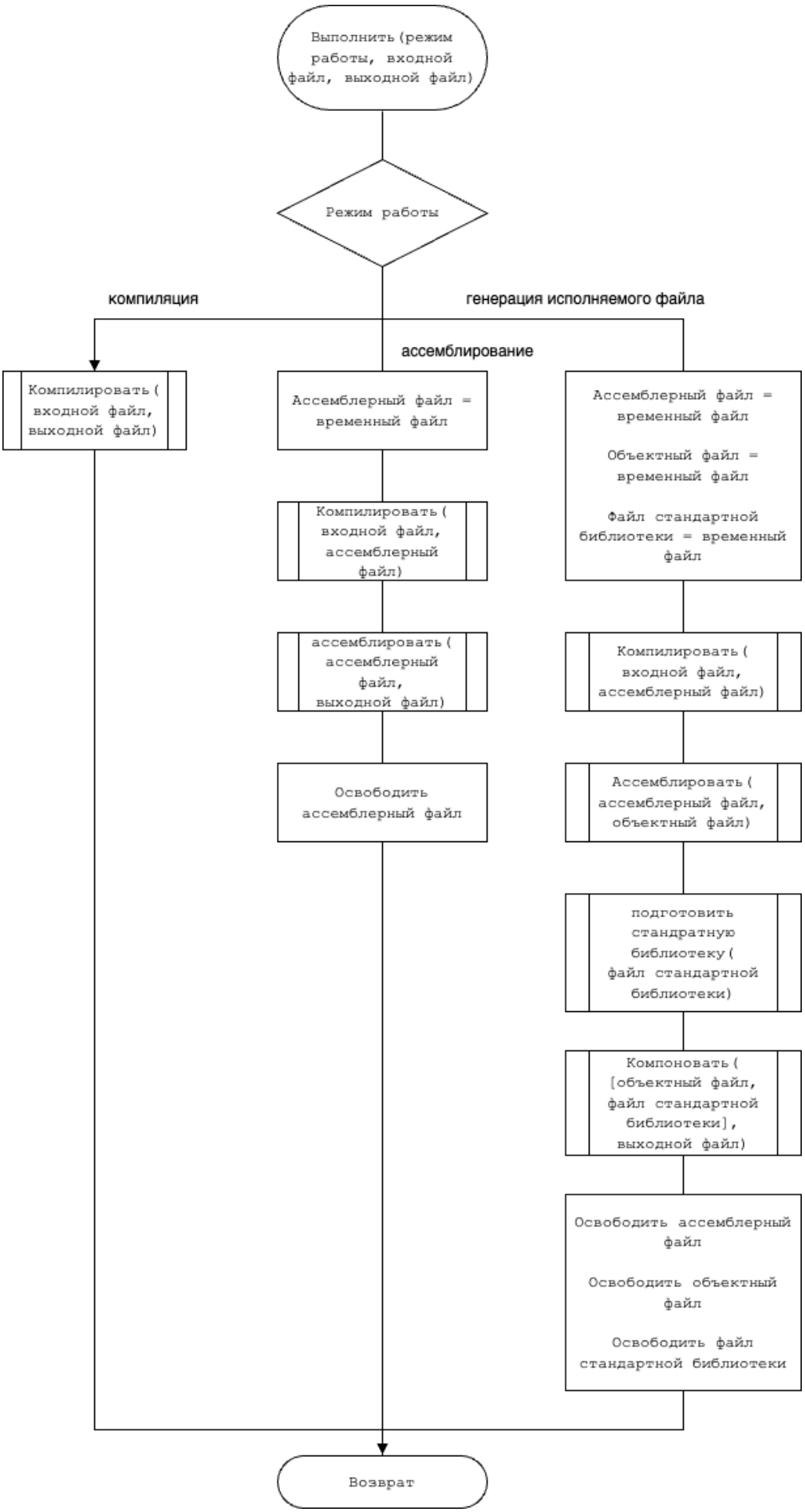

Рисунок 8 — Схема алгоритма подпрограммы «выполнить»

Поскольку язык обладает стандартной библиотекой (подробнее этом изложено в подразделе 2.5), существует подпрограмма подготовки к компоновке файла стандартной библиотеки. Алгоритм данной подпрограммы показан на рисунке 9. Алгоритм соотвествует аналогичному для кода из входного файла, но оперирует заранее заготовленными ассемлерными листингами для создания объектного файла.

Рисунок 9 — Схема алгоритма подпрограммы подготовки стандартной библиотеки

2.4 Разработка компонента разрбота текста

Выбранная библиотека для построения генераторов комбинаторных парсеров, Nom, позволяет реализовать разбор выражений методом рекурсивного спуска [6]. Также по техническому заданию необходимо

реализовать разбор ряда конструкций в синтаксисе обратной польской записи. В связи с данными ограничениями аксиома языка описывает подходящий под требования конкатенативный язык. Аксиома изображена на рисунке 10. Определение аксиомы утверждает, что каждый «терм» окружен либо другими «термами», либо разделителями, либо началом и окончанием файла.

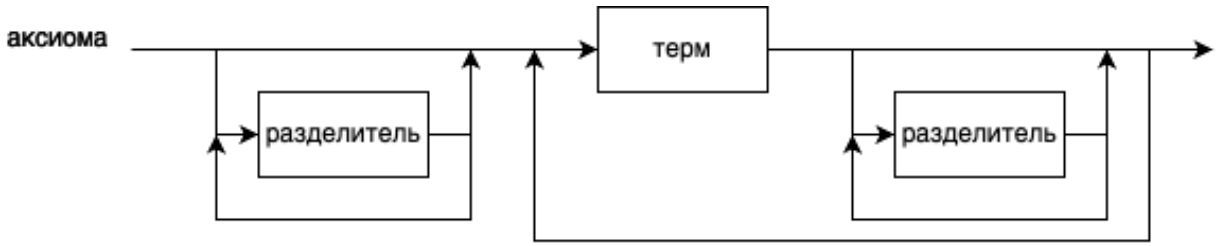

Рисунок 10 — Синтаксическая диаграмма аксиомы исходного языка

Одним из главных правил в грамматике является правило «терм», обозначающее некоторую операцию. Рисунок 11 показывает синтаксическую диаграмму правила «терм». «Терм» означает синтаксическую единицу, команду на исходном языке.

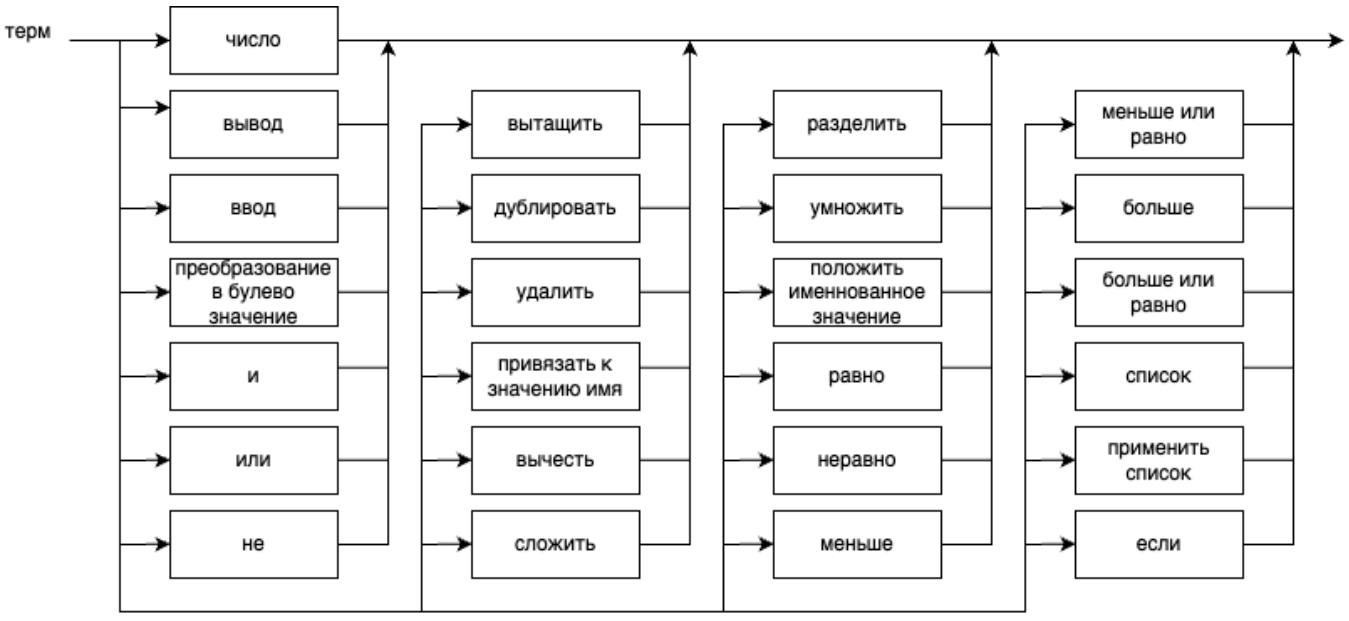

Рисунок 11 — Синтаксическая диаграмма «терм»

Между термами могут располагаться разделители, в том числе и комментарии (комментарий располгается от своего начала и до конца строки). Разделителями являются проблемы, переводы строки, табы. Синтаксическая диаграмма правил «разделитель» и «комментарий» показана на рисунке 12.

Остальные правила изображены на рисунках 13, 14, 15. Данные правила задают непосредственно команды исходного языке. На рисунке 15 представлена синтаксическая диаграмма списков – важного элемента языка, который, являясь рекурсивным вкупе с операцией ветвления, позволяет языку удовлетворить полноте по Тьюрингу.

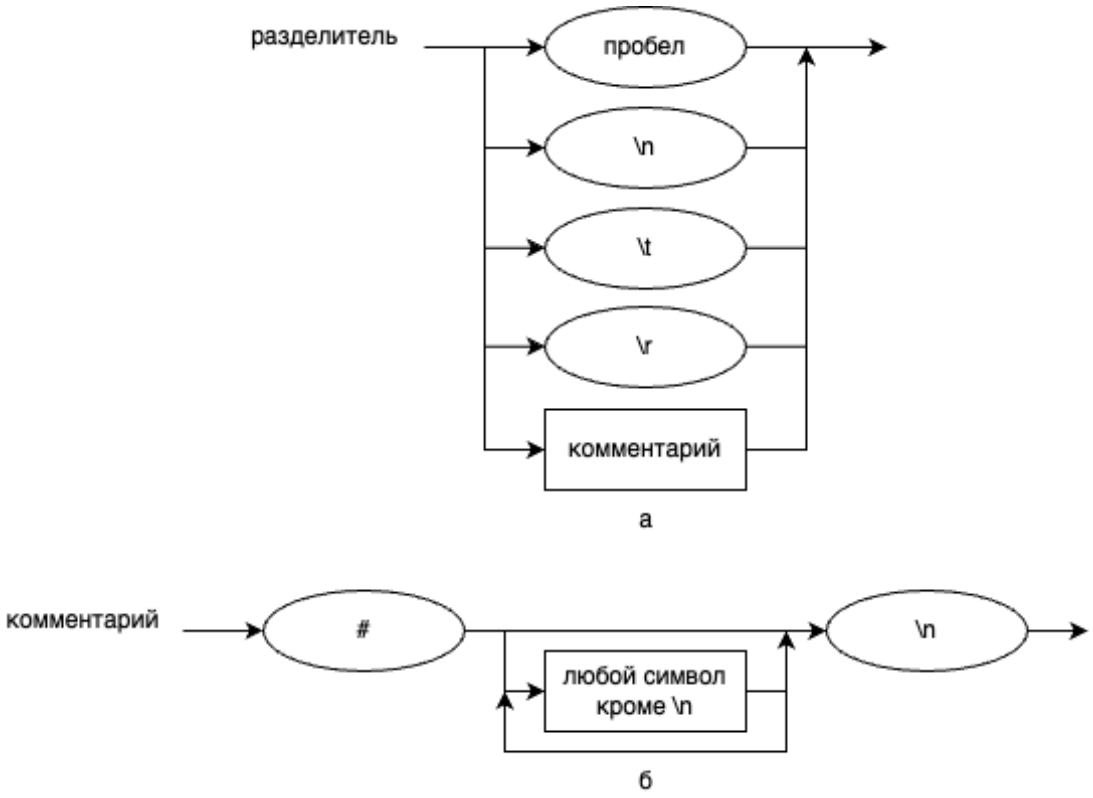

Рисунок 12 — Синтаксические диаграммы: а – «разделитель»; б – «комментарий»

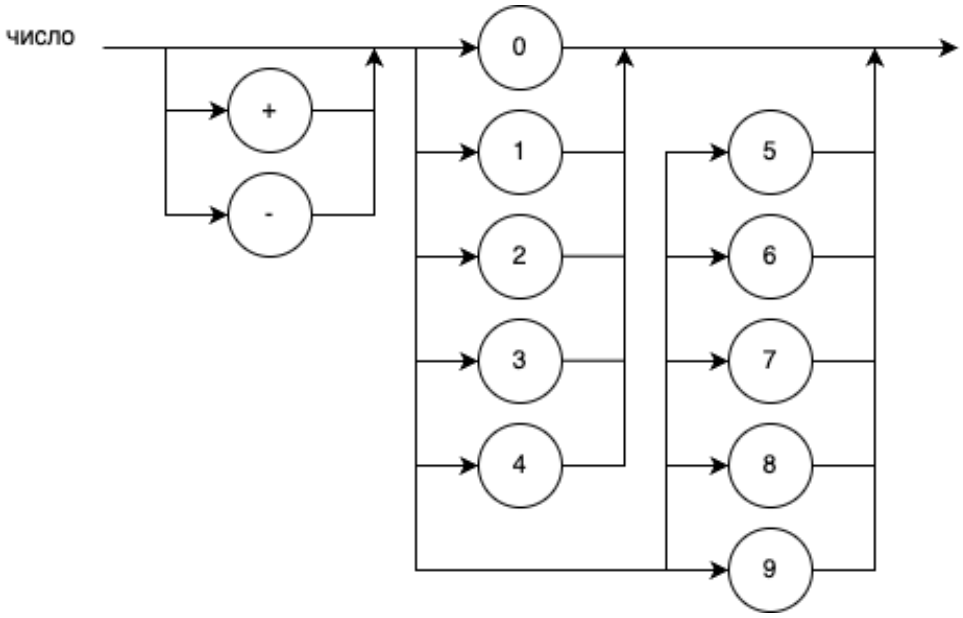

Рисунок 13 — Синтаксическая диаграмма «число»

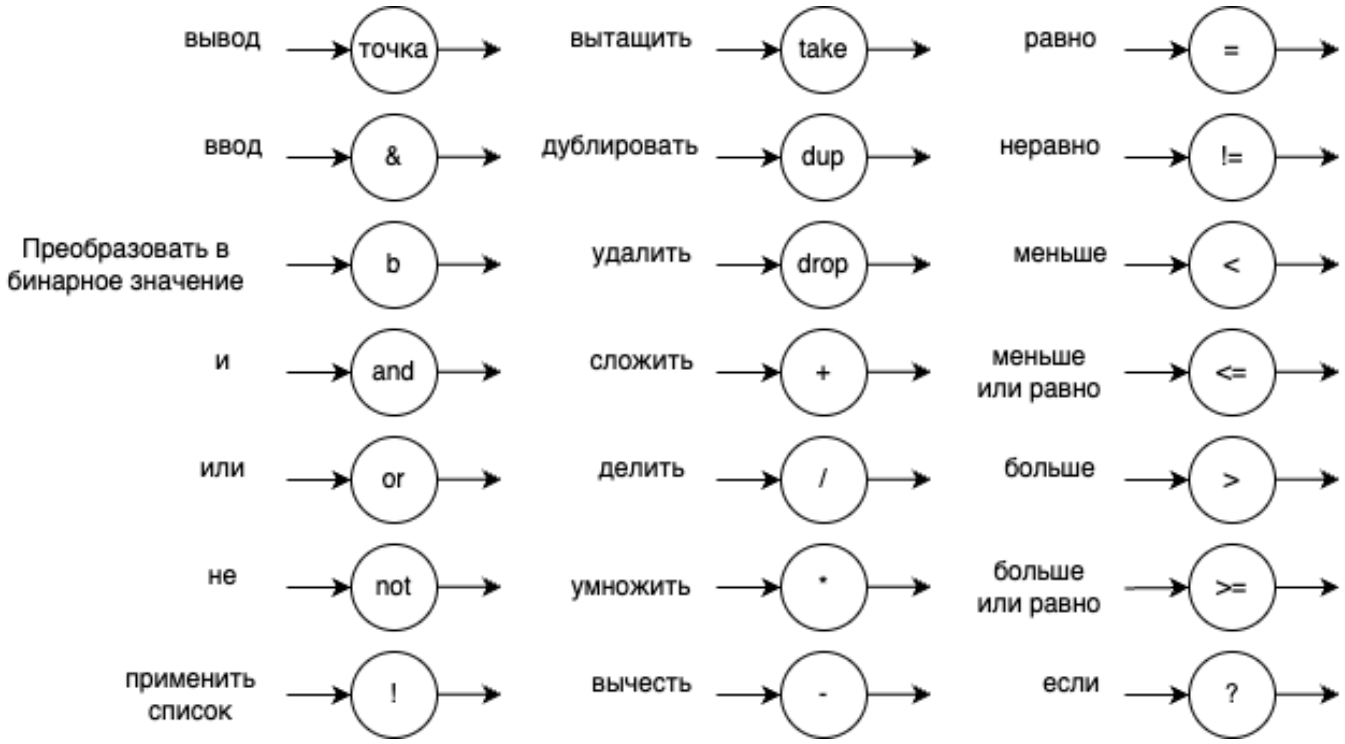

Рисунок 14 — Синтаксические диаграммы правил для некоторых термов

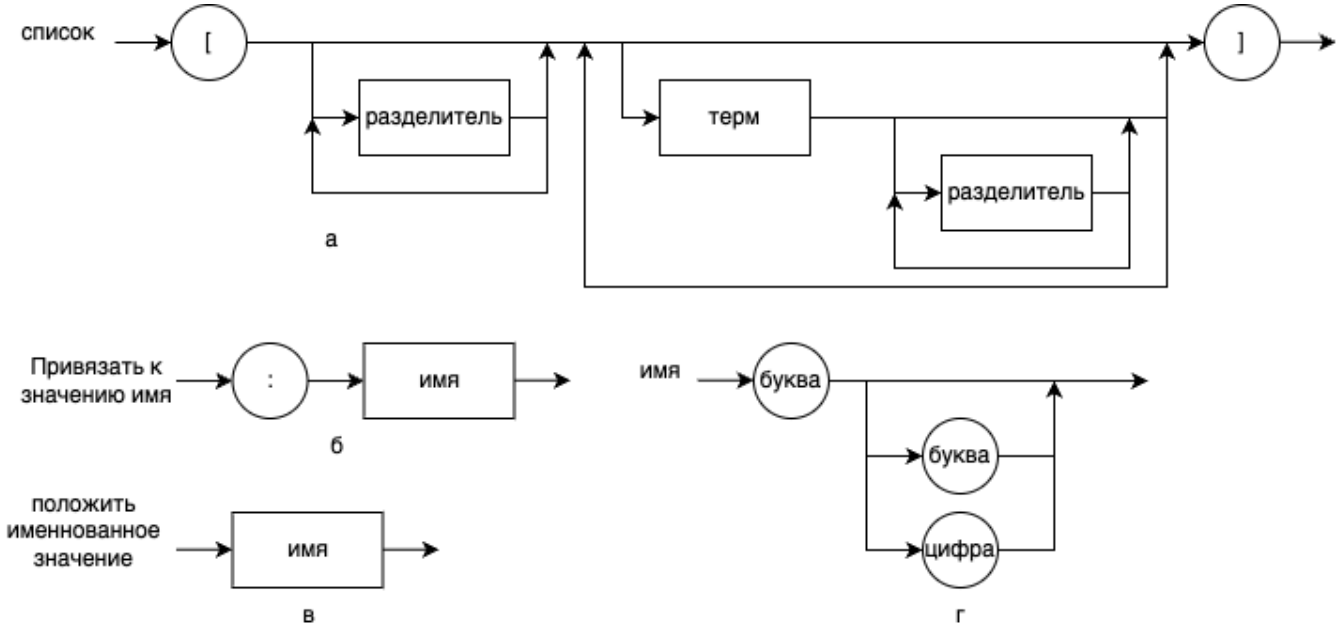

Рисунок 15 — Синтаксические диаграммы правил: а – «список»; б – «привязать к имени»; в – «положить именнованное значение»; г – «имя»

Для создания парсера выбрана библиотека Nom, написанная в парадигме комбинаторых парсеров. Составление парсера при комбинаторном подходе подразумевает использование подпрограмм-генераторов парсеров, в аргументах которых указываются необходимые для создания парсера параметры, и которые в результате вызова возвращают готовый к работе

парсер. Важным аспектом при работе с генераторами парсеров является их возможность «комбинировать парсеры»: для создания возвращаемого парсера они могут использовать уже созданные парсеры, поданные в качестве аргументов. Комбинаторные парсеры позволяют близко к диаграммам описывать правила грамматики. Листинг 1 демонстрирует подпрограмму, осуществляющую разбор аксиомы языка. Данная подпрограмма наглядно иллюстрирует принципы комбинаторного подхода.

Листинг 1 — Подпрограмма разбора аксиомы языка

```
pub fn axiom<'s, E: ParseError<&'s str> + ContextError<&'s
str>>(
    inp: &'s str,
) -> IResult<&'s str, Vec<Term>, E> {
    delimited(
        many0(separator),
        many0(term.and(many0(separator))).map(|term_pairs| {
            term_pairs
                .into_iter()
                .map(|term_pair| term_pair.0)
                .collect()
        }),
        many0(separator),
    )
    .parse(inp)
}
```

Описания некоторых частей, использованных при построении подпрограммы разрабора аксиомы, приведено ниже:

– delimeted, генератор, который позволяет окружить данный парсер парсерами перед и после, игнорируя их результаты работы;

– separator, парсер, соответствующий правилу «разделитель»;

– many0, генератор, использующий поданный парсер 0 или больше раз пока это возможно;

– term, парсер, соответствующий правилу «терм».

Поскольку язык конкатенативен, его абстрактное синтаксическое дерево (далее – АСТ) вырождено в список, элементами которого являются структура, описывающая некоторую операцию (результат разбора парсера term).

Результатом разбора исходного кода является AST, которое после отработки парсера передается на следующий этап обработки.

2.5 Разработка компонента компиляции

На основе AST, полученного после разбора исходного кода (процесс описан в подразделе 2.4), создается ассемблерный листинг программы. Чтобы выполнить данную задачу, необходимую каждой операции сопоставить заготовку (шаблон) на языке ассемблера, удовлетворяющую семантике операции [8]. Именно при разработке таких заготовок решается задача адаптации стекового языка (предназначенного для стековой машины) под регистровую машину.

Решение задачи адаптации заключается в низкоуровневой эмуляции стековой машины. В частности эмуляция стека операндов представлена выделенной под стек памятью и указателями на вершину и основание стека. Применение списков операций реализуется за счет стека вызовов. Оставшиеся операции реализуются за счет инструкций, взаимодействующих со стеками операндов и вызовов.

Операции и соответствующие им шаблоны на языке ассемблера представлены в таблице 5.

Шаблоны описаны с использованием dsl, разработанного для удобной генерации кода на языке ассемблера. Для создания инструкции используется макрос i, который принимает мнемонику инструкции, а затем аргументы. Описание возможных аргументов приведено ниже:

– reg, работа с регистом;

– indirect_register, значение в памяти по адресу из регистра;

– opexpr, сырая формула;

– op::label, подстановка символа;

– op::literal, подстановка литерала.

Для последующего развития предусмотрена стандартная библиотека. В текущей версии решения реализованы функции ввода/вывода, завершения работы приложением. Библиотека собирается аналогично исходному коду в

объектный файл, а затем компонуется с объектным файлом с точкой входа программы.

Таблица 5 — Некоторые операции и их ассемблерные шаблоны

Операция	Шаблон
Положить число	`i!(Sub, reg!(Ebx), Op::Literal(OP_SIZE_BYTES)), i!(Mov, indirect_register!(Ebx), OP_SIZE, Op::Literal(*number as i64))`
Добавить	`i!(Mov, reg!(Eax), indirect_register!(Ebx)), i!(Add, reg!(Ebx), Op::Literal(OP_SIZE_BYTES)), i!(Add, indirect_register!(Ebx), reg!(Eax)),`
Делить	`i!(Mov, reg!(Edi), indirect_register!(Ebx)), i!(Add, reg!(Ebx), Op::Literal(OP_SIZE_BYTES)), i!(Xor, reg!(Rdx), reg!(Rdx)), i!(Mov, reg!(Rax), indirect_register!(Ebx)), i!(Cltq), i!(Cqto), i!(Div, reg!(Edi)), i!(Mov, indirect_register!(Ebx), reg!(Eax)),`
Вывести	`i!(Call, oplabel!(STD_PRINT_FN_LABEL))`
Дублировать	`i!(Mov, reg!(Eax), indirect_register!(Ebx)), i!(Sub, reg!(Ebx), Op::Literal(OP_SIZE_BYTES)), i!(Mov, indirect_register!(Ebx), reg!(Eax)),`
Удалить	`i!(Add, reg!(Ebx), Op::Literal(OP_SIZE_BYTES))`
Вытащить	`i!(Xor, reg!(Rcx), reg!(Rcx)), i!(Mov, reg!(Ecx), indirect_register!(Ebx)), i!(Add, reg!(Ebx),`

Продолжение таблицы 5

	`Op::Literal(OP_SIZE_BYTES)),`
	`i!(Cmp, reg!(Ecx), opexpr!("dword 0")),`
	`i!(Jz, opexpr!(no_exch_label)),`
	`i!(label!(exch_cycle_label.as_str())),`
	`i!(Mov, reg!(Eax), opexpr!(format!`
	`("[EBX+ECX*{OP_SIZE_BYTES}]"))),`
	`i!(Mov, reg!(Esi), opexpr!(format!`
	`("[EBX+ECX*{OP_SIZE_BYTES}-`
	`{OP_SIZE_BYTES}]"))),`
	`i!(Mov, opexpr!(format!`
	`("[EBX+ECX*{OP_SIZE_BYTES}]")), reg!(Esi),`
	`i!(Mov, opexpr!(format!`
	`("[EBX+ECX*{OP_SIZE_BYTES}-`
	`{OP_SIZE_BYTES}]")), reg!(Eax)),`
	`i!(Sub, reg!(Ecx), opexpr!("dword 1")),`
	`i!(Jnz, oplabel!(exch_cycle_label)),`
	`i!(label!(no_exch_label.as_str())),`

Применить список	`i!(Add, reg!(Ebx),`
	`Op::Literal(OP_SIZE_BYTES)),`
	`i!(Call, opexpr!(format!("[EBX-`
	`{OP_SIZE_BYTES}]"))),`

Преобразовать в булево значение	`i!(Cmp, indirect_register!(Ebx), opexpr!`
	`("dword 0")),`
	`i!(Mov, reg!(Eax), Op::Literal(1)),`
	`i!(Cmovz, reg!(Eax), opexpr!(format!`
	`("[{DWORD_ZERO_LABEL}]"))),`
	`i!(Mov, indirect_register!(Ebx), reg!(Eax)),`

Отрицание	`i!(Xor, indirect_register!(Ebx), opexpr!`
	`("dword -1")),`
	`i!(Mov, reg!(Eax), Op::Literal(1)),`
	`i!(Cmp, indirect_register!(Ebx), opexpr!`
	`("dword 0")),`
	`i!(Cmovz, reg!(Eax), opexpr!(format!`
	`("[{DWORD_ZERO_LABEL}]"))),`
	`i!(Add, indirect_register!(Ebx), reg!(Eax)),`

Продолжение таблицы 5

Привязать значение к имени	`i!(label!(name), opexpr!(format!("resq 1")))`
	`i!(Mov, reg!(Rax), indirect_register!(Ebx)),`
	`i!(Add, reg!(Ebx),`
	`Op::Literal(OP_SIZE_BYTES)),`
	`i!(Mov, opexpr!(format!("[{name}]")), reg!`
	`(Rax)),`

Положить именнованное значение	`i!(Mov, reg!(Rax), opexpr!(format!`
	`("[{name}]"))),`
	`i!(Sub, reg!(Ebx),`
	`Op::Literal(OP_SIZE_BYTES)),`
	`i!(Mov, indirect_register!(Ebx), reg!(Rax)),`

3 Выбор стратегии тестирования и разработка тестов

3.1 Описание выбранных стратегий, способов, методов тестирования

В качестве основной стратегиии выбрано функциональное тестирование. Данная стратегия основывается на принципе «черного ящика», что делает ее максимально близкой к опыту, который получает пользователь при использовании программного решения [4].

В рамках стратегии функционального тестирования реализованы сквозные автотесты, покрывающие функциональность приложения. Данные автотесты гарантируют корректность работы основных путей программного продукта, а следовательно и требуемое качество пользовательского опыта. Скозные автотесты реализованы за счет вспомогательной программы, которая собирает основную программу-компилятор и через средства операционной системы проверяет работоспособность за счет вызовов программы и проверки результатов работы, как если бы это выполнял сам пользователь. Для составления тестов используется метод эквивалентного разбиения.

Для компонента парсера используется структурное тестирование. Модульные тесты, написанные в рамках стратегии структурного тестирования, позволяют гарантировать корректность работы каждой части парсера и компонента в целом. Наличие модульных тестов обусловленно сложностью устройства парсера и его хрупкостью при внесении изменений. Для составления тестов используется метод покрытия по операторам.

Таким образом, в рамках тестирования программного продукта используется комбинированный подход.

3.2 Функциональное тестирование программного решения

Таблица 6 иллюстрирует выделенные классы эквивалентности. Для выделения классов эквивалентности использованы соображения о возможных значениях флагов вызова программы и возможных используемых конструкций в коде программм на исходном языке.

Таблица 6 — Выделенные классы эквивалентности

Параметр разбиения	Правильные классы эквивалентности	Неправильные классы эквивалентности
Флаги вызовы	Известные флаги	Неизвестный флаги
Входной файл	Существующий входной файл	Несуществующий входной файл
Синтаксис кода	Корректный синтаксис, корректное использование конструкций языка	Неизвестный символ, Неизвестное имя

Таблица 7 показывает фрагмент тестов по методу эквивалентного разбиения.

Таблица 7 — Фрагменты сквозных тестов

№	Класс эквивалентности	Аргументы вызова	Входной файл	Ожидаемый результат	Результат теста	Вывод
1	Неизвестный флаг	-a	Нет	Ошибка, неизвестный флаг	Ошибка, неизвестный флаг	Корректная работа
2	Неизвестный флаг	--abc	Нет	Ошибка, неизвестный флаг	Ошибка, неизвестный флаг	Корректная работа
3	Несущетвующий файл	Нет	Несуществующий файл	Ошибка, файл не найден	Ошибка, файл не найден	Корректная работа
4	Неизвестный символ	Нет	/	Ошибка, неизвестный символ /	Ошибка, неизвестный символ /	Корректная работа

Продолжение таблицы 7

5	Неизвестное имя	Нет	foo	Ошибка, неизвестное имя foo	Ошибка, неизвестное имя foo	Корректная работа
6	Известный флаг	-h	Нет	Сообщение помощи	Сообщение помощи	Корректная работа
7	То же	--help	Нет	Сообщение помощи	Сообщение помощи	Корректная работа
8	«»	-V	Нет	Сообщение с версией программы	Сообщение с версией программы	Корректная работа
9	«»	--version	Нет	Сообщение с версией программы	Сообщение с версией программы	Корректная работа
10	Известный флаг, существующий входной файл, корректный синтаксис	Нет флагов	Пустой файл	Генерация исполняемого файла	Генерация исполняемого файла	Корректная работа
6	То же	-c	Пустой файл	Генерация объектоного файла	Генерация объектоного файла	Корректная работа
7	«»	--assemble-only	Пустой файл	Генерация объектоного файла	Генерация объектоного файла	Корректная работа
8	«»	-S	Пустой файл	Генерация ассемблерного листинга	Генерация ассемблерного листинга	Корректная работа

Продолжение таблицы 7

9	«»	--compile-only	Пустой файл	Генерация ассемблерного листинга	Генерация ассемблерного листинга	Корректная работа
10	Корректная работа чисел и операции вывода	Нет	1 .	Исполняемый файл, вывод: 1	Исполняемый файл, вывод: 1	Корректная работа
11	Корректная работа операций ввода и вывода	< (echo 1)	& .	Исполняемый файл, вывод: 10	Исполняемый файл, вывод: 1	Корректная работа
12	Корректная трансляция комментариев	Нет	1 . # 123 +	Исполняемый файл, вывод: 1	Исполняемый файл, вывод: 1	Корректная работа
13	Корректная трансляция операции сложения	Нет	1 1 + .	Исполняемый файл, вывод: 2	Исполняемый файл, вывод: 2	Корректная работа
14	Корректная трансляция операции вычитания	Нет	1 2 - .	Исполняемый файл, вывод: −1	Исполняемый файл, вывод: −1	Корректная работа

В таблице выше приведен фрагмент сквозных тестов. Остальные тесты составлены аналогичным образом.

3.3 Структурное тестирование компонента парсера

При использовании метода покрытия операторов должны быть тесты, покрывающие все операторы тестируемого компонента. В таблице 8 приведены результаты структурного тестирования.

Таблица 8 — Модульные тесты для компонента парсера

№	Покрываемые операторы	Код на исходном языке	Фактический результат	Ожидаемый результат	Вывод
1	Подпрограмма аксиомы	Пустая строка	{} (Пустое АСТ)	{} (Пустое АСТ)	Корректная работа
1	Подпрограмма аксиомы	;	Ошибка, неизвестный символ	Ошибка, неизвестный символ	Корректная работа
1	Подпрограмма комментариев	1 # foo	{ 1 }	{ 1 }	Корректная работа
2	Подпрограмма int	42	{ int(42) }	{ int(42) }	Корректная работа
3	Подпрограмма add	1)	{ add }	{ add }	Корректная работа
4	Подпрограмма sub	–	{ sub }	{ sub }	Корректная работа
5	Подпрограмма mul	*	{ mul }	{ mul }	Корректная работа
6	Подпрограмма div	/	{ div }	{ div }	Корректная работа
7	Подпрограмма print	.	{ print }	{ print }	Корректная работа
8	Подпрограмма dup	dup	{ dup }	{ dup }	Корректная работа
9	Подпрограмма drop	drop	{ drop }	{ drop }	Корректная работа
10	Подпрограмма take	take	{ take }	{ take }	Корректная работа
11	Подпрограмма list	[1]	{ list }	{ list }	Корректная работа

Продолжение таблицы 8

12	Подпрограмма apply	!	{ apply }	{ apply }	Корректная работа
13	Подпрограмма and	and	{ and }	{ and }	Корректная работа
14	Подпрограмма or	or	{ or }	{ or }	Корректная работа
15	Подпрограмма not	not	{ not }	{ not }	Корректная работа
16	Подпрограмма equals	==	{ equals }	{ equals }	Корректная работа
17	Подпрограмма not_equals	!=	{ not_equals }	{ not_equals }	Корректная работа
18	Подпрограмма less	<	{ less }	{ less }	Корректная работа
19	Подпрограмма greater	>	{ greater }	{ greater }	Корректная работа
20	Подпрограмма less_equals	<=	{ less_equals }	{ less_equals }	Корректная работа
21	Подпрограмма greater_equals	>=	{ greater_equals }	{ greater_equals }	Корректная работа
22	Подпрограмма if	?	{ if }	{ if }	Корректная работа
23	Подпрограмма bool	b	{ bool }	{ bool }	Корректная работа
24	Подпрограмма bind	:foo	{ bind }	{ bind }	Корректная работа
25	Подпрограмма put	foo	{ put }	{ put }	Корректная работа
26	Подпрограмма scan	&	{ scan }	{ scan }	Корректная работа

Как демонстрирует таблица выше, компонент парсера работает корректно и распознает все лексемы.

ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы был спроектирован компилятор для стекого языка с синтаксисом на основе обратной польской записи.

Цель приложения заключается в создании исполняемых, объектных файлов и ассемблерных листингов на основе кода на исходном языке. В ходе работы был проведен анализов процессов, определена структура программного решения, созданы грамматики синтаксиса интерфейса и исходного языка, разработаны алгоритмы, реализован парсер исходного языка. В результате создан программный продукт, полностью удовлетворяющий техническому заданию.

Тестирование подвердило надежность и корректность реализации функциональности в приложении. Сквозные тесты, написанные при помощи метода эквивалентного разбиения, гарантируют работоспособность всего приложения, как черного ящика, и создание клиентского опыта, соответствующего ожиданиям. Модульные тесты парсера проверяют на ошибки работу парсера и его соотвествие требованиям, а также ускоряют разработку за счет автоматического тестирования сложного компонента. Предоставляемая компилятором гарантия корректности при работе с памятью повысила качество программного обеспечения.

Дальнейшее развитие решения предполагает внедрение оптимизаций, улучшающих быстродействие и потребление памяти генерируемых программ, а также расширение стандартной библиотеки.

В процессе был получен опыт разработки прикладного приложения-компилятора с использованием языка программирования Rust, тестирования при помощи сквозных тестов, создания парсеров и кодогенераторов, консольных интерфейсов. Также закреплен материал дисциплины «Технология разработки программных систем».

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Спецификация ассемблера архитектуры x64 [электронный ресурс]. URL: https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf (дата обращения: 05.10.2024)
2. Философия UNIX [электронный ресурс]. URL: http://www.catb.org/esr/writings/taoup/html/ch01s06.html (дата обращения: 03.10.2024)
3. Документация Rust [электронный ресурс]. URL: https://www.rust-lang.org/learn (дата обращения: 29.09.2024)
4. Иванова Г.С. Технология программирования: Учебник для вузов. 3-е изд., стер. – М.: КНОРУС, 2018. 334 с.
5. Документация библиотеки Clap [электронный ресурс]. URL: https://docs.rs/crate/clap/latest (дата обращения: 15.10.2024)
6. Документация библиотеки Nom [электронный ресурс]. URL: https://docs.rs/crate/nom/latest (дата обращения: 05.10.2024)
7. Идиомы проектирования интерфейсов командной строки [электронный ресурс]. URL: https://clig.dev/ (дата обращения: 06.10.2024)
8. Иванова Г.С. Лекции по дисциплине "Машино-зависимые языки и основы компиляции"

ПРИЛОЖЕНИЕ А

ТЕХНИЧЕСКОЕ ЗАДАНИЕ

Листов 8

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ **ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ**

КАФЕДРА **КОМПЬЮТЕРНЫЕ СИСТЕМЫ И СЕТИ (ИУ6)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.01 Информатика и вычислительная техника**

БАКАЛАВРСКАЯ ПРОГРАММА **09.03.01/03 Вычислительные машины, комплексы, системы и сети**

КОМПИЛЯТОР ДЛЯ ЯЗЫКА ПРОГРАММИРОВАНИЯ НА ОСНОВЕ ОБРАТНОЙ ПОЛЬСКОЙ ЗАПИСИ

Техническое задание на курсовую работу
по дисциплине Технология разработки программных систем

Листов 8

Студент _гр. ИУ6-53Б__ _____ __В.К. Залыгин____
(Группа) (Подпись, дата) (И.О. Фамилия)

Руководитель курсовой работы,
(Ст. преподаватель) _____ __Б.И. Бычков____
(Подпись, дата) (И.О. Фамилия)

Москва, 2024

1 ВВЕДЕНИЕ

Настоящее техническое задание распространяется на разработку компилятора для стекового языка программирования (далее — исходный язык) с синтаксисом на основе обратной польской записи (postfix language compiler) [PLC]. Решение могут использовать разработчики, создающие программы на исходном языке с целью описания алгоритмов обработки данных над целыми числами, а также компиляции кода в объектные и исполняемые файлы под машины с операционной системой семейства Linux и архитектурой процессора x86-64, генерации ассемблерных листингов.

Стековые языки программирования используются в сферах, требующих высокой производительности и минимализма. Стековая архитектура, лежащая в основе данных языков, позволяет оперировать данными, хранящимися на стеке, и последовательно выполнять операции без необходимости использовать переменные. Вследствие этого код на стековых языках часто более лаконичен и прост для анализа. Также такой подход делает языки более гибкими, позволяя создавать и комбинировать функции как объекты первого класса.

По сравнению с аналогичными компиляторами (forth, joy, cat) преимуществом данной разработки является простота исходного языка и малое количество синтаксических конструкций, что позволяет программисту быстро освоить парадигму исходного языка и получить практические навыки построения программ.

2 ОСНОВАНИЯ ДЛЯ РАЗРАБОТКИ

Программа PLC разрабатывается по тематике кафедры.

3 НАЗНАЧЕНИЕ РАЗРАБОТКИ

Назначение PLC заключается в трансляции текста на исходном языке в текст на языке ассемблера и сборки программ в объектные и исполняемые файлы. Разработчики

могут использовать компилятор для составления программ обработки данных над целыми числами.

4 ТРЕБОВАНИЯ К ПРОГРАММНОМУ ИЗДЕЛИЮ

4.1 Требования к функциональным характеристикам

4.1.1 Выполняемые функции

– генерация ассемблерного кода в синтаксисе Intel из текстов на исходном языке;

– создание объектных файлов из текстов на исходном языке;

– создание исполняемых файлов из текстов на исходном языке.

4.1.2 Исходные данные

Исходные данные должны быть представлены текстом программы на исходном языке и флагами, указывающими на тип выходного файла (ассемблерный листинг, объектный файл, исполняемый файл) и его имя.

Исходный язык должен поддерживать следующие конструкции:

- сохранение целых 32-разрядных чисел со знаком (элементов стека) в стек операндов;

- арифметические операции с целыми числами (сложение, вычитание, умножение, деление целочисленное);

- операции работы со стеком (дублирование, удаление, перемещение элементов на вершину элементов стека);

- операция создания списка команд;

- операция применения списка команд;

- ветвление;

- операции сравнения чисел (равенство, неравенство, больше, меньше);

- операция преобразования числа в булево значение;

- операции над булевыми значениями (и, или, не);

- операция присваивания имени элементу;

- операция добавления элемента по имени на стек;

- операция вывода элемента в стандартный поток вывода;

- операция ввода элемента из стандартного потока ввода;

4.1.3 Результаты:

– в случае успешной операции — файл с ассемблерным листингом или объектный файл, или исполняемый файл;

– в случае неуспешной операции — ошибка с описанием проблемы (ошибка в синтаксисе текста программы на исходном языке, ошибка отсутствия необходимых зависимостей компилятора).

4.2 Требования к надежности

4.2.1 Предусмотреть контроль синтаксической корректности текста на исходном языке.

4.2.3 Предусмотреть контроль консистентности флагов, передаваемых при вызове программы-компилятора.

4.3 Условия эксплуатации

Условия эксплуатации в соответствии с СанПиН 2.2.2/2.4.1340-03.

4.4 Требования к составу и параметрам технических средств

4.4.1 Программное обеспечение должно функционировать на IBM-совместимых персональных компьютерах.

4.4.2 Минимальная конфигурация технических средств, на которых развернут компилятор:

4.4.2.1 Архитектура процессора x86-64

4.4.2.2 Количество ядер процессора ..... 1 Шт.

4.4.2.3 Объем ОЗУ ..... 1 Гб.

4.4.3 Требования к конфигурации технических средств, на которых исполняется скомпилированная программа: процессор должен поддерживать набор команд x86-64.

4.5 Требования к информационной и программной совместимости

4.5.1 Программное обеспечение должно работать под управлением операционных систем семейства GNU/Linux.

4.5.2 Программное обеспечение должно иметь интерфейс командной строки.

5 ТРЕБОВАНИЯ К ПРОГРАММНОЙ ДОКУМЕНТАЦИИ

5.1 Разрабатываемые программные модули должны быть самодокументированы, т.е. тексты программ должны содержать все необходимые комментарии.

5.2 В состав сопровождающей документации должны входить:

5.2.1 Расчетно-пояснительная записка на 25-30 листах формата А4 (без приложений 5.2.2 и 5.2.3).

5.2.2 Техническое задание (Приложение А).

5.2.3 Руководство программиста (Приложение Б).

5.3 Графическая часть должна быть включена в расчетно-пояснительную записку в качестве иллюстраций:

5.3.1 Схема структурная программного обеспечения.

5.3.2 Схемы алгоритмов.

5.3.3 Функциональная диаграмма программного обеспечения.

5.3.4 Синтаксическая диаграмма грамматики исходного языка.

5.3.5 Диаграмма вариантов использования.

5.3.6 Синтаксическая диаграмма консольного интерфейса приложения в виде РБНФ.

5.3.7 Таблицы тестов.

6 СТАДИИ И ЭТАПЫ РАЗРАБОТКИ

Этап	Содержание этапа	Сроки и объем	Представляемые результаты	
			Спецификации и программный продукт	**Документы**
1.	Выбор темы, составление задания, решение организационных вопросов	1..2 недели (10 %)	-	**Заполненный бланк задания на курсовую работу – вывешивается на сайт кафедры для получения утверждающей подписи заведующего кафедрой**
2.	Анализ предметной области, разработка ТЗ. Исследование методов решения, выбор основных проектных решений	3..4 недели	Результаты декомпозиции предметной области. Эскизный проект: интерфейс, схемы, возможно, часть программы (выбранные готовые решения).	Фрагмент расчетно-пояснительной записки с обоснованием выбора средств и подходов к разработке
3.	**Сдача ТЗ**	**4 неделя (25 %)**	**-**	**Техническое задание – утверждается руководителем**
4.	Проектирование и реализация основных компонентов – ядра программы	5..7 недели	Технический проект основной части: структура программы, алгоритмы программ. Программный продукт, реализующий основные функции (демонстрируется руководителю)	Фрагмент расчетно-пояснительной записки с обоснованием разработанных спецификаций Тексты части программного продукта, реализующего основные функции.
5.	**Сдача прототипа программного продукта**	**7 неделя (50 %)**	**Прототип программного продукта – демонстрируется руководителю**	
6.	Разработка компонентов, обеспечивающих функциональную полноту	8..10	Рабочий проект программы. Готовая программа	Черновик расчетно-пояснительной записки. Тексты программного продукта.
7.	**Сдача программного продукта**	**11 неделя (75 %)**	**Готовая программа – оценивается руководителем в баллах**	**-**
8.	Тестирование программы и подготовка документации	12..14	Тесты и результаты тестирования.	РПЗ и Руководство пользователя.
9.	**Оформление и сдача документации**	**14 неделя (90 %)**	**–**	**Расчетно-пояснительная записка и Руководство пользователя – проверяются и подписываются руководителем**

Этап	Содержание этапа	Сроки и объем	Представляемые результаты	
			Спецификации и программный продукт	**Документы**
10.	Защита курсовой работы	15..16 недели (100%)	–	Доклад (3-5 минут). Защита курсовой работы. Подписанная документация – вывешивается на сайт кафедры

7 ПОРЯДОК КОНТРОЛЯ И ПРИЕМКИ

7.1 Порядок контроля

Контроль выполнения осуществляется руководителем еженедельно.

7.2 Порядок защиты

Защита осуществляется комиссии преподавателей кафедры.

7.3 Срок защиты

Срок защиты: 15-16 недели.

8 ПРИМЕЧАНИЕ

В процессе выполнения работы возможно уточнение отдельных требований технического задания по взаимному согласованию руководителя и исполнителя

ПРИЛОЖЕНИЕ Б

РУКОВОДСТВО ПРОГРАММИСТА

Листов 5

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ _ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ_____

КАФЕДРА _КОМПЬЮТЕРНЫЕ СИСТЕМЫ И СЕТИ_____

НАПРАВЛЕНИЕ ПОДГОТОВКИ __**09.03.01 Информатика и вычислительная техника**

Компилятор для языка программирования на основе обратной польской записи

Руководство программиста

Листов 5

Студент ИУ6-53Б _____ В.К. Залыгин
(Группа) (Подпись, дата) (И.О. Фамилия)

Руководитель курсовой работы _____ Б.И. Бычков
(Ст. преподаватель) (Подпись, дата)

1 Общие сведения о программном продукте

Настоящий документ был сформирован специально для программного обеспечения компилятора языка с синтаксисом на основе обратной польской записи (далее – исходный язык). Компилятор предоставляет возможности компиляции программ, написанных на исходном языке, в исполняемые файлы под набор команд x64 и операционную систему Linux.

2 Требования к системе

Компьютер, на котором запускается компилятор, должен работать под управлением операционной системой Linux и иметь установленные программы nasm и ld.

Компьютер, на котором запускаются скомпилированные компилятором исполняемые файлы, должен работать под управлением ОС Linux и поддерживать набор команд x64.

3 Описание установки и запуска

Для debian-like дистрибутивов ОС Linux достаточно выполнить следующее:

1) убедиться, что в системе установлены sh и curl;

2) открыть терминал;

3) выполнить команду curl -sSfL https://raw.githubusercontent.com/vzalygin/plc/refs/heads/master/install.sh | sh

4) дождать завершения установки и закрыть терминал.

Для остальных дистрибутивов необходимо выполнить следующее:

1) установить сборщик пакетов cargo по официальной инструкции (ссылка на официальную инструкцию: https://doc.rust-lang.ru/book/ch01-01-installation.html);

2) склонировать репозиторий с компилятором (ссылка на репозиторий: https://github.com/vzalygin/plc);

3) выполнить команду cargo build --refease в корневой папке репозитория

4) дождаться сборки компилятора;

5) переместить собранный исполяемый файл mv target/release/plc /usr/bin.

4 Инструкция по работе

Обращение к программе происходит через терминал с помощью команд, начинающихся с plc. После имени plc возможно указать несколько флагов. Если данный набор флагов недопустим, то компилятор выведет сообщение об этом. Компилятор поддерживает несколько флагов, регулирующих его режим работы:

– -h (длинный --help), вызов информационного сообщения;

– -V (длинный --version), вызов сообщения с версией компилятора;

– -c (длинный --assemble-only), только ассемблирование в ассемблерный листинг;

– -S (длинный --compile-only), только создание объектного файла, без компоновки;

– -o (длинный --output), указание пути до выходного файла, путь необходимо написать после флага;

При отсутствии флагов компилятор при вызове собирает исполняемый файл по пути ./output.

5 Описание исходного языка

Исходный язык имеет синтаксис на основе обратной польской записи, что означает, что в языке оператор идет после своих операндов. Исходный язык относится к стековым языкам, поскольку оперирует одним глобальным хранилищем – стеком, который поддерживает 2 операции, положить значение и вытащить значение. Язык поддерживает операции над 32-битными целочисленными значениями со знаком, над бинарными значениями (бинарное подмножество целочисленных значений – 0 и 1), над списками операций. Список доступных операций с их описанием приведен ниже:

– «число», положить число на стек;

– «.», взять число со стека и подать его на stdout;

– «&», прочитать число из stdin и положить его на стек;

– «+», взять 2 числа со стека и сложить, результат положить на стек:

– «-», взять 2 числа со стека и вычесть, результат положить на стек, вычитаемым является число;

– «*», взять 2 числа со стека и умножить, результат положить на стек;

– «/», взять 2 числа со стека и разделить целочисленно, результат положить на стек, делителем является число;

– «dup», взять элемент со стека, сгенерировать такой же элемент и положить оба элемента на стек;

– «drop», взять эемент со стека и выбросить его;

– «take», взять число со стека и вытащить из стека на вершину стека элемент, находящийся на глубине, равной вытащенному числу;

– «[операции]», сформировать список операций и положить его на стек;

– «!», взять список со стека и положить операции внутри него на стек;

– «b», взять число и преобразовать его в бинарное значение по правилу, что если число равно 0, то бинарное значение тоже равно 0, иначе бинарное значение равно 1;

– «==», взять два числа со стека и сравнить их, положить бинарное значение, означающее верность их равенства;

– «!=», взять два числа со стека и сравнить их, положить бинарное значение, означающее верность их неравенства;

– «>», взять два числа и сравнить их, положить бинарное значение, означающее веронсть отношения «больше» между ними;

– «>=», взять два числа и сравнить их, положить бинарное значение, означающее веронсть отношения «больше или равно» между ними;

– «<», взять два числа и сравнить их, положить бинарное значение, означающее веронсть отношения «меньше» между ними;

– «<=», взять два числа и сравнить их, положить бинарное значение, означающее веронсть отношения «меньше или равно» между ними;

– «and», взять два элемента и выполнить поразрядное И между ними, результат положить на стек (если элементы – бинарные значения, операция между ними соответствует логическому И);

– «or», взять два элемента и выполнить поразрядное ИЛИ между ними, результат положить на стек (если элементы – бинарные значения, операция между ними соответствует логическому ИЛИ);

– «not», взять число со стека и выполнить поразрядное НЕ, результат положить на стек;

– «?», взять со стека бинарное значение и два других элемента, если бинарное значение означает верно, то положить на стек последний вытащенный элемент, иначе положить предпоследний вытащенный элемент;

– «:имя», взять элемент со стека и привязать к нему указанное имя;

– «имя», положить на стек элемент, привязанный к указанному имени.

Для всех операций, берущих со стека несколько операндов, верно, что первым операндом является последний вытащенный из стека элемент.

6 Пример программы на исходном языке

Листинг Б.1 содержит в себе пример программы на исходном языке, вычисляющей факториал:

Листинг Б.1 — Пример программы, вычисляющей факториал

```
[
    dup               # dup N for the next iterations
    [                 # if current N greater than 1
        dup 1 -       # copy N and decrement for the next
iteration
        fac!          # call the next iteration
        *             # after call multiply current N and the
product of the next iterations
    ]
    [                 # if current N is 1
                      # then do nothing
    ]
    [2 take 1 > ]!    # get current N and compare is that
greater then 1
    ?!                # execute if and chosen branch
] :fac                # bind a list to a "fac" name

&                     # input N
fac!                  # put and apply the list
.                     # print the result
```

ПРИЛОЖЕНИЕ В

ФРАГМЕНТ КОДА

Листов 4

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ _ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ_____

КАФЕДРА _КОМПЬЮТЕРНЫЕ СИСТЕМЫ И СЕТИ_____

НАПРАВЛЕНИЕ ПОДГОТОВКИ __**09.03.01 Информатика и вычислительная техника**

Компилятор для языка программирования на основе обратной польской записи

Фрагмент исходного текста программы

Листов 4

Студент ИУ6-53Б _____ В.К. Залыгин
(Группа) (Подпись, дата) (И.О. Фамилия)

Руководитель курсовой работы _____ Б.И. Бычков
(Ст. преподаватель) (Подпись, дата)

Москва 2024 г.

Исходный код программы расположен в открытом репозитории: https://github.com/vzalygin/plc.

В листинге В.1 в качестве фрагмента исходного кода приведено содержимое компонента сборки.

Листинг В.1 — Фрагмент исходного текста программы

```
use anyhow::{anyhow, Result};
use std::{
    env,
    fs::File,
    io::Write,
    path::{Path, PathBuf},
    process::Command,
};

use crate::translator::Asm;

const TMP_SUBDIR: &str = "plc";

pub fn link_to_executable_file<'a>(
    object_files_paths: &'a [&Path],
    output_path: &'a Path,
) -> Result<&'a Path> {
    {
        let output_path = output_path
            .to_str()
            .ok_or(anyhow!("path contains non-utf8
characters"))?;

        let mut ld_command = Command::new("ld");
        let mut ld_command = ld_command
            .args(["-dynamic-linker", "/lib64/ld-linux-
x86-64.so.2"])
            .args(["-o", output_path])
            .arg("-lc");

        for object_files_path in object_files_paths {
            ld_command = ld_command.arg(
                object_files_path
                    .to_str()
                    .ok_or(anyhow!("path contains non-utf8
characters"))?,
            )
        }

        let ld_exit_code: std::process::ExitStatus =
```

Продолжение листинга B.1

```
ld_command.status()?;

        if !ld_exit_code.success() {
            let msg = match ld_exit_code.code() {
                Some(code) => format!("ld returned code
{code}"),
                None => "ld was interrupted".to_string(),
            };

            return Err(anyhow!(msg));
        }
    }

    Ok(output_path)
}

pub fn make_object_file<'a>(asm_file_path: &'a Path,
output_path: &'a Path) -> Result<&'a Path> {
    {
        let output_path = output_path
            .to_str()
            .ok_or(anyhow!("path contains non-utf8
characters"))?;
        let asm_file_path = asm_file_path
            .to_str()
            .ok_or(anyhow!("path contains non-utf8
characters"))?;

        let nasm_exit_status = Command::new("nasm")
            .args(["-f", "elf64"])
            .arg("-w-orphan-labels") // TODO: set only in
release mode
            .args(["-o", output_path])
            .arg(asm_file_path)
            .status()?;

        if !nasm_exit_status.success() {
            let msg = match nasm_exit_status.code() {
                Some(code) => format!("nasm returned code
{code}"),
                None => "nasm was interrupted".to_string(),
            };

            return Err(anyhow!(msg));
        }
    }
```

Продолжение листинга B.1

```
    Ok(output_path)
}

pub fn make_asm_file(asm: Asm, output: &Path) -> Result<&Path>
{
    let code = asm.into_assembly();

    let mut file = File::create(output)?;

    let _ = file.write(code.as_bytes())?;

    Ok(output)
}

pub fn make_tmp_path() -> PathBuf {
    env::temp_dir()
        .join(TMP_SUBDIR)
        .join(uuid::Uuid::new_v4().to_string())
}

pub fn check_tmp_dir() -> Result<()> {
    std::fs::create_dir_all(
      env::temp_dir().join(TMP_SUBDIR)).map_err(|e| e.into()
    )
}
```