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.

РЕФЕРАТ

Расчетно-пояснительная записка состоит из 29 страниц, включающих в себя 15 рисунков, 4 таблиц, 0 источников и 2 приложения.

КОМПИЛЯТОР, СТЕКОВЫЙ ЯЗЫК, ОБРАТНАЯ ПОЛЬСКАЯ ЗАПИСЬ, LINUX, АРХИТЕКТУРА 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.

СОДЕРЖАНИЕ

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 опарируют 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 за счет 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 кода исходного языка;

– сборка объектный файлов из кода исходного языка;

– составление ассемблерных листингов кодов на исходном языке.

При сравнении с существующими решениями 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 задания и выбор технологии, языка и среды разработки

В соответствии с требованиями технического задания необходимо разработать программу, которая транслироватьк коды 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. 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. 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). Указанные особенности Rust позволяют писать безопасное решение и недопускать ошибки в программном обеспечении. В таблице 1 показаны результаты сравнения языков программирования.

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

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.

Таблица 1 — 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

При создании программного обеспечения целесообразно проводить разработку нисходящим способом. 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 влития.

Для создания интерфейса командной строки рационально использовать готовую библиотеку описания интерфейса – Clap. Для разбора исходных кодов можно использовать комбинаторный подход и библиотеки, предоставляющие набор компонентов для построения генераторов комбинаторых парсеров. В данном случае используется библиотека Nom. С целью ускорения разработки рационально использовать готовые решения для ассемблирования и компоновки. Под целевую платформу (Linux x64) одними из самых распространенных

являются ассемблер nasm и компоновщик ld, они используются в рамках данной разработки.

Характеристики разработки показаны в таблице 2.

Таблица 2 — Характеристики разработки

Характеристика	Значение
Язык программирования	Rust
Среда разработки	Visual Studio Code
Система контроля версий	Git
Используемые библиотеки и зависимости	Clap, Nom, Nasm, Ld
Подход к разработке	Нисходящий
Поддерживаемые платформы	Linux
Поддерживаемые наборы команд	x64

1.2 Анализ процессов

В соответствии с техническим заданием 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 1 представлена функциональная диаграмма процесса трансляции программы при помощи программного решения.

Рисунок 2 уточняет блок A0, процесс трансляции исходного кода.

Рисунок 3 уточняет блок A3, процесс сборки.

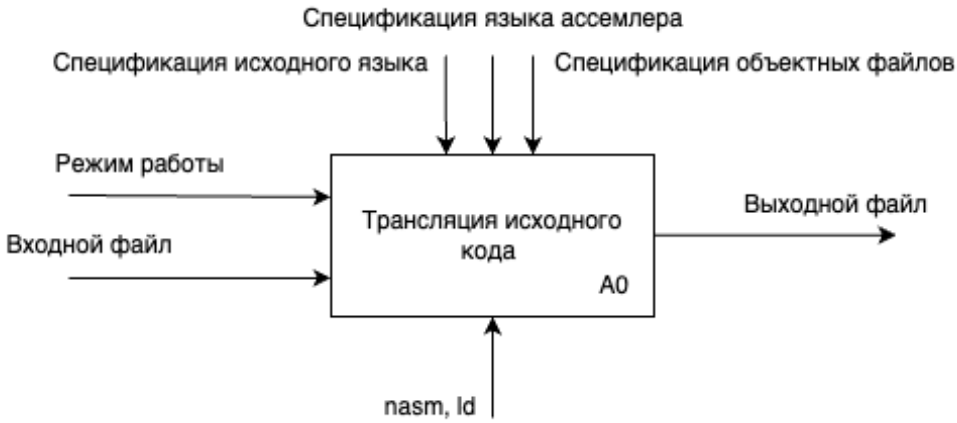

Рисунок 1 — Функциональная диаграмма

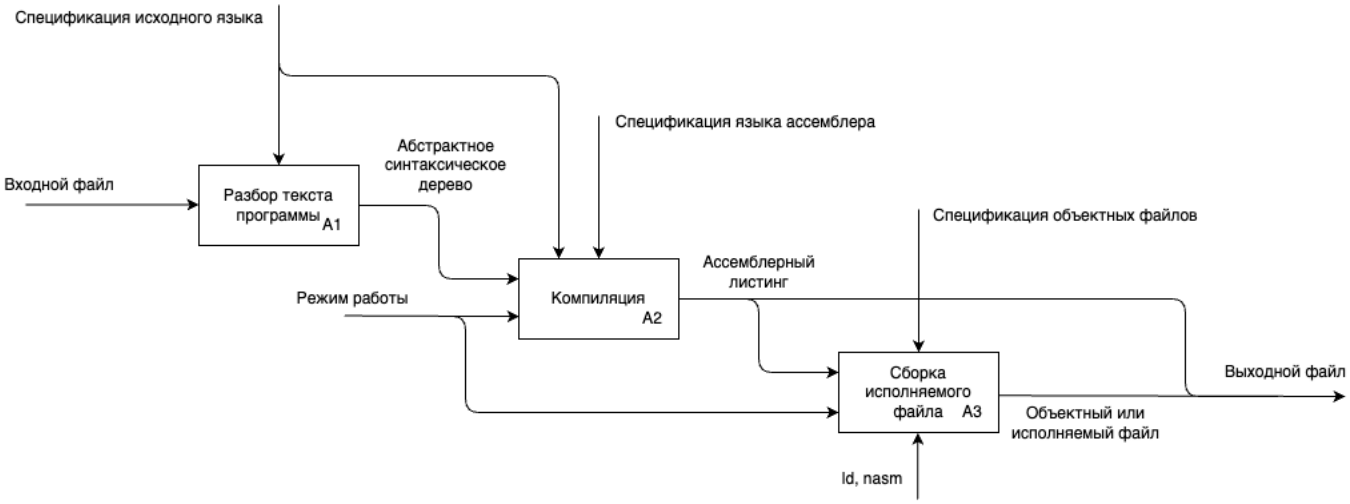

Рисунок 2 — Функциональная диаграмма, уточняющая процесс трансляции

Рисунок 3 — Функциональная диаграмма, уточняющая процесс сборки

1.3 Анализ вариантов использования

Поскольку техническое задание предполагает реализацию различных вариантов использования программы, целесообразно показать их на диаграмме вариантов использования. Рисунок 4 показывает возможности использования программного обеспечения.

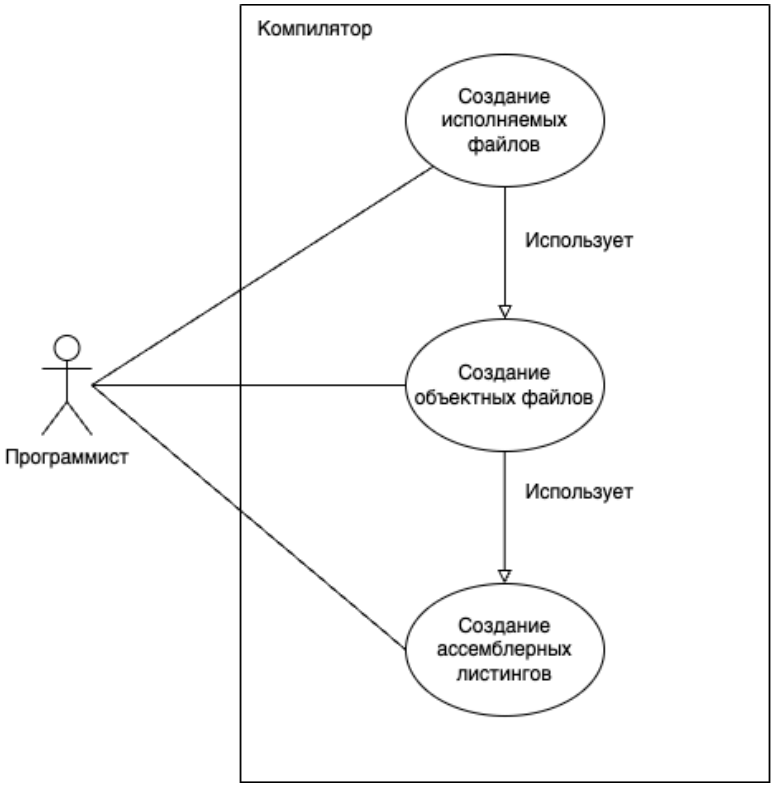

Рисунок 4 — Диаграмма вариантов использования

2 Проектирование структуры и компонентов программного продукта

2.1 Проектирование структуры приложения

Согласно подразделу 1.1 расчетно-пояснительной записки в рамках разработки был выбран структурный подход. Для работы при таком подходе необходимо уточнить структурную схему программного решения. На рисунке 5 изображена структура проекта.

Рисунок 5 — Структурная схема

Описание частей структурной схемы приведено ниже:

– программа-компилятор, главная часть программного решения;

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

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

– компонент разбора текстов, включает в себя подпрограммы для разрбора тестов исходного языка;

– компонент компиляции, содержит подграммы, 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 1.1, необходимо начинать разработку от компонентов верхнего уровня, постепенно спускаясь вниз к компонентам нижних уровней. После уточнения структурной схемы в пункте 2.1 наиболее верним 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. Форма изображена на рисунке 6. Аксиомой грамматики является нетерминал «plc».

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

Рисунок 6 — РБНФ интерфейса

Интерфейс командной строки соответствует принятым идиомам проектирования интерфейсов для консольных приложений. 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 3.

Таблица 3 — Поддерживаемые флаги

Тип флага	Краткая форма флага	Длинная форма флага	Назначение
Информационный	-h	--help	Вывод сообщения с информацией о приложении и доступных действиях
Информаицонный	-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

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.

2.3 Разработка алгоритмов

В подразделе 1.2 описаны функциональные диаграммы процессов, в рамках которых используется решение. Для переноса процессов в программное обеспечение необходимо описать алгоритмы, соответствующие процессам. В качестве представления алгоритом целесообразно использовать схемы алгоритмов. На рисунке 7 представлен алгоритм для основной подпрограммы. В рамках основной подграммы происходит выбор режима, а также вызов агрегирующей подпрограммы «выполнить».

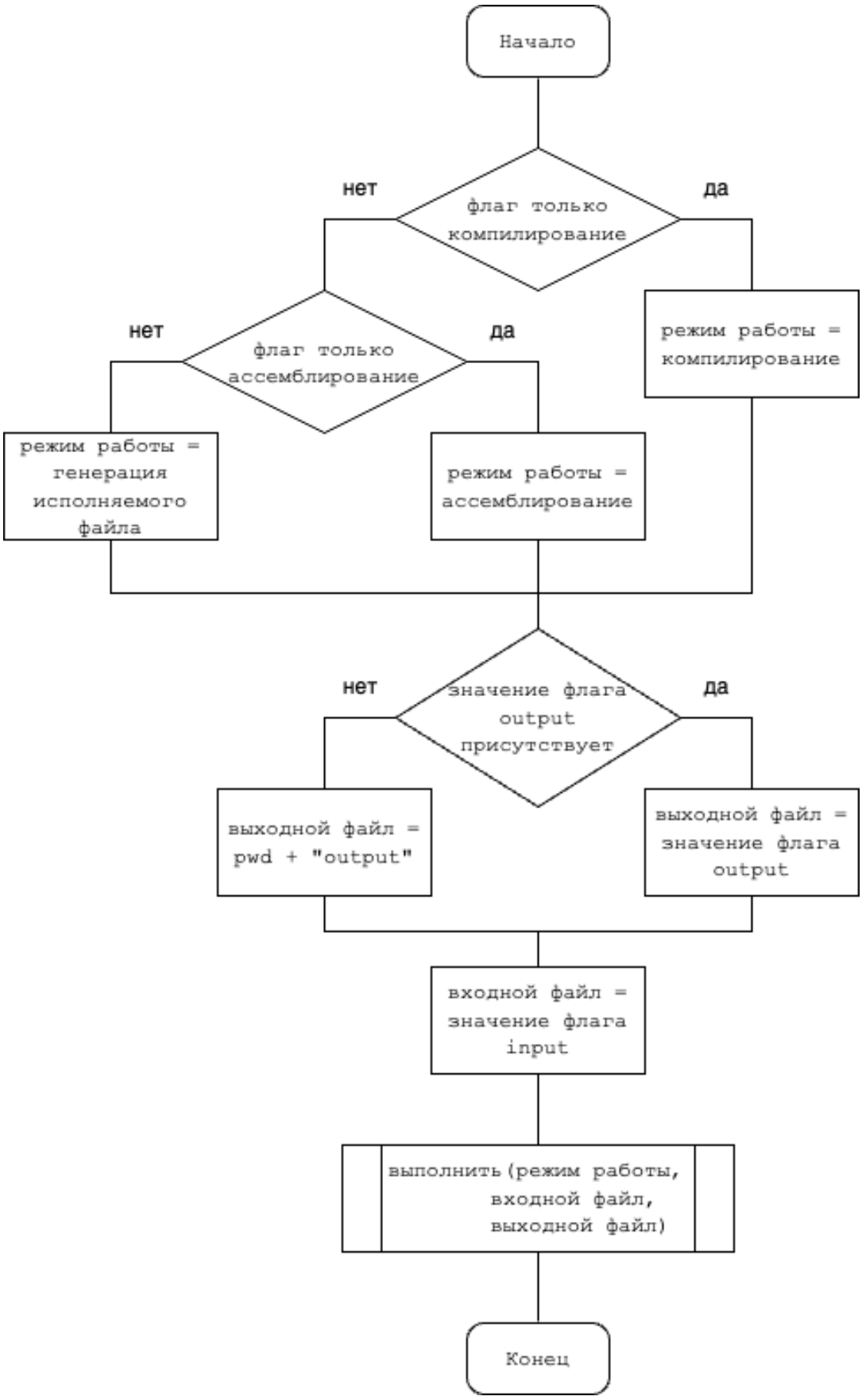

Рисунок 7 — Алгоритм работы основной подпрограммы

Схема алгоритма работы библиотечной подпрограммы «выполнить» представлена на рисунке 8. В зависимости от режима работы подграмма 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.

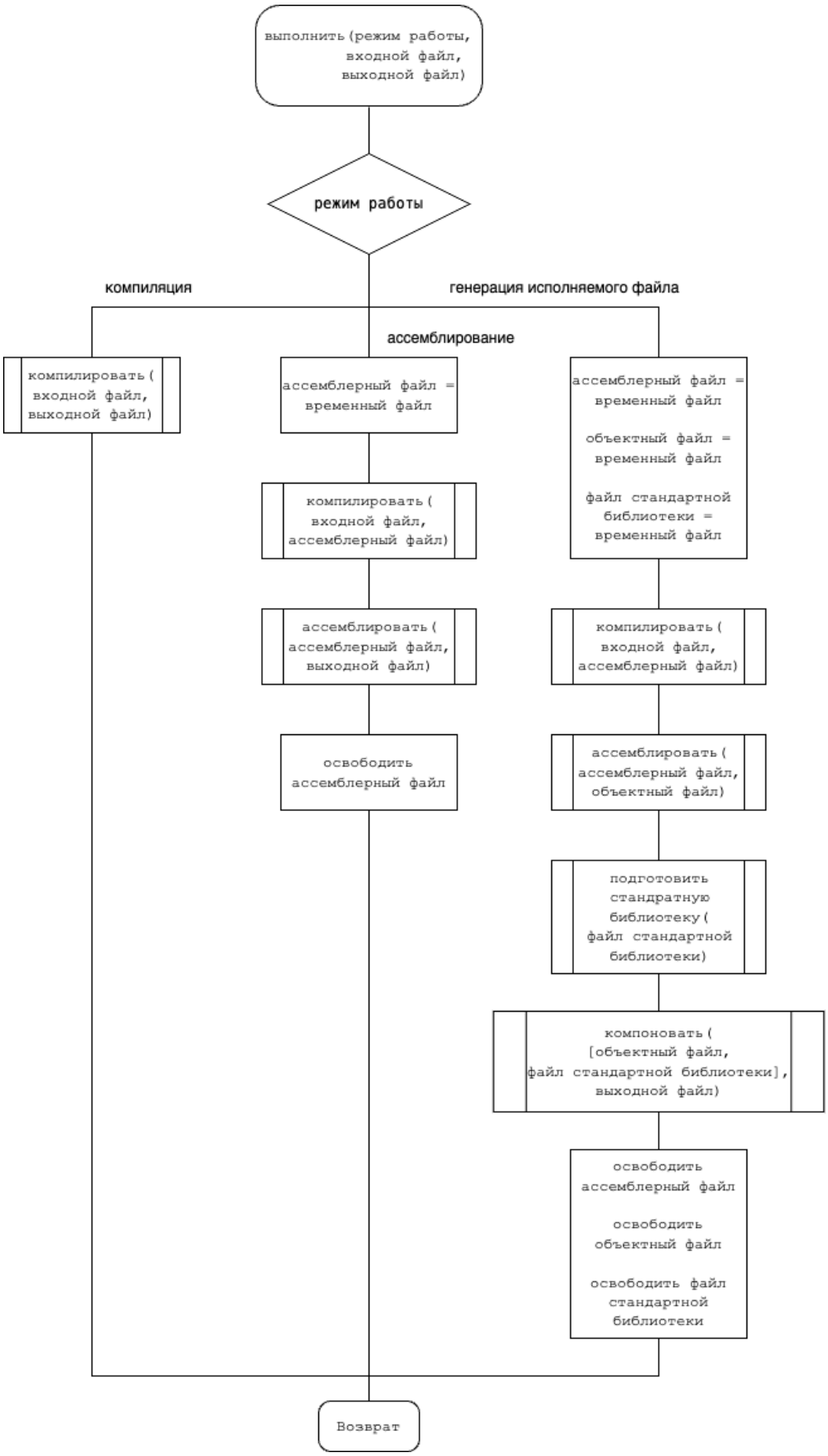

Рисунок 8 — Схема алгоритма подпрограммы «выполнить»

Поскольку язык обладает стандартной библиотекой (подробнее об 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 синтаксиса грамматики исходного языка и парсера

Выбранная библиотека для построения генераторов комбинаторных парсеров, Nom, позволяет реализовать разбор выражений методов рекурсивного спуска. 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.

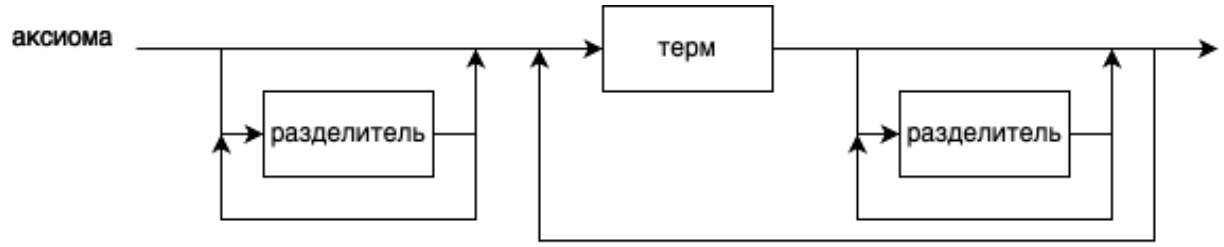

Рисунок 10 — Синтаксическая диаграмма аксиомы исходного языка

Одним из самых 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.

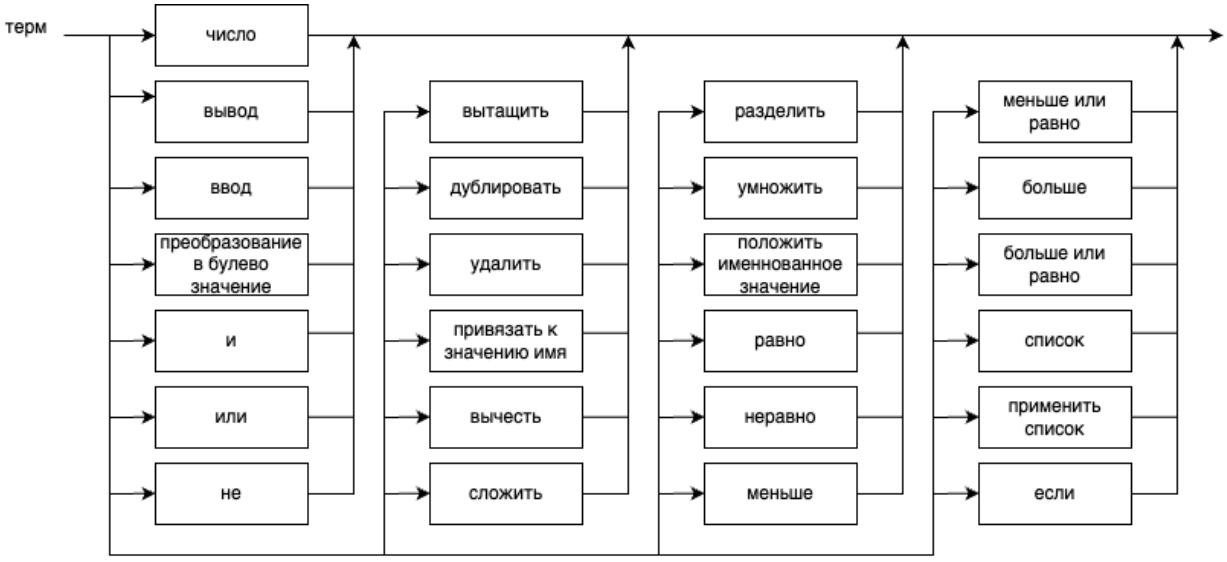

Рисунок 11 — Синтаксическая диаграмма «терм»

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, позволяет языку удовлетворить полноте по Тьюрингу.

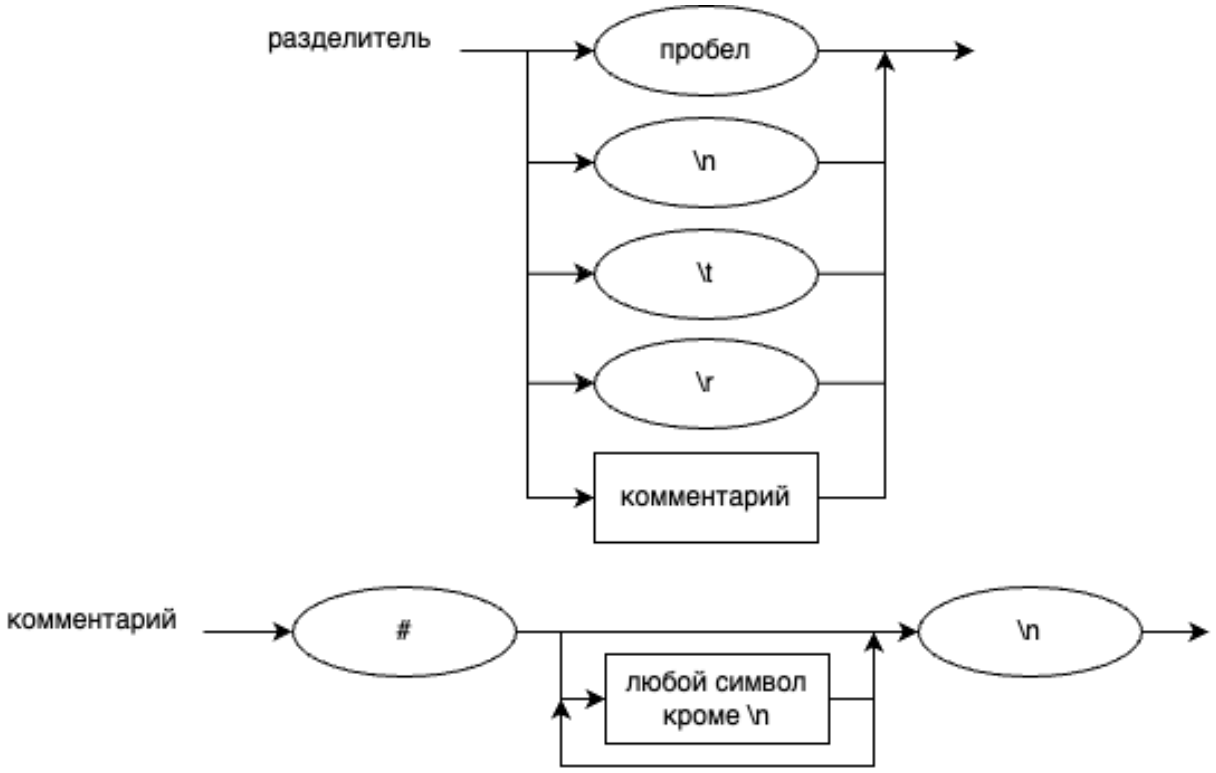

Рисунок 12 — Синтаксические диаграммы «разделитель», «комментарий»

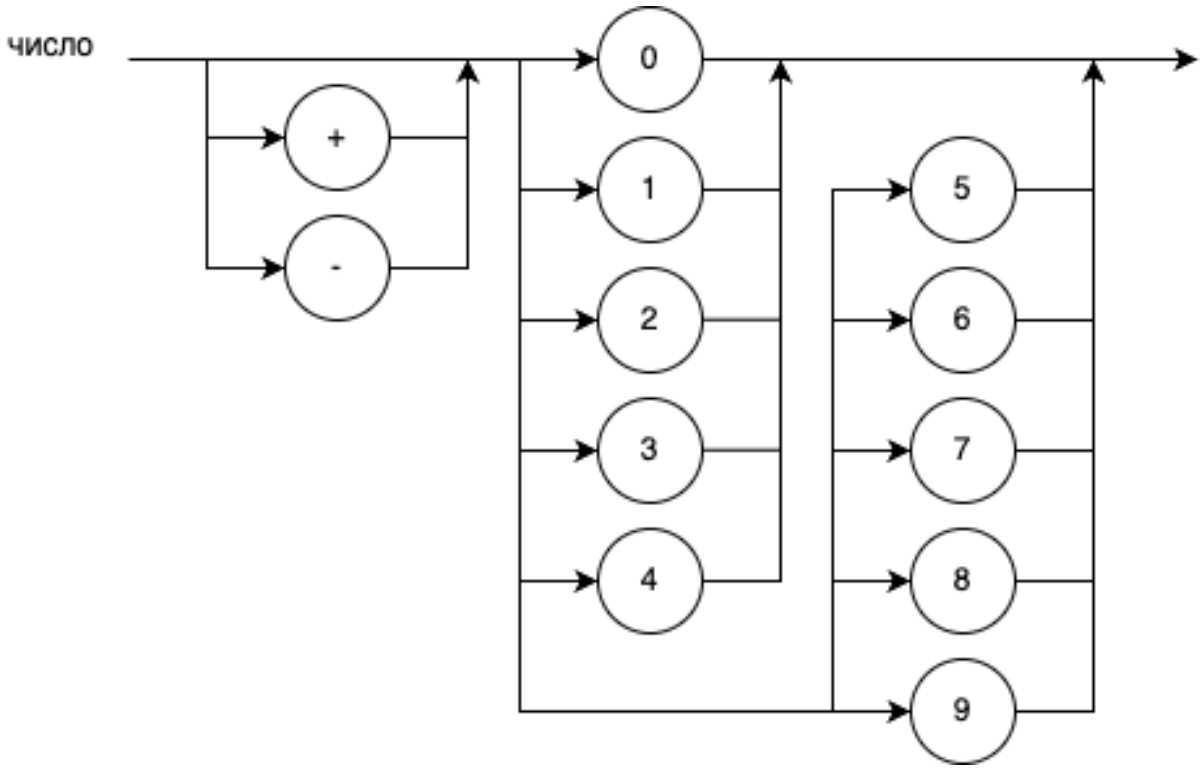

Рисунок 13 — Синтаксическая диаграмма «число»

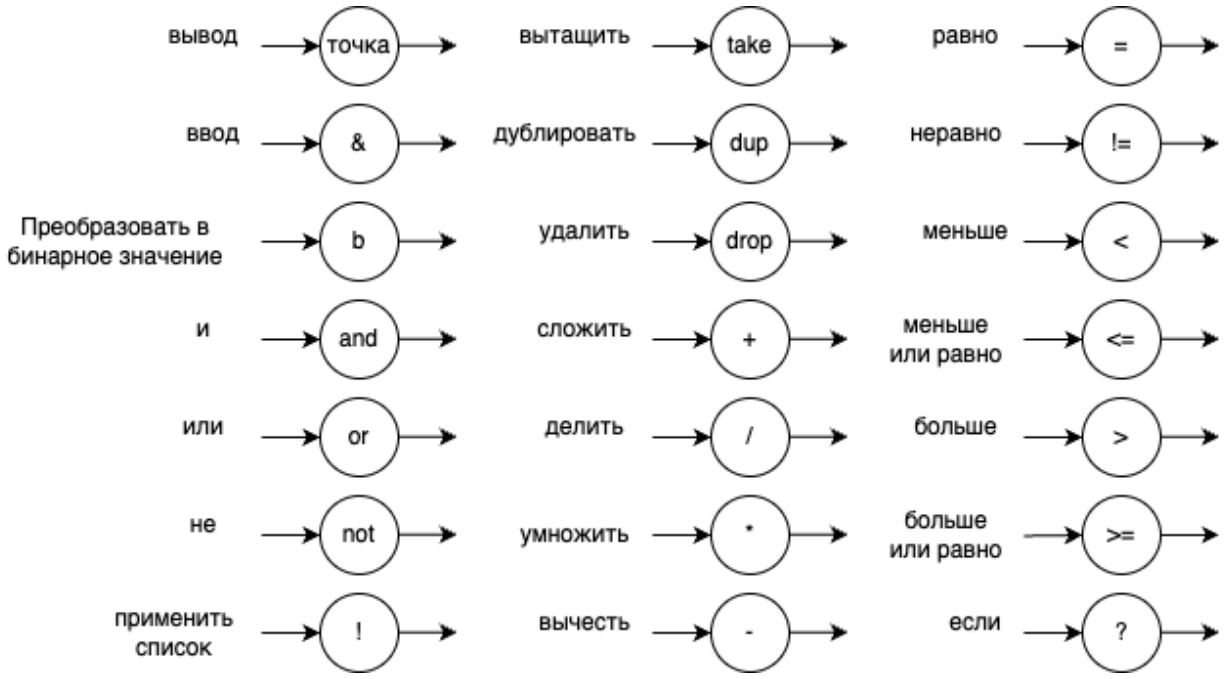

Рисунок 14 — Синтаксические диаграммы правил для некоторых термов

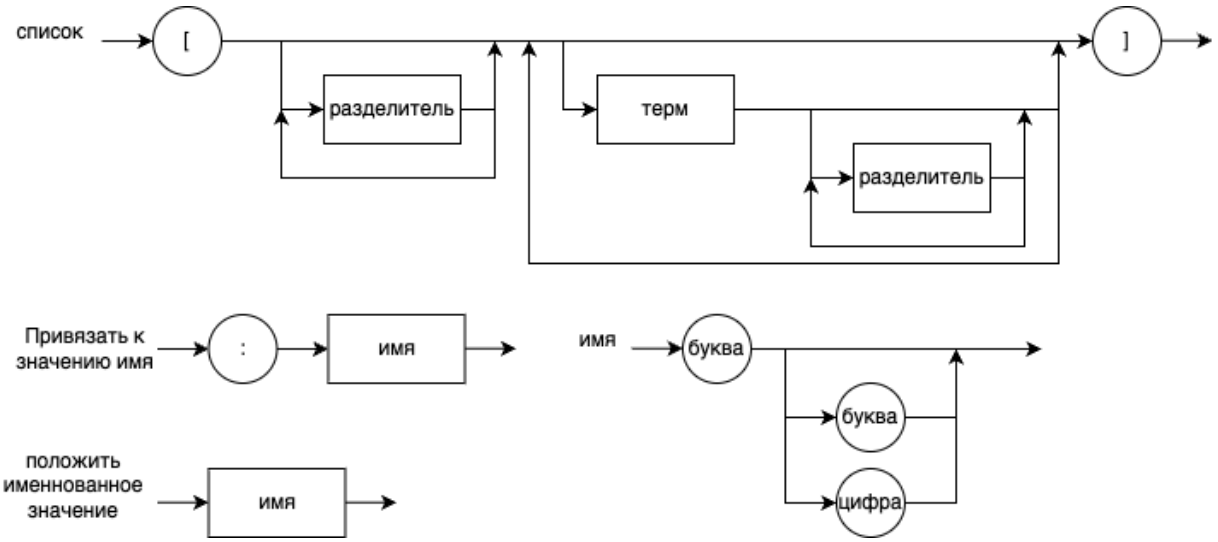

Рисунок 15 — Синтаксические диаграммы правил «список», «имя», «привязать к имени», «положить именнованное значение»

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 конкатенативнен, его абстрактное дерево вырождено в список, элементами которого являются структура, описывающая некоторую операцию (результат разбора парсера term). Результатом разбора исходного кода является абстрактное синтаксическое дерево, которое после отработки парсера передается на следующий этап обработки.

2.5 Разработка подпрограммы-компилятора

2.5.1 Разработка генератора ассемблерных кодов

На основе абстрактного синтаксического дерево, полученного после разбора исходного кода (процесс описан в подразделе 2.4), создается ассем-

блерный листинг программы. Чтобы выполнить данную задачу, необходимую каждой операции сопоставить заготовку (шаблон) на языке ассемблера, удовлетворяющую семантике операции. 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 4.

Таблица 4 — Некоторые операции и 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), 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)),`

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

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, выхода приложения. Библиотека собирается аналогично исходному коду в объектный файл, а затем компонуется с объектным файлом с точкой входа программы.

3 Выбор стратегии тестирования и разработка тестов

ЗАКЛЮЧЕНИЕ

ПРИЛОЖЕНИЕ А

ПРИЛОЖЕНИЕ Б