ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САРАТОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ИМЕНИ ГАГАРИНА Ю.А.»

ИНСТИТУТ ПРИКЛАДНЫХ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И КОММУНИКАЦИЙ

Кафедра «Прикладные информационные технологии»

Отчет по преддипломной практике

Место прохождения практики	Администрация Калининского муниципального района Саратовской области
Время прохождения практики	4 мая 2022 г. по 1 июня 2022 г.

	ФИО	Подпись	Дата
Выполнил студент группы б1-ПИНФ41	Нефедов Данил Вадимович		
Руководитель от кафедры	Бровко Александр Валерьевич		

Итоговая оценка по защите результатов деятельности на практике	

Саратов 2022

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **3**

1 Описание задачи и анализ существующих решений **6**

1.1 Описание предметной области 6

1.2 Постановка задачи 8

1.3 Обзор существующих аналогов 11

1.3.1 8cc 12

1.3.2 Nanopass Framework 12

1.3.3 AMaCC 13

1.3.4 catc 13

1.4 Сравнительный анализ существующих решений 13

2 Описание проектирование системы **16**

2.1 Описание методов компиляции 16

2.1.1 Лексический анализ 16

2.1.2 Синтаксический анализ 17

2.1.3 Построение абстрактного синтаксического дерева . . . 19

2.1.4 Кодогенерация 20

2.2 Проектирование системы 20

2.2.1 Определение информационных потоков системы . . . 20

2.2.2 Описание системных процессов 22

2.2.3 Описание потоков данных 23

2.2.4 Описание пакетов системы 25

2.2.5 Описание компонентов 26

2.2.6 Описание потоков информации по UML 26

3 Описание практической реализации **29**

3.1 Компилятор 29

3.1.1 Система сборки 29

3.1.2 Лексический анализ 32

3.1.3 Синтаксический анализ 33
3.1.4 Построение абстрактного синтаксического дерева . . . 34
3.1.5 Кодогенерация 35
3.1.6 Интерфейс командной строки 35
3.1.7 Экспорт промежуточных представлений в формате JSON 37
3.2 Серверная часть 38
3.2.1 Nginx 39
3.2.2 Go 39
3.2.3 JSON 41
3.3 Графический интерфейс компилятора 42
3.3.1 Выбор средств разработки 42
3.3.2 Описание пользовательского интерфейса 49

ЗАКЛЮЧЕНИЕ **53**

ПРИЛОЖЕНИЕ А **55**

ВВЕДЕНИЕ

Цели преддипломной практики: получение практических навыков выполнения исследований, разработки и реализации архитектуры программного обеспечения в ходе выполнения научно-исследовательской работы в области разработки компиляторов.

Задачи преддипломной практики:

— Анализ предметной области и существующих аналогов программного обеспечения компиляторов, направленных на образование.

— Разработка архитектуры программного обеспечения.

— Разработка программного обеспечения компилятора с возможностью интроспекции всех промежуточных представлений, а также графический пользовательский интерфейс к нему.

— Ознакомление с основными требованиями, предъявляемыми к содержанию и оформлению научно-исследовательских работ.

— Развитие навыков оформления рабочих результатов в виде научно-технических отчетов.

Преддипломная практика направлена на формирование навыковой составляющей следующих компетенций:

— ОК-7: способность к самоорганизации и самообразованию.

— ПК-6: способность собирать детальную информацию для формализации требований пользователей заказчика.

— ПК-24: способность готовить обзоры научной литературы и электронных информационно-образовательных ресурсов для профессиональной деятельности.

Компилятор многим видется как некий «магический артефакт», своеобразный черный ящик, созданный могучими волшебниками-программистами, недоступный для понимания простым смертным.

Книги, посвященные созданию компляторов, внушают благоговейный ужас каждому программисту, так же как «Критика чистого разума» Канта внушает ужас каждому философу. Чрезвычайная академичность материала и необходимость пробираться через тонны математических конструкций и абстрактных построений твердо закрепила за этими книгами и, соответственно, предметной областью, репутацию непреступных крепостей мира компьютерных наук [1].

Серьезные компиляторы чрезвычайно сложны для понимания, включают в себя десятки миллионов строк кода и, порой, десятилетия труда сотен инженеров.

На сегодня существует довольно серьезная проблема в отрасли: *найти специалистов в сфере разработки комплияторов сложно* [2].

Во многом в этом виновата некоторая элитарность, высокий порог вхождения и низкая популярность предметной области относительно современных трендов, например таких как веб-программирование.

Решение данной проблемы должно быть комплексным: необходимо устранить сразу несколько факторов, среди которых можно выделить: а) высокий порог вхождения в предметную область; б) низкая популярность по сравнению с современными «модными» технологиями; в) стереотип о сухости и академичности темы.

Обратим особое внимание на первую проблему из списка и постараемся понять, чем она может быть обусловлена.

Высокий порог вхождения в любою технологию или предметную область среди прочего может быть обусловлен: а) отсутствием доступных учебных материалов; б) необходимостью иметь глубокие базовые знания и навыки, т.н. *background*.

Целью выпускной квалификационной работы поставлено решение проблемы высокого порога вхождения в предметную область.

В качестве средства для достижения цели была поставлена задача создания компилятора языка C, который сможет послужить хорошей отправной точкой в изучении предметной области начинающими специалистами.

1 Описание задачи и анализ существующих решений

1.1 Описание предметной области

Компилятор — это просто программа по преобразованию текста из одного вида в другой. Входной формат текста называется *языком исходного кода*, а выходной — *целевым языком* [3].

Сама необходимость такого преобразования заключается в том, что языком исходного кода является некоторый язык программирования высокого уровня, а в качестве целевого языка выступает язык ассемблера. Программа на языке высокого уровня в абсолютном большинстве случаев просто не может быть напрямую исполнена на CPU, который «понимает» только язык ассемблера.

Обычно процесс компиляции заключается в том, чтобы сперва проанализировать исходный код и построить его *семантическую интерпретацию*, а затем уже *синтезировать* соответствующий код на целевом языке.

Та часть компилятора, которая занимается анализом исходного кода называется *фронтендом*, а часть, которая синтезирует код на целевом языке — *бекендом*. В хорошо спроектированном компиляторе фронтенд не имеет никакого представления о целевом языке, а бекенд — об исходном, единственно общее между ними — это знание семантического представления.

Со временем сложилась устоявшаяся структура как фронтенда, так и бекенда компилятора, ставшая уже классической. Ниже вкратце будут рассмотрены модули этой структуры.

Самым первым модулем фронтенда компилятора считается *лексический анализатор*. Его задача заключается в том, чтобы превратить поток символов исходного текста в ряд отдельных лексических единиц языка, называемых *токенами*. К примеру, таких как: целочисленные константы, идентификаторы, фигурные и круглые скобки.

После обработки входного текста лексическим анализатором полученный поток токенов подается на вход *синтаксическому анализатору*, кото-

рый превращает этот ряд токенов в *синтаксическое дерево*, соответствующее синтаксису исходного языка.

Стоит отметить, что фазы парсинга исходного текста хорошо изучены и формализованы [4]. Существует отдельная область компьютерной лингвистики, которая изучает формальные языки. Особо отметим достижения профессора Ноама Чомского, который сумел составить классификацию всех формальных языков, суммаризованной в иерархии названной его именем — *иерархии Чомского*. Это позволило построить математический аппарат и создать мощные формализмы, способные проводить разбор формальных языков различных видов.

Интересно, что в некоторых компиляторах стадии лексического и синтаксического анализа объединяются. Дело в том, что регулярные языки, которые достаточно мощны для разбора исходного текста на лексические единицы, находятся ниже контекстно-независимых языков в вышеупомянутой иерархии. Иначе говоря, любой регулярный язык является подмножеством некоторого контекстно-независимого. Это позволяет описать лексическую структуру языка прямо в контекстно-независимой грамматике [5].

Следующим шагом итоговое синтаксическое дерево передается на вход *семантическому анализатору*, который выполняет семантический анализ программы, возвращая в качестве результата *абстрактное синтаксическое дерево* (англ. *Abstract Syntax Tree*, AST). Помимо всего прочего, такое дерево лишено излишних деталей синтаксического дерева, которые затрудняют дальнейшую его обработку [6].

Следующей и последней задачей фронтенда является генерация *промежуточного представления* (англ. *intermediate representation*, кратк. *IR*) из AST дерева. Это представление, в отличие от AST, является языконезависимым и подходит для представления кода на разных языках программирования. Обычно IR почти полностью состоит из выражений и операторов потока управления.

С этого этапа начинается работа бекенда компилятора, где на первом этапе над полученным IR выполняются *проходы оптимизации* (англ.

optimization passes) с целью увеличения эффективности кода. Простейшим примером такого прохода является *свертка констант* при котором в выражениях заранее вычисляются операнды, значение которых известно во время компиляции [7].

После завершения всех проходов оптимизации IR передается модулю *кодогенерации*, задача которого состоит в том, чтобы из полученного IR сгенерировать код на целевом языке. Кодогенератор выбирает соответствующие нужные инструкции, распределяет машинные регистры для хранения значений и размещает инструкции в нужном порядке.

После генерации кода на целевом языке компилятор может выполнить так называемые *микро-оптимизации* (англ. *peephole optimizations*) [8], которые изменяют небольшой набор инструкций на эквивалентный набор, который имеет большую производительность.

Наконец, после всех этих манипуляций, компилятор может выдать финальный код на целевом языке.

1.2 Постановка задачи

Разрабатываемый компилятор толжен послужить хорошей отправной точкой для всех, кто пытается разобраться в области построения компиляторов. Несмотря на субъективность формулировки задачи, что вероятно присуще всем работам в области образовательных процессов, мы постараемся выделить конкретные требования к компиляторы.

Первым и довольно очевидным требованием к решению будет его *простота*. Основополагающим принципом при разработке компилятора должен быть так называемый *принцип KISS*. Он был предложен в 1960 военно-морскими силами США и гласит, что большинство систем работают лучше, если они сделаны как можно проще, а следовательно простота должна быть ключевой целью проектирования. Справедливость этого тезиса подтверждается его многочисленными вариациями, всплывающими в культуре на протяжении веков [9].

Следующим требованием выделим *необходимость показать наиболее идеоматичные и устоявшиеся алгоритмы, методы и архитектуру построения компилятора*. Это позволит учащимся лучше понять основополагающие принципы работы программы, а кроме того даст фундаментальные знания предмета, которые в дальнейшем можно будет применить как в индустрии, так и в академической сфере.

Вспомним, что компиляция это преобразование текста с одного языка в другой. Так как обычно эти языки находятся на очень разных уровнях абстракции и в целом сильно отличаются друг от друга, у нас нет никакого способа произвести преобразование напрямую. Поэтому мы придумали промежуточные языки и превратили процесс компиляции в ряд простых преобразований из одного языка в другой.

Таким образом, следующим нашим требованием будет реализовать *возможность просмотра всех промежуточных представлений исходного кода*. Это поможет учащимся получить так называемый *инсайт* о том, что действительно происходит в процессе компиляции.

Теперь нам необходимо выбрать исходный язык для нашего компилятора. Имеет смысл предъявить к языку следующие требования:

1. *Распространенность*. Язык должен быть известен наиболее широкому кругу учащихся. Нельзя написать компилятор языка X, будучи незнакомым с языком X.

2. *Широкое применение*. Язык должен использовать в реальной работе, это не должен быть «игрушечный» язык, иначе учащимся будет просто не интересно работать над ним.

3. *Простота*. Язык должен быть максимально простым и в то же время использоваться в реальной работе.

Под все эти критерии идеально подходит язык C:

1. Язык C является вводным языком для большинства академических курсов, посвященных информатике и компьютерным наукам [10].

2. Уже на протяжении сорока лет является одним из самых популярных языков в мире. Он используется для разработки самых распространенных программных решений, таких как Microsoft Windows, Linux, OpenSSL и миллионов других.

3. Несмотря на свою мощь и выразительность язык крайне прост как для написания и чтения кода, так и для компиляции [11].

Стоит отметить, что понятие «язык C» это своего рода сфеерический конь в вакууме. Язык появился в 70-х годах прошлого века и за все это время претерпел колоссальное количество, как оффициальных так и не очень, изменений, что породило огромное количество версий и диалектов языка.

Оффициальные версии языка определяются организациями ISO и IEC, а конкретно комитетом ISO/IEC JTC 1/SC 22, который отвечает за разработку и принятие стандарта языка [12].

Мы должны помнить, что нашей задачей является написание дидактического примера, а не промышленного компилятора. Поэтому мы должны отсечь все возможности языка, которые будут усложнять реализацию, не привнося при этом никакого существенного вклада в образовательную ценность проекта.

Когда-то давно комитет C89 при разработке самого первого стандарта языка C поставил для себя задачу «сохранить дух языка» [13]. Такой же цели должны придерживаться и мы: убрать лишнее из реализации, сохранив при этом дух языка.

Следующим шагом будет выбор целевого языка компилятора, другими словами нам необходимо выбрать под какую платформу будет генерироваться ассемблерный код.

В целом все языки ассемблера крайне похожи между собой, потому что нижележащие принципы работы любого из них примерно одинаковы [14]. Наиболее очевидным и, в то же время, подходящим выбором для нас будет архитектура Intel 64: она уже многие годы доминирует на рынке персональных

компьютеров и подавляющие число студенов имеют процессоры, которые реализуют именно её.

Одним из самых больших недостатков этой архитектуры является её сложность, ставшая результатом давней политики корпорации Intel по сохранению обратной совместимости их процессоров. Мы могли бы выбрать архитектуру ARM или RISC-V, которые более просты и минималистичны [15], но:

1. Для Intel 64 написано колоссально большее количество учебных материалов чем по любой другой архитектуре.

2. Сложности Intel 64 в полной мере проявляются только при очень низкоуровневом программирование, что в нашем случае нивелирует эффекты этой особенности архитектуры.

3. Для запуска кода под архитектуру, отличающуюся от родной архитектуры процессора, студентам придется использовать средства виртуализации, которые требуют дополнительного обучения и могут быть несовместимы с некоторым оборудованием.

Резюмируя все вышеизложенное, сформулируем нашу задачу: *разработать дидактический пример компилятора языка C, поддерживающий основные конструкции языка и обеспечивающий возможность просмотра всех промежуточных представлений исходного кода, применив при разработке идеоматичные алгоритмы и методы построения компиляторов*.

1.3 Обзор существующих аналогов

Перед реализация программного обеспечения необходимо проанализировать аналогичное или схожее программное обеспечение, существующее на рынке, провести сравнение с разрабатываемым решением, а также возможность его использования для решения поставленных задач.

1.3.1 8cc

Одним из самых популярных из простых компиляторов языка C является 8cc, который реализует язык C11 и способен скомпилировать сам себя.

При его написании автор старался сохранять код как можно более простым и читаемым, чтобы он мог послужить хорошим примером для узучения различных техник, применяемых в компиляции.

Стадии препроцессинга и лексического анализа объединены в один проход. Используется, написанный в ручную, парсер рекурсивного спуска, который строит сразу AST, пропуская этап построения синтаксического дерева.

Компилятор предоставляет возможность вывести результаты препроцессинга, построения AST дерева и ассемблерного кода под платформу Intel 64.

1.3.2 Nanopass Framework

Nanopass Framework является встраиваемым предметно-ориентированным языком для создания компиляторов, который фокусируется на создании небольших этапов прохода по исходному кода и большого количества промежуточных представлений. Фреймворк уменьшает количество вспомогательного кода, требуемого для создания компилятора, что сделать их более простыми для понимания и поддержки.

Фреймворк поощряет создание небольших проходов компиляции крайне незначительных по сложности. Например, это могут быть такие проходы как «удаление всех точек с запятой» или «преобразование операции присваивания». Последовательное применение десятков или даже сотен таких проходов позволяет получить программу на целевом языке.

Существуют реализации Nanopass на таких языках как Scheme и Racket, оба из которых являются функциональными.

1.3.3 AMaCC

Проект AMaCC (расш. *Arguably Minimalist ARM C Compiler*) — это компилятор подмножества языка C89 под 32-х битную архитектуру ARM. Создавался в педагогических целях для изучения компиляторов, линкеров и загрузчиков.

Реализует две модели выполнения: JIT-компиляция под ARM-бекенд, а также генерация исполняемых файлов в формате ELF для систем GNU/Linux.

AMaCC использует смешение рекурсивного спуска и высходящего парсинга. Простое стековое AST генерируется через вызовы функций парсинга `stmt` и `expr`, на вход которым лексер подает токены. Функция `expr` выполняет простые оптимизация над литералами. Полученный AST преобразуется в стековое VM IR функцией `gen`. Наконец, функция `codegen` генерирует ARM32 инструкции из IR, которые могут быть либо выполнены функцией `jit`, либо переданы фунции `elf32` для генерации исполняемого файла.

1.3.4 catc

Компилятор catc поддерживает подмножество языка C и реализует кодогенерацию под платформу Intel 64. Фазы лексического и синтаксического анализа реализованны с помощью инструментов построения парсеров Lex и Yacc.

1.4 Сравнительный анализ существующих решений

Найти дидактические примеры компилятора языка C довольно сложно, так как таких разработок крайне мало. К сожалению, даже найденные с таким трудом, готовые решения не удовлетворяют всем выдвинутым требованиям к проекту.

После проведенного анализа существующих решений можно сделать следующие выводы:

— Большинство продуктов реализуют небольшое подмножество языка C, кроме компилятора «8cc», который практически полностью реализует версию языка C11.

— Ни один из проанализированных компиляторов не предоставляет достаточных возможностей интроспекции промежуточных представлений исходного кода в процессе компиляции.

— Ни один из проанализрованных компиляторов не может служить хорошим примером идеоматичных и устоявшихся архитектурных принципов построения компиляторов. Ни в одном из них нет архитектуры, где каждый модуль представляет из себя законченный функциональный блок определенного этапа компиляции.

— Большая часть компиляторов содержит ненужный для решения поставленных нами задач функционал, что усложнит понимание их внутренней структуры и реализации студентам. В качестве примера такого функционала можно привести JIT-компиляцию в проекте «AMaCC» или почти полное соответствие стандарту C11 в компиляторе «8cc».

— Большинство проектов реализовано на той или иной версии языка C. Исключение составляет Nanopass Framework, который имеет реализации на языках Scheme и Racket.

— Большая часть компиляторов имеет довольно сложный для понимания исходный код. Сложность кода является, в том числе, следствием отсутствия деления программы на отдельные независимые функциональные узлы. В архитетуре решений наблюдается нарушения принципа единой ответственности.

Таким образом, было принятно решение разрабатывать свой собственный компилятор языка C.

Таблица 1 – Сравнительная характеристика существующих решений и разрабатываемого проекта

Параметр	8cc	AMaCC	Nanopass Framework	catc
Интроспекция	AST	IR	—	нет
Целевая платформа	Intel 64	ARM32	—	Intel 64
Модульность архитектуры	слабая	средняя	—	слабая
Сложность исходного кода	средняя	высокая	—	средняя
Версия языка	C11	подмножество C89	—	*(не указано)*
Язык реализации	C11	C89	Scheme или Racket	C99

2 Описание проектирование системы

2.1 Описание методов компиляции

2.1.1 Лексический анализ

Первым шагом в процессе компиляции является лексический анализ. Лексический анализтор преобразует сырой исходный код, представленный в виде потока символов, поток групп символов, которые мы называем *токенами*. Токены — это осмысленные лексические единицы, из которых строится его грамматика.

Самым распространенным способом преобразования потока символов в набор токенов является использование *регулярных выражений*.

Регулярные выражения широко применяются и за пределами области разработки компиляторов. Утилита `grep` использует их для поиска различных паттернов внутри текстовых файлов.

Набор строк, определяемый регулярными выражениями называется *регулярным множеством*. В контексте лексического анализа каждый токен предствлен регулярным множеством, чья структура определяется регулярным выражением. Конкретный элемент токена из регулярного множества называют *лексемой*.

Наше определение регулярного выражения начнется с конечного множества символов, или *словаря* (обозначается как Σ). Обычно этот словарь является набором символов, используемым компьютером.

Пустые, или нулевые, строки разрешены и обозначаются символом λ. Этим символов обозначают пустой буфер в котором ни один из символов еще не был распознан регулярным выражением. Этим же символом обозначают опциональную часть токена, например, целочисленный литерал может начинаться с плюса или минуса, а если он беззнаковый, то с λ.

Строки строятся из символов множества Σ при помощи операции *конкатенации* (объединение отдельных символов для образования целостной строки).

Операция конкатенации расширяется на множество строк следующим образом. Пусть P и Q множества строк. Символ $\in$ обозначает принадлежность ко множеству. Если $s_1 \in P$ и $s_2 \in Q$, то строка $s_1 s_2 \in (PQ)$. Небольшие конечные множества обычно обозначаются простым перечислением своих элементов. Скобки исполбзуются для отделения выражений, а $|$, *оператор альтерации*, используется для обозначения альтернатив. Например, множество десятичных цифр можно представить следующим образом:

$$D = (0 \,|\, 1 \,|\, 2 \,|\, 3 \,|\, 4 \,|\, 5 \,|\, 6 \,|\, 7 \,|\, 8 \,|\, 9).$$

Введем третью операцию — *замыкание Клини*. Оператор * обозначает постфиксную операцию замыкание Клини. Например, пусть P будет набором строк. Тогда P^* представляет собой все строки, сформированные конкатенацией нуля или более элементов из P.

2.1.2 Синтаксический анализ

Следующим этапом после лексического анализа является *синтаксический анализ* или *парсинг*.

Регулярные языки, являясь чрезвычайно мощным и в то же время простым инстурментом, обладают одним фундаментальным ограничением: они не могут распознавать рекурсивные структуры.

Однако, языки программирования в большинстве случаев требуют возможности определять рекурсивные элементы, типа вложенных блоков или подвыражений.

Анализ и классификаци формальных языков, проведенные Ноамом Чомским, поместили регулярные языки на нижний уровень в одноименной иерархии. Однако на уровне выже расположились *контестно-свободные языки*, которые обладают достаточными возможностями для распознавания рекурсивных элементов, что делает их рабочей лошадкой синтаксических анализаторов для большинства языков программирования.

Формально, контестно свободная грамматика определяется как четверака $G = (V, \Sigma, R, S)$, где

1. V — это конечное множество. Каждый элемент $v \in V$ называют *нетерминальным символом* или *переменной*. Иногда их также называют *синтаксическими категориями*. Каждая переменная определяет дочерний язык языка, определяемого G.

2. Σ — это конечное множество , непересекающийся с V. Множество терминалов определяет алфавит языка, определенного грамматикой G.

3. R — это конечное отношение в $V \times (V \cup \Sigma)^*$. Элементы множества R называют *правилами* или *продукциями* грамматики.

4. S — это начальная переменная, используемый для представления всего предложения.

Задача распознавания потока символов контекстно-свободной грамматикой и называется парсингом. Эта проблема хорошо изучена и существует ряд классических и провереных алгоритмов парсинга. Невдаваясь в подробности, остановимся на одном из самых простых и эффективных методах парсинга — *рекурсивном спуске*.

Рекурсивный спуск это подвид нисходящих парсеров в котором каждая процедура реализует один из нетерминальных символов грамматики. Таким образом структура реализации такого парсера повторяет структуру самой грамматики, которую этот парсер распознает.

Предиктивный рекурсивный спуск не требует бэктрекинга. Таким парсеры возможны только для LL(k) грамматик, которые являтются контекстно-свободными грамматиками, для которых существует положительное целое k, которое позволяет парсеру определить какую продукцию использовать просмотрев только k следующих токенов. Таким образом, LL(k) грамматики исключают все неоднозначные грамматики, а также все леворекурсивные грамматики.

Рекурсивный спуск с бектрекингом это техника, при которой парсер определяет подходящую продукцию, пробуя все продукции подряд и делая возврат к предыдущему состоянию при ошибке. Такой парсер не ограничивается LL(k) грамматикам, но не может гарантировать завершение работы, если только грамматика не является LL(k).

2.1.3 Построение абстрактного синтаксического дерева

Конкретное синтаксическое дерево полностью совпадает с грамматикой языка. К сожалению, получаемое в процессе парсинга дерево выходит слишком большим и “зашумленными” и не может быть эффективно использовано для дальнейшего анализа программы. Например, деревья разбора обычно сохраняют в себе такие незначимые для дальнейшего процесса компиляции детали как узлы пунктуационных элементов типа фигурных скобок, точек с запятой и прочих.

Для решения этой проблемы было придумано такое представление программы как *абстрактное синтаксическое дерево* (англ. *AST*). Такие деревья получаются гораздо меньшего размера и таким образом, с ним проще работать в дальнейших стадиях компиляции.

Превращение AST в CST является крайне нетривиальной задачей и каких конкретных устоявшихся алгоритмов для ее решения нет.

Существует такой язык как ASDL, который используется для декларативного описания узлов абстрактного синтаксического дерева и дальнейшей генерации кода для разных языков, который эти узлы описывает. Однако он служит исключительно в целях описания структуры AST, а не конвертации CST в AST.

Из-за этой проблемы парсеры обычно генерируют сразу AST дерево, минуя этап формирования отдельного CST. Однако нам этот метод не подходит, потому что нам необходимо иметь возможность интроспекции как CST, так и AST.

Таким образом, единственным вариантом для нас остается так называемый метод ноу-хау, в котором мы будем в ручную проходить по CST дереву и

строить соответствующие узлы AST. Такой же подход использует компилятор Groovy [16].

2.1.4 Кодогенерация

Кодогенерация является последним этапом процесса компиляции и представляет собой генерацию кода на выходном языке, который может быть напрямую исполнен целевой платформой, из некоторого промежуточного представления.

В нашем случае, мы будем генерировать ассемблерный код под архитектуру Intel 64 с System V ABI в синтаксисе AT&T напрямую из AST.

2.2 Проектирование системы

2.2.1 Определение информационных потоков системы

Перед стадией разработки на этапе проектирования необходимо определить информационные потоки системы. Информационные потоки помогут лучше понять через какие этапы проходят основные процессы, которые были подвержены автоматизации, какие данные используются и что получает на выходе пользователь системы.

На сегодняшний день одним из удобных и популярных способов представления информационных потоков являются IDEF диаграммы. IDEF диаграммы — это методология для решения моделирования сложных систем, данные диаграммы позволяют отображать и анализировать различные процессы или деятельность пользователей системы в различных разрезах. Широту и глубину обследования процессов определяет сам разработчик, благодаря знаниям и опыту которого создаваемая модель не перегружается излишними процессами и информацией.

Для построения высокоуровневой модели системы необходимо создать диаграмму IDEF0. Данная диаграмма предполагает наличие четко сформулированной цели единственного субъекта моделирования и одной точки зрения,

она поможет более точно выявить главные процессы в системе. На диаграмме указывают какие данные поддаются на вход, какие данные получает пользователь в результате данного процесса, какие модули и компоненты программного обеспечения принимают в этом участие.

IDEF0 является методологией функционального моделирования. Она используется для создания функциональной модели, которая отображает структуру и функции системы, а также потоки информации и материальных объектов, связывающих эти функции. Компонентами синтаксиса IDEF0 являются блоки, стрелки, диаграммы и правила.

Диаграмма IDEF0 состоит из *блоков* и *стрелок*, направление которых заданы правила. Блоки изображаются в виде прямоугольников и содержат в себе описание той функции, которую они представляют, и номер. Номер блока должен быть расположен в правом нижнем углу.

К каждому блоку должно вести несколько стрелок, такие как:

— *левые стрелки* (направлены в блок) обозначают информацию или продукты, которую функция получает на входе;

— *правые стрелки* (направлены из блока) обозначают информацию или продукты, которую функция дает на выходе;

— *верхние стрелки* (направлены в блок) обозначают документы, которые регламентируют работы системы;

— *нижние левые стрелки* (направлены в блок) обозначают механизмы, которые влияют на работу нашей системы;

— *нижние правые стрелки* (направлены из блока) обозначают обращение к дополнительной информационной системе, которая существует совершенно отдельно от нашей и которая необходима для осуществления процесса или функции. Может как присутствовать на схеме, так и нет.

На рисунке ниже представлена диаграмма IDEF0 процесса компиляции одной единицы трансляции.

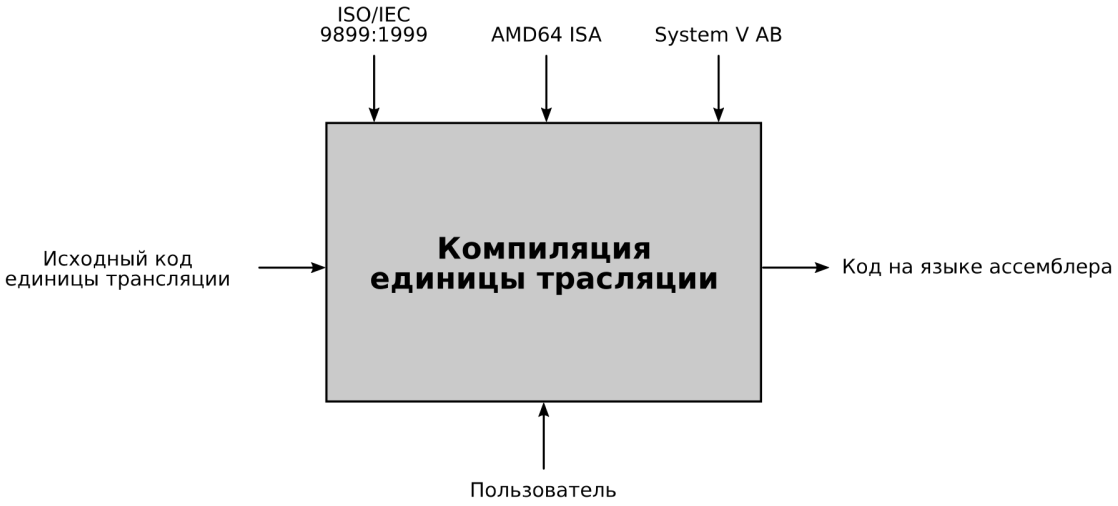

Рисунок 2.1 – Комплексная диаграмма IDEF0 процесса компиляции единицы трансляции.

Также на этапе проектирования полезно сделать более детальную декомпозицию комплексной диграммы IDEF0, что позволяет более подробно рассмотреть структуру и функции изучаемой системы.

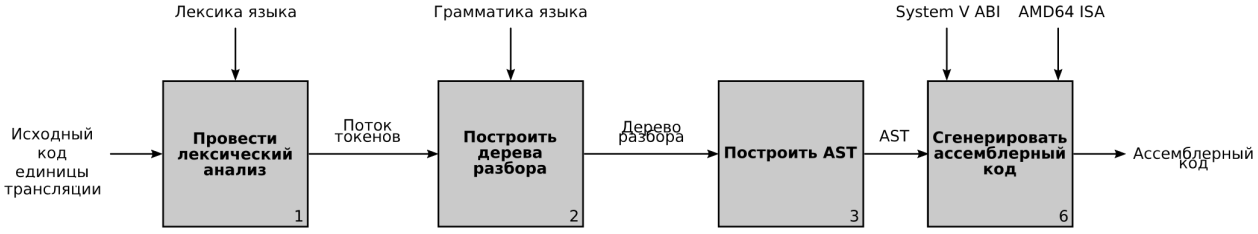

Рисунок 2.2 – Декомпозиця комплексной диаграммы IDEF0.

2.2.2 Описание системных процессов

Методология IDEF3 — это методология описания процессов. В отличие от IDEF0 она рассматривает последовательность выполнения задач, а также причинно-следственные связи между ними. Диаграммы IDEF3 состоят из блоков, описывающих функции, стрелок-связей и перекрёстков, которые показывают, как именно выполняются процессы.

Блок (функциональный элемент) представляет собой прямоугольник, разделенный на три другие прямоугольника: один большой наверху и два маленьких друг рядом с другом внизу. В верхнем прямоугольнике содержится имя функции, в нижнем левом номер его выполнения, в нижнем правом, при необходимости, находится ссылка на другую функцию. Связи бывают простыми, относительными и связями с условием.

Перекрёстки подразделяются на следующие типы: *И*, *ИЛИ*, *синхронное И*, *синхронное ИЛИ*, а также *исключающее ИЛИ*.

2.2.3 Описание потоков данных

Диаграммы потоков данных (англ. *DFD*) — это способ представления процессов обработки информации. Они показывают, как информация перемещается из одной функции к другой. Подобное представление потока данных отражает движение объектов, их хранение и распространение.

DFD состоит из следующих компонентов: внешняя сущность, процесс, поток данных и хранилище данных [17]. Внешняя сущность представляет собой источник или приёмник информации и изображается прямоугольником с прямыми углами. Процессы в DFD — это функции системы, преобразующие входы и выходы, которые изображаются как прямоугольники со скругленными углами. Потоки данных изображаются стрелками. Если стрелка соединяет какую-либо функцию с хранилищем данных, то на ней должно быть отображено имя, отражающее содержание данного потока. Хранилище данных является прообразом базы данных и изображается как прямоугольник без правой стороны.

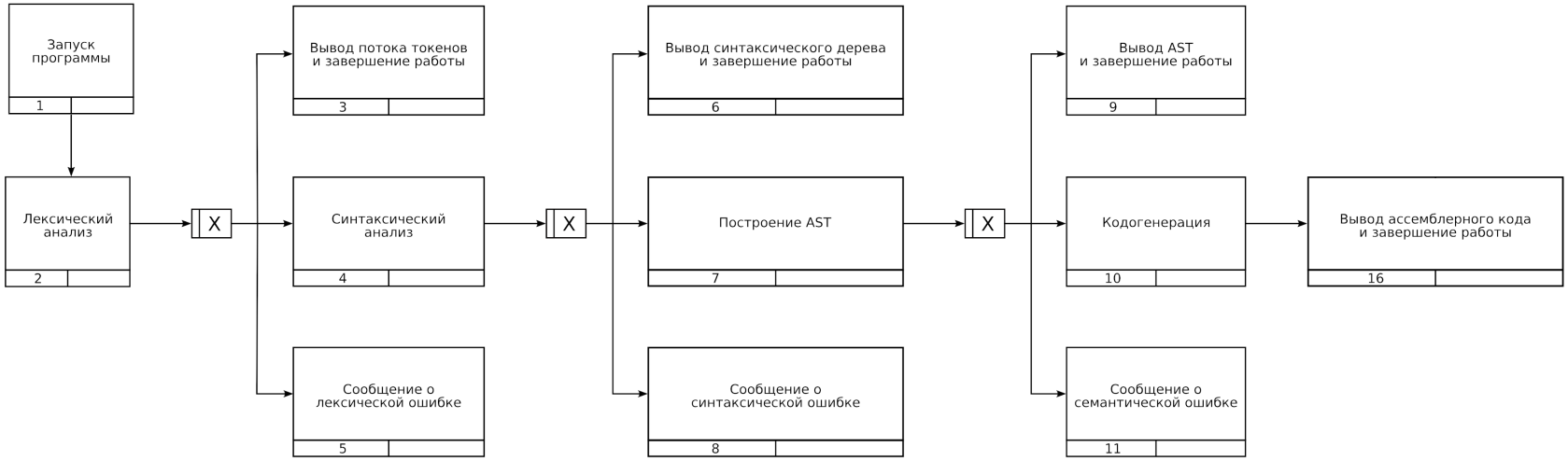

Рисунок 2.3 – Диаграмма описания процессов по методологии IDEF3.

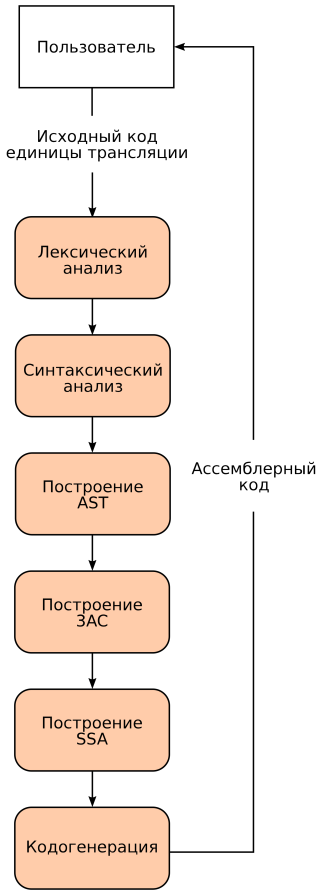

Рисунок 2.4 – Описание потоков данных по методологии DFD.

2.2.4 Описание пакетов системы

Диаграмма пакетов — это структурная диаграмма UML, которая показывает структуру проектируемой системы на уровне пакетов [18]. Обычно на диаграмме изображаются следующие элементы: пакет, пакетированный элемент, зависимость, импортируемый элемент, импорт пакета, объединение пакета.

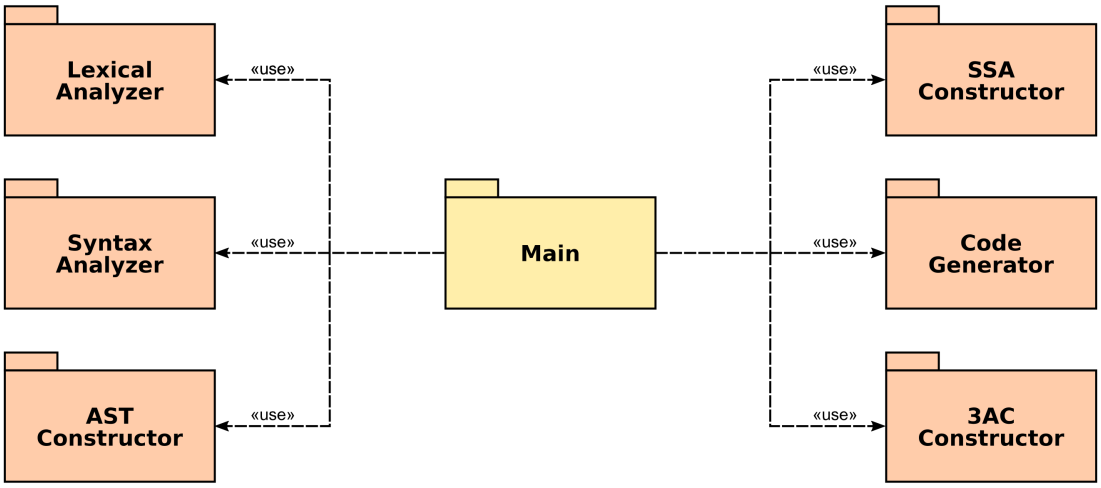

Рисунок 2.5 – Диаграмма описания пакетов системы по методологии UML.

2.2.5 Описание компонентов

Диаграмма компонентов показывает компоненты, предоставляемые и требуемые интерфейсы, порты и отношения между ними [19]. Этот тип диаграмм используется в компонентно-ориентированной разработке для описания систем с сервисно-ориентированной архитектурой [20].

Компонентно-ориентированная разработка основана на предположения, что ранее сконструированные компоненты могут быть переиспользованы или, при необходимости, заменены на некие «эквивалентные» или «совместимые» компоненты.

2.2.6 Описание потоков информации по UML

Диаграмма потока информации — это поведенческая диаграмма UML, которая показывает обмен информацией между сущностями системы на высоком уровне абстракции. Потоки информации могут быть полезны для описания циркуляции информации через систему через представление аспектов

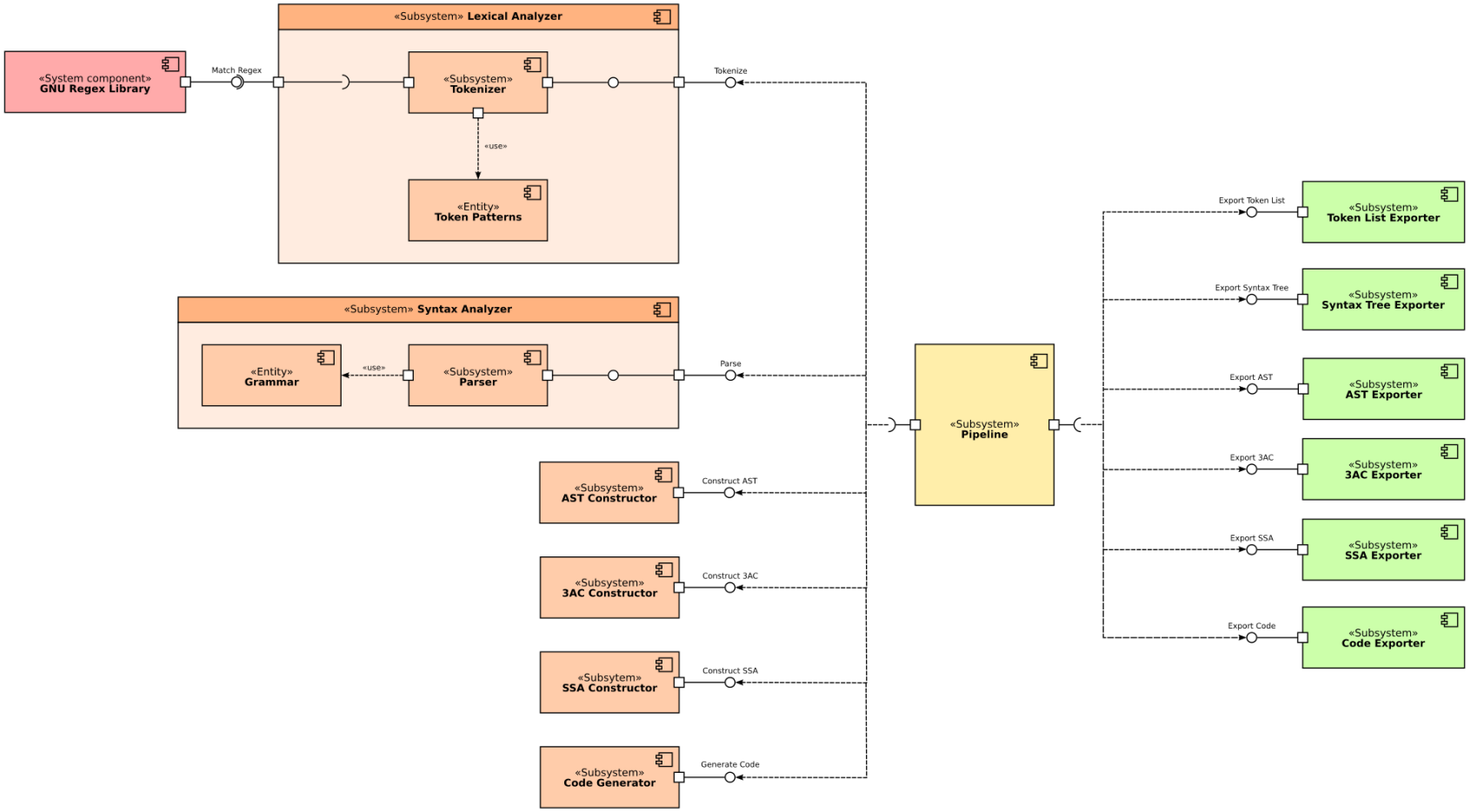

Рисунок 2.6 – Диаграмма описания компонентов, выполненная по методологии UML.

модели, которые еще не полностью специфицированы или недостаточно детализированы.

Потоки информации не показывают природу информации, механизмы передачи, порядок обмена или какие-либо контрольные условия [21]. Единицы информации могут быть использованы для представления информации, которая протекает через систему вместе с информационными потоками еще прежде, чем их реализация будет проработана.

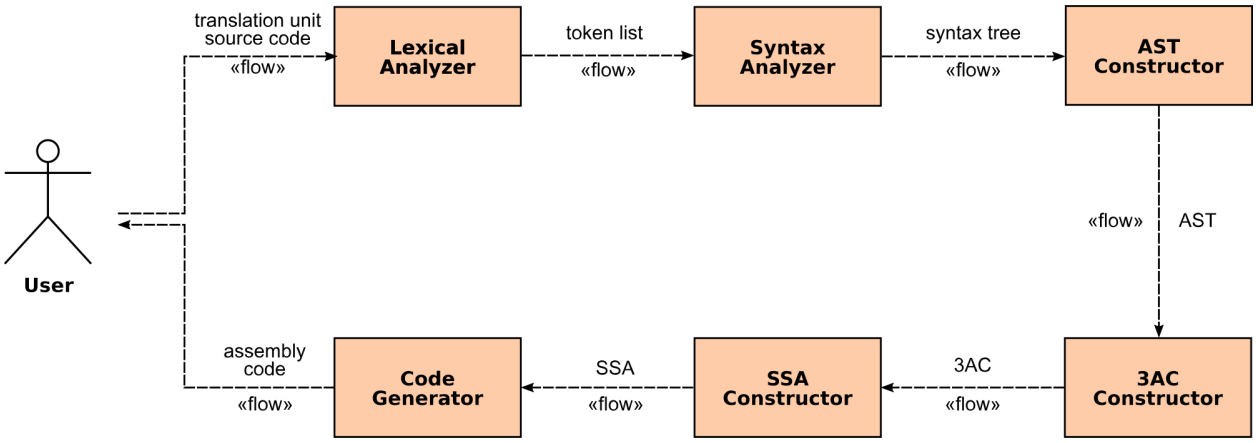

Рисунок 2.7 – Описание потоков информации системы с помощью диаграммы потока информации методологии UML.

3 Описание практической реализации

3.1 Компилятор

В качестве средств для разработки компилятора были выбраны инструменты из экосистемы GNU/Linux.

3.1.1 Система сборки

В качестве системы сборки есть три наиболее распространенных решения: GNU Make, CMake и GNU Autotools.

3.1.1.1 GNU Make

В настоящее время существует множество утилит для сборки с отслеживанием зависимостей, но Make является одной из самых распространенных, в первую очередь благодаря тому, что она была включена в Unix, начиная с PWB/UNIX 1.0, в которой было множество инструментов, предназначенных для задач разработки программного обеспечения. Первоначально он был создан Стюартом Фельдманом в апреле 1976 года в Bell Labs. Фельдман получил награду ACM Software System Award 2003 года за создание этого широко распространенного инструмента.

Make обычно используется для сборки исполняемых программ и библиотек из исходного кода. Однако в целом Make применим к любому процессу, который включает выполнение произвольных команд для преобразования исходного файла в конечный результат. Например, Make можно использовать для обнаружения изменений, внесенных в файл изображения (источник), а действия по преобразованию могут заключаться в преобразовании файла в определенный формат, копировании результата в систему управления контентом, а затем отправке электронной почты заранее определенному набору пользователей с указанием того, что вышеуказанные действия были выполнены.

Среди ключевых достоинств GNU Make можно выделить:

— Широкая распространенность и повсеместное использование.

— Проста и логичность написания правил сборки.

— Универсальность, позволяющая собирать произвольные артефакты, а не только исполняемые файлы.

Из недостатоков можно отметить:

— Не предусмотрена автоматическая пересборка файлов исходного кода при изменении заголовочных файлов, что требует дополнительной логики в Makefile.

— В больших проектах Makefile очень быстро становится сложно сопровождать.

3.1.1.2 CMake

CMake — это кроссплатформенное бесплатное программное обеспечение с открытым исходным кодом для автоматизации сборки, тестирования, упаковки и установки программного обеспечения с помощью независимого от компилятора метода. CMake не является системой сборки, а скорее генерирует файлы сборки другой системы. Он поддерживает иерархии каталогов и приложения, зависящие от множества библиотек. Он используется совместно с родными средами сборки, такими как Make, Qt Creator, Ninja, Android Studio, Apple's Xcode и Microsoft Visual Studio. Он имеет минимальные зависимости, требуя только компилятор C++ в собственной системе сборки.

Сборка программы или библиотеки с помощью CMake представляет собой двухэтапный процесс. Сначала из конфигурационных файлов (`CMakeLists.txt`), написанных на языке CMake, создаются (генерируются) стандартные файлы сборки. Затем используются встроенные инструменты сборки платформы (т.н. *native toolchain*) для фактической сборки программ.

Файлы сборки конфигурируются в зависимости от используемого генератора (например, *Unix Makefiles* для make). Опытные пользователи могут также создавать и включать дополнительные генераторы Makefile для поддержки своих специфических потребностей в компиляторе и ОС. Сгенерированные файлы обычно помещаются (с помощью флага `cmake` в папку вне исходника (сборка вне исходника), например, `build/`.

Каждый проект сборки в свою очередь содержит свой собственный файл `CMakeCache.txt` и каталог `CMakeFiles` в каждом (под)каталоге проекта, включаемом командой `add_subdirectory(...)`, что помогает избежать или ускорить регенерацию при многократном запуске.

После создания Makefile (или альтернативного) поведение сборки может быть точно настроено через свойства цели или через глобальные переменные `CMAKE_...`-префикс. Последнее не рекомендуется для конфигураций только для целей, поскольку переменные также используются для конфигурирования самого CMake и установки начальных значений по умолчанию.

Из преимуществ CMake можно выделить:

— Кроссплатформенность и интеграция с IDE.

Из недостатков:

— Двухступенчатый процесс сборки, который может быть непривычен и непонятен студентам.

— Зависимость от вторичной системы сборки.

3.1.1.3 GNU Autotools

GNU Autotools, также известный как GNU Build System, — это набор инструментов программирования, предназначенный для помощи в создании пакетов исходного кода, переносимых на многие Unix-подобные системы.

Сделать программу переносимой может быть непросто: компилятор C отличается от системы к системе; некоторые библиотечные функции отсут-

ствуют в некоторых системах; заголовочные файлы могут иметь разные имена. Одним из способов решения этой проблемы является написание условного кода с выделением блоков кода с помощью директив препроцессора (`#ifdef`); но из-за большого разнообразия сред сборки такой подход быстро становится неуправляемым. Autotools предназначен для более удобного решения этой проблемы.

Autotools является частью набора инструментов GNU и широко используется во многих пакетах свободных программ и пакетов с открытым исходным кодом. Его составные инструменты являются свободным программным обеспечением, лицензируемым по лицензии GNU General Public License со специальными лицензионными исключениями, позволяющими использовать его с несвободным программным обеспечением.

К преимуществам GNU Autotools относят:

— Гибкость при конфигурации программного обеспечения под различные платформы.

— Широкая распространенность в экосистеме GNU/Linux.

Однако GNU Autotools не раз сталкивался с резкой критикой, которая в основом отмечает сложность и запутанность процесса сборки.

3.1.1.4 Сравнение

Поскольку мы делаем большой фокус на простоту и элегантность проекта, наилучшим выбором для нас станет использование GNU Make.

3.1.2 Лексический анализ

Модуль лексического анализатора пользуется встроенной в GNU C Library реализацией регулярных выражений.

В исходном коде мы в декларативном стиле определяем список токенов, а также соответствующие им регулярные выражения. Для того, чтобы избежать дублирования кода мы используем классическую технику X Macro.

При вызове основной процедуры лексера `tokenize()` мы начинаем проходить по строке исходного кода символ за символом и пытаться распознать следующий токен последовательно с помощью каждого регулярного выражения.

В конце концов, если одно из выражений удается распознать мы создаем новый токен с соответствующим типом и записываем информацию о лексеме, полученную с помощью инструмента групп захвата регулярных выражений.

Таким образом мы распознаем весь поток символов и превращаем его в поток токенов.

В случае, если на определенном этапе анализа, нам не удалось распознать оставшийся поток символов ни одним из регулярных выражений, мы возвращаем ошибку, сигнализирующую о лексической ошибке ввода.

3.1.3 Синтаксический анализ

Для синтаксического анализа мы используем алгоритм рекурсивного спуска. Однако, вместо того чтобы вручную писать парсер, было принято решения создать небольшой генератор парсеров, который мог бы самостоятельно генерировать код рекурсивного спуска из переданного ему декларативного описания грамматики.

Мы описываем грамматику, используя похожий на EBNF синтаксис. В процессе сборки, генератор парсера преобразует это описание в исходный файл на языке C с соответствующими процедурами, распознающими нетреминальный символы.

Далее этот сгенерированный парсер линкуется вместе со всеми остальными модулями компилятора.

Сам парсер начинает распознавания потока токенов, рассматривая его к единицу трансляции. Далее он пытается распознать оставшийся поток токенов по-очередно с помощью каждой продукции. В случае, если продукция не была распознана, выполняется бэктрекинг, который возвращает курсор парсера на прежнее место в потоке токенов.

В случае успешного распознавания потока токенов, парсер возвращает конкретное синтаксическое дерево, которое хранит информацию о синтаксический и лексических единицах, а также о положении каждого токена в исходном коде.

В случае неудачного распознавания нетерминального символа парсер возвращает ошибку, сигнализирующую о синтаксической ошибке. Отдельным видом такой ошибки является неожиданный конец ввода.

3.1.4 Построение абстрактного синтаксического дерева

Следующим этапом мы проходим по конкретному синтаксическому дереву, полученному от парсера и пытаемся построить AST. Все происходит полностью в ручном процессе.

На этом этапе мы пытаемся определить типы идентификаторов в их объявлениях, их спецификаторы типов, типы выражений и прочее. Иными словами, заполнить атрибуты в соответствующих описаниях узлов AST.

Каждый узел AST представляет из себя структуру `struct ast`, которая состоит из объединения и поля члена перечисления. Объединение содержит структуры конкретных типов AST узлов, например `struct ast_transunit` или `struct ast_parmdecl`. Какой конкретно элемент объединения выбран в данный момент определяет поле `kind` обобщенной структуры.

В некоторых случаях, из CST не может быть построенно AST, потому что в исходном коде присутствовала семантическая ошибка. В таком случае генератора AST вовзращает встреченную ошибку с подробным описанием.

Рассмотрим пример, когда во входном коде присутствует конструкция

```
signed unsigned int a;
```

которая является абсолютно корректной с синтаксической точки зрения. С точки зрения семантики она является ошибочной так как стандарт языка прямо запрещает подобную комбинацию спецификаторов типа.

В таком случае генератор AST вернет вызывающей стороне ошибку с кодом `C2A_ERR_BAD_TYPE_SPECIFIERS`.

3.1.5 Кодогенерация

3.1.6 Интерфейс командной строки

Компилятор предоставляет интерфейс командной строки (англ. *CLI*). Пользователь должен передать компилятору ключ соответствующий представлению кода, которое он хочет получить (например, `-ast` и `-cst` для абстрактного и конкретного синтаксических деревьев соответственно). Также опционально может быть передан флаг `-json`, который указывает компилятору на то, что вывод должен быть представлен в формате JSON.

Следующим образом выглядят различные промежуточные представления, полученные от компилятора:

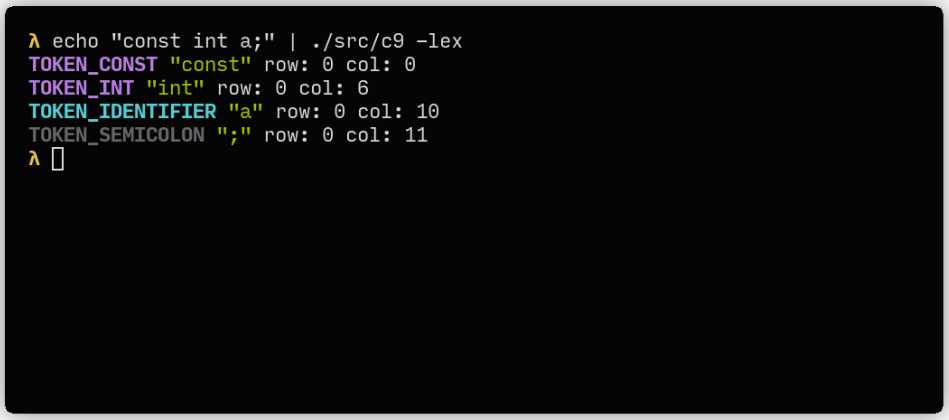

Рисунок 3.8 – Результат работы лексического анализа. Можно видеть поток получившихся токенов, их лексемы и позицию в исходном коде (соответствующие строка и столбец).

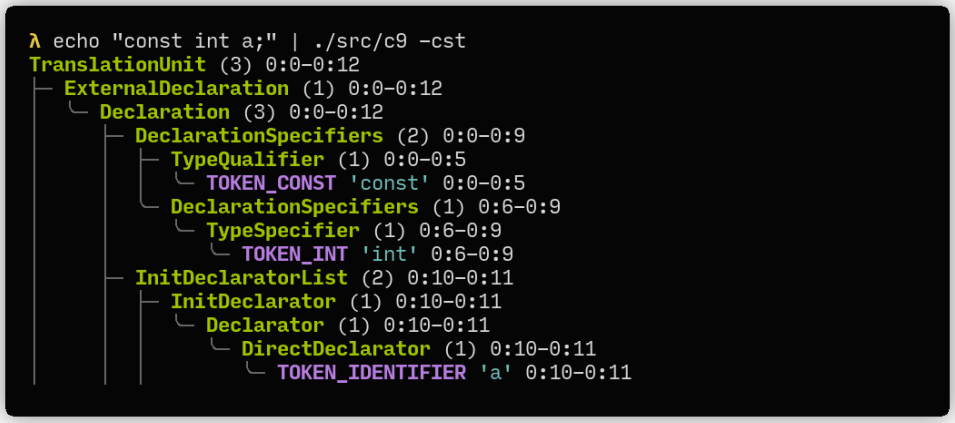

Рисунок 3.9 – Результат работы парсера в CLI представляет из себя визуализированное дерево разбора.

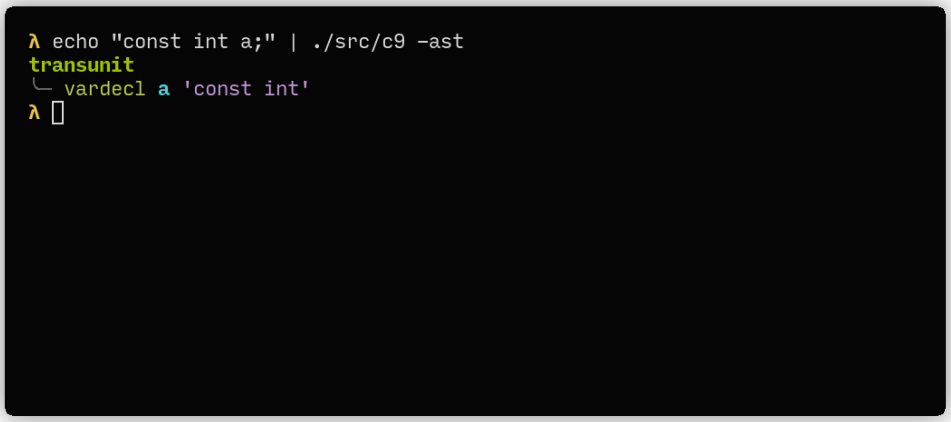

Рисунок 3.10 – Построенное из дерева разбора AST.

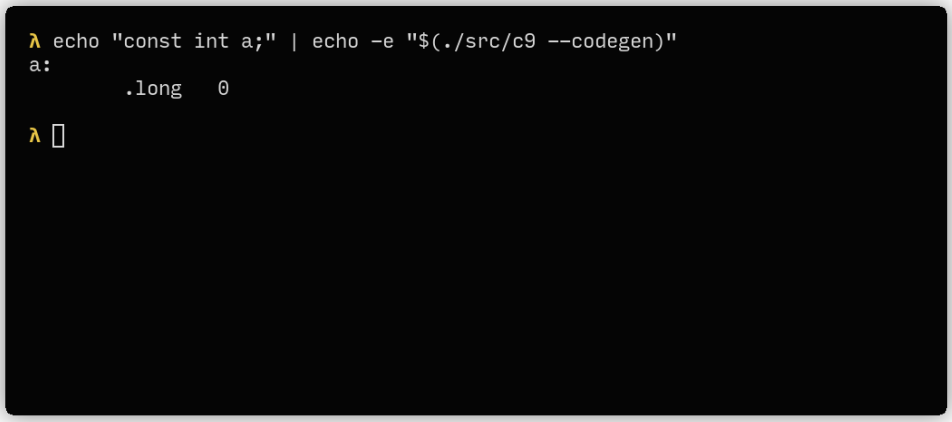

Рисунок 3.11 – Сгенерированный ассемблерный код.

3.1.7 Экспорт промежуточных представлений в формате JSON

Для того дальнейшней обработки результатов интроспекции различных представлений кода, в компиляторе предусмотрена возможность экспорта вывода в формате JSON. За эту опцию отвечает флаг `-json` в CLI. Ниже на рисунках показан экспорт в JSON различных представлений кода.

```
λ echo "const int a;" | ./src/c9 -lex --json | jq
{
  "tokens": [
    {
      "type": "TOKEN_CONST",
      "lexem": "const",
      "row": 0,
      "col": 0,
      "metatype": "keyword"
    },
    {
      "type": "TOKEN_INT",
      "lexem": "int",
      "row": 0,
      "col": 6,
```

Рисунок 3.12 – Экспорт в JSON потока токенов, полученного от лексического анализатора.

```
λ echo "const int a;" | ./src/c9 -cst --json | jq
{
  "name": "TranslationUnit",
  "type": "nonterminal",
  "nchildren": 3,
  "start": {
    "row": 0,
    "col": 0
  },
  "end": {
    "row": 0,
    "col": 12
  },
  "children": [
    {
```

Рисунок 3.13 – Экспорт в JSON дерева разбора, полученного от синтаксического анализатора.

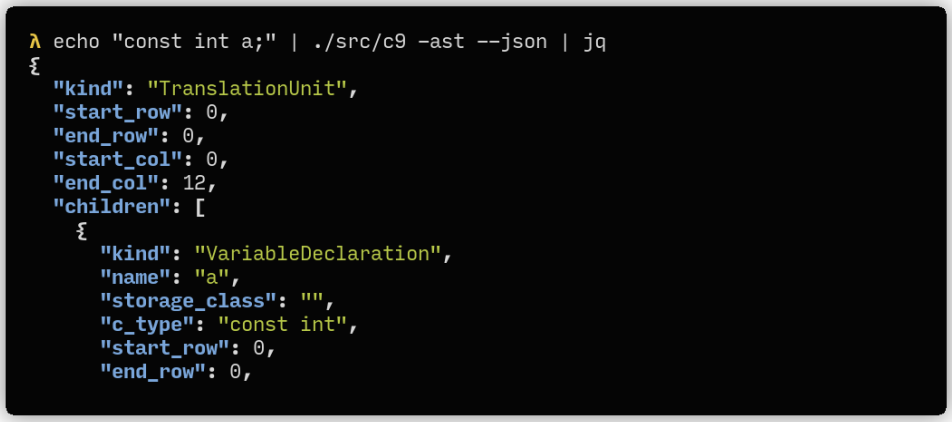

Рисунок 3.14 – Экспорт в JSON построенного AST.

Рисунок 3.15 – Экспорт в JSON сгенерированного ассебмлерного кода.

3.2 Серверная часть

Серверная часть веб-приложения для компилятора работает следующим образом. Сервер предоставляет открытое API с несколькими эндпоинтами, каждый из которых позволяет получить одно из промежуточных представлений кода. Когда пользователь делает запрос к сайту, веб-сервер пришлет ему статические файлы клиентского кода, которые делают асинхронные AJAX запросы к вышеупомянутому API.

В качестве веб-сервера был выбран Nginx как один из самых популярных и легко конфигурируемых. В качестве языка для написания бекенда был

выбран язык Go, так как он позволяет максимально быстро и просто создавать серверные приложения. Ниже немного расскажем про каждую из этих технологий и конкретную логику их работы в нашем случае.

3.2.1 Nginx

Nginx, произносится как “engine X”, является веб-сервером с открытым исходным кодом, который, после своего первоначального успеха в качестве веб-сервера, теперь также используется в качестве обратного прокси, HTTP-кэша и балансировщика нагрузки.

Некоторые известные компании, использующие Nginx, включают Autodesk, Atlassian, Intuit, T-Mobile, GitLab, DuckDuckGo, Microsoft, IBM, Google, Adobe, Salesforce, VMWare, Xerox, LinkedIn, Cisco, Facebook, Target, Citrix Systems, Twitter, Apple, Intel и многие другие.

Nginx был первоначально создан Игорем Сысоевым, его первый публичный релиз состоялся в октябре 2004 года. Игорь изначально задумывал программное обеспечение как ответ на проблему C10k, которая связана с проблемой производительности при обработке 10 000 одновременных соединений.

Поскольку его корни лежат в оптимизации производительности при масштабировании, Nginx часто превосходит другие популярные веб-серверы в эталонных тестах, особенно в ситуациях со статическим контентом и/или большим количеством одновременных запросов.

Мы используем Nginx для раздачи статических файлов фронтенда веб-приложения. Однако все запросы на эндпоинты `/codegen`, `/genast`, `/tokenize` и `/parse` проксируются на слушающий на локальном хосте бекенд, который предоставляет API компилятора.

3.2.2 Go

Go — это компилируемый язык с открытым исходным кодом и сильной типизацией, написанный для создания параллельного и масштабируемо-

го программного обеспечения. Язык был изобретен в Google Робом Пайком, Кеном Томсоном и Робертом Гризмером. Go предназначен для создания простого, надежного и эффективного программного обеспечения.

Go был разработан в Google в 2007 году для повышения производительности программирования в эпоху многоядерных, сетевых машин и больших кодовых баз. Разработчики хотели снять критику с других языков, используемых в Google, но сохранить их полезные характеристики:

1. Статическая типизация и эффективность во время выполнения (как C).
2. удобочитаемость и удобство использования (как Python или JavaScript).
3. Высокопроизводительные сети и многопроцессорность.

Его разработчики были в первую очередь мотивированы их общей неприязнью к C++.

Go был публично анонсирован в ноябре 2009 года, а версия 1.0 была выпущена в марте 2012 года. Go широко используется в производстве в Google и во многих других организациях и проектах с открытым исходным кодом.

В ноябре 2016 года шрифты Go и Go Mono были выпущены шрифтовыми дизайнерами Чарльзом Бигелоу и Крисом Холмсом специально для использования в проекте Go. Go — гуманистический сансериф, напоминающий Lucida Grande, а Go Mono — моноширинный. Оба шрифта соответствуют набору символов WGL4 и были разработаны так, чтобы быть разборчивыми с большой высотой x и четкими формами букв. И Go, и Go Mono соответствуют стандарту DIN 1450, поскольку имеют косой ноль, строчную l с хвостиком и прописную I с засечками.

В апреле 2018 года оригинальный логотип был заменен стилизованным GO, наклоненным вправо с обтекаемыми линиями. (Талисман суслика остался прежним).

Наш бекенд, написанный на Go, работает следующим образом. При запуске процесса мы создаем встроенный в стандартную библиотеку языка

веб-сервер, который принимает запросы на такие эндпоинты как `/codegen`, `/genast`, `/tokenize` и `/parse`.

При получении запроса на эндпоинт, мы запускаем в дочернем процессе наш компилятор с соответствующими флагами и передаем ему полученный при запросе исходный код. Компилятор возвращает бекенду ответ в формате JSON, который бекенд затем пересылает клиенту в качестве ответа.

3.2.3 JSON

JSON — это открытый стандартный формат файлов и формат обмена данными, который использует человекочитаемый текст для хранения и передачи объектов данных, состоящих из пар атрибут-значение и массивов (или других сериализуемых значений). Это распространенный формат данных, который находит разнообразное применение в электронном обмене данными, в том числе при взаимодействии веб-приложений с серверами.

JSON — это независимый от языка формат данных. Он был заимствован из JavaScript, но многие современные языки программирования включают код для генерации и разбора данных в формате JSON. В именах файлов JSON используется расширение `.json`.

JSON вырос из потребности в протоколе связи между сервером и браузером в режиме реального времени без использования плагинов браузера, таких как Flash или Java-апплеты, которые доминировали в начале 2000-х годов.

Предшественник библиотек JSON был использован в детском игровом проекте по торговле цифровыми активами под названием Cartoon Orbit на Communities.com (где ранее работали все соучредители State Software) для Cartoon Network, который использовал плагин для браузера с собственным форматом обмена сообщениями для манипулирования элементами Dynamic HTML (эта система также принадлежит 3DO). После открытия ранних возможностей AJAX, digiGroups, Noosh и другие использовали фреймы для передачи информации в визуальное поле браузера пользователя без обновления визуального контекста веб-приложения, реализуя богатые веб-приложения в

реальном времени, используя только стандартные возможности HTTP, HTML и JavaScript в Netscape 4.0.5+ и IE 5+.

Крокфорд впервые определил и популяризировал формат JSON. Соучредители State Software согласились создать систему, использующую стандартные возможности браузеров и обеспечивающую уровень абстракции для веб-разработчиков, позволяющий создавать веб-приложения с состоянием, которые имеют постоянное дуплексное соединение с веб-сервером, удерживая два открытых соединения по протоколу передачи гипертекста (HTTP) и утилизируя их до истечения стандартного времени браузера, если обмен данными не происходит. Соучредители провели круглый стол и проголосовали за то, как назвать формат данных — JSML (JavaScript Markup Language) или JSON (JavaScript Object Notation), а также под каким типом лицензии сделать его доступным. Чип Морнингстар разработал идею State Application Framework в компании State Software.

JSON в основном предназначен для обмена данными между приложениями. Парсинг данных из одного приложения в другое с помощью JSON очень прост благодаря его независимому от языка формату данных. Почти каждый язык программирования имеет поддержку JSON через официальные и сторонние разработчики. Теперь, ссылаясь на мой пример, JSON можно использовать как файл конфигурации или временного хранения данных для любого приложения. Существует важный факт, что JSON не имеет функции абстрактного типа данных (ADT) из-за своего формата сериализации данных, который нарушает непрозрачность ADT, потенциально раскрывая частные детали реализации.

3.3 Графический интерфейс компилятора

3.3.1 Выбор средств разработки

Для разработки веб-интерфейса для компилятора было принято решение использовать классические технологии создания веб-страниц HTML, CSS и JavaScript.

3.3.1.1 HTML

Язык разметки гипертекста или HTML — это стандартный язык разметки для документов, предназначенных для отображения в веб-браузере. Ему могут помогать такие технологии, как каскадные таблицы стилей (CSS) и языки сценариев, такие как JavaScript.

Веб-браузеры получают HTML-документы с веб-сервера или из локального хранилища и отображают документы в виде мультимедийных веб-страниц. HTML описывает структуру веб-страницы семантически и изначально включал подсказки для внешнего вида документа.

HTML-элементы — это строительные блоки HTML-страниц. С помощью HTML-конструкций изображения и другие объекты, такие как интерактивные формы, могут быть встроены в отображаемую страницу. HTML предоставляет средства для создания структурированных документов, обозначая структурную семантику для текста, такого как заголовки, абзацы, списки, ссылки, цитаты и другие элементы. Элементы HTML обозначаются тегами, написанными с использованием угловых скобок. Такие теги, как `<img />` и `<input />`, непосредственно вводят содержимое на страницу. Другие теги, такие как `<p>`, окружают и предоставляют информацию о тексте документа и могут включать другие теги в качестве подэлементов. Браузеры не отображают HTML-теги, но используют их для интерпретации содержимого страницы.

HTML может встраивать программы, написанные на языке сценариев, таком как JavaScript, который влияет на поведение и содержание веб-страниц. Включение CSS определяет внешний вид и расположение контента. Консорциум Всемирной паутины (W3C), бывший разработчик стандартов HTML и нынешний разработчик стандартов CSS, поощряет использование CSS вместо явного представления HTML с 1997 года. Форма HTML, известная как HTML5, используется для отображения видео и аудио, в основном с использованием элемента `<canvas>` в сотрудничестве с JavaScript.

3.3.1.2 CSS

Каскадные таблицы стилей (CSS) — это язык таблиц стилей, используемый для описания представления документа, написанного на языке разметки, таком как HTML. CSS является краеугольным камнем технологии World Wide Web, наряду с HTML и JavaScript.

CSS разработан для разделения представления и содержания, включая макет, цвета и шрифты. Такое разделение может улучшить доступность содержания; обеспечить большую гибкость и контроль при определении характеристик представления; позволить нескольким веб-страницам совместно использовать форматирование путем указания соответствующего CSS в отдельном `.css` файле, что уменьшает сложность и повторение структурного содержания; и позволить кэшировать `.css` файл для улучшения скорости загрузки страницы между страницами, которые совместно используют этот файл и его форматирование.

Разделение форматирования и содержимого также позволяет представить одну и ту же страницу разметки в разных стилях для различных методов визуализации, например, на экране, в печати, голосом (через речевой браузер или программу чтения с экрана) и на тактильных устройствах на основе шрифта Брайля. В CSS также есть правила для альтернативного форматирования, если контент доступен на мобильном устройстве.

Название “каскадный” происходит от указанной схемы приоритетов для определения того, какое правило стиля применяется, если определенному элементу соответствует более одного правила. Эта каскадная схема приоритетов предсказуема.

3.3.1.3 JavaScript

JavaScript, часто сокращенно JS, — это язык программирования, который является одной из основных технологий Всемирной паутины, наряду с HTML и CSS. Более 97% веб-сайтов используют JavaScript на стороне клиента

для поведения веб-страниц, часто используя сторонние библиотеки. Все основные веб-браузеры имеют специальный движок JavaScript для выполнения кода на устройствах пользователей.

JavaScript — это высокоуровневый, часто компилируемый язык, соответствующий стандарту ECMAScript. Он имеет динамическую типизацию, объектную ориентацию на основе прототипов и функции первого класса. Это мультипарадигма, поддерживающая событийно-управляемый, функциональный и императивный стили программирования. Он имеет интерфейсы прикладного программирования (API) для работы с текстом, датами, регулярными выражениями, стандартными структурами данных и объектной моделью документа (DOM).

Стандарт ECMAScript не включает никаких средств ввода/вывода (I/O), таких как сетевые средства, средства хранения данных или графические средства. На практике веб-браузер или другая система выполнения предоставляет API JavaScript для ввода/вывода.

Движки JavaScript первоначально использовались только в веб-браузерах, но сейчас являются основными компонентами некоторых серверов и разнообразных приложений. Наиболее популярной системой выполнения для такого использования является Node.js.

3.3.1.4 jQuery

jQuery — это библиотека JavaScript, предназначенная для упрощения обхода и манипулирования деревом DOM HTML, а также обработки событий, CSS-анимации и AJAX. Это бесплатное программное обеспечение с открытым исходным кодом, использующее разрешительную лицензию MIT. По состоянию на май 2019 года, jQuery используется 73% из 10 миллионов самых популярных веб-сайтов. Веб-анализ показывает, что это самая широко используемая библиотека JavaScript с большим отрывом, по крайней мере, в 3-4 раза больше, чем любая другая библиотека JavaScript.

Синтаксис jQuery разработан для облегчения навигации по документу, выбора элементов DOM, создания анимации, обработки событий и разработки приложений AJAX. jQuery также предоставляет разработчикам возможность создавать подключаемые модули поверх библиотеки JavaScript. Это позволяет разработчикам создавать абстракции для низкоуровневого взаимодействия и анимации, продвинутых эффектов и высокоуровневых, тематически настраиваемых виджетов. Модульный подход к библиотеке jQuery позволяет создавать мощные динамические веб-страницы и веб-приложения.

Набор основных функций jQuery — выбор, обход и манипулирование элементами DOM - поддерживаемый его селекторным движком (названным “Sizzle” с версии 1.3), создал новый “стиль программирования”, объединяющий алгоритмы и структуры данных DOM. Этот стиль повлиял на архитектуру других JavaScript-фреймворков, таких как YUI v3 и Dojo, позже стимулировав создание стандартного API Selectors.

Microsoft и Nokia поставляют jQuery на свои платформы. Microsoft включает его в Visual Studio для использования в рамках фреймворков Microsoft ASP.NET AJAX и ASP.NET MVC, а Nokia интегрировала его в платформу разработки виджетов Web Run-Time.

3.3.1.5 DarkReader

DarkReader — это расширение для браузера, которое позволяет с легкостью превращать любые сайты в сайты с темной темой, которая гораздо меньше напрягает глаза при длительном простмотре веб-страниц.

Темный режим — это дополнительный режим, который можно использовать для отображения в пользовательском интерфейсе преимущественно темных поверхностей. Он позволяет снизить уровень света, излучаемого экранами устройств, сохраняя при этом минимальные коэффициенты цветовой контрастности, необходимые для удобства чтения. Преимущества темного режима заключаются в том, что он улучшает визуальную эргономику, снижая напряжение глаз, облегчая настройку экранов в соответствии с текущими

условиями освещенности и обеспечивая комфорт использования ночью или в темной обстановке.

Кроме того, он экономит заряд батареи, позволяя использовать устройство в течение длительного времени без подзарядки. Обычно темную тему можно выключить или включить с помощью заметного значка-переключателя на экране. Кроме того, он размещается в меню или в настройках приложения.

Мы используем это расширение в формате библиотеки JavaScript для нашего сайта, чтобы обеспечить пользователям более приятный опыт использования.

3.3.1.6 Resizable.js

Resizable.js — это JavaScript библиотека, которая позволяет создавать блочные элементы веб-страницы с изменяемыми с помощью курсора мыши размерами.

Эта библиотека позволит пользователям настраивать экраны приложения нужным им образом, фокусируя внимания на том или ином промежуточном представлении кода.

3.3.1.7 CodeMirror

CodeMirror — библиотека на языке JavaScript, предоставляющий редактор кода в браузере с открытым исходным кодом, доступный под лицензия MIT. Работает в таких браузерах и программах как: Firefox, Chrome, и Safari, а также Light Table, Adobe Brackets, Bitbucket, и многих других проектах. CodeMirror имеет богатый программный API и ориентирован на расширяемость.

Первая версия редактора была написана в начале 2007 года для консоли веб-сайта Eloquent JavaScript. Код был впервые собран и выпущен под названием CodeMirror в мае 2007 года. Эта версия была основана на такой функциональной возможности браузеров как атрибут `contentEditable`. Это перечисляемый атрибут, указывающий, должен ли элемент редактироваться

пользователем. Если это так, браузер изменит свой виджет, чтобы разрешить редактирование.

В конце 2010 года Ace, еще один редактор кода на основе JavaScript, впервые применил новые методы реализации и продемонстрировал, что даже в JavaScript можно обрабатывать документы, состоящие из многих тысяч строк, без снижения производительности. Это побудило переписать CodeMirror по тем же принципам. В результате появилась версия 2, которая больше не полагалась на `contentEditable`, что значительно улучшило производительность редактора.

Мы используем эту библиотеку для того, чтобы предоставить пользователю возможность вводить код в интерактивном режиме прям в веб-приложении.

3.3.1.8 Tippy.js

Tippy.js — это очень гибкая библиотека, предоставляющая возможность создавать всплывающие подсказки при наведении мыши на ссылку или иной элемент веб-страницы. У него есть много своих тем оформления внешнего вида, включая возможность своей кастомизации. Плагин Tippy.js подключает в себя различные эффекты анимации при появлении всплывающих подсказок (англ. *tooltip*). В нем есть практически любой параметр на изменение, к примеру, скорость появления, место появления, множество вложений и очень многое другое.

Мы используем эту библиотеку для того, чтобы предоставить пользователям функцию всплывающих подсказок к AST, которые содержат более подробную информацию об атрибутах узлов.

3.3.1.9 Highlighter.js

Highlight.js — это подсветка синтаксиса, написанная на JavaScript. Он работает как в браузере, так и на сервере. Он может работать практически с

любой разметкой, не зависит от других фреймворков и имеет автоматическое определение языка.

Мы используем эту библиотеку для того чтобы подсвечивать синтаксис листинга ассемблерного кода, полученного на этапе кодогенерации.

3.3.2 Описание пользовательского интерфейса

Весь пользовательский интерфейс веб-приложения состоит из пяти модулей: редактор исходного текста, визуализаторы потоков токенов, CST и AST, а также модуль отображения результатов кодогенерации.

Рисунок 3.16 – Внешний вид интерфейса веб-приложения. На картинке можно видеть все модули UI.

Рассмотрим более подробно модуль редактора исходного текста. В качестве редактора, как уже было сказано выше мы используем библиотеку CodeMirror. Редактор поддерживает подсветку синтаксиса, подсветку парных скобок, а также мощнейший движок эмуляции редактора Vim.

Как можно видеть на скриншотах, в редакторе подсвечиваются строки кода, которые пользователь выделил курсором мыши в любом другом модуле.

При наличие ошибки в исходном коде, в редакторе будет отображено диагностическое сообщение, полченное от компилятора:

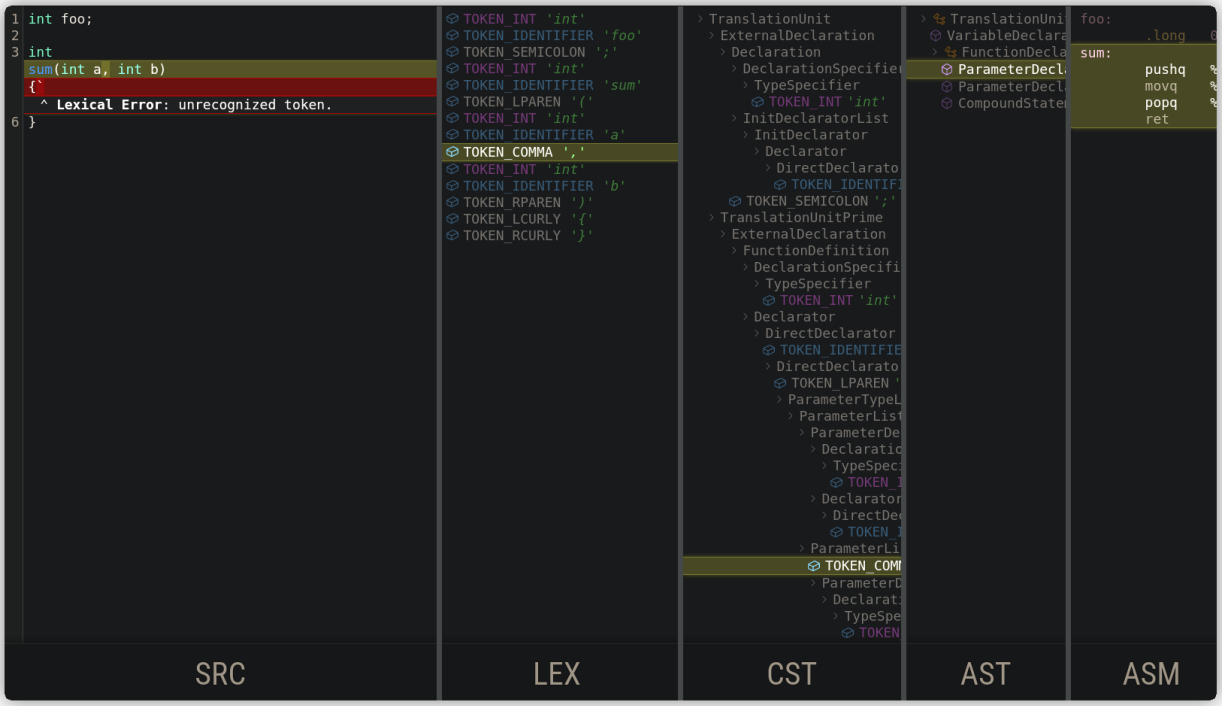

Рисунок 3.17 – Вывод диагностического сообщения в редакторе при лексической ошибке в исходном коде.

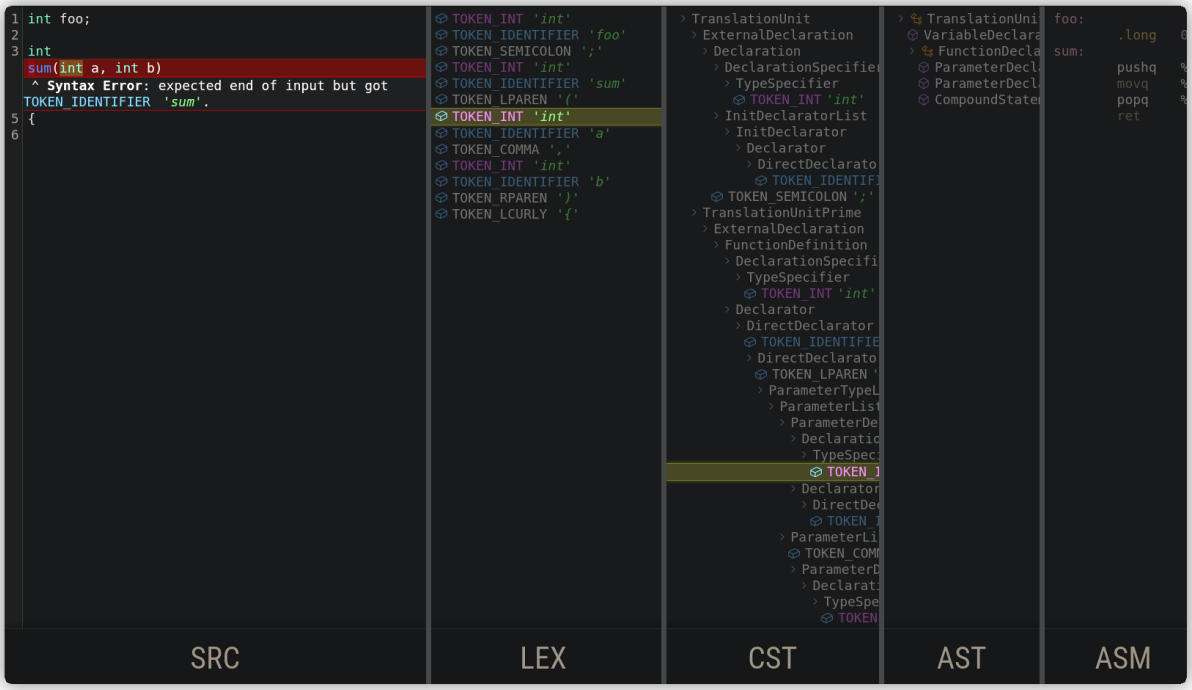

Рисунок 3.18 – Вывод диагностического сообщения в редакторе при обнаружении синтаксической ошибки при парсинге исходного кода.

Далее остановим наше внимание на модуле отображения дерева разбора. Модуль представляет собой визуализацию CST, полученного после прохода синтаксического анализа. В модуле присутствует функционал сворачивания дерева на любом из нужных улов. При наведении курсором мыши на узел, он и его дочернии узлы подсвечиваются, а неактивные части дерева затемняются для более комфортной работы с представлениями.

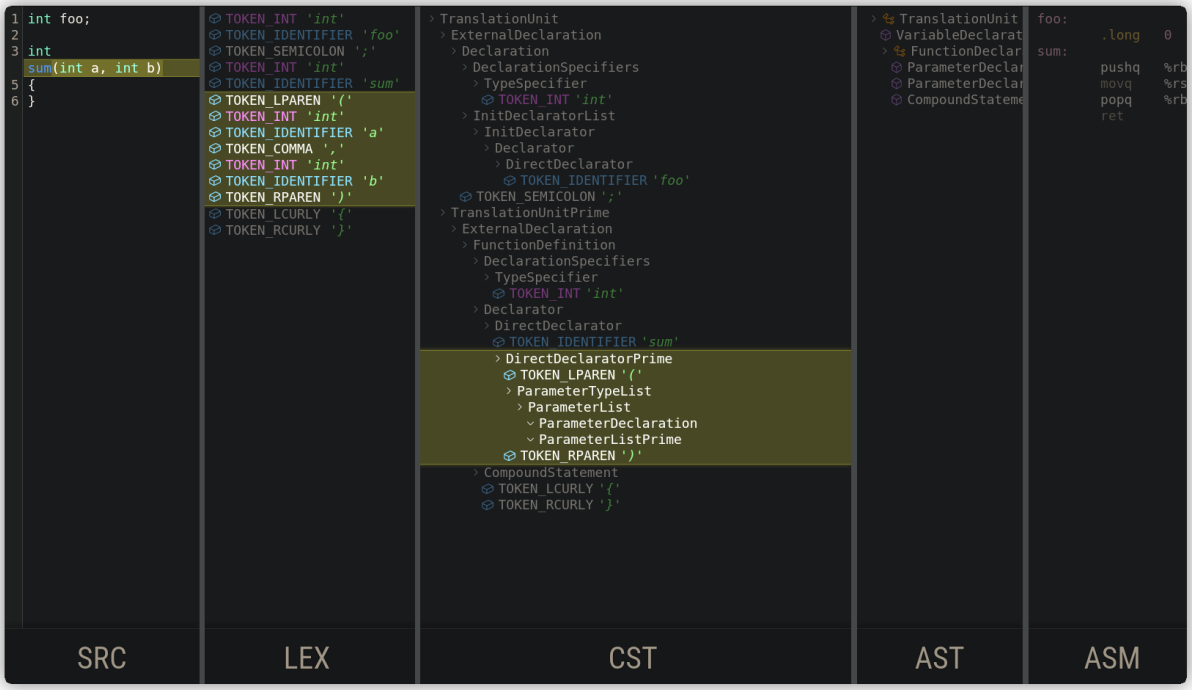

Рисунок 3.19 – На изображении можно наблюдать функционал сворачивания узлов дерева и выделение поддерва при наведении курсором мыши на элемент.

ЗАКЛЮЧЕНИЕ

В результате преддипломной практики было разработано программное обеспечение компилятора с возможность интроспекции промежуточных представлений, а также графический интерфейс пользователя к нему.

Для достижения поставленных целей были выполнены следующие задачи:

— Обоснование необходимости разработки программного обеспечения.

— Анализ предметной области.

— Сравнительный анализ существующих решений компиляторов для образовательных целей.

— Определение функционала системы.

— Обоснование выбора используемых технологий.

— Разработка и реализация компилятора.

— Разработка клиент-серверного приложения графического пользовательского интерфейса компилятора.

— Тестирование разработанного программного обеспечения.

Также:

— Были развиты навыки разработки архитектуры программного обеспечения.

— Были развиты навыки оформления рабочих результатов в виде научно-технического отчета.

— Были изучены основные требования, предъявляемые к содержанию и оформлению научно-исследовательских работ.

— Цель выпускной квалификационной работы была достигнута и все поставленные задачи для достижения цели были успешно.

Список использованной литературы

1. *Aho A. V.*, *Sethi R.*, *Ullman J. D.* Compilers: Principles, Techniques, and Tools. — Reading, Massachusetts : Addison-Wesley, 1986. — С. 585. — ISBN 0-201-10088-6.
2. *One Two Three*. Why is it so hard to recruit for compiler[-related] jobs? — URL: `https://softwareengineering.stackexchange.com/q/221413/359711`. Software Engineering Stack Exchange.
3. *Grune D.*, *Reeuwijk K. van*, *Bal H. E.* Modern Compiler Design. — 2-е изд. — ISBN 978-1-4614-4698-9.
4. *Sermutlu E.* Automata, Formal Languages, and Turing Machines. — 2020. — С. 348.
5. *babou*. lexers vs parsers. — URL: `https://stackoverflow.com/a/24165329/8086115`. StackOverflow.
6. *Ekstrand M.* What is the difference between an Abstract Syntax Tree and a Concrete Syntax Tree? — URL: `https://stackoverflow.com/a/1888973/8086115`. StackOverflow.
7. *Muchnick S. S.* Advanced Compiler Design Implementation. — Morgan Kaufmann, 1997. — ISBN 978-1-55860-320-2.
8. *Fischer C. N.*, *Cytron R. K.*, *Richard J. LeBLANC J.* Crafting a Compiler. — Addison-Wesley, 2010. — ISBN 978-0-13-606705-4.
9. *Misra R. B.* Global IT Outsourcing: Metrics for Success of All Parties // Journal of Information Technology Cases and Applications. — 2004. — Т. 6, вып. 3. — С. 21.
10. C Takes Programming Language Popularity Crown: TIOBE. — 2020. — URL: `https://insights.dice.com/2020/05/05/c-takes-programming-language-popularity-crown-tiobe`.
11. *Kernighan B. W.*, *Ritchie D. M.* The C Programming Language. — Prentice Hall, 1988. — С. 288.

12. *ISO*. ISO/IEC 9899:1999: Programming languages — C. — 1999.

13. *INCITS J11*, *SC22 WG14*. Rationale for International Standard — Programming Languages — C. — 2003.

14. *Harris D. M.*, *Harris S. L.* Digital Design and Computer Architecture. — 2-е изд. — Morgan Kaufmann, 2013.

15. *Hennessy J. L.*, *Patterson D. A.* Computer Architecture: A Quantiative Approach. — 5-е изд. — Morgan Kaufmann, 2012.

16. *Baumann J.* AST Transformations: Compiler Phases and Syntax Trees. — URL: `https://joesgroovyblog.blogspot.com/2011/09/ast-transformations-compiler-phases-and.html`.

17. *Gane C.*, *Sarson T.* Structured Systems Analysis: Tools and Techniques. — New York : Improved Systems Technologies, 1977. — С. 373. — ISBN 978-0930196004.

18. *Martin R. C.* UML for Java Programmers. — Prentice Hall, 2003. — ISBN 0-13-142848-9.

19. *Буч Г.*, *Рамбо Д.*, *Якобсон И.* Краткая история UML. — 2-е изд. — Москва : ДМК Пресс, 2006. — С. 14. 496 с. — ISBN 5-94074-334-X.

20. *Douglass B.* Real-Time UML. — 3-е изд. — Newnes, 2004. — ISBN 9780321160768.

21. *Penker M.*, *Eriksson H.-E.* Business Modeling with UML. — John Wiley & Sons, 2000. — ISBN 0-471-29551-5.

ПРИЛОЖЕНИЕ А

САРАТОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ ГАГАРИНА Ю.А.

УТВЕРЖДАЮ	УТВЕРЖДАЮ
Руководитель (должность, наименование предприятия — заказчика АС)	Руководитель (должность, наименование предприятия — заказчика АС)
Личная подпись Расшифровка подписи	Личная подпись Расшифровка подписи
Печать	Печать
Дата	Дата

Программного обеспечения «Компилятор языка С»

ТЕХНИЧЕСКОЕ ЗАДАНИЕ

На 21 листе

Действует с «___» _____ 2022 г.

СОГЛАСОВАНО

Руководитель (должность, наименование согласующей организации)

Личная подпись Расшифровка подписи

Печать

Дата

Саратов 2022

1 ОБЩИЕ ПОЛОЖЕНИЯ

1.1 Полное наименование системы и ее условное обозначение

Полное наименование системы: «Автоматизированная информационная система “Компилятор языка С”».

Краткое наименование системы: АИС «Компилятор».

1.2 Номер договора (контракта)

Шифр темы: АИС-КА-ФА-07.

Номер контракта: №1/11-11-11-001 от 29.09.2021.

1.3 Наименования организации-заказчика и организаций-участников работ

Заказчиком системы является Саратовский Государственный технический университет имени Гагарина Ю.А. Адрес заказчика: 410054, г. Саратов, ул. Политехническая, д. 77.

Разработчиком системы является Нефедов Данил Вадимович. Адрес разработчика: 415011, г. Саратов, ул. Московская, д. 27.

1.4 Перечень документов, на основании которых создается система

Основанием для разработки компилятора «*CC99*» являются следующие документы и нормативные акты:

— ISO/IEC 9989:1999. Programming languages — C.

— Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1: Basic Architecture.

— The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C & 2D: Instruction Set Reference.

— The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C & 3D: System Programming Guide.

— The Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers.

— System V Application Binary Interface. AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models).

— IEEE Std 1003.1-2017.

— ISO/IEC JTC 1 Working Group. Rationale for International Standard — Programming Languages — C.

— Documentation for GNU binutils 2.37.

1.5 Плановые сроки начала и окончания работы по созданию системы

Плановый срок начала работ по созданию АИС «Компилятор» — 29 сентября 2021 года.

Плановый срок окончания работ по созданию АИС «Компилятор» — 28 февраля 2022 года.

1.6 Порядок оформления и предъявления заказчику результатов работ по созданию системы

Исходный код разработанного программного обеспечения, список технических средств к закупке, а также инструкция по установке, настройке, запуске рабочего окружения и его сопровождение должна быть передана Заказчику в установленные им сроки. Приёмка и проверка работоспособности системы будет осуществлена комиссией, сформированной заказчиком.

1.7 Перечень нормативно-технических документов, методических материалов, использованных при разработке ТЗ

При создании и разработки программного обеспечения и проектно-эксплуатационной документации необходимо руководствоваться требованиями следующий документов:

— ГОСТ 34.602-89.

1.8 Определения, обозначения и сокращения

Сокращение	Расшифровка
ТЗ	Техническое задание
АИС	Автоматизированная информационная система
ЯП	Язык программирования
ABI	Application Binary Interface
AST	Abstract Syntax Tree, абстрактное синтаксическое дерево
3AC	Three Address Code
SSA	Static Single Assignment form

2 НАЗНАЧЕНИЕ И ЦЕЛИ СОЗДАНИЯ СИСТЕМЫ

2.1 Назначение системы

АИС «Компилятор» создается для образовательных целей и предназначена для обучения студентов и прочих желающих внутреннему устройству и построению в частности компиляторов, а в общем случае — программ-трансляторов с одного языка программирования в другой.

2.2 Цели создания системы

Основными целями создания АИС «Компилятор» являются:

— Создание компилятора языка C, который полностью или частично соответствует международному стандарту ISO/IEC 9899:1999.

— Создание компилятора, который поможет студентам и прочим желающим разобраться в работе программ-компиляторов и программ-трансляторов с одного ЯП в другой.

— Создание компилятора, который сможет скомпилировать сам себя (так называемая раскрутка, англ. *bootstrapping*).

Для реализации поставленных целей система должна решать следующие задачи:

— Переводить программу, написанную на ЯП C версии определенной и описанной в международном стандарте ISO/IEC 9899:1999 в язык ассемблера GNU Assembler с синтаксисом AT&T для архитектуры Intel® 64 в соответствии System V ABI.

— Иметь простой и понятный исходный код.

— Быть простой в использовании.

— Иметь возможность скомпилировать свой собственный исходный код.

3 ТРЕБОВАНИЯ К СИСТЕМЕ

3.1 Требования к системе в целом

3.1.1 Требования к структуре и функционированию системы

3.1.1.1 Перечень подсистем, их назначение и основные характеристики

В состав АИС «Компилятор» должны входить следующие подсистемы:

— Подсистема лексического анализа (лексический анализатор).

— Подсистема синтаксического анализа (синтаксический анализатор).

— Подсистема построения AST.

— Подсистема перевода AST в 3AC.

— Подсистема перевода 3AC в SSA.

— Подсистема перевода SSA в код ассемблера GNU Assembler с синтаксисом AT&T для архитектуры Intel® 64 в соответствии с System V ABI.

Подсистема лексического анализа предназначена для а) разбиения исходного кода входной программа, представленного в качестве массива байт в определенной кодировке, на отдельные лексические единицы ЯП C; б) выявления и диагностики лексических ошибок во входной программе.

Подсистема синтаксического анализа предназначена для а) перевода потока токенов, полученных от лексического анализатора в синтаксическое дерево по грамматике, соответствующей грамматике языка C, описанного международным стандартом ISO/IEC 9899:1999; б) выявления и диагностики синтаксических ошибок во входной программе.

Подсистема построения AST предназначена для а) перевода синтаксического дерева, полученного от синтаксического анализатора в AST; б) выявления и диагностики семантических ошибок во входной программе.

Подсистема перевода AST в 3AC предназначена для перевода абстрактного синтаксического дерева, полученного из подсистемы построения AST, в 3AC.

Подсистема перевода 3AC в SSA предназначена для перевод 3AC, полученного из предыдущей подсистемы в SSA.

Предназначение подсистемы перевода SSA в код ассемблера GNU Assembler с синтаксисом AT&T для архитектуры Intel® 64 в соответствии с System V ABI очевидно из ее названия.

3.1.1.2 Требования к способам и средствам связи для информационного обмена между компонентами системы

Требования не предъявляются.

3.1.1.3 Требования к характеристикам взаимосвязей создаваемой системы со смежными системами

АИС «Компилятор» должна взаимодействовать со следующими смежными системами:

— Операционная система.

3.1.1.4 Требования к режимам функционирования системы

Для АИС «Компилятор» определены следующие режимы функционирования:

— Нормальный режим функционирования.

В нормальном режиме функционирования системы АИС «Компилятор» программное обеспечение обеспечивает возможность функционирования по запросу пользователя.

3.1.1.5 Требования по диагностированию системы

Требования не предъявляются.

3.1.1.6 Перспективы развития, модернизации системы

АИС должна иметь возможность дальнейшей модернизации как программного обеспечения, так комплекса технических средств.

Необходимо предусмотреть возможность добавления одного или нескольких проходов оптимизации SSA; возможность добавления других бэкендов компилятора — то есть генерация другого ассемблерного кода, кода для другой архитектуры и/или ABI.

3.1.2 Требования к численности и квалификации персонала системы

Для эксплуатации АИС «Компилятор» определены следующие роли:

— Пользователь.

Основными возможностями пользователя являются:

— Компиляция исходной программы.

— Просмотр промежуточных стадий трансляции.

Пользователи системы должны иметь опыт работы с любой командной оболочкой совместимой со стандартом IEEE Std 1003.1-2017.

3.1.3 Показатели назначения

Требования на время отклика системы не налагаются и зависят от входных данных, переданных системе на обработку.

Система должна предусматривать возможность масштабирования по производительности и объему обрабатываемой информации без модификации ее программного обеспечения путем модернизации используемого комплекса технических средств.

3.1.4 Требования к надежности

Требования к надежности не предъявляются, так как это позволит сильно упростить исходный код АИС для понимания.

Система не обязана сохранять работоспособность и обеспечивать восстановление своих функций при возникновении каких-либо внештатных ситуаций.

3.1.5 Требования к эргономике и технической эстетике

АИС не должна иметь визуальный графический интерфейс. Все взаимодействие с АИС должно производится из командной оболочки соответствующей стандарту IEEE Std 1003.1-2017.

3.1.6 Требования к транспортабельности для подвижных АС

Требования не предъявляются.

3.1.7 Требования к эксплуатации, техническому обслуживанию, ремонту и хранению компонентов системы

Система должна быть рассчитана на эксплуатацию на персональных компьютерах пользователей.

3.1.8 Требования к защите информации от несанкционированного доступа

Требования не предъявляются.

3.1.9 Требования по сохранности информации при авариях

Требования не предъявляются.

3.1.10 Требования к защите от влияния внешних воздействий

Требования не предъявляются.

3.1.11 Требования к патентной чистоте

Разработка системы должна осуществляться в рамках рекомендаций по стандартизации Р50.1.028-2001 «Информационные технологии поддержки жизненного цикла продукции. Методология функционального моделирования».

3.1.12 Требования по стандартизации и унификации

— ГОСТ 18421-93.

— ГОСТ 19.001-77.

— ГОСТ 19.005-85.

— ГОСТ 19.402-78.

— ГОСТ Р 51904-2002.

3.1.13 Дополнительные требования

Дополнительные требования не предъявляются.

3.2 Требования к функциям (задачам), выполняемым системой

3.2.1 Подсистема лексического анализа

Подсистема лексического анализа должна разбивать исходный код входной программы, представленной в качестве массива байт в определенной кодировке, на отдельные лексические единицы ЯП C.

Подсистема лексического анализа должна предоставлять возможность экспортировать результат своей работы (полученный поток токенов) в произвольном формате.

Также для подсистемы допускается выявление и диагностика лексических ошибок во входной программе с выводом диагностических сообщений пользователю, что соответствует требованиям ISO/IEC 9899:1999.

3.2.2 Подсистема синтаксического анализа

Подсистема синтаксического анализа должна переводить потока токенов, полученных от лексического анализатора в синтаксическое дерево по грамматике, соответствующей грамматике языка C, описанного международным стандартом ISO/IEC 9899:1999.

Подсистема лексического анализа должна предоставлять возможность экспортировать результат своей работы (полученное синтаксическое дерево) в произвольном формате.

Также для подсистемы допускается выявление и диагностика синтаксических ошибок во входной программе с выводом диагностических сообщений пользователю, что соответствует требованиям ISO/IEC 9899:1999.

3.2.3 Подсистема построения AST

Данная подсистема должна переводит синтаксическое дерево, полученное от синтаксического анализатора в AST.

Также подсистема должна предоставлять возможность экспортировать результат своей работы (полученное абстрактное синтаксическое дерево) в произвольном формате.

Также для подсистемы допускается выявление и диагностика семантических ошибок во входной программе с выводом диагностических сообщений пользователю, что соответствует требованиям ISO/IEC 9899:1999.

3.2.4 Подсистема перевода AST в 3AC

Подсистема перевода AST в 3AC должна переводить абстрактное синтаксическое дерево в 3AC. Также подсистема должна предоставлять возможность экспортировать результат своей работы (полученный 3AC) в произвольном формате.

3.2.5 Подсистема перевода 3AC в SSA

Подсистема перевода 3AC в SSA должна переводить 3AC в SSA. Также подсистема должна предоставлять возможность экспортировать результат своей работы (полученный SSA) в произвольном формате.

3.2.6 Подсистема перевода SSA в код ассемблера

Подсистема должна переводить SSA в код ассемблера GNU Assembler с синтаксисом AT&T для архитектуры Intel® 64 в соответствии с System V ABI. Полученный результат должен быть выведен пользователю любым способом. Например, с помощью вывода в стандартный поток вывода или файл, указанный пользователем.

3.3 Требования к видам обеспечения

3.3.1 Требования к математическому обеспечению системы

Для лексического анализа допустимо и желательно применение готовых библиотек программных кодов, реализующих функции работы с регулярными выражениями.

Для синтаксического анализа требуется использовать алгоритм рекурсивного спуска как наиболее простого. Так как грамматика языка C является контекстно-зависимой, потребуется применение постпроцессинга полученного синтаксического дерева.

3.3.2 Требования к лингвистическому обеспечению системы

Исходя из целей, АИС должна быть разработка с применением языка программирования C версии, соответствующей международному стандарту ISO/IEC 9899:1999. Также допустимо применение такого языка как GNU Make для создания системы сборки АИС.

Языком взаимодействия с пользователем является английский язык (американский).

Окончательное требования к кодировке входных данных уточняется в процессе создания программного обеспечения АИС и не обязано быть согласовано с Заказчиком.

3.3.3 Требования к программному обеспечению системы

При проектировании и разработке системы необходимо использовать только библиотеки программных кодов с открытым исходным кодом.

Базовой программной платформой должна являться POSIX-совместимая операционная система, в которой используется GNU C Library.

3.3.4 Требования к техническому обеспечению

Требования к техническому обеспечению не предъявляются.

3.3.5 Требования к метрологическому обеспечению

Требования к метрологическому обеспечению не предъявляются.

4 СОСТАВ И СОДЕРЖАНИЕ РАБОТ ПО СОЗДАНИЮ (РАЗВИТИЮ) СИСТЕМЫ

Этап	Содержание работ	Результаты работ
1	Разработка документов технического проекта АИС «Компилятор»	Документы технического проекта АИС «Компилятор»
2	Проектирование, создание и тестирование программного обеспечения АИС «Компилятор»	Программное обеспечение АИС «Компилятор»
3	Приемка АИС «Компилятор»	Акт о приемке АИС «Компилятор»

5 ПОРЯДОК КОНТРОЛЯ И ПРИЕМКИ СИСТЕМЫ

5.1 Виды, состав, объем и методы испытаний системы

Каждая подсистема АИС должна пройти комплексное тестирование для проверки корректности работы. Должно быть также проведено интеграционное тестирование каждой подсистемы по отдельности и всех вместе для выявления проблем взаимодействия как внутри подсистем, так и подсистем между собой.

Кроме того АИС должна пройти инспекцию кода комиссией, в состав которой входят представители Заказчика и Исполнителя. Инспекция кода должна подтвердить, что исходный код АИС соответствует предъявляемым к нему требованиям качества.

5.2 Общие сведения к приемке работ по стадиям

Сдача-приемка осуществляется комиссией, в состав которой входят представители Заказчика и Исполнителя. По результатам приемки подписывается акт приемочной комиссии.

5.3 Статус приемочной комиссии

Статус приемочной комиссии определяется Заказчиком до проведения испытаний.

6 ТРЕБОВАНИЯ К СОСТАВУ И СОДЕРЖАНИЮ РАБОТ ПО ПОДГОТОВКЕ ОБЪЕКТА АВТОМАТИЗАЦИИ К ВВОДУ СИСТЕМЫ В ДЕЙСТВИЕ

Требования не предъявляются.

7 ТРЕБОВАНИЯ К ДОКУМЕНТИРОВАНИЮ

Требования не предъявляются.

8 ИСТОЧНИКИ РАЗРАБОТКИ

Учебники, учебные пособия, научные работы и другие материалы по а) построению трансляторов кода; б) формальным языкам; в) системному и низкоуровневому программированию. Документы, приведенные в пункте 1.4.