Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное 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 **09.03.01/03 Вычислительные машины, комплексы, системы и сети**

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ БАКАЛАВРА НА ТЕМУ:

Компилятор стекового языка программирования

Студент ИУ6-83Б (Группа) _____ (Подпись, дата) В.К. Залыгин (И.О. Фамилия)

Руководитель ВКРБ _____ (Подпись, дата) Б.И. Бычков (И.О. Фамилия)

Нормоконтролер _____ (Подпись, дата) _____ (И.О. Фамилия)

2026 г.

Задание.

Календарный план.

АННОТАЦИЯ

Расчетно-пояснительная записка посвящена процессу разработки компилятора стекового языка программирования. В исследовательской части рассмотрены различные особенности стековых языков программирования, проведено их сравнение, сделаны выводы об области применимости стековых языков. Затем описаны алгоритмы типизации для стековых языков, а также рассмотрена структура и модель исполнения виртуальной машины WebAssembly. В конструкторской части определена структура компилятора, выполнены проектирование и разработка компонентов, проведено комплексное тестирование программного обеспечения. Технологическая часть посвящена разработке технологии формирования SSA-представления для программ на стековых языках.

ANNOTATION

The settlement explanatory note is devoted to the development process of a stack programming language compiler. In the research part, various features of stack programming languages are considered, their comparison is carried out, conclusions are drawn about the field of applicability of stack languages. Then the typing algorithms for stack languages are described, and the structure and execution model of the WebAssembly VM are considered. The structure of the compiler has been defined in the design part, the design and development of components have been completed, and comprehensive software testing has been conducted. The technological part is devoted to the development of SSA representation generation technology for programs in stack languages.

РЕФЕРАТ

Расчетно-пояснительная записка состоит из 76 страниц, включающих в себя 8 рисунков, 11 таблиц, 13 источников и 3 приложения.

КОМПИЛЯТОР, СТЕКОВЫЙ ЯЗЫК, КОНКАТЕНАТИВНЫЙ ЯЗЫК, CAT, WASM, SSA

Объектом разработки является компилятор стекового языка программирования.

Цель работы – разработка и проектирование программы, представляющей собой компилятор стекового языка программирования Cat в модуль WebAssembly. Компилятор позволяет верифицировать и компилировать программы на исходном языке в байт-код.

Разрабатываемое программное обеспечение предназначено для использования программистами, которые пишут программы обработки данных на языке программирования Cat.

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ 7
ВВЕДЕНИЕ 9
1 Анализ синтаксиса и семантики стековых языков программирования 11
1.1 Аналитический обзор стековых языков программирования 11
1.2 Сравнительный анализ стековых языков программирования 15
1.2.1 Сравнение подходов к построению синтаксиса 15
1.2.2 Сравнение подходов к типизации и оптимизации программ 17
1.2.3 Выводы 18
1.3 Анализ средств вывода типов стековых языков программирования . . . 19
1.4 Анализ устройства байт-кода и модели исполнения WebAssembly 23
2 Проектирование, разработка и тестирование компилятора стекового языка программирования 27
2.1 Анализ процесса компиляции 27
2.2 Анализ задания и выбор технологии, языка и среды разработки 31
2.3 Разработка структуры программного обеспечения 32
2.4 Разработка компонентов приложения 34
2.4.1 Разработка интерфейса командной строки 35
2.4.2 Разработка парсера 38
2.4.3 Разработка статического анализатора 41
2.4.4 Разработка кодогенератора 44
2.5 Описание стратегий, способов, методов и проведения тестирования . . 46
2.5.1 Функциональное тестирование программного решения 47
2.5.2 Структурное тестирование компонента парсера 49
3 Разработка технологии построения SSA представления для программ на стековых языках 51
ЗАКЛЮЧЕНИЕ 58
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 59
ПРИЛОЖЕНИЕ А ТЕХНИЧЕСКОЕ ЗАДАНИЕ 60
ПРИЛОЖЕНИЕ Б ФРАГМЕНТ ИСХОДНОГО КОДА 67
ПРИЛОЖЕНИЕ В КОПИИ ЛИСТОВ ГРАФИЧЕСКОЙ ЧАСТИ 70

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящем отчете о ВКР применяют следующие сокращения и обозначения:

.NET CIL – .NET Common Intermediate Language, промежуточный язык (байт-код) платформы .NET, в который транслируются программы и который исполняется средой выполнения (CLR) или преобразуется в машинный код

.NET CLR – .NET Common Language Runtime, виртуальная машина платформы .NET, исполняющая программы на языке .NET CIL

S-выражение – скобочный формат записи, в котором на первом месте внутри скобок стоит оператор, а затем перечисляются операнды

Алгоритм Хиндли-Милнера – алгоритм вывода типов в полиморфных системах типов, автоматически определяющий типы выражений на основе их структуры и правил типизации без явных аннотаций (при соблюдении условий применимости системы)

Виртуальная машина – программная (либо программно-аппаратная) среда, реализующая абстрактную вычислительную модель и обеспечивающая выполнение программ в заданном формате (байт-код, промежуточное представление) независимо от конкретной аппаратной архитектуры

Интерпретатор – программа, которая выполняет исходный код, покомандно анализируя и исполняя команды «на лету»

Комбинатор – функция высшего порядка, задающая способ комбинирования других функций или программных фрагментов

Компилятор – программа, выполняющая преобразование исходного текста программы на языке программирования в эквивалентную форму представления (как правило, объектный код, машинный код или промежуточное представление), пригодную для последующего исполнения или дальнейшей трансляции

Лексема – минимальная значимая единица исходного текста программы, выделяемая на этапе лексического анализа (например, идентификатор, ключевое слово, литерал, оператор, разделитель)

Нетерминал – символ формальной грамматики, который может быть заменён последовательностью терминалов и/или нетерминалов по правилам вывода и используется для построения синтаксических конструкций языка

Нитевой код – способ представления программы в виде последовательности ссылок (адресов) на подпрограммы или примитивы исполнения, используемый для ускорения интерпретации и уменьшения объёма кода в стековых системах

Обратная польская запись – форма записи выражений, в которой операторы следуют за операндами (постфиксная нотация)

Рантайм – (англ. runtime, время исполнения) окружение программы, в котором она исполняется

РБНФ – расширенная форма Бекуса-Наура, нотация задания формальных грамматик, расширяющая БНФ средствами записи повторений, альтернатив, опциональных элементов и группирования

Стек – структура данных, организованная по принципу «последним пришёл – первым вышел» (LIFO), поддерживающая операции добавления и извлечения элементов с одной стороны, называемой вершиной стека

Терминал – символ формальной грамматики, который не подлежит дальнейшему разложению по правилам вывода и входит в алфавит конечных строк языка

ВВЕДЕНИЕ

Стековые (или стек-ориентированные) языки программирования характеризуются применением стека данных в качестве основного механизма передачи информации и хранения результатов вычислений. Стек-ориентированность позволяет программам на таких языках компактно выглядеть и эффективно исполняться.

Исторически первым стековым языком стал Forth, разработанный Чарльзом Муром в начале 1970-х годов. В дальнейшем в анализируемой области появляются новые разработки, вводящие формальные основы теории стековых языков. В начале 2000-х годов возрос интерес исследователей к более высокоуровневым стековым языкам. Был предложен термин «конкатенативность» для обозначения семейства стековых языков, в которых программа воспринимается как функция, преобразующая последовательность аргументов в последовательность результатов, а конкатенация функций в тексте эквивалентна их композиции во время выполнения (отсюда и название – конкатенативные языки).

Модель исполнения стекового языка – это некая (чаще всего виртуальная) стековая машина, имеющая стек данных, операционный стек и набор команд для работы со стеком. Например, реализация Forth оперирует стеком данных для хранения аргументов и результатов и стеком возвратов (операционным стеком) для адресов возврата при вызове подпрограмм [1]. Программа представляет собой последовательность команд, которые могут быть либо встроенными примитивами (например, арифметические операции, операции над стеком; интринсиками), либо определенными пользователем субпрограммами. В терминах стековых языков команды принято называть словами, или комбинаторами, или функциями. Каждое слово либо модифицирует состояние стека, либо изменяет поток исполнения программы. Большинство инструкций в такой машине берёт необходимые операнды с вершины стека, выполняет вычисление и кладёт полученный результат обратно на стек. Когда выполнение программы завершено, результат обычно считается

находящимся на вершине стека, откуда его можно извлечь для дальнейшего использования.

Поток исполнения программы контролируется при помощи счетчика команд (program counter), который указывает на текущую команду. Для большинства команд поток исполнения линеен: после выполнения одной команды машина берет на исполнение следующую команду (то есть инкрементирует значение счетчика до следующей команды). Исключение составляют команды условного и безусловного переходов, в некоторых реализациях стековых машин также присутствуют команды циклов. Для таких команд машина может менять значение счетчика неким особым образом согласно семантике конкретной команды. Следует обратить внимание, что отсутствие команд циклов не означает невозможность итерирования: в таком случае цикличность исполнения может быть достигнута при помощи команд ветвления и рекурсивных вызовов.

Таким образом модель исполнения (машина) содержит в себе следующие компоненты:

– стек данных;

– стек возвратов;

– счетчик команд;

– словарь поддерживаемых команд (в число которых могут входить команды модификации стека данных, а также команды модификации стека возвратов и счетчика команд), который может быть расширен в процессе исполнения программы.

1 Анализ синтаксиса и семантики стековых языков программирования

1.1 Аналитический обзор стековых языков программирования

Начиная с семидесятых годов прошлого века был создан ряд стековых языков. Далее рассмотрены некоторые из них, каждый из которых привнёс нововведения в область стековых языков. В обзор не включены, однако также заслуживают внимания:

– конкатенативный язык программирования Kitten, который продолжает идеи языка Cat в статическом анализе и представляет способ явного указания типов;

– .NET CIL и байт-код Java, которые также являются языками под стековые виртуальные машины (.NET CLR и JVM соответственно).

Forth – первый известный стековый конкатенативный язык, созданный Чарльзом Муром в начале 1970-х годов как расширяемый язык и интерактивная среда: ключевые слова можно добавлять прямо во время работы системы и таким образом формировать собственное подмножество языка под конкретную предметную область [1]. Синтаксис основан на обратной польской, или постфиксной, нотации: программа – это последовательность слов, разделённых пробелами. Интерпретатор идёт слева направо по программе и ищет лексемы в словаре доступных слов, после чего либо выполняет их, либо трактует как число и кладёт на стек. Язык и среда исполнения едины: система работает в режимах интерпретации и компиляции, переключаясь между ними. Для добавления нового слова или отложенных вычислений используется режим компиляции, для исполнения – режим интерпретации. Стандартной библиотеки в привычном смысле нет – функциональность наращивается подключаемыми наборами слов (word sets), зависящими от конкретной реализации. При этом ANS Forth (стандарт языка) задаёт базовый набор слов «core word set» со стековыми операциями, арифметикой, сравнениями, управлением потоком и примитивами памяти [2]. Система типов у Forth «бестиповая»: данные представлены машинными словами фиксированной разрядности, а корректность трактовки содержимого стека

контролирует программист, что повышает гибкость и предсказуемость времени выполнения за счёт единой семантики для любых данных, но усложняет статический анализ и проверку ошибок. Для Forth предложена оптимизация занимаемого программой пространства техникой « нитевого кода»: программа по сути является последовательностью безусловных переходов, каждый из которых ведёт к инструкциям, выполняющим ту или иную команду. Работа с памятью осуществляется через явные операции чтения и записи по адресу, что даёт низкоуровневое управление. Область применения Forth – прошивки, встроенные и ресурсно-ограниченные системы, системы реального времени и космическая техника.

На основе идей Forth в 2001 году Манфред фон Тун представил язык Joy как попытку формализации логики стековых языков: каждая команда в Joy обозначает унарную функцию вида «stack -> stack»: на вход подаётся стек данных (состояние программы), в процессе применения команды происходит его модификация и передача следующей команде, значения и подпрограммы при этом также передаются через стек. Программу на Joy можно рассматривать как математическую функцию «stack -> stack» без побочных эффектов [3]. Язык является динамически типизированным, за проверку типов несет ответственность программист, однако ему в этом могут помогать типы, описанные явно в объявлениях функций. Синтаксис Joy также основан на постфиксной записи. Фон Тун вводит идею комбинаторов – функций высшего порядка, которые принимают другие функции (в терминах стековых языков их принято называть цитатами) и образуют некий алгоритм обработки данных (например, комбинатор линейной рекурсии «primrec» принимает цитату тела рекурсии и цитату условия рекурсии). В стандартной поставке присутствует несколько библиотек, сгруппированных по областям применения: библиотеки «agglib» и «seqlib» содержат обобщённые операции над агрегатами (неупорядоченными множествами, строками и списками) и последовательностями, «numlib» включает числовые функции и численные методы, а «mtrlib» предназначена для работы с матрицами [4].

В 2006 году Кристофер Диггинс опубликовал несколько технических отчётов [5,6], в рамках которых описывает алгоритм статической типизации для стековых языков, в качестве примера используя свой экспериментальный язык Cat. Cat наследует идею Joy: любая программа рассматривается как преобразование вида «stack -> stack», а основная операция – композиция таких преобразований, реализуемая простой конкатенацией лексем в исходном тексте программы. В плане синтаксиса Cat аналогичен предыдущим и использует обратную польскую запись [6]. Грамматика показана на листинге 1 в виде РБНФ.

Листинг 1 — РБНФ грамматики Cat

```
программа    ::= { подпрограмма | определение } ;
подпрограмма ::= { терм } ;
терм         ::= слово | литерал | цитата ;

определение  ::= "define" имя подпрограмма ";" ;
цитата       ::= "[" подпрограмма "]" ;

литерал      ::= число | буль ;
слово        ::= имя | "+" | "-" | "*" | "/" ;
имя          ::= буква { буква | цифра | "_" | "-" } ;
число        ::= [ "+" | "-" ] цифра { цифра } ;
буль         ::= "true" | "false" ;
цифра        ::= "0"-"9" ;
буква        ::= "a"-"z" | "A"-"Z" ;
```

Статическая типизация в контексте стековых языков подразумевает, что в момент компиляции возможно построить конфигурацию стека (его размер и «форму» – типы значений, которые на нём лежат) для произвольного момента времени исполнения. Таким образом, вывод типов позволяет проводить статический анализ и трансформации. Подробности статического вывода типов описываются в подразделе 1.3. В языке присутствует несколько встроенных команд, их описание приведено в таблице 1. Помимо них в язык встроены арифметические и логические команды, а также команды сравнения чисел. Язык использует три типа данных – числа, булевы значения и цитаты.

Таблица 1 — Встроенные команды Cat

Команда	Описание
apply	Применить цитату с вершины стека к остальному стеку.
quote	Захватить вершину стека в цитату.
compose	Выполнить композицию двух цитат и положить на стек цитату-результат композиции.
dup	Положить на стек дубль вершины стека.
pop	Удалить значение с вершины стека.
swap	Поменять значение на вершине стека со значением под ним.
cond	Комбинатор ветвления, выбирает элемент на стеке или элемент под ним в зависимости от условия.
while	Комбинатор для цикла «пока».

Ещё одним ответвлением от Joy стал язык Factor, впервые опубликованный в 2003 году Славой Пестовым [7]. Преемственность Factor прослеживается по нескольким деталям: синтаксис на основе обратной польской записи, аналогичная идея использования и способ создания новых комбинаторов. Ключевая особенность – это первый прикладной высокоуровневый стековый язык общего назначения. Его можно так назвать благодаря богатой стандартной библиотеке (которая включает в себя, например, веб-фреймворк и парсер XML) и инструментарию разработки, собранному в одной IDE и включающему в себя компилятор с развитой системой анализов, интерактивный отладчик, браузер документации и инспектор объектов. Как и Joy, язык динамический, но имеет отличие: при объявлении новой функции программист должен указать её тип, а компилятор, в свою очередь, должен проверить этот тип на соответствие содержимому функции. Внутренний механизм этой проверки описывается как абстрактная интерпретация программы: компилятор (а точнее, модуль «Stack-checker» внутри него) симулирует эффекты команды на абстрактном стеке, при встрече ветвлений анализирует обе ветви и унифицирует состояния, а несовместимость размера или формы стека превращает в ошибки

времени компиляции [8]. На основе анализа типов компилятор проводит высокоуровневые и низкоуровневые оптимизации. Также дополнительно используются динамические проверки времени исполнения. За счет этих возможностей Factor можно назвать применимым для решения задач прикладного программирования.

Наконец рассматривается спецификация WebAssembly – одноимённых стекового языка и виртуальной машины, которые были разработаны и выпущены в 2017 году группой компаний W3C, Mozilla, Google, Microsoft, Apple и других. Wasm определяет низкоуровневый переносимый байт-код, архитектуру и семантику набора команд под виртуальную стековую машину, запускаемую преимущественно (но не только) в веб-браузерах наравне с другими движками [9]. В исходных целях проектирования Wasm зафиксированы требования к компактности двоичного представления, быстрой однопроходной валидации и компиляции, а также к «песочнице», пригодной для запуска недоверенного кода с производительностью низкоуровневого кода. Как стековый язык Wasm использует неявный стек данных: инструкции потребляют значения с вершины стека и помещают результаты обратно, при этом реализация не обязана хранить «настоящий» стек – спецификация описывает интерпретацию стека как набора анонимных регистров. Это первый из представленных в обзоре языков, который является целью компиляции, а не сам компилируется во что-либо. Поддержка компиляции в Wasm имеется для множества высокоуровневых языков, в первую очередь для полностью компилируемых языков C/C++/Rust. Для языка существует два представления: текстовое (формат «Wat» с синтаксисом на S-выражениях [10]), удобное для чтения человеком, и бинарное, понимаемое виртуальной машиной. Подробнее устройство байт-кода Wasm и модели его исполнения рассмотрены в подразделе 1.4.

1.2 Сравнительный анализ стековых языков программирования

1.2.1 Сравнение подходов к построению синтаксиса

Таблица со сравнением языков по синтаксису представлена в таблице 2.

Таблица 2 — Сравнение синтаксиса языков (сводная таблица)

Язык	Форма записи	Особенности синтаксиса
Forth	Постфиксная	Управляющие конструкции реализуются как слова времени компиляции.
Joy	Постфиксная	Цитаты используются как данные, управление через комбинаторы («i», «ifte», «primrec/linrec/binrec»).
Cat	Постфиксная	Синтаксис близок к Joy.
Factor	Постфиксная	Обязательное указание эффектов над стеком при объявлении функций.
Wasm	S-выражения	Нетрадиционная форма записи синтаксических конструкций

В общем случае в рассмотренных языках прослеживается следующая закономерность. Для языков, которые используются непосредственно программистами для написания программ вручную, применяется характерный конкатенативный синтаксис, что соответствует изначально заложенным в языки данного множества концепциям минималистичности и простоты. Вероятно, некоторую роль в сохранении такой формы играет фактор историчности: исторически первый стековый язык (Forth) обладал конкатенативным синтаксисом.

В то же время синтаксис языков, которые являются целями трансляции, представлен большим разнообразием. В частности, Wasm использует S-выражения – тоже одну из достаточно распространённых форм синтаксиса для функциональных языков. S-выражения также соответствуют концепции простоты, поскольку РБНФ грамматики таких языков имеет относительно малый размер.

Таким образом, отличительная черта синтаксиса стековых языков – минималистичный дизайн, который упрощает как механизмы разбора программ за счёт упрощённого парсера, так и работу программиста за

счёт меньшего объёма требуемых знаний о синтаксисе и высокой степени читаемости.

1.2.2 Сравнение подходов к типизации и оптимизации программ

Сравнение подходов к типизации приведено в таблице 3.

Таблица 3 — Сравнение подходов к типизации

Язык	Тип типизации	Доступный анализ программ
Forth (1971)	Бестиповой	Корректность интерпретации содержимого стека обеспечивается соглашениями и выявляется на этапе исполнения.
Joy (2001)	Динамическая	Динамическая проверка типов при выполнении.
Factor (2003)	Динамическая	Модуль «stack-checker» выполняет роль статического анализатора: абстрактная интерпретация эффектов слов на абстрактном стеке и проверка согласованности ветвей.
Cat (2006)	Статическая	Вывод типов и верификация формы и размера стека, гарантии согласованности ветвлений и инвариантов циклов.
Wasm (2017)	Статическая	Однопроходная валидация модуля по типам и структурированному управлению, предотвращение underflow и несогласованности типов до исполнения.

Согласно таблице 3 в процессе развития стековые языки постепенно улучшают свои возможности динамического и статического анализа. Бестиповость сменяется динамическими типами (предложена идея описания типов), затем были предложены алгоритмы вывода, позволяющие фиксировать типы в момент компиляции.

Основные техники оптимизации показаны в таблице 4.

Таблица 4 — Специфичные техники оптимизации для стековых языков

Оптимизация	Описание
Нитевой код (Forth)	Представление программы как последовательности ссылок на слова; снижает размер кода и удешевляет интерпретацию.
Разворачивание слов времени компиляции (Forth)	Трансляция управляющих конструкций в примитивные переходы и последовательности слов, уменьшающая накладные расходы на управление.
Инлайнинг цитат и макрорасширение (Factor)	Снижение накладных расходов вызовов и диспетчеризации за счёт разворачивания и специализации на этапе компиляции.

Оптимизации в стековых языках можно условно разделить на две группы: преобразования представления и диспетчеризации команд (они ускоряют интерпретатор или виртуальную машину) и оптимизации, использующие результаты анализа семантики (стек-эффекты, типы, структуру управления), что позволяет убирать лишние проверки и преобразовывать программу без нарушения корректности. Стековые языки изначально поддерживают техники первой группы вследствие своей структуры (например, нитевой код Forth). С развитием подходов к статическому анализу становятся доступны оптимизации из второй группы – Cat и Wasm активно применяют оптимизации, основанные на эквивалентных преобразованиях структуры программы, для увеличения скорости работы и снижения размера программы.

1.2.3 Выводы

В развитии стековых языков прослеживается тенденция: изначально они работают на низких уровнях абстракции благодаря простоте рантайма (по сравнению с нестековыми языками), высокой переносимости и оптимизациям, доступным при меньшей стоимости за счёт структуры программ и модели исполнения (например, нитевого кода Forth). Со временем стековые языки обзаводятся формальной базой, которая позволяет точно описывать работу программ и проводить развитый статический анализ и трансформации кода,

нацеленные на уменьшение объёма потребляемой памяти и уменьшение скорости работы.

Одной из вершин развития стековых языков можно назвать Wasm. Исходные цели проектирования во многом задают условия, подходящие для использования стекового языка, на основе которых выросла система, сочетающая сразу несколько изначально трудносовместимых преимуществ: высокую портируемость на разные платформы (по сравнению с регистровыми виртуальными машинами), высокую производительность (например, по сравнению с традиционными движками ECMAScript) за счёт низкоуровневых интерфейсов и оптимизаций, а также верифицируемость и, отчасти, безопасность исполнения. Wasm вбирает в себя теоретические и практические наработки в области стековых языков, решая с их помощью сложную задачу исполнения прикладного кода с учётом заданных требований.

В результате анализа можно сделать вывод, что современная ниша стековых языков – это быстрые виртуальные машины, запускаемые на широком спектре устройств. Это утверждение также подтверждается фактом, что спецификации самых распространенных виртуальных машин (JVM, .NET CLR, Wasm) используют стековую модель исполнения кода.

1.3 Анализ средств вывода типов стековых языков программирования

Использование стека данных в качестве единого хранилища порождает несколько возможных проблем типизации. Первая – проблема несовпадения типов, когда на вход некоторой команды с вершины стека поступают данные неправильного типа. Вторая – проблема исчерпания стека, когда команда при выполнении пытается взять данные с пустого стека. Такие ошибки приводят к аварийным завершениям программ и тем самым снижают их надёжность. В то же время от подобных ошибок можно защититься, если в момент компиляции программы статически определять состояния стека во все будущие моменты времени исполнения и в случае обнаружения ошибки блокировать компиляцию программы как небезопасной. Просчёт возможных состояний стека для программы на стековом языке называется выводом её типа.

В стековых языках тип некоторого выражения определяется эффектом, который это выражение оказывает на конфигурацию стека данных при выполнении. В свою очередь конфигурация стека состоит из размера стека (сколько данных на нём лежит) и «формы» стека (данные каких типов на нём лежат). Задача вывода типов для стековых языков сводится к нахождению типа для всей программы – того, как программа изменит конфигурацию стека при выполнении.

В общем случае выводом типов в языках программирования называется процесс, который позволяет установить неуказанный явно тип некоторого выражения в тексте программы по ограничениям окружения, в котором это выражение встречается. Разные языки обладают разными возможностями вывода в зависимости от размера области программы, которая используется для вывода типа. Некоторые языки допускают простые выводы, основанные на анализе только самого выражения, для которого выводится тип. Например, вывод типов в языке программирования Java ограничивается автоматическим определением типа переменной при её объявлении по выражению, которое инициализирует данную переменную. Другие языки идут дальше и позволяют выводить типы по совокупности выражений и совокупности переменных, задействованных в этих выражениях в рамках функций. К примеру, Rust позволяет выводить тип переменной из контекста выражений, где эта переменная используется: так, если функция возвращает целое 32-битное число, то для гипотетической переменной, из которой выполняется возврат значения и которая может быть объявлена где угодно внутри такой функции, выводится, соответственно, тип целого 32-битного числа. Наконец, существуют языки, вывод типов в которых осуществляется в рамках всей единицы трансляции – зачастую это языки с функциональной парадигмой. В таком случае типы для всей программы можно не указывать вовсе, так как все они будут выведены неявно при компиляции.

Для стековых языков алгоритмы вывода типов всей программы (последний случай, рассмотренный в предыдущем абзаце) актуальны, так как

зачастую программисту затруднительно самому вычислить тип программы, что и порождает ошибки работы со стеком данных.

Для вывода типов в функциональных языках используется алгоритм Хиндли-Милнера. Вкратце алгоритм при запуске над несколькими выражениями сводится к двум шагам: сначала вычисляются ограничения, которые задаются формой выражений (например, одна переменная равна другой или переменная участвует в сложении, а потому должна быть складываемой). Эти ограничения составляют систему уравнений, для которой на втором шаге ищется решение – такой набор типов, при подстановке которых уравнения будут тождественно верными. Если набор найден, то типы выведены успешно, иначе – ошибка типизации.

Согласно идее, предложенной Кристофером Диггинсом, для стековых языков использование алгоритма Хиндли-Милнера также возможно, но с поправкой на row-полиморфизм [5]. Row-полиморфизм – полиморфизм по размеру строки («row»), то есть по состоянию стека данных. Все команды row-полиморфны и оперируют всем стеком сразу, который состоит из хвоста (типового вектора, обозначаемого большими латинскими буквами) и единичных значений, лежащих на вершине стека (они обозначаются маленькими латинскими буквами либо названиями типов, например «a» и «Bool»). Для цитат используется запись со скобками, показывающая отложенность описанных вычислений; например, конфигурация стека, на котором лежит цитата, записывается как «A (A -> B)».

Диггинс демонстрирует возможность статического вывода типов на примере своего экспериментального стекового языка Cat. Для встроенных команд приводятся типы, показанные на листинге 2.

Стоит обратить внимание на ограничения, которые ставят описанные в листинге 2 типы. Для условного оператора «if» тип требует, чтобы обе ветви, принимая на вход одну и ту же конфигурацию стека, возвращали тоже одну и ту же измененную конфигурацию стека (при этом конкретные значения на стеке, конечно, могут различаться). Для цикла «while» тип требует, чтобы после

выполнения цикла конфигурация стека оставалась такой же, как и на момент начала цикла.

Листинг 2 — Типы встроенных команд Cat

```
// Применение цитаты, которая приводит конфигурацию S к
конфигурации R, к стеку S
apply   : (S (S -> R) -> R)
// Формирование цитаты, путем захвата значения с вершины стека
quote   : (S a -> S (R -> R a))
// Композиция двух цитат
compose : (S (B -> C) (A -> B) -> S (A -> C))
// Дублирование значения на вершине стека
dup     : (S a -> S a a)
// Удаление значения с вершины стека
pop     : (S a -> S)
// Перестановка двух значений с вершины стека
swap    : (S a b -> S b a)
// Примитив ветвления, выбирающий то или иное значения со
стека в зависимости от значения Bool
cond    : (S Bool a a -> S a)
// Примитив циклических операций, выполняющий тело (цитата на
вершине) пока верно условие (цитата под вершиной)
while   : ((S -> R Bool) (R -> S) S -> S)
```

Алгоритм вывода типа заключается в последовательном «сцеплении» типов команд, образующих программу. В своем техническом отчете Диггинс приводит текстовое описание алгоритма и пример, в котором происходит вывод типа для программы «[dup] apply» [5]. В результате применения алгоритма получается следующая цепочка типов: «A a -> A a (A a -> A a a) -> A a a» (примечание, в отчете приводятся только начальная и конечная конфигурации стека, но аналогичным образом возможно построить и промежуточные конфигурации). Как видно, цитата и команда «apply» взаимно уничтожаются, из-за чего результирующий тип эквивалентен применению одиночной команды «dup» с точностью до промежуточных конфигураций.

Предложенный алгоритм имеет ограничение, заключающееся в невозможности вычислять рекурсивные типы. Из-за этого ограничения нельзя вычислить тип для программ, которые имеют рекурсивный поток управления. Также нельзя вычислить тип программ, предполагающих рекурсию за счет использования конструкции «dup apply» (объяснить это можно так: «apply»

предполагает, что на стеке лежит некоторая цитата, которая переводит стек под собой из начального состояния в некоторое другое, но при этом «dup» говорит, что сама цитата также является частью этого начального состояния под собой). Такие ограничения можно снять, если расширить систему типов дополнительными конструкциями.

Таким образом, для стековых языков возможно статически вывести форму и размер стека и в случае наличия ошибок заранее их заметить. Признаком исчерпания стека служит наличие значений на вершине стека в начальной конфигурации программы – то есть программа ожидает, что на стеке до её запуска уже что-то лежит, хотя это не так. Признак несовпадения типов выражается в невозможности найти решение системы уравнений в процессе очередной сцепки типов команд в программе.

1.4 Анализ устройства байт-кода и модели исполнения WebAssembly

Байт-код WebAssembly (или, сокращённо, Wasm) спроектирован как низкоуровневая переносимая платформа с околонативной производительностью вычислений [10]. Поддерживаются два формата описания программ Wasm: бинарное представление, понимаемое виртуальной машиной Wasm, и текстовое представление в синтаксисе S-выражений, удобное для чтения человеком. В зависимости от цели одну и ту же программу можно представить в обоих вариантах.

Единицей трансляции Wasm является модуль – описанный спецификацией Wasm файл, содержащий коды программ, наборы метаданных, информацию об импортах и экспортах и другие составляющие [9]. Модуль является самостоятельной единицей, готовой к запуску на виртуальной машине. Для представления чисел используется формат little-endian (младшие разряды чисел записываются по младшим адресам). При кодировании векторов (последовательностей значений) в начале блока данных записываются 4 байта, задающие длину вектора, а затем кодируемые данные. Строки кодируются как байтовые векторы UTF-8.

Содержимое модуля состоит из преамбулы (значение «\0asm» и 4 байта номера версии) и основной части – набора секций. Каждая секция состоит

из байта-идентификатора, четырёх байт, задающих размер секции, и полезной нагрузки в соответствии с идентификатором секции. Секции могут идти в произвольном порядке, но рекомендуется располагать их по мере увеличения идентификаторов.

Помимо модуля, Wasm определяет также несколько других сущностей, которые описываются в модуле:

– функции – блоки исполняемого кода;

– таблицы ссылок – механизм реализации указателей на функции, динамических вызовов;

– глобальные переменные и константы – набор типизированных изменяемых или неизменяемых значений фиксированного типа;

– линейная память – непрерывные последовательности байт;

– теги – механизм исключений.

Сущности каждого вида объединяются в индексные пространства, которые используются для ссылок между секциями. Если сущность из адресного пространства задается несколькими секциями (что верно для функций, таблиц и линейной памяти, у которых объявление и определение разделены на разные секции, подробнее в таблице 5), то нумерация в этих секциях идет параллельно (например, функция с индексом 1 задается объявлением с индексом 1 и определением с индексом 1). Внутри индексного пространства нумерация сквозная и начинается с 0. Сначала нумеруются импортированные сущности в порядке перечисления в секции импорта, затем аналогично в порядке перечисления в соответствующих секциях нумеруются внутренние сущности.

Описание каждой секции приведено в таблице 5.

Вызов функций возможен либо напрямую через указание индекса нужной функции либо косвенно через таблицу ссылок. Для операций над данными используется стек данных. Чтобы выполнить некую операцию, необходимо положить на стек требуемые данные, операция берет операнды со стека и кладет на стек результат выполнения. Функции действуют

аналогично. Аналогично описанию в подразделе 1.3, тип функции описывает преобразование верхушки стека, которое осуществляет функция.

Таблица 5 — Описание секций модуля

Ид.	Название	Описание
0	Custom section	Игнорируется виртуальной машиной, хранит произвольные данные. Может использоваться для различных целей, в том числе для отладки.
1	Type section	Объявления доступных для использования типов функций (типы параметров и типы результатов). Остальные секции ссылаются на данную при необходимости указать тип.
2	Import section	Объявления сущностей, которые импортируются в модуль для работы. Определения импортируемых сущностей предоставляют модули, которые их экспортируют.
3	Function section	Объявления типов для определенных в модуле функций, имеет параллельную с секцией Code нумерацию.
4	Table section	Объявления таблиц ссылок, имеет параллельную с секцией Element нумерацию.
5	Memory section	Объявления линейной памяти.
6	Global section	Объявления глобальных переменных.
7	Export section	Объявления публичных сущностей, которые могут быть импортированы другими модулями.
8	Start section	Объявление индекса функции, которую необходимо вызвать при инстанцировании модуля.
9	Element section	Объявления элементных сегментов – заполнения таблиц.
10	Code section	Объявления тел функций.
11	Data section	Объявление сегментов данных.
12	Data count section	Опциональное указание числа сегментов данных (Data section), которое ускоряет валидацию модуля.
13	Tag section	Объявления тегов для реализации механизма исключений.

Спецификация Wasm использует идею «непрозрачных» указателей, которая выражается в том, что с точки зрения программы представление указателей не раскрывается, они выступают лишь значениями, которые ссылаются на что-то. Оперирование указателями осуществляется через отдельные команды. Указатели нельзя сохранять в линейную память – только в таблицы ссылок.

Wasm различает восстановимые и невосстановимые ошибки. Невосстановимые ошибки называются «trap» и приводят к аварийному завершению программы. Восстановимые ошибки реализованы как исключения. При возникновении исключения указывается его тег и машина раскручивает стек вызовов программы, пока не найдется обработчик соответствующего тега или стек вызовов будет раскручен полностью, и программа аварийно завершится.

Таким образом, спецификация WebAssembly предлагает низкоуровневую виртуальную машину. За счёт низкого уровня, небольшого количества абстракций и ряда инженерных решений (таких как стековая архитектура и непрозрачные указатели) достигается, во-первых, малая удельная стоимость исполнения команд, что делает программы Wasm сопоставимыми по скорости с нативными программами, а во-вторых, снижается сложность реализаций виртуальной машины для различных аппаратных платформ. Вследствие этого расширяется круг устройств, для которых реализована виртуальная машина и на которых возможно исполнение Wasm-программ. Эти преимущества делают Wasm подходящей платформой для запуска аппаратно-независимых программ на множестве устройств, что активно используется в веб-разработке и облачных сервисах.

2 Проектирование, разработка и тестирование компилятора стекового языка программирования

2.1 Анализ процесса компиляции

Компилятор служит промежуточным звеном между человеком, пишущим программу, и компьютером, который будет исполнять заданные программой команды. Таким образом, человек и компилятор образуют пару, ответственную за представление описания задач компьютеру в понятном для него виде – в виде низкоуровневых команд. Чтобы программу назвали хорошей, она должна выполнять поставленные задачи, притом делать это с минимальным количеством ошибок и эффективно использовать доступные ресурсы (далее по тексту под оптимальностью программы понимается эффективность использования ресурсов компьютера, а под оптимизацией – увеличение степени оптимальности). Каждый участник пары «человек и компилятор» имеет свою зону ответственности в достижении заданных для программы требований. Компилятор частично за минимизацию ошибок и частично за эффективность, проверяя и совершенствуя то, что описал человек на исходном языке.

Таким образом, можно выделить три задачи компилятора:

1) трансляция программ на исходном языке в целевое представление;

2) проверка программы на наличие ошибок (верификация);

3) оптимизация программы.

В зависимости от специфики предметной области эти задачи могут изменяться как в количестве, так и в определениях. Список выше представлен для предметной области стековых языков.

Процесс компиляции заключается в последовательном преобразовании входных данных – текста программы на исходном языке (в контексте этой работы, подмножестве языка Cat) – в некоторое количество промежуточных форм и наконец в финальную форму выходного файла.

Каждая из форм программы служит некоторой цели. Исходная форма обычно приспособлена для работы человека – она должна обеспечивать удобство читаемости программы, ее создания и редактирования. Стоит

отметить, что описанная в этой форме программа может уступать в эффективности, но должна превосходить в понятности и читаемости остальные формы программы.

Финальная форма полностью соответствует формату файла, ожидаемому на выходе в результате компиляции, то есть модулю WebAssembly. За счет исходной и финальной форм компилятор выполняет первую задачу.

Возможные назначения промежуточных форм более разнообразны и зачастую связаны с выполнением какой-либо из задач компилятора (которые описывались выше).

Как уже было сказано в подразделе 1.3, для проверки программы на стековом языке на наличие ошибок работы со стеком возможно использовать один из методов статического анализа – типизацию программы. Результат применения этого метода – представление типа программы. Это и есть первое из промежуточных представлений. Его назначение – выполнение второй поставленной выше задачи компилятора.

Для выполнения третьей задачи компилятора, оптимизации, было создано немало методов и техник. Оптимизация основывается на переписывании программы таким образом, чтобы наблюдаемые извне эффекты сохранились в неизменном виде, а сама программа стала более оптимальной. На текущий момент в индустрии компиляторов популярностью пользуется форма SSA представления (single static assignment, единственное статическое присвоение). SSA форма удобна для проведения большого количества оптимизаций, так как за счёт своих свойств заметно упрощает алгоритмы анализа и трансформации. Подробнее об SSA представлении рассказывается в разделе 3. Таким образом, ещё одно промежуточное представление – SSA форма – закрывает последнюю задачу, стоящую перед компилятором [11].

Для представления процесса трансляции можно воспользоваться методологией IDEF0. Контекстная диаграмма показана на рисунке 1.

Контекстная диаграмма отражает компилятор как единую функцию A0 «скомпилировать». Входом функции является текст программы, поступающий слева. Управляющими воздействиями выступают спецификация исходного

языка, управляющие флаги и спецификация WebAssembly, задающие правила разбора исходной программы, режим работы компилятора и требования к результирующему модулю. Выходами функции являются модуль WebAssembly и сообщения об ошибках. Механизмом выполнения служат вычислительные ресурсы компьютера, на которых реализуется процесс компиляции.

Диаграмма декомпозиции на рисунке 2 уточняет процесс компиляции.

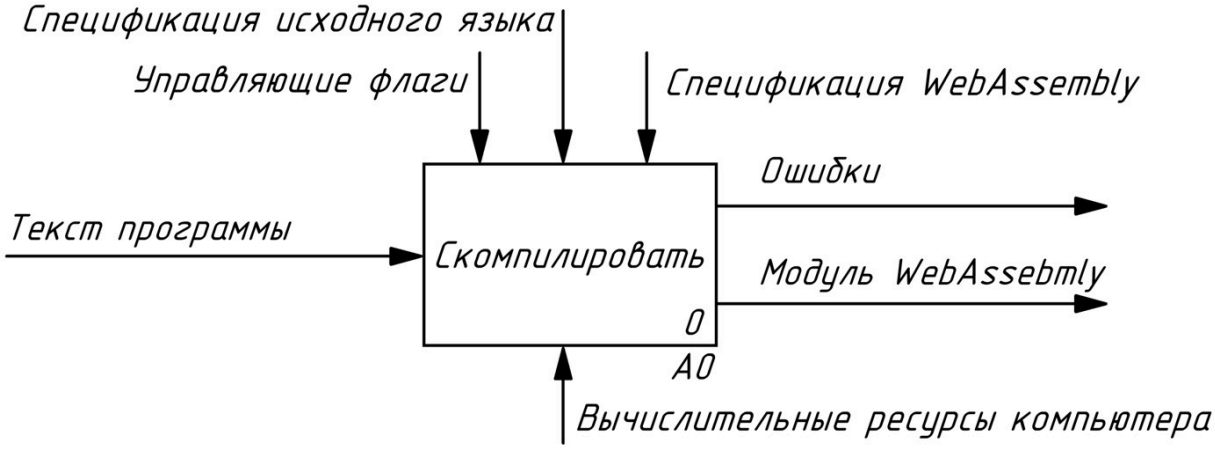


Рисунок 1 — Контекстная диаграмма

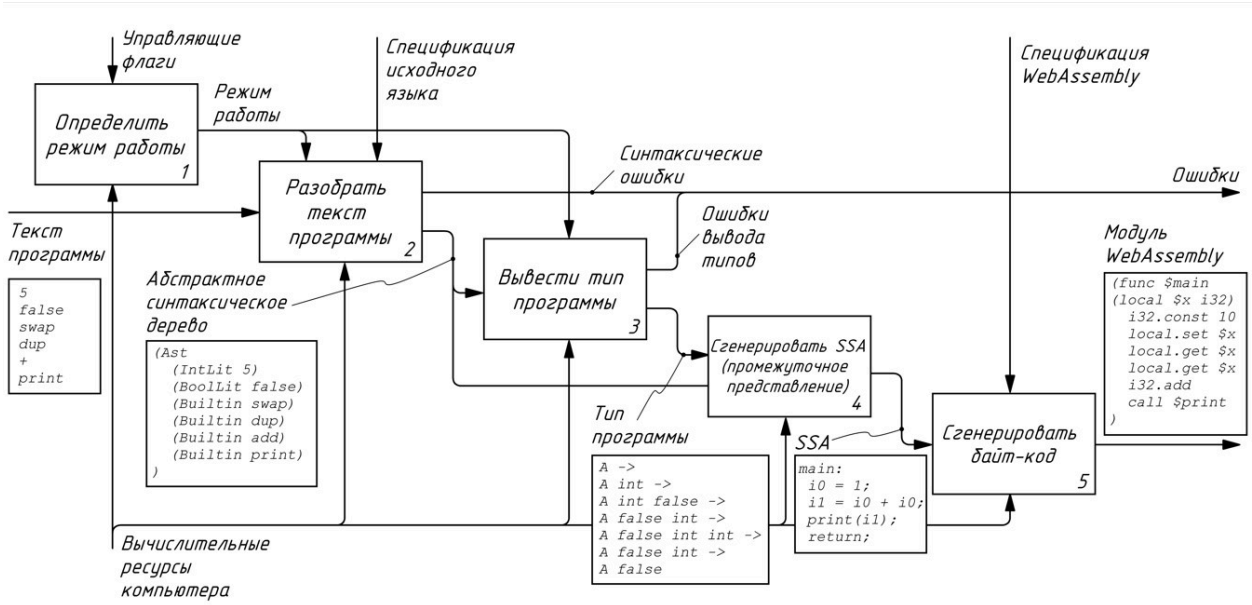


Рисунок 2 — Диаграмма декомпозиции

Процесс компиляции разделён на пять функций. Согласно техническому заданию, компилятор должен реализовывать несколько режимов работы: проверка программы на корректность и генерация модуля WebAssembly или только проверка. Этой логикой занимается блок 1, который по управляющим флагам определяет режим работы компилятора. Результатом работы блока

является выбранный режим работы, который управляет прохождением программы по последующим этапам обработки. В блоке 2 выполняется разбор текста программы в соответствии со спецификацией исходного языка. Результатом этого этапа является абстрактное синтаксическое дерево (AST). При невозможности корректного разбора формируются синтаксические ошибки, которые выводятся как один из результатов работы компилятора.

В блоке 3 по построенному абстрактному синтаксическому дереву выводится тип программы. Если вывод типов завершается успешно, получается представление типа программы, иначе формируются ошибки вывода типов. Полученное представление типа используется на следующем этапе. В блоке 4 на основе абстрактного синтаксического дерева и выведенного типа строится SSA представление, выступающее промежуточной формой для дальнейшего преобразования программы. Наконец, в блоке 5 по SSA представлению и с учетом спецификации WebAssembly генерируется итоговый байт-код в виде модуля WebAssembly.

Связи между блоками отражают последовательное преобразование форм программы: исходный текст преобразуется в абстрактное синтаксическое дерево, затем для него выводится тип, после чего строится SSA представление и на его основе создаётся конечный модуль WebAssembly. Вычислительные ресурсы компьютера выступают механизмом выполнения для этапов разбора, вывода типов, построения SSA представления и генерации байт-кода, то есть обеспечивают фактическую реализацию всех преобразований программы. На рисунке 2 также показаны листинги, демонстрирующие различные представления обрабатываемой программы в зависимости от этапа обработки.

По диаграмме на рисунке 2 можно сделать вывод: компиляция программы возможна только при её корректности. Для некорректной программы не получится построить абстрактное синтаксическое дерево либо вывести её тип. Обе структуры данных необходимы для построения SSA представления, которое, в свою очередь, используется для генерации байт-кода. Следовательно, для некорректной программы не существует представления в виде байт-кода, а значит, компиляция невозможна.

2.2 Анализ задания и выбор технологии, языка и среды разработки

В соответствии с требованиями технического задания необходимо разработать программу, которая может выполнять трансляцию кода на исходном языке. Компилятор должен обеспечивать поддержку ряда синтаксических конструкций, представляющих исходный язык. Модули байт-кода, являющиеся результатом работы компилятора, должны соответствовать спецификации WebAssembly [9]. Программное обеспечение должно работать под управлением ОС Linux и иметь интерфейс командной строки.

Исторически к программам-компиляторам предъявляются требования по скорости работы, нативности и наличию интерфейса командной строки. Иными словами, привычный компилятор – это скомпилированное нативное CLI-приложение без сборщика мусора.

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 сборщика мусора, а сам язык обладает наиболее строгой системой типов среди предложенных [12]. Указанные особенности Rust позволяют писать безопасное решение и не допускать ошибок в программном обеспечении. В таблице 6 показаны результаты сравнения языков программирования.

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

Поскольку процесс компиляции можно описывать как последовательность вызовов функций, поэтапно преобразующих код от исходного языка до байт-кода, рационально использовать структурный подход. Структурный подход также является идиоматичным при разработке на Rust.

Компилятор Rust автоматически контролирует состояние памяти, проверяя корректность работы с памятью на этапе компиляции. Проверка осуществляется за счет концепций «владения» и «заимствования». При написании программы программист должен объяснить компилятору, как

используется память в программе, а компилятор должен проверить, что такое использование безопасно. Преимуществом такого подхода по сравнению с другими автоматическими решениями является то, что все проверки осуществляются до начала исполнения программы.

Таблица 6 — 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

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

Наконец критерий «современные методы разработки» учитывает поддержку языком принятых в профессиональном сообществе стандартов написания кода и программных продуктов.

При создании программного обеспечения целесообразно проводить разработку нисходящим способом, как одним из рекомендуемых [13]. Версионирование программного обеспечения осуществляется при помощи инструмента git.

2.3 Разработка структуры программного обеспечения

Согласно подразделу 2.2 в рамках разработки был выбран структурный подход. Для работы при таком подходе необходимо уточнить структурную схему программного решения.

В проведенном в подразделе 2.1 анализе процесса компиляции выделено несколько функций: определение режима работы, разбор текста программы, вывод ее типа, генерация SSA и генерация байт-кода. Уместно использовать метод пошаговой детализации и для каждой функции выделить свой структурный компонент, который будет отвечать за выполнение этой функции. Также возможно дополнить список компонентами, которые не показаны на диаграмме декомпозиции A0 из-за недостаточной степени детализации, но имеют заметную с точки зрения структуры роль. Общий список компонентов и их описания приведены в таблице 7.

Таблица 7 — Компоненты программного обеспечения

Название	Назначение компонента
Программа-компилятор	Точка входа в программное обеспечение, управление остальными компонентами, определение режима работы
Компонент интерфейса командной строки	Предоставление интерфейса взаимодействия с программным обеспечением посредством терминала
Парсер	Разбор текстов программ, построение абстрактного синтаксического дерева
Статический анализатор	Проведение анализов и трансформаций программы: вывод типов, построение SSA представления, оптимизация SSA представления
Кодогенератор	Создание модулей WebAssembly согласно SSA представлению

На рисунке 3 изображена структурная схема проекта, составленная с точки зрения управления.

Рисунок 3 — Структурная схема программного обеспечения

На приведённой схеме компонент «Программа-компилятор» является главной частью программного обеспечения, которая обеспечивает управление остальными модулями. Компонент интерфейса командной строки предоставляет программисту способ взаимодействия с программой при помощи терминала и занимается разбором вводимых запросов, которые затем передаются управляющему компоненту. Парсер, статический анализатор и кодогенератор совместно выполняют функции трансляции программы согласно процессу на рисунке 2.

2.4 Разработка компонентов приложения

После определения структуры приложения необходимо разработать его компоненты согласно выбранному нисходящему подходу. Для повышения наблюдаемости и качества отладки каждый компонент должен уметь генерировать для оператора сообщения разной степени критичности, а уровень критичности должен задаваться оператором при вызове программного обеспечения. Уровни критичности сообщений приведены далее в порядке убывания приоритета:

– сообщения с невосстановимыми ошибками,

– сообщения с предупреждениями,

– отладочные сообщения.

Компоненты должны обладать низким уровнем сцепленности и высокой связностью, что оказывает положительное влияние на технологичность программы.

Далее рассмотрена разработка каждого компонента согласно подразделу 2.3.

2.4.1 Разработка интерфейса командной строки

Интерфейс командной строки представляет собой способ взаимодействия с программным обеспечением при помощи терминала. Поскольку согласно техническому заданию необходимо обеспечить работоспособность программы в рамках Unix-подобной операционной системы Linux, дальнейшее описание подразумевает использование стандартного для Unix-подобных систем терминала на основе командной оболочки Unix. Взаимодействие строится на основе модели «запрос-ответ»: оператор вызывает требуемую программу при помощи написания команды, которая состоит из названия программы, набора передаваемых параметров, а также операторов управления стандартными потоками ввода, вывода и ошибок.

В командной оболочке Unix принято соглашение об именовании параметров (флагов): перед именем параметра идет префикс в виде одного или двух символов «-», для именования параметра могут использоваться длинные и короткие имена. Короткие имена отличаются от длинных тем, что состоят только из одного символа. Для коротких имен префикс состоит из одного символа «-», для длинных – из двух «-». У параметров могут быть аргументы, которые в случае наличия записываются следом за именем параметра через пробельные символы. Аргументы могут быть также у программы как таковой, а не у конкретного параметра, в таком случае аргументы перечисляются в конце команды вызова.

При завершении программа возвращает код ответа, обозначающий результат вызова. Если код ответа равен 0, то программа завершилась успешно. В ином случае считается, что в работе программы имелись ошибки. Интерфейс компилятора должен соответствовать данной схеме обратной связи.

Для построения грамматики необходимо определить требования к интерфейсу. Согласно техническому заданию, необходимо поддержать указание различных режимов работы, что может осуществляться через выбор одного из двух доступных параметров: компиляция программы или только верификация. В качестве аргумента программы необходимо поддержать указание пути до файла с текстом программы, а также флаг, указывающий на имя результирующего файла с байт-кодом. Должны быть флаги, регулирующие уровень сообщений от компонентов компилятора. Далее приводится таблица 8, содержащая информацию о параметрах.

Таблица 8 — Допустимые параметры при вызове компилятора

Короткое имя	Длинное имя	Аргументы	Описание	Обязатель-ность
«-f»	«--verificate»	–	Включение режима верификации	Нет
«-c»	«--compile»	–	Включение режима компиляции	Нет
«-q»	«--quite»	–	Отключение вывода всех сообщений	Нет
–	–	Имя файла	Название для входного файла с программой	Да
«-o»	«--output»	Имя файла	Название для генерируемого выходного файла	Нет
–	«--log»	«debug» или «warn» или «error»	Определение общего уровня сообщений	Нет
–	«--log-parser»	«debug» или «warn» или «error»	Определение уровня сообщений парсера	Нет

Продолжение таблицы 8

Короткое имя	Длинное имя	Аргументы	Описание	Обязатель-ность
–	«--log-analyzer»	«debug» или «warn» или «error»	Определение уровня сообщений статического анализатора	Нет
–	«--log-codegen»	«debug» или «warn» или «error»	Определение уровня сообщений кодогенератора	Нет

Имя выходного файла по умолчанию соответствует имени входного файла без постфикса расширения. Например, для файла «foo.kol» выходной файл будет обладать именем «foo».

Значения для параметров уровня сообщений имеют следующую логику:

– при значении «debug» должны отображаться отладочные сообщения и остальные более критичные сообщения;

– при значении «warn» должны отображаться сообщения с предупреждениями и сообщения с ошибками;

– при значении «error» должны отображаться только сообщения с ошибками.

По умолчанию все параметры уровня сообщений имеют значение «warn».

Наконец, грамматику интерфейса командной строки можно представить в виде расширенной формы Бекуса-Наура (РБНФ), что и показано в листинге 3.

Правило «path» в грамматике соответствует корректной строке пути в рамках файловой системы Linux.

Для реализации интерфейса командной строки для программного продукта принято решение использовать популярную в экосистеме языка Rust библиотеку Clap, которая использует декларативный подход к описанию консольного интерфейса приложения.

Листинг 3 — РБНФ грамматики интерфейса командной строки компилятора

```
compiler  ::= "kolenka" { short | long } path ;
short     ::= "-f" | "-c" | "-q" | "-o" path ;
long      ::= "--verificate"
            | "--compile"
            | "--quite"
            | "--output" path
            | "--log", log_level
            | "--log-parser" log_level
            | "--log-analyzer" log_level
            | "--log-codegen" log_level ;
log_level ::= "debug" | "warn" | "error" ;
```

2.4.2 Разработка парсера

Согласно результатам анализа процесса компиляции в подразделе 2.1, а также структурной схеме из подраздела 2.3, компонент-парсер должен принимать на вход текстовое представление программы, осуществлять его разбор, а затем возвращать разобранный текст в виде абстрактного синтаксического дерева либо набор найденных ошибок.

Грамматика исходного языка Cat уже была представлена ранее в листинге 1. Синтаксис языка конкатенативен и представлен в постфиксной форме, при которой операнды записываются перед оператором (например, «2 2 +»), в противовес привычной инфиксной форме, при которой оператор записывается между операндами (например, «2 + 2»). Такая форма синтаксиса соответствует модели исполнения стекового языка, в которой, прежде чем выполнить некоторую операцию с данными, необходимо положить эти данные на стек – операнды должны идти по тексту программы раньше своих операторов.

За счёт конкатенативного синтаксиса структура абстрактного синтаксического дерева вырождается в абстрактный синтаксический список, элементы которого являются командами на исходном языке программирования. Команды включают в себя набор встроенных операций (показаны в таблице 1), операции добавления на стек числовых и булевых литералов, а также логические и арифметические операции. Структура, задающая абстрактное синтаксическое дерево, показана в листинге 4.

Листинг 4 — Структура абстрактного синтаксического дерева

```
struct Ast {
    pub(crate) program: Vec<AstNode>,
}

type Program = Vec<AstNode>;

enum AstNode {
    Int { value: i32 },
    Bool { value: bool },
    BuiltinIdentifier { value: Builtin },
    Identifier { value: String },
    Quote { value: Program },
    Define { id: String, value: Program },
}

enum Builtin {
    Eval,
    If,
    Add,
    Sub,
    Mul,
    Div,
    Less,
    LessOrEq,
    Great,
    GreatOrEq,
    Pop,
    Dup,
    Swap,
}
```

Структура «Ast» задаёт абстрактное синтаксическое дерево, определение «Program» задаёт программу как последовательность команд, перечисление «AstNode» задаёт узел абстрактного синтаксического дерева, перечисление «Builtin» описывает встроенные команды.

Для разбора выражений можно использовать идею комбинаторных парсеров. При таком подходе парсер – это функция, поглощающая строку и возвращающая распознанный терминал, нетерминал либо ошибку. Построение парсеров для сложных правил осуществляется при помощи комбинаторов – функций высшего порядка, которые принимают в качестве аргументов функции-парсеры и затем комбинируют их для достижения необходимой

логики. В качестве основы используются простейшие парсеры терминалов: например, парсер, поглощающий одиночный символ.

Далее рассматривается пример построения комбинаторного парсера строки «hello». Парсер должен возвращать положительный ответ в случае нахождения слова в переданной строке или ошибку в случае его отсутствия. Пусть парсер одной буквы обозначается как «char({letter})», где «{letter}» – некоторая буква. Пусть также дан комбинатор «сцепка», который принимает на вход два парсера и возвращает новый парсер, поочерёдно вызывающий для входящей строки два переданных, и который обозначается как «bind({parser1}, {parser2})», где «{parser1}» и «{parser2}» – передаваемые парсеры. Построенный на основе комбинаторного подхода парсер слова «hello» будет выглядеть как сцепление нескольких простейших парсеров букв при помощи введённого комбинатора, что и показано на листинге 5.

Листинг 5 — Комбинаторный парсер слова «hello»

```
bind(char(h),
bind(char(e),
bind(char(l),
bind(char(l),
     char(e)))))
```

Библиотека Nom предоставляет инструментарий на языке Rust для построения комбинаторных парсеров. В её арсенале имеются различные комбинаторы, а также набор простейших парсеров. Библиотека поддерживает описание позиционных ошибок, что помогает формировать для пользователя понятные сообщения о синтаксических ошибках с указанием конкретного места, где возникла та или иная ошибка. На листинге 6 показан пример построенного парсера. В листинге «all_consuming», «delimited» являются комбинаторами, а «multispace0» и «terms» – комбинируемыми парсерами. Функция «map» осуществляет преобразование распознанной строки в структуру «Ast».

В рамках работы принято решение использовать комбинаторные парсеры и библиотеку Nom за счёт возможности более быстрого прототипирования

и реализации грамматик по сравнению с традиционными подходами. При помощи Nom реализован парсер, работающий по методу рекурсивного спуска.

Листинг 6 — Комбинаторный парсер правила «program» грамматики языка Cat

```
fn program<'a, E: ParseError<&'a str>>(input: &'a str)
  -> IResult<&'a str, Ast, E> {
  map(
    all_consuming(
      delimited(
        multispace0,
        terms,
        multispace0)
      ),
    |terms| Ast { program: terms },
  )(input)
}
```

2.4.3 Разработка статического анализатора

Статический анализатор является сердцем компилятора и выполняет семантический анализ программ, а также проводит их трансформации. Анализатор делится на две части: модуль вывода типов и модуль построения SSA представления программы. Модули работают последовательно – сначала производится вывод типов программы, затем на основе выведенных типов строится промежуточное представление.

Вывод типов представляет собой первый и глобальный этап анализа всей программы. Описание типизации стековых программ приведено в подразделе 1.3. С точки зрения разработки, в данном аспекте стоят задачи формализации приведённого алгоритма и выбора структур данных. Формальное описание приведено в виде схемы алгоритма на рисунке 4.

Тип программы выводится путем присоединения типов узлов, из которых она состоит. Алгоритм проходится по каждому узлу программы и пытается сцепить тип узла с типом программы, получившимся до него.

Для выполнения сцепки крайняя правая конфигурация стека типа программы должна совпадать с крайней левой конфигурацией стека сцепляемого узла.

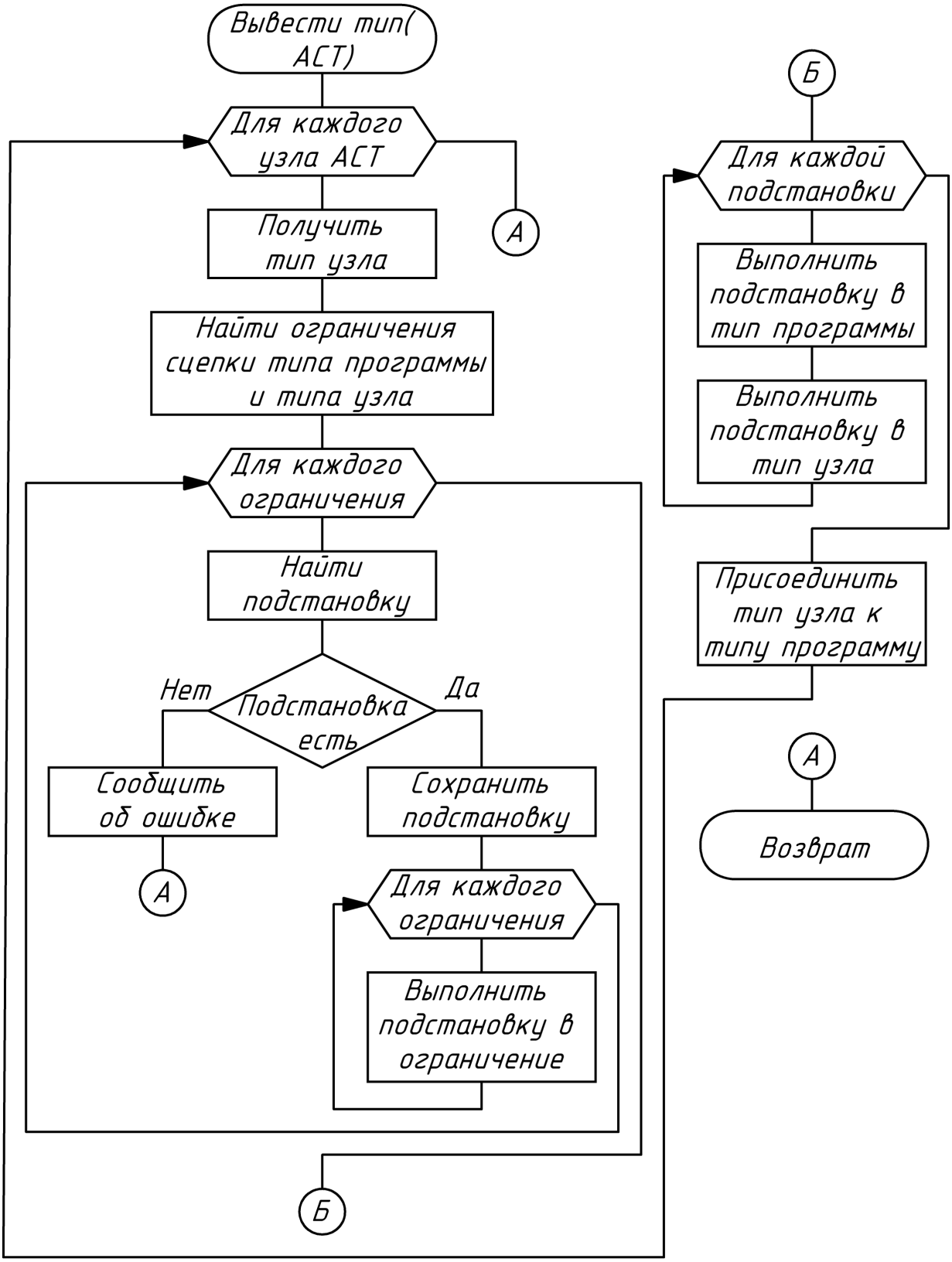


Рисунок 4 — Схема алгоритма вывода типов

Например, для сцепки типов «A -> A int» и «B -> B bool» (сцепки программы «5» и команды «true») необходимо добиться совпадения конфигураций «A int» и «B». В приведённом примере, как и в большинстве случаев,

конфигурации изначально не совпадают, и их необходимо унифицировать. Для унификации алгоритм ищет ограничения: в рассматриваемом примере конфигурации «A int» и «B» должны быть эквивалентны – это и есть найденное ограничение. Полученный таким образом набор ограничений является системой уравнений, которая затем последовательно решается: для каждого ограничения ищутся наиболее общие удовлетворяющие подстановки. В примере такой подстановкой будет «B => A int», то есть хвост стека «B» должен быть заменён на хвост «A int». Если найти подстановку не удаётся, то алгоритм завершается сообщением об ошибке. Каждая подстановка выполняется для всех ограничений, затем поиск подстановки повторяется уже для следующего ограничения. Набор ограничений впоследствии применяется к типу узла и типу программы. В рамках примера после применения подстановки типы выглядят следующим образом: «A -> A int» и «A int -> A int bool» – как видно, крайняя правая конфигурация типа программы (показана первой) теперь совпадает с крайней левой конфигурацией узла (показана второй). Операция присоединения заключается в наращивании типа программы конфигурациями стека присоединяемого типа, кроме его самой левой конфигурации: например, «A -> A int -> A int bool». После сцепки типов всех узлов алгоритм завершает работу.

После вывода типов начинает работать модуль промежуточного представления. В его зоне ответственности – генерация и трансформация промежуточного представления программы с целью оптимизации. Технология построения промежуточного представления в форме SSA описана в разделе 3.

После построения SSA представления выполняются алгоритмы оптимизации. Поскольку такие алгоритмы описаны во множестве источников, их рассмотрение выходит за рамки этой работы. Тем не менее в компиляторе используется ряд базовых оптимизаций:

– удаление неиспользуемых определений,

– удаление мертвого кода,

– инлайнинг функций (включение содержимого функций в код других функций).

2.4.4 Разработка кодогенератора

Описание структуры байт-кода и модели исполнения WebAssembly (сокращённо Wasm) дано в подразделе 1.4. Построение байт-кода Wasm связано с двумя вопросами:

1) генерация байт-кода из SSA представления,

2) работа с окружением из программы.

Генерация осуществляется при помощи линейного прохода по SSA представлению: каждая команда транслируется согласно шаблону в аналогичную конструкцию байт-кода. Для минимизации ошибок в работе используется библиотека «wasm-encoder». Библиотека предоставляет набор примитивов для автоматической генерации бинарного файла и берёт на себя задачи кодирования структур в корректный байт-код. В то же время «wasm-encoder» никак не контролирует структуру принимаемого описания – ответственность за корректную структуру и соблюдение всех конвенций лежит на плечах кода компилятора.

Для SSA представления с листинга 10 генерируется следующий байт-код. На листинге 7 приведено текстовое представление байт-кода (имеет название Wat).

В листинге 7 для взаимодействия с внешним миром программа использует функции «$read» и «$print» для ввода и вывода соответственно. Это встроенные функции стандартной библиотеки языка, которые компилятор неявно добавляет в байт-код в процессе кодогенерации. Компилятор Rust поддерживает WebAssembly в качестве цели компиляции. В связи с этой особенностью для написания стандартной библиотеки удобно использовать отдельный проект на Rust, в котором перечислены все функции, подмешиваемые в компилируемые модули. Изначально проект стандартной библиотеки компилируется в отдельный Wasm-модуль, который упаковывается в поставку разрабатываемого компилятора. При кодогенерации получается модуль с непосредственным кодом компилируемой программы. Компилятор осуществляет слияние модуля стандартной библиотеки и модуля программы при помощи библиотеки «wasm-merge». Итоговый модуль содержит все

необходимые определения для запуска – саму программу, а также набор вспомогательных функций стандартной библиотеки. На рисунке 5 показана схема создания итогового модуля.

Листинг 7 — Wat-описание программы «read dup 0 > [1 +] [1 -] if print»

```
(module
  (func $read (result i32)
  ;; содержимое функции чтения числа из стандартного потока
ввода
  )
  (func $print (param $i1 i32)
  ;; содержимое функции вывода числа в стандартного потока
вывода
  )
  (func $main_q1 (param $i1 i32) (result i32)
    local.get $i1
    i32.const 1
    i32.add
  )
  (func $main_q2 (param $i1 i32) (result i32)
    local.get $i1
    i32.const 1
    i32.sub
  )
  (func $main
    (local $i1 i32)
    (local $i5 i32)
    call $read
    local.set $i1
    ;; b1 = greater(i1, 0)
    local.get $i1
    i32.const 0
    i32.gt_s
    if
      local.get $i1
      call $main_q1
      local.set $i5
    else
      local.get $i1
      call $main_q2
      local.set $i5
    end
    local.get $i5
    call $print
  )
  (export "main" (func $main))
)
```

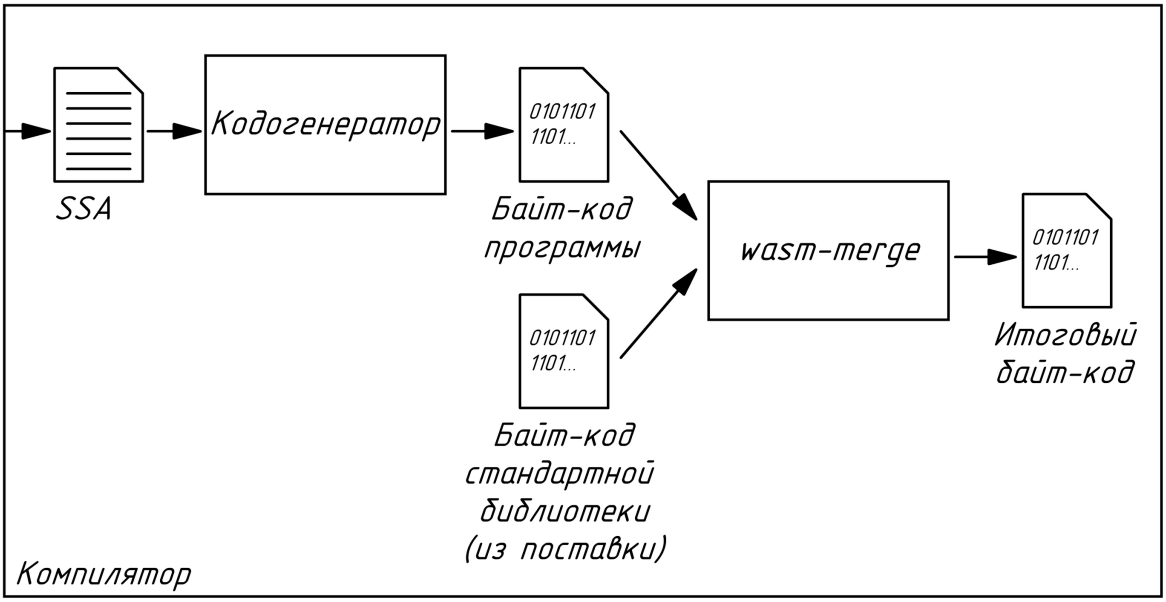


Рисунок 5 — Схема создания модуля WebAssembly

2.5 Описание стратегий, способов, методов и проведения тестирования

В качестве основной стратегии выбрано функциональное тестирование. Данная стратегия основывается на принципе «чёрного ящика», что делает её максимально близкой к опыту, который получает пользователь при использовании программного решения [13].

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

Для компонента парсера используется структурное тестирование. Модульные тесты, написанные в рамках стратегии структурного тестирования, позволяют гарантировать корректность работы каждой части парсера и компонента в целом. Наличие модульных тестов обусловлено сложностью

устройства парсера и его хрупкостью при внесении изменений. Для составления тестов используется метод покрытия по операторам.

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

2.5.1 Функциональное тестирование программного решения

Таблица 9 иллюстрирует выделенные классы эквивалентности. Для выделения классов эквивалентности использованы соображения о возможных значениях флагов вызова программы и возможных конструкциях, используемых в коде программ на исходном языке.

Таблица 9 — Выделенные классы эквивалентности

Параметр разбиения	Правильные классы эквивалентности	Неправильные классы эквивалентности
Флаги вызова	Известные флаги	Неизвестные флаги
Входной файл	Существующий входной файл	Несуществующий входной файл
Синтаксис	Корректный синтаксис, корректное использование конструкций языка	Неизвестный символ, Неизвестное имя
Типизация	Корректный тип программы	Несовпадение типов, исчерпание стека

Таблица 10 показывает фрагмент тестов по методу эквивалентного разбиения.

Таблица 10 — Фрагменты сквозных тестов

№	Класс эквивалентности	Аргументы вызова	Входной файл	Ожидаемый результат	Результат теста	Вывод
1	Неизвестный флаг	-a	Нет	Ошибка, неизвестный флаг	Ошибка, неизвестный флаг	Корректная работа
2	Известный флаг	--verificate	1 2 +	Программа успешно прошла проверку	Программа успешно прошла проверку	Корректная работа

Продолжение таблицы 10

№	Класс эквивалентности	Аргументы вызова	Входной файл	Ожидаемый результат	Результат теста	Вывод
3	Существующий входной файл	--verificate	1 2 +	Программа успешно прошла проверку	Программа успешно прошла проверку	Корректная работа
4	Несуществующий входной файл	--verificate	Нет	Ошибка, входной файл не найден	Ошибка, входной файл не найден	Корректная работа
5	Корректный синтаксис	--verificate	1 2 +	Программа успешно прошла проверку	Программа успешно прошла проверку	Корректная работа
6	Неизвестный символ	--verificate	1 @	Синтаксическая ошибка, неизвестный символ	Синтаксическая ошибка, неизвестный символ	Корректная работа
7	Неизвестное имя	--verificate	foo	Ошибка, неизвестное имя	Ошибка, неизвестное имя	Корректная работа
8	Корректный тип программы	--verificate	1 2 +	Программа успешно прошла проверку	Программа успешно прошла проверку	Корректная работа
9	Несовпадение типов	--verificate	true 1 +	Ошибка типизации, несовпадение типов	Ошибка типизации, несовпадение типов	Корректная работа
10	Исчерпание стека	--verificate	+	Ошибка типизации, исчерпание стека	Ошибка типизации, исчерпание стека	Корректная работа

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

2.5.2 Структурное тестирование компонента парсера

При использовании метода покрытия операторов должны быть тесты, покрывающие все операторы тестируемого компонента. В таблице 11 приведены результаты структурного тестирования.

Таблица 11 — Модульные тесты для компонента парсера

№	Покрываемые операторы	Код на исходном языке	Фактический результат	Ожидаемый результат	Вывод
1	Подпрограмма аксиомы	Пустая строка	(Ast)	(Ast)	Корректная работа
2	Ненулевая программа, слово-имя, встроенные слова + - * /	dup_1-a + - * /	(Ast (Identifier dup_1-a) (Builtin add) (Builtin sub) (Builtin mul) (Builtin div))	(Ast (Identifier dup_1-a) (Builtin add) (Builtin sub) (Builtin mul) (Builtin div))	Корректная работа
3	Числовой литерал со знаком	−123	(Ast (IntLit −123))	(Ast (IntLit −123))	Корректная работа
4	Булевы литералы	true false	(Ast (BoolLit true) (BoolLit false))	(Ast (BoolLit true) (BoolLit false))	Корректная работа
5	Цитата с вложенной программой	[1 dup]	(Ast (Quote (IntLit 1) (Builtin dup)))	(Ast (Quote (IntLit 1) (Builtin dup)))	Корректная работа
6	Определение слова с именем и телом	define mul_2-1 dup * ;	(Ast (Define mul_2-1 (Builtin dup) (Builtin mul)))	(Ast (Define mul_2-1 (Builtin dup) (Builtin mul)))	Корректная работа

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

3 Разработка технологии построения SSA представления для программ на стековых языках

Как уже было сказано в подразделе 2.1, промежуточное представление используется для анализа и оптимизации программ. В научном сообществе для обозначения промежуточного представления зачастую используется краткое англоязычное сокращение «HIR», расшифровывающееся как «High Intermediate Language» – «высокоуровневый промежуточный язык». В дальнейшем для краткости промежуточный язык будет обозначаться как HIR.

SSA (single static assignment, статическое единственное представление) форма – это подмножество HIR, широко используемое для проведения трансформаций над кодом. Причиной тому является набор свойств, которыми обладает SSA представление и которые позволяют упростить алгоритмы анализа и трансформации программы с целью максимизации некоторых её характеристик. Представление с единственным статическим присваиванием – это представление кода, при котором у каждой переменной есть единственное определение, сопровождающееся инициализацией [11].

Алгоритм построения SSA сводится к нескольким шагам:

1) проименовать используемые переменные и составить граф потока данных,

2) построить граф потока управления,

3) составить промежуточное представление с несколькими определениями,

4) свести получившееся представление к SSA.

На рисунке 6 показаны зависимости между структурами данных, используемыми для построения SSA. Стрелка от блока «АСТ» к блоку «Тип программы» проведена условно, так как в общем случае тип может выводиться и без абстрактного синтаксического дерева.

Потоком данных называется взаимодействие данных между собой в программе. В стековом языке без вывода типов изначально неизвестно, данные какого типа хранятся в той или иной ячейке стека в тот или иной момент времени исполнения. Нельзя также отследить, какие команды с какими

данными связаны. Построение потока данных возможно выполнить на основе проведённого вывода типов. В выведенном типе программы можно отследить, когда в стеке появилось то или иное значение, какая команда его произвела, а какая – использовала. Это возможно за счёт требований унификации для сцепки – идентификатор переменной «протаскивается» по всей программе за счёт подстановок с его участием. Именование переменных – это проставление явного имени ячейке стека, в которой на определённом промежутке времени хранится одно и то же значение.

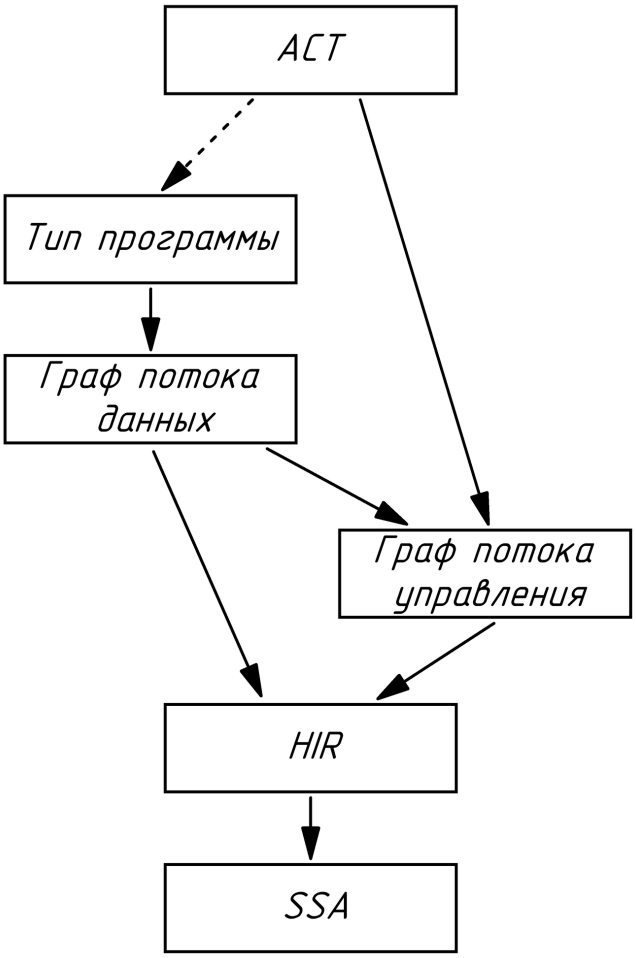


Рисунок 6 — Граф зависимостей между структурами данных

Для именования переменных необходимо сопоставить выведенный тип программы и последовательность команд. Для каждой команды существует правило сопоставления, результатом применения которого становится список переменных, которые команда использует, и список переменных, которые команда производит. Например, результатом сопоставления команды «+» и типа «A a b -> A C» станет список используемых переменных, состоящий из «a»

и «b», и список производимых переменных, состоящий из «c». Таким образом, можно построить граф потока данных в программе.

Пример потока изображён на рисунке 7. В окружностях записаны команды, в прямоугольниках – переменные (в том числе цитаты). Пунктирными стрелками показана зависимость по продукции, сплошными – зависимость по использованию, штрих-пунктирными – очерёдность команд в программе. Переменные именованы при помощи нумерации в рамках каждого типа (переменные численного типа начинаются с «i», булевого – с «b», цитаты – с «q»). Следует заметить, что переменных обобщённого типа в таком представлении быть не может: если некая цитата работает с переменной обобщённого типа, то на этапе формирования потока данных необходимо инстанцировать эту цитату под все возможные конкретные типы используемых переменных.

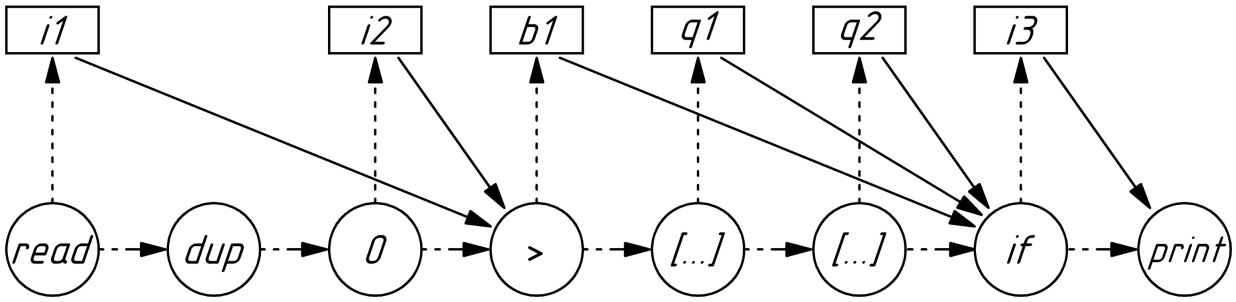


Рисунок 7 — Поток данных для программы «read dup 0 > [1 +] [1 -] if print»

Поток управления – путь, по которому вычислитель проходит по командам программы. Построить граф потока управления можно на основе абстрактного синтаксического дерева и потока данных: для команд вычисления цитат и вызова определений поток управления переходит к выполнению команд, соответствующих цитате или определению, затем возвращается и переходит к следующей по очереди команде; для остальных команд управления поток линеен – поочерёдно выполняются последовательно расположенные команды. Так как цитаты – это тоже данные, то зависящие от них команды (например, команда ветвления «if») используют граф потока данных для определения того, от каких именно цитат они зависят.

Пример графа потока управления показан на рисунке 8. На рисунке показано, как определения цитат, а также команда ветвления преобразуются в развилку, вызовы подпрограмм, а затем в схождение.

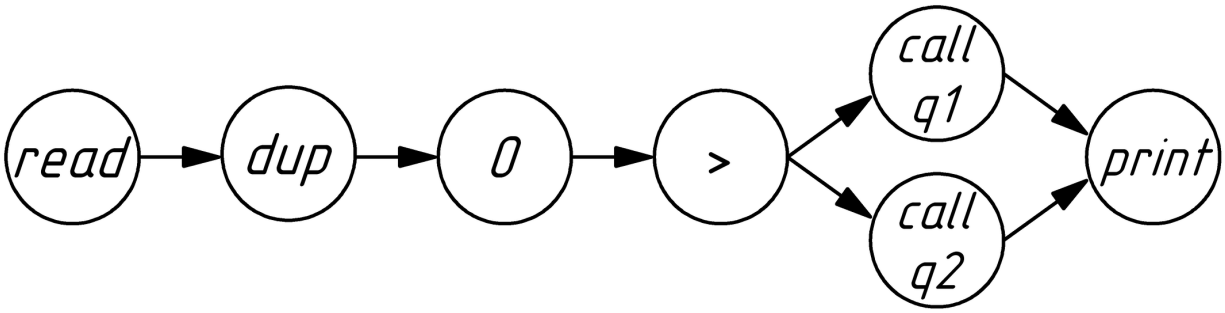


Рисунок 8 — Граф потока управления для программы «read dup 0 > [1 +] [1 -] if print»

После определения графов потока управления и данных следует построение промежуточного представления. Однако предварительно необходимо описать его структуру, чему и посвящены следующие три абзаца.

HIR – это регистровый (в значении, что для хранения данных используются именованные ячейки памяти, а не стек) язык, использующий трёхадресный код. HIR состоит из последовательности выражений, на каждое из которых можно ссылаться при помощи меток и которые могут быть объявлениями новых переменных либо переходами на определённые метки условно или безусловно. Граф потока управления проходит линейно по выражениям одно за другим и может меняться при встрече с безусловным или условным оператором перехода.

Программа в HIR разделена на базовые блоки. Базовый блок – последовательность выражений, которая начинается с метки и заканчивается условным или безусловным переходом к новой метке. Все функции строятся из базовых блоков. Грамматика языка показана на листинге 8.

Нетерминалы «alphabet» и «digits» представляют собой латинский алфавит и цифры соответственно. Для краткости правила их продукции не показаны.

Построение HIR выполняется при помощи наложения потоков управления и данных на структуру промежуточного представления. Компилятор проходит по потоку управления и составляет заготовку HIR. Для

каждой команды используется шаблон её трансформации в HIR, в котором не заполнены указания переменных. Затем при помощи потока данных между командами проставляются зависимости по данным (указания переменных).

Листинг 8 — РБНФ HIR

```
program    ::= { function } ;
function   ::= header { base_block } ;
header     ::= name "(" {param} ")" ":" ;
base_block ::= label { statement } ( if | goto | return ) ;
statement ::= var "=" function "(" { var } ")" ";" ;
if         ::= "if" "(" var  ")" "then" goto "else" goto ";" ;
goto       ::= "GOTO" label ";" ;
return     ::= "RETURN" { var };
label      ::= name ":" ;
param      ::= name ;
var        ::= name ;
function   ::= name ;
name       ::= ( alphabet | "_" ) { alphabet | digits |
"_" } ;
```

На листинге 9 показан пример HIR. Цитаты выделены как отдельные функции – так реализуется семантика их вызова, описанного в потоке управления. Команда ветвления в свою очередь трансформируется в несколько базовых блоков. Области видимости переменных и меток ограничены определениями функций.

Листинг 9 — HIR для программы «read dup 0 > [1 +] [1 -] if print»

```
main_q1(i1):
    i2 = imm(1);
    i3 = add(i1, i2);
    RETURN i3;
main_q2(i1):
    i2 = imm(1);
    i3 = sub(i1, i2);
    RETURN i3;
main():
    i1 = read();
    i2 = imm(0);
    b1 = greater(i1, i2);
    if (b1) then GOTO l1 else GOTO l2;
  l1:
    i3 = main_q1(i1);
    GOTO l3;
```

Продолжение листинга 9

```
  l2:
    i3 = main_q2(i1);
    GOTO l3;
  l3:
    i4 = print(i3);
    RETURN;
```

На заключительном этапе HIR приводится в соответствие SSA форме. Как можно заметить, листинг 9 не является SSA по причине двойного определения переменной «i3». Чтобы устранить это, приведение к SSA работает по следующему алгоритму [11]:

1) дублирующиеся переменные разводятся по разным именам;

2) в точках схождения потока управления объявляется новая переменная, равная функции φ от разведенных переменных;

3) код еще раз анализируется на наличие дублирующихся переменных (так как на пункте 2 возникли новые переменные, неучтенные в пункте 1).

Представленный алгоритм не является самым эффективным среди возможных, однако является одним из самых простых и был выбран именно по этой причине.

Таким образом, SSA представление в качестве примера может иметь следующий вид, показанный на листинге 10.

Листинг 10 — SSA форма для программы «read dup 0 > [1 +] [1 -] if print»

```
main_q1(i1):
    i2 = imm(1);
    i3 = add(i1, i2);
    RETURN i3;
main_q2(i1):
    i2 = imm(1);
    i3 = sub(i1, i2);
    RETURN i3;
main():
    i1 = read();
    i2 = imm(0);
    b1 = greater(i1, i2);
    if (b1) then GOTO l1 else GOTO l2;
  l1:
```

Продолжение листинга 10

```
    i3 = main_q1(i1);
    GOTO l3;
  l2:
    i4 = main_q2(i1);
    GOTO l3;
  l3:
    i5 = PHI(i3, i4);
    i6 = print(i5);
    RETURN;
```

ЗАКЛЮЧЕНИЕ

В результате выполнения квалификационной работы бакалавра разработано программное обеспечение компиляции стекового языка программирования. Основное назначение заключается в создании исполняемых файлов на основе текстов программ.

В ходе работы проведен анализ синтаксиса и семантики стековых языков программирования, выделение и сравнение их особенностей, рассмотрены средства статического анализа стековых языков программирования, структура байт-кода и модель исполнения WebAssembly, сформулировано определение области использования стековых языков. На основе результатов анализа сформированы требования к программной системе, определены задачи.

Для выполнения технического задания выполнен анализ процесса компиляции, определены средства разработки, спроектирован и разработан компилятор с возможностью проведения оптимизаций на основе современной теории компиляторостроения. Выделены и разработаны компоненты компилятора, необходимые средства представления программ. По окончанию разработки проведено функциональное и структурное тестирование, подтвердившие отсутствие ошибок в компиляторе и соответствие требованиям технического задания.

Также разработана технология построения SSA представления для программ на стековых языках, которая была использована при разработке одного из компонентов программного обеспечения.

В качестве направлений развития можно выделить расширение ряда поддерживаемых оптимизаций, повышение качества сообщений об ошибках, расширение списка поддерживаемых целей компиляции.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1) Forth documentation [Электронный ресурс]. URL: https://www.forth.com/resources/forth-programming-language/ (дата обращения: 25.09.2025).

2) ANS Forth standard [Электронный ресурс]. URL: https://forth-standard.org/standard/words (дата обращения: 25.09.2025).

3) Mathematical foundations of Joy [Электронный ресурс]. URL: https://hypercubed.github.io/joy/html/j02maf.html (дата обращения: 10.10.2025).

4) The prototype implementation of Joy [Электронный ресурс]. URL: https://hypercubed.github.io/joy/html/j09imp.html (дата обращения: 10.10.2025).

5) Christopher Diggins. Simple Type Inference for Higher-Order Stack-Oriented Languages. 2008.

6) Christopher Diggins. Typing Functional Stack-Based Languages. 2008.

7) Factor wiki [Электронный ресурс]. URL: https://concatenative.org/wiki/view/Factor (дата обращения: 20.11.2025).

8) Slava Pestov, Daniel Ehrenberg, Joe Groff. Factor: a dynamic stack-based programming language. 2010.

9) WebAssembly Specification 2.0 [Электронный ресурс]. URL: https://webassembly.github.io/spec/versions/core/WebAssembly-2.0.pdf (дата обращения: 11.11.2025).

10) Understanding WebAssembly text format [Электронный ресурс]. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Guides/Understanding_the_text_format (дата обращения: 11.11.2025).

11) Владимиров К.И. Оптимизирующие компиляторы. Структура и алгоритмы: 978–5–17–167965–1. — Москва: АСТ, 2024. 272 с.

12) Документация Rust [Электронный ресурс]. URL: https://www.rust-lang.org/learn (дата обращения: 29.01.2026).

13) Иванова Г.С. Технология программирования: Учебник для вузов. 3-е изд., стер. – Москва: КНОРУС, 2018. 334 с.

ПРИЛОЖЕНИЕ А

ТЕХНИЧЕСКОЕ ЗАДАНИЕ

Листов 6

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

УТВЕРЖДАЮ

Заведующий кафедрой ___ИУ6___

_____ А.В. Пролетарский

« __ » _____ 2026 г.

КОМПИЛЯТОР СТЕКОВОГО ЯЗЫКА ПРОГРАММИРОВАНИЯ

Техническое задание

Листов 6

Студент	ИУ6-83Б (Группа)	_____ (Подпись, дата)	В.К. Залыгин (И.О. Фамилия)
Руководитель		_____ (Подпись, дата)	Б.И. Бычков (И.О. Фамилия)

2026 г.

1 ВВЕДЕНИЕ

Настоящее техническое задание распространяется на разработку программы «Компилятор стекового языка программирования», используемой для трансляции в байт-код Wasm и верификации программ и предназначенной для создания исполняемых файлов по текстам программ, написанных программистом на конкатенативном стековом языке программирования Cat.

Актуальность разработки обусловлена новизной выбора цели трансляции. С точки зрения языка, прямая трансляции в WebAssembly позволит избежать дополнительных накладных расходов, связанных с тяжеловесной машиной .NET CLR, увеличить скорость работы и снизить потребление памяти Cat-программ. С точки зрения виртуальной машины, текстовый формат программ которой основан на использовании S-выражений, WebAssembly получит инструмент описания программ при помощи конкатенативного языка, что является более распространенным для стековых языков.

2 ОСНОВАНИЯ ДЛЯ РАЗРАБОТКИ

Компилятор для стекового языка программирования разрабатывается в соответствии с тематикой кафедры «Компьютерные системы и сети».

3 НАЗНАЧЕНИЕ РАЗРАБОТКИ

Основное назначение компилятора заключается в создании исполняемых файлов путем трансляции и верификации написанных программистом на исходном языке Cat текстов в байт-код, соответствующий спецификации виртуальной машины WebAssembly.

4 ИСХОДНЫЕ ДАННЫЕ, ЦЕЛИ И ЗАДАЧИ

4.1 Исходными данными для разработки являются следующие материалы:

4.1.1 Christopher Diggins. Simple Type Inference for Higher-Order Stack- Oriented Languages. 2008.

4.1.2 Владимиров К.И. Оптимизирующие компиляторы. Структура и алгоритмы: 978–5–17–167965–1. — Москва: АСТ, 2024. 272 с.

4.1.3 WebAssembly Specification 2.0 [Электронный ресурс]. URL: https://webassembly.github.io/spec/versions/core/WebAssemblyU2.0.pdf (дата обращения: 11.11.2025).

4.1.4 Cat language repository [Электронный ресурс]. URL: https://github.com/cdiggins/catlanguage (дата обращения: 13.10.2025).

4.2 Цель работы

Целью работы является прототип компилятора стекового языка программирования для трансляции и верификации программ и создания исполняемых файлов.

4.3 Решаемые задачи

4.3.1 Анализ синтаксиса и семантики различных стековых языков программирования, выделение и сравнение их особенностей.

4.3.2 Анализ средств статического анализа стековых языков программирования.

4.3.3 Анализ структуры байт-кода и модели исполнения WebAssembly.

4.3.4 Выбор технологии и средств разработки для программного обеспечения (далее ПО).

4.3.5 Разработка структуры программного обеспечения и определение его компонентов.

4.3.6 Проектирование программных компонентов.

4.3.7 Реализация компонентов ПО с использованием выбранных средств разработки.

4.3.8 Сборка программного обеспечения и его комплексное тестирование.

4.3.9 Разработка технологии построения SSA-представления текстов программ на стековых языках.

5 ТРЕБОВАНИЯ К ПРОГРАММНОМУ ИЗДЕЛИЮ

5.1 Требования к функциональным характеристикам

5.1.1 Выполняемые функции:

– статический анализ программы на предмет наличия ошибок несовпадения типов и исчерпания стека;

– генерация байт-кода WebAssembly.

5.1.2 Исходные данные:

– текст программы на исходном языке, который является подмножеством языка программирования Cat;

– управляющие флаги, регулирующие работу компилятора: выбор режима работы (только верификация или верификация с последующей кодогенерацией), уровня критичности выводимых в процессе работы сообщений, названия выходного файла.

5.1.3 Результаты:

– в случае успешной операции – файл с байт-кодом WebAssembly или сообщение, подтверждающее корректность программы;

– в случае неуспешной операции – сообщение с описанием ошибки.

5.2 Требования к надежности

5.2.1 Предусмотреть контроль консистентности управляющих флагов, передаваемых при вызове компилятора.

5.2.2 Предусмотреть контроль синтаксической корректности текста на исходном языке.

5.3 Условия эксплуатации

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

5.4 Требования к составу и параметрам технических средств

Минимальная конфигурация технических средств, на которых может быть запущен компилятор:

– количество ядер процессора – 1 шт;

– объем ОЗУ – 1 Гб.

5.5 Требования к информационной и программной совместимости

5.5.1 Программа должна иметь интерфейс командной строки.

5.5.2 Программа должна работать под управлением операционных систем семейства Linux.

5.6 Требования к маркировке и упаковке

Требования к маркировке и упаковке не предъявляются.

5.7 Требования к транспортированию и хранению

Требования к транспортировке и хранению не предъявляются.

5.8 Специальные требования

Сгенерировать установочную версию ПО.

6 ТРЕБОВАНИЯ К ПРОГРАММНОЙ ДОКУМЕНТАЦИИ

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

6.2 В состав сопровождающей документации должны входить:

6.2.1 Расчетно-пояснительная записка на 55-65 листах формата А4 (без приложений).

6.2.2 Техническое задание (Приложение А).

6.2.3 Руководство программиста (Приложение Б).

6.2.4 Исходный текст программного модуля вывода типов (Приложение В).

6.3 Графическая часть должна быть выполнена на 6 листах формата А1 (копии формата А3/А4 включить в качестве приложений к расчетно-пояснительной записке):

6.3.1 Схема процесса трансляции программы в байт-код WebAssembly и ее исполнения.

6.3.2 Результаты анализа стековых языков.

6.3.3 Схема алгоритма вывода типов.

6.3.4 Функциональная диаграмма программного обеспечения.

6.3.5 Диаграмма компоновки.

6.3.6 Схема алгоритма построения SSA-представления текстов программ на стековых языках.

7 ТЕХНИКО-ЭКОНОМИЧЕСКИЕ ПОКАЗАТЕЛИ

Требования не предъявляются.

8 СТАДИИ И ЭТАПЫ РАЗРАБОТКИ

Этапы разработки курсовой работы указаны в таблице А.1.

Таблица А.1 — Этапы разработки

№	Название этапа	Срок даты, %	Отчетность
1	2	3	4
1.	Разработка технического задания	9.02.2026 – 28.02.2026, 5%	Утвержденное техническое задание и задание на выпускную квалификационную работу
2.	Анализ требований и уточнение спецификаций (эскизный проект)	28.02.2026 – 14.03.2026, 25%	Спецификации ПО
3.	Проектирование структуры программной системы, проектирование компонентов (технический проект)	14.03.2026 – 01.04.2026, 50%	Диаграмма компоновки, функциональная диаграмма общего процесса работы, схема алгоритма вывода типов
4.	Реализация компонентов и автономное тестирование компонентов, сборка и комплексное тестирование	01.04.2026 – 14.05.2026, 75%	Тексты программных компонентов, тесты, результаты тестирования
5.	Разработка документации	14.05.2026 – 25.05.2026, 90%	Расчетно-пояснительная записка
6.	Прохождение нормоконтроля, проверка на антиплагиат, получение рецензии, подготовка доклада и предзащита	25.05.2026 – 6.06.2026, 95%	Иллюстративный материал, доклад, рецензия, справки о нормоконтроле и проценте плагиата
7.	Защита выпускной квалификационной работы	1.06.2026 – 04.07.2026, 100%	–

9 ПОРЯДОК КОНТРОЛЯ И ПРИЕМА

9.1 Порядок контроля

Контроль выполнения осуществляется руководителем еженедельно.

9.2 Порядок защиты

Защита осуществляется перед государственной экзаменационной комиссией (ГЭК).

9.3 Срок защиты

Срок защиты определяется в соответствии с планом заседаний ГЭК.

10 ПРИМЕЧАНИЕ

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

ПРИЛОЖЕНИЕ Б

ФРАГМЕНТ ИСХОДНОГО КОДА

Листов 2

Листинг Б.1 — Фрагмент исходного кода

```
/// Вывод (сцепка, chaining) общего типа для двух
последовательных типов
///
/// Выходная конфигурация `lhs` должна быть сопоставлена с
входной конфигурацией `rhs` (T-COMPOSE rule).
/// Сопоставление конфигураций -- попарное сопоставление
переменных на верхушках стеков конфигураций. Сопоставление для
цитат -- сопоставление их входных и выходных конфигураций.
/// В процессе сопоставления генерируются ограничения, для
которых затем ищется наиболее общее решение -- унификация.
Если решение не существует, то имеет место ошибка типизации.
fn chain(lhs: &Type, rhs: &Type, ctx: &mut Context) ->
Result<Type, TypingError> {
    let (mut lhs, mut rhs) = (lhs.clone(), rhs.clone());
    ctx.emit_debug(format!("types lhs {} rhs {}", lhs, rhs));

    let mut constraints: Vec<Constraint> =
constrain_chain(&lhs, &rhs, &mut ctx.step());
    let mut replacements: Vec<Replacement> = vec![];
    ctx.emit_debug(format!("constraints {}",
fmt_vec(&constraints)));

    {
        let ctx = &mut ctx.step();
        while !constraints.is_empty() {
            let constraint = constraints.pop().unwrap();
            ctx.emit_debug(format!("solve constraint {}",
constraint));
            let replacement = chain_solve(constraint)?;
            ctx.emit_debug(format!("by replacement {}",
replacement));
            let mut new_constraints: Vec<Constraint> = vec![];
            for constraint in &constraints {
                let mut constraints = constraint
                    .clone()
                    .apply_replacement(&replacement, &mut
ctx.step());
                new_constraints.append(&mut constraints);
            }
            replacements.push(replacement);
            constraints = new_constraints;
        }
    }

    ctx.emit_debug(format!("replacements {}",
fmt_vec(&replacements)));
```

Продолжение листинга Б.1

```
    for replacement in replacements {
        lhs = lhs.apply_replacement(&replacement);
        rhs = rhs.apply_replacement(&replacement);
    }

    ctx.emit_debug(format!("chained types lhs {} rhs {}", lhs,
rhs));

    Ok(lhs.append(rhs.seq.into_iter().skip(1)))
}
```

ПРИЛОЖЕНИЕ В
КОПИИ ЛИСТОВ ГРАФИЧЕСКОЙ ЧАСТИ

Листов 6

ВКРБ включает в себя следующий перечень графического материала:

1) схема процесса трансляции программы в байткод WebAssembly и ее исполнения;

2) результаты анализа стековых языков;

3) схема алгоритма вывода типов;

4) функциональная диаграмма программного обеспечения;

5) диаграмма компоновки;

6) схема алгоритма построения SSA-представления текстов программ на стековых языках.

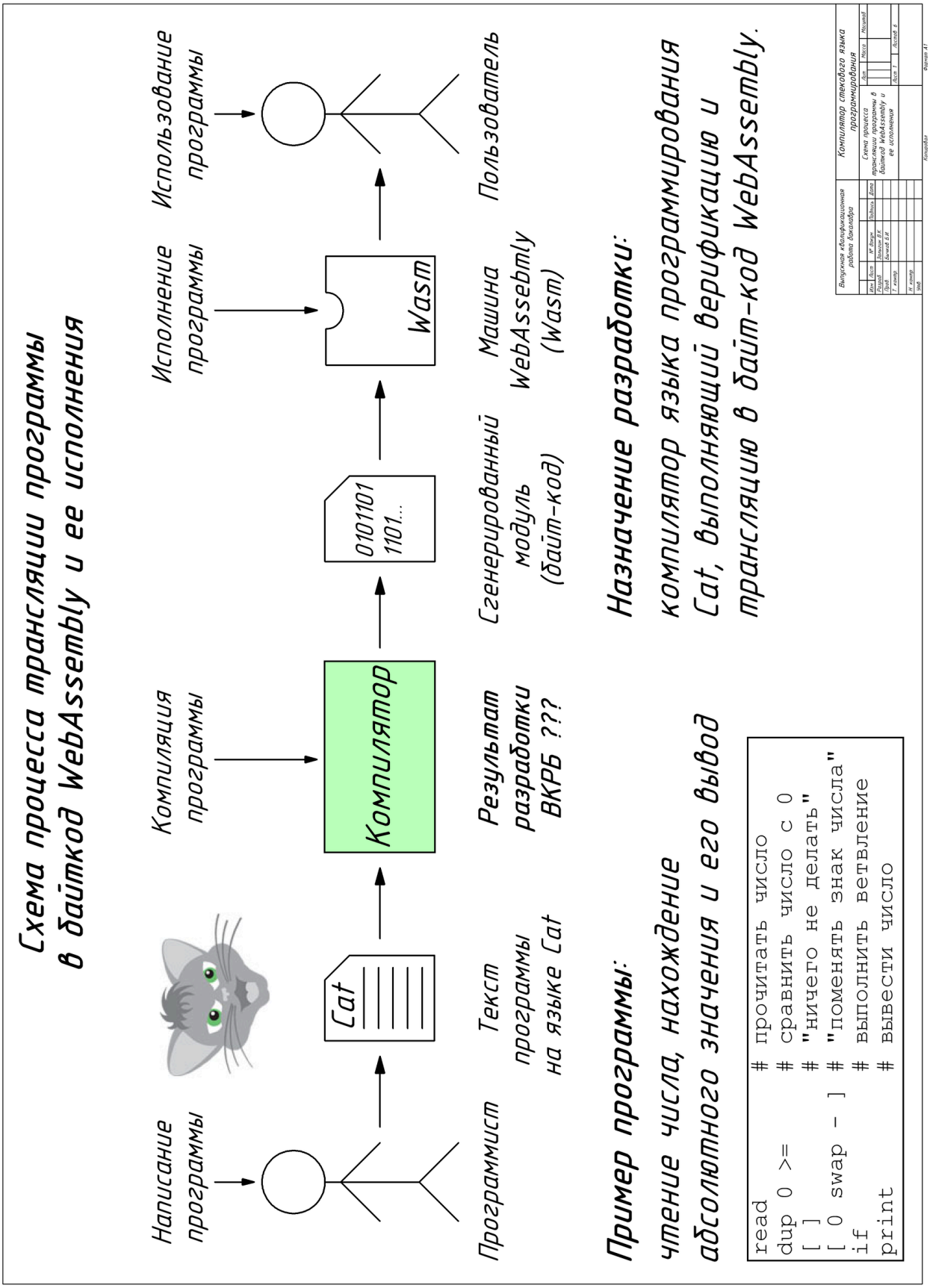


Рисунок В.1 — Схема процесса трансляции программы в байткод WebAssembly и ее исполнения

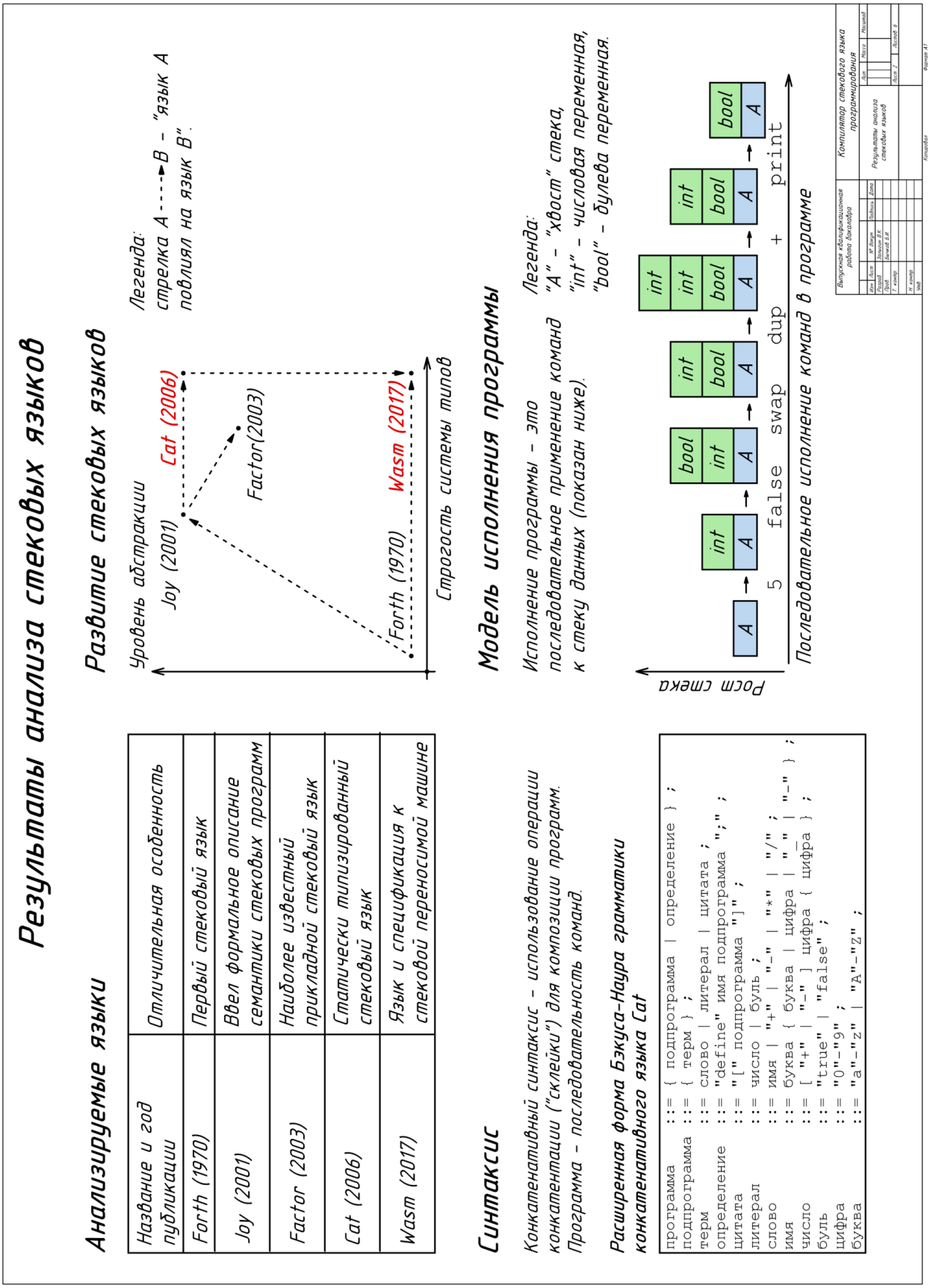
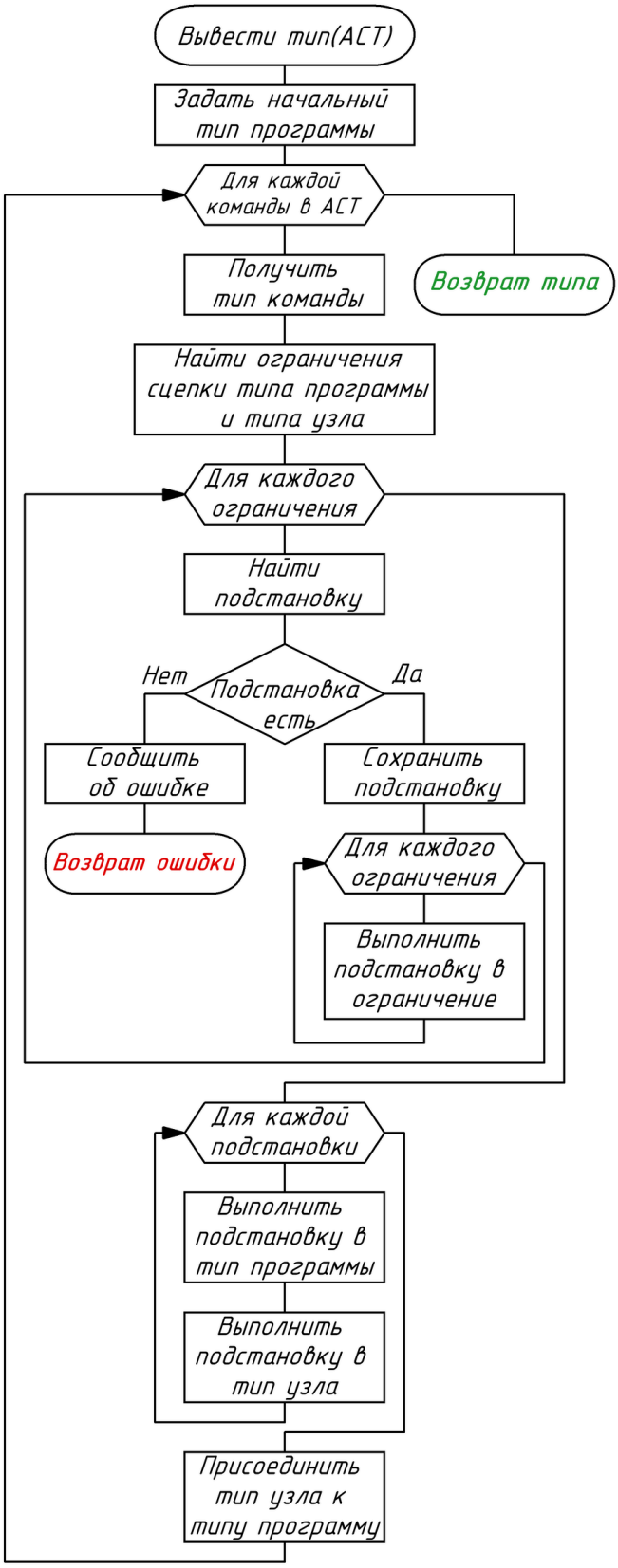


Рисунок В.2 — Результаты анализа стековых языков

Схема алгоритма вывода типов



Вывод типов – последовательное применение ко всей программе алгоритма Хиндли-Минлера.

Для каждой команды в программе:

1. составить систему линейных уровнений из ограничений на типы;

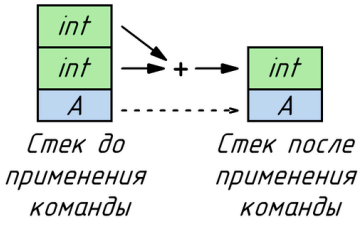
2. найти наиболее общего решение системы.

Если решения нет, возникает ошибка вывода типов.

Тип программы – пара {F, T}, где "F" – конфигурация стека до применения программы, "T" – конфигурация стека после применения программы.

Пример: программа "+" – сложение двух чисел.

Программа потребляет со стека два числа и кладет на стек результат их сложения:



Тип программы "+" записывается как

```
A int int -> A int
```

Выпускная квалификационная работа бакалавра					Компилятор стекового языка программирования			
Изм	Лист	№ докум	Подпись	Дата	Схема алгоритма вывода типов	Лит	Масса	Масштаб
Разраб		Зыкин В.К						
Пров		Бычков Б.И				Лист 3	Листов 6	
Т. контр								
Н. контр								
Утв								

Копировал Формат А1

Рисунок В.3 — Схема алгоритма вывода типов

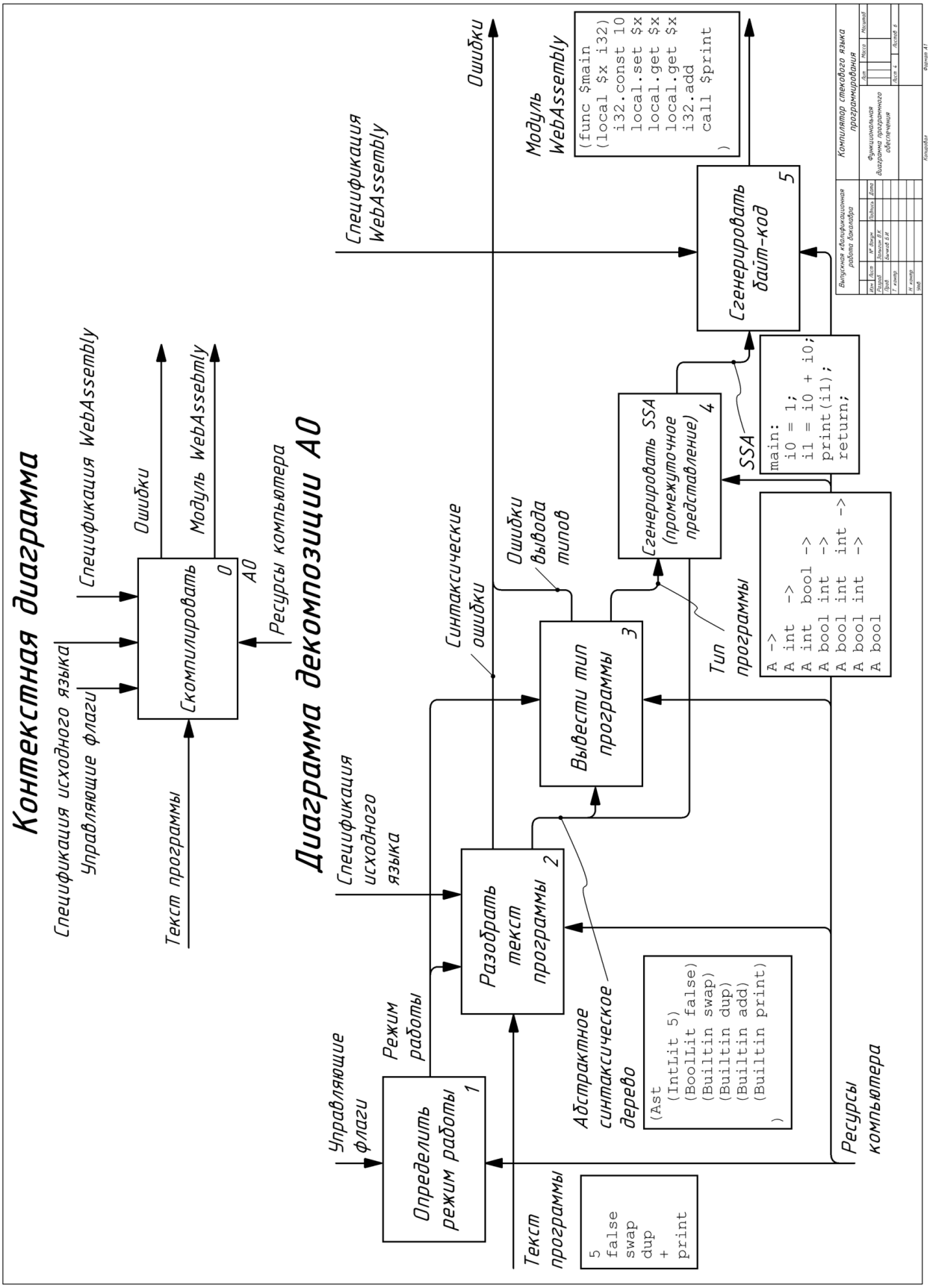


Рисунок В.4 — Функциональная диаграмма программного обеспечения

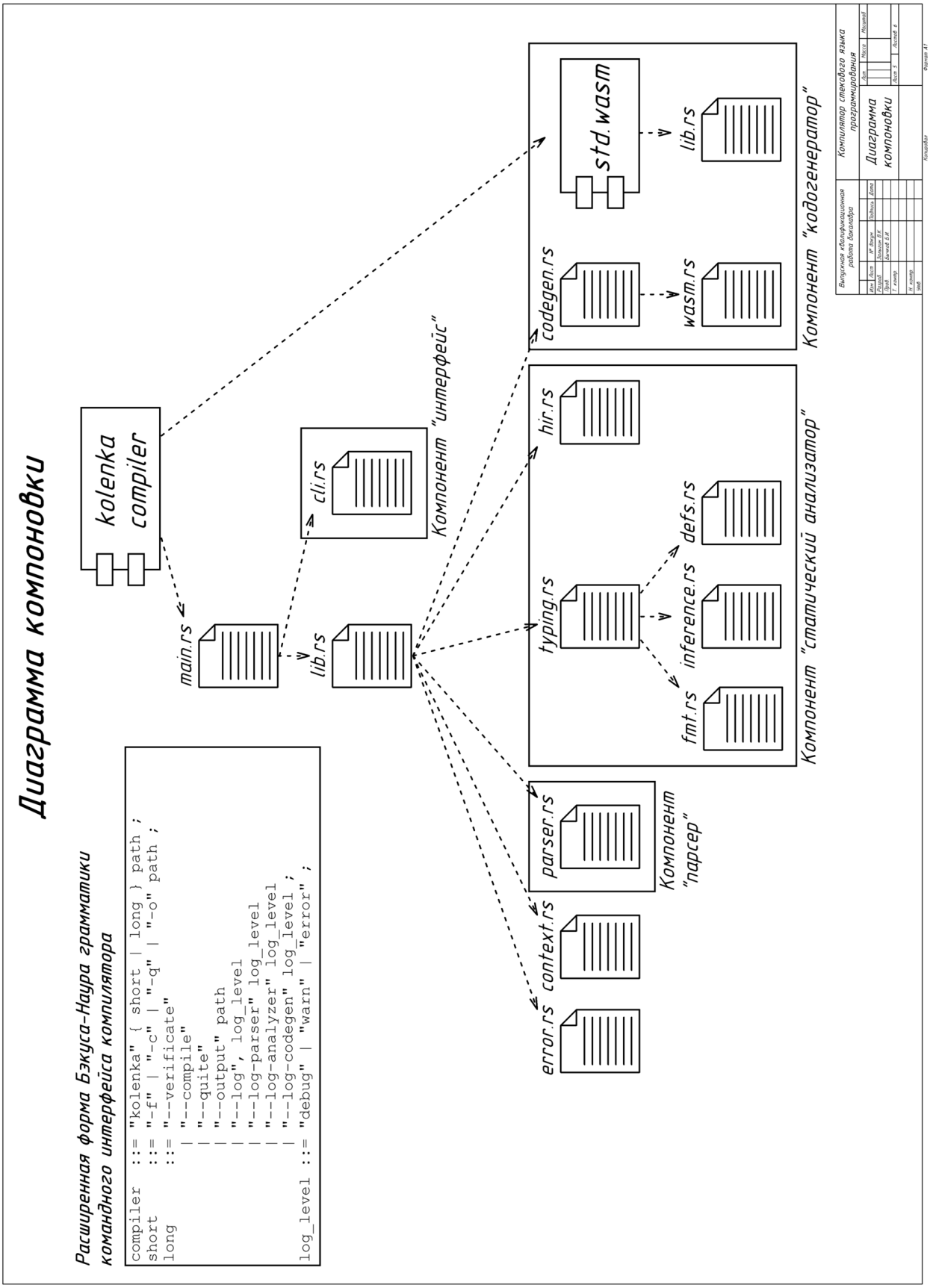


Рисунок В.5 — Диаграмма компоновки

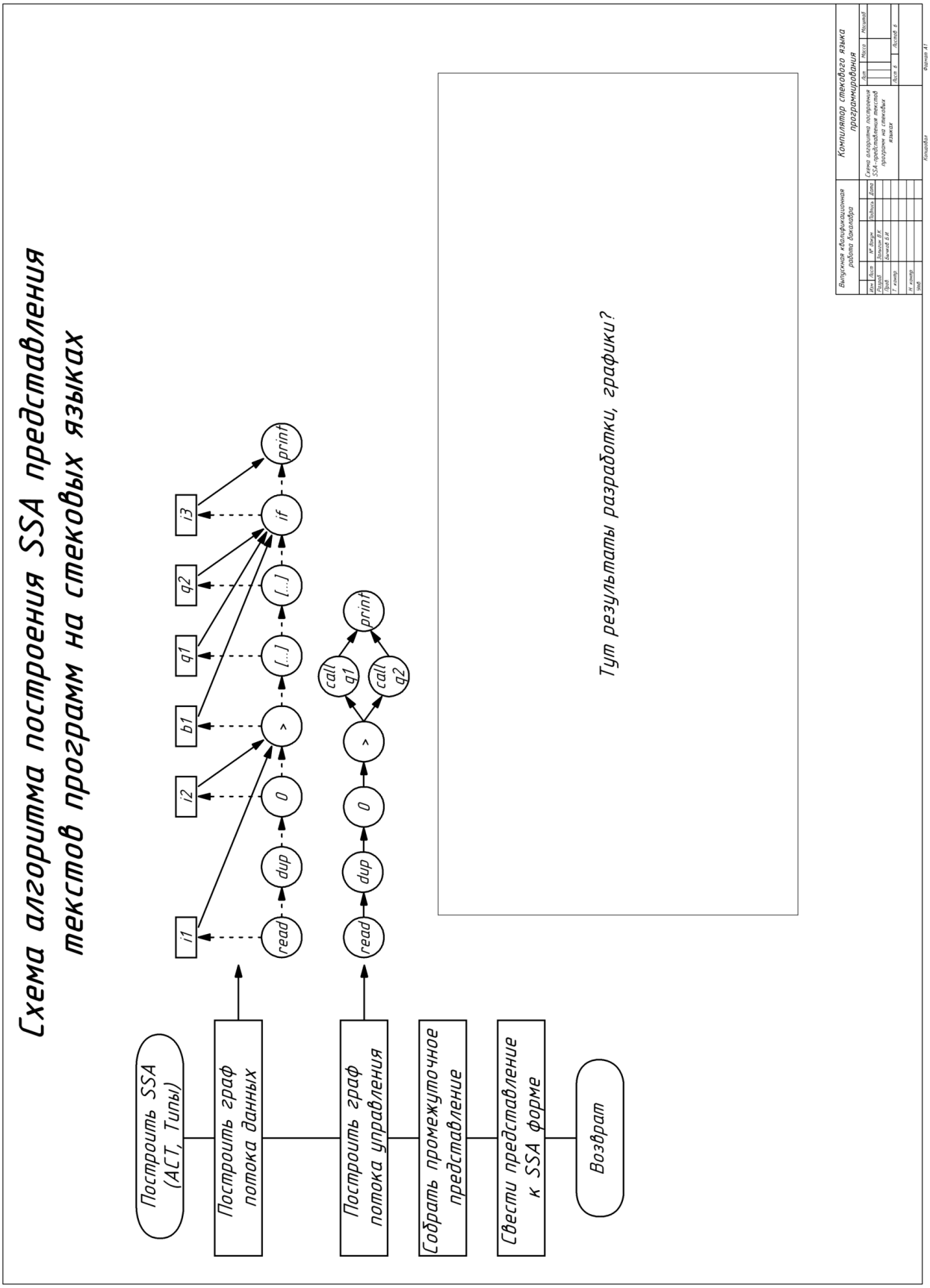


Рисунок В.6 — Схема алгоритма построения SSA-представления текстов программ на стековых языках