Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

Московский государственный технический университет имени Н.Э. Баумана
(МГТУ им. Н.Э. Баумана)

Факультет: «Информатика и системы управления»
Кафедра: «Теоретическая информатика и компьютерные технологии»

Расчетно-пояснительная записка к
выпускной квалификационной работе

Подходы к компиляции функциональных языков программирования с динамической типизацией в переносимый код для виртуальной машины

Научный руководитель: _____ (А. В. Дубанов)
(подпись, дата)

Студент группы ИУ9-41М: _____ (А. В. Беляев)
(подпись, дата)

Москва, 2020

АННОТАЦИЯ

Работа посвящена исследованию способов представления программ с помощью математических моделей. Рассматривается предметная область и разные варианты моделей. Путем сравнения выбирается наиболее подходящий в рамках работы вариант — моделирование программ посредством математической логики и лямбда-исчисления.

По ходу работы после каждого выполненного шага приводится минималистичный пример программной апробации подхода.

Исследованы подходы к компиляции программ и специфика этих подходов в случае компиляции функциональных программ. Для проверки гипотез реализованы части компилятора.

Из полученных в ходе апробации подходов частей собирается компилятор и представляется в качестве практического результата работы. На нем проверяется достижимость целей и задач поставленных в исследовании. Выполнена оценка возможности дальнейшего исследования.

Пояснительная записка к выпускной квалификационной работе содержит 66 страниц текста формата А4, 8 рисунков, 2 таблицы, 20 листингов и список используемой литературы, включающий 16 библиографических источников.

Содержание

ВВЕДЕНИЕ

Исторически сложилось, так, что интернет сравнительно недавно стал неотъемлемой частью жизни. На заре компьютерной эпохи и вплоть до недавнего времени компьютеры имели достаточно ограниченную область применения

При этом развитие технологий происходит не равномерно — где-то благодаря “цифровизации” только появляются первые компьютеры, в то время, как где-то происходит смена очередного поколения устройств и способов доступа в интернет.

Но, так или иначе, любое развитие начиналось с появления программ на настольные компьютеры. Таким образом, исторически сложилось, что объем программного обеспечения под настольные компьютеры в разы превышает объем программ, работающих в сети интернет.

Несмотря на то, что это различие постепенно снижается и в какой-то момент появится перевес в другую сторону, на это понадобится значительное количество времени, т.к. объемы кодовых баз достигают миллионы строк на единицу программного обеспечения.

Использование программ с помощью сети интернет значительно упрощает многие процессы как для разработчика, так и, в большей степени, для пользователя — привычные действия не требуют привязки к конкретному устройству, а лишь требуют наличия браузера и подключения к сети интернет.

При этом “перенос” программ в интернет трудозатратен и зачастую не может быть осуществлен в принципе из-за объемов кодовых баз.Автоматизация же процесса переноса ставится во главу угла, т.к. позволяет выполнить процесс с меньшими затратами. При этом гарантируется целостность целевой программы, а также сохранение ее быстродействия.

В качестве такого автоматизированного процесса может служить компиляция — процесс переноса программ на входном языке в, как правило, более низкоуровневую версию. В терминах данной работы под низкоуровневой программой понимается аналог исходной программы, работающий в сети интернет.

В данной работе будут исследованы аспекты подобного переноса программ, начиная от построения математической модели и заканчивая реализацией компилятора для апробации подходов.

1 ПОСТАНОВКА ЗАДАЧИ

Целью работы является исследование подходов к компиляции функциональных языков программирования, т.к. они в значительной мере отличаются от таковых для «классических» императивных языков.

В ходе работы будут рассмотрены способы построения математических моделей программ и оперирования этими программами. Для апробации гипотез будет реализован компилятор.

Это, в свою очередь, дает возможность говорить о практической цели работы: решение выработанное в данной работе обеспечит выполнение в среде веб (посредством веб-браузера) программ, разработанных для настольных компьютеров.

Были поставлены следующие задачи:

- Исследовать способы представления программ с помощью математических моделей.
- Изучить способы компиляции функциональных языков программирования.
- Выбрать наиболее эффективный подход из рассмотренных.
- Реализовать компилятор для проверки гипотез, выдвинутых в ходе работы.

2 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

В природе происходит множество процессов и явлений. Их описание чрезвычайно сложно, а попытка моделирования этих процессов затрудняется без множества существенно важных деталей. Если эти детали не учитываются или искажены, моделирование становится невозможным в принципе.

Тем не менее, с помощью строгой формализации в соответствии с некоторыми правилам, эти процессы могут быть упрощены без потери важных деталей.

Перевод процесса или явления в формальное описание системы называется математическим моделированием. Дисциплина «Математическое моделирование» занимается построением подобных моделей и важный момент здесь — соблюдение баланса между простотой математической модели и полнотой описываемой ею реальной системы. Так, упрощение модели позволяет описать ее меньшим количеством параметров, что позволяет производить вычисления с меньшими трудозатратами. Но обратная сторона — потеря «связи с реальностью», т.к. с большой вероятностью построенная модель уже не описывает процесс во всей его полноте или описывает совершенно другой процесс. Использование ее в дальнейших вычислениях может привести к неожиданным результатам, что катастрофически скажется на результатах всего исследования.

В свою очередь сложная или переусложненная модель приводит к «удорожанию» вычислений и обесцениванию результатов, внимание может быть приковано к неключевым параметрам притом, что «проверка вручную» в случае такой системы будет невозможна. Трудозатраты — существенный фактор, т.к. При неправильном построении сложность эксперимента может оказаться дороже потенциальных результатов. Оче-

видно, если проведение исследования более затратно, чем выгода от получения результатов исследования, то такое исследование несостоятельно.

2.1 Аксиоматическая семантика

Очевидно, выполнение программы, как и процесс ее компиляции могут быть формализованы математическим описанием. Существенный фактор здесь состоит в том, что рассматривается выполнение программы, написанной в функциональном стиле, т.к. этот «стиль» имеет тесную связь с математикой. Об этом будет сказано чуть позднее в данной работе. Но для начала необходимо дать определение области, в рамках которой будет происходить работа и все дальнейшие эксперименты и проверки гипотез.

Информатика (Computer Science, CS) — широкая область наук, включающая все аспекты обработки информации от теоретического исследования алгоритмов и структур данных до практической реализации программного и аппаратного обеспечения.

В рамках данной работы рассматривается подобласть информатики — **Теоретическая информатика** (Theoretical Computer Science, TCS) — чуть более узкая область, занимающаяся исследованиями, связанными с математическими, логическими и формальными проблемами информатики. Вот лишь некоторые из направлений:

- Исследование алгоритмов и теория вычислимости.
- Теория языков программирования, формальных языков.
- Вычислительная сложность.
- Логика и формальная семантика.

Нас в рамках данной работы интересует последний пункт, включающий логику высказываний, логику предикатов и, в частности, формальную семантику. Эти направления имеют тесную связь с практическими задачами, например, построением компиляторов, что является одной из задач, поставленной в данной работе.

Таким образом, область работы была сформулирована в общем смысле. Тем не менее, поскольку теоретическая информатика достаточно широка, как область исследования, необходимо перейти к направлению в более узком смысле. В данном случае нас интересует подобласть, называемая «семантикой» [7].

Семантика (Semantics) — дисциплина более прикладная к теме данной работы. Она занимается формализацией значений конструкций языков программирования путем построения их математических моделей и включает следующее:

- Определение семантических моделей.
- Отношения между семантическими моделями.
- Отношения между вычислимостью и нижележащими математическими структурами, например, логикой и теорией множеств.

Семантика описывает процессы, выполняемые компьютером во время исполнения программы и помогает понять значения конструкций, указанных, например в Листинге 1.

Листинг 1: Примеры конструкции языка программирования

```
if 1 == 1 then S1 else S2


i=0; while(i < 5)do{ i++; }
```

```
i=0; do{ i++; }while(i <= 4);
```

Так, семантика позволяет «улавливать смысл» выражений и понять, что выполнение конструкции $if..then..else..$ эквивалентно выполнению лишь $S1$, т.к. условие $1 = 1$ всегда истинно. А 2 цикла ниже (первый цикл — $while..do..$ — с предусловием, второй — $do..while..$ — с постусловие) эквивалентны между собой логически, хотя и отличаются семантически. Но процессы, происходящие во время выполнения программы могут быть описаны с разных сторон. Следовательно, и семантика, описывающая их делится на подвиды. Рассмотрим лишь некоторые из них.

Операционная семантика — наиболее распространенный вид семантики. Он используется для связывания синтаксических конструкций языка программирования с некоторой абстрактной машиной (автоматом), на которой выполняются программы на этом языке. Семантика в данном случае определяет набор правил, в соответствии с которыми работает абстрактная машина. Более формально, автомат, находясь в некотором начальном состоянии , задает значение терма (смысл конструкции языка) путем перехода в конечное состояние.

Следующий вид семантики — **денотационная** (denote, обозначение). Она занимается сопоставлением выражений программы с математическими объектами. В качестве значения терма принимается некий математический объект (например, функция или число) и задается функция интерпретации этого объекта, переводящая их в другую область. В случае языка программирования, смысл выражений на нем описывается на другом языке — метаязыке, задающем значения конструкций входного языка. Это находит отражение в конструировании компиляторов, но не входит в основное содержание данной работы.

Интерпретационная семантика описывает конструкции языков программирования высокого уровня в терминах языков низкого уровня, например, ассемблера или машинных кодов. Очевидно, это находит отражение в самых прикладных аспектах программирования, но, тем не менее, так же не является основным направлением исследования.

Аксиоматическая семантика — интересующее нас в данной работе направление — занимается связыванием практических аспектов — синтаксических конструкций языка программирования — с математическим смыслом через набор аксиом, правил вывода. Эта семантика не разделяет смысл конструкций и логические формулы, их описывающие. Наоборот, вывод эти формул и является смыслом конструкций языка. Каноническим примером аксиоматической семантики является логика Хоара (Hoare logic) — о ней будет сказано далее.

Таким образом, по аналогии с доказательством теорем, аксиомы и правила вывода (обозначающие операторы программы), будучи примененными к некоторым входным переменным (удовлетворяющим некоторым ограничениям) порождают выходные переменные, тоже удовлетворяющие соответствующим ограничениям.

Результат выполнения программы в свою очередь является доказательством того, что выходные данные представляют значения функции, вычисленной по входным данным.

2.2 Модели вычислений

Все способы описания поведения программ, которые были рассмотрены выше так или иначе связаны с конкретными реализациями программ так или иначе, привязанных к какому-либо языку или стилю написания кода.

Но дело в том, что если привязываться к этим параметрам, можно получить множество разных моделей одной и той же программы — все они в конечном счете выполняют одну и ту же логику, но то, как программы реализованы влияет на то, как они исполняются. Этого необходимо избежать и одним из способов абстрагироваться от этого является переход к конкретной модели вычислений, как к обобщенной реализации программы.

Модель вычисления описывает зависимость выходных значений математической функции от входных. При этом, работа и эффективность любого алгоритма в рамках модели вычислений может быть оценена независимо от реализации и без привязки к конкретному языку и технологии.

2.2.1 Конечный автомат

Конечные автоматы (Finite state machine, FSM) широко применяются при моделировании систем и вычислений. Они представляют из себя математическую абстракцию устройства, имеющего один вход и выход и в любой момент времени находящегося в одном состоянии из всего множества возможных. Автомат описывается в терминах

- входа.
- выхода.
- множества операций по переводу входных данных в выходные.

Широко известными примерами автоматов являются **Машина Тьюринга** — «классическая» модель вычисления, «перебирающая» ленту из входных символов согласно заданным правилам и **Виртуальная машина** — программная реализация абстрактного автомата посредством

интерпретации.

Абстрактные автоматы подходят для моделирования систем, работающих *последовательно* и имеющих сравнительно небольшое количество дискретных состяний. Каждое из этих состяний имеет отношение к детали моделируемой задачи.

Более формально, автомат имеет

- конечное число дискретных состояний $S_1, \ldots, S_k$, одно из которых является начальным состоянием.
- конечное число m входных символов $I_1, \ldots, I_m$.
- конечное число n выходных символов $O_1, \ldots, O_n$.
- набор правил перехода, определяющих следующее состояние $\bar{s}\ (s, I_1, \ldots, I_m)$ для каждого состояния s и набора входных данных $(I_1, \ldots, I_m)$.
- набор правил вывода, определяющих выходные значения $O_1(s, I_1, \ldots, I_m), \ldots, O_n(s, I_1, \ldots, I_m)$ для каждого состояния s и входов $I_1, \ldots, I_m$.

Автомат начинает работу с начального состояния S_1 и выполняет переход в новое состояние $\bar{s}$ в зависимости от текущего состояния s и текущего входа. Выходные значения продиктованы текущим состоянием.

Распространенным представлением автомата является граф, узлы которого представляют состояния, а ребра — переходы между состояниями.

Несмотря на то, что автомат по определению может выдавать выходные значения, основываясь одновременно и на входных значениях, и на текущем состоянии, выходные значения обычно отражают только текущее состояние, но не входные значения. Такие автоматы получили

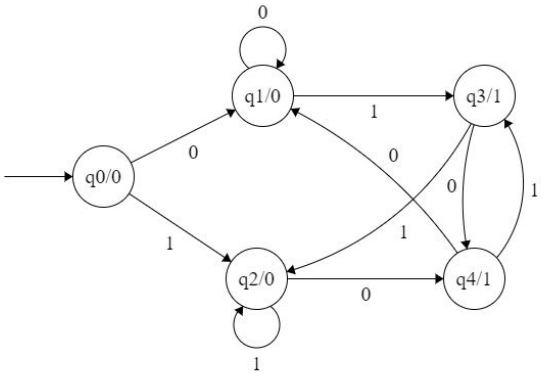

Рис. 1: Автомат Мура

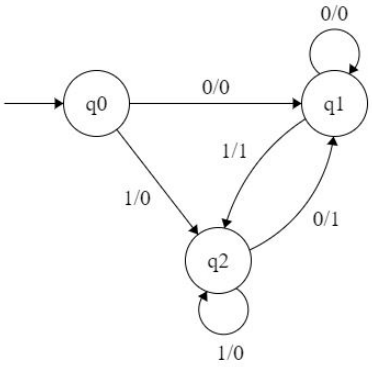

Рис. 2: Автомат Мили

название **Автоматы Мура** (Moore machine). Диаграмма таких автоматов обычно содержит одновременно название состояния и выходное значение в нем [6].

В противоположность в **автоматах Мили** (Mealy machine) выходное значение одновременно зависит от состояния и входа. На диаграммах это отражается в маркировке ребер. На Рисунках 1 и 2 показаны представления обоих типов автоматов.

Возможность концептуально задать автомат любого размера и, соответственно, провести вычисления практически любых масштабов делает конечные автоматы неплохими кандидатами в качестве модели вычислений. Автоматы хорошо изучены и широко используются.

Но, говоря о вычислениях в общем смысле, встает вопрос, подходят ли автоматы для моделирования *всех возможных* вычислений. Если любые возможные вычисления могут быть выполнены некоторым конечным автоматом, значит ли это, что необходимо из перечислимого набора автоматов лишь выбрать подходящий? Такой подход, очевидно, игнорирует практические аспекты, такие, как затраты и производительность.

Чтобы ответить на этот вопрос, необходимо изучить недостатки ко-

нечных автоматов. Для этого представим следующую задачу: разработать автомат, проверяющий сбалансированность скобочной последовательности. Так, последовательность «(()())» является сбалансированной, а «()())» — не является. Для простоты можно закодировать открывающуюся и закрывающуюся скобки с помощью «1» и «0».

Сложность этой задачи состоит в ограниченности хранилища автомата. Если скобочная последовательность состоит из k элементов, потребуется не менее $k + 1$ разных состояний для моделирования каждого шага процесса.

С практической точки зрения, при наличии ограничений, автоматы подходят для моделирования вычислений. Любой автомат несет хранилище строго определенных размеров. Балансировка скобочной последовательности это простой пример — любая строка конечного размера может быть проверена некоторым автоматом, но для любого такого ограничения на размер строки, мы можем подобрать строку, которая, будучи все еще ограниченной, не укладывается в хранилище автомата.

Таким образом, необходимо изучить альтернативные модели вычислений.

2.2.2 Сети Петри

Для моделирования параллельных вычислений некоторое время применялись сети Петри (Petri nets) — граф, состоящий из вершин двух типов — *позиций* и *переходов*, соединенных ребрами. При этом вершины одного типа не могут быть соединены непосредственно. Позиции в данном случае схожи с состояниями автомата, но это не совсем так. В данном случае в позициях могут быть расположены *метки* (или токены), способные перемещаться по сети. Состоянием же всей сети Петри

называется распределение метод по позициям в отдельный момент времени.

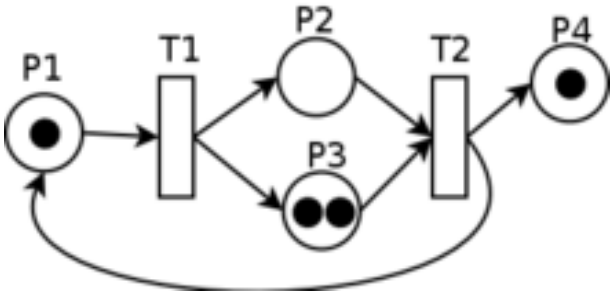

Рис. 3: Сеть Петри

На Рисунке 3 представлен пример сети Петри, тройки вида $N = (P, T, F)$. Здесь $P = (P_1, \ldots, P_4)$ — *позиции*, полоски $T = (T_1, T_2)$ — *переходы*, а черные точки (F) — *метки*.

События в сети происходят либо мгновенно, либо размеренно. То, что события могут произойти одновременно, представляет сложность в восприятии и использовании: на одном атомарном шаге системы токены (метки) из одной позиции могут полностью исчезнуть и появиться в других позициях.

Так же, по аналогии с конечными автоматами, сети Петри подходят для моделирования потока управления (control flow), но не для представления данных в конкретный момент времени [8]

Тем не менее, параллельные вычисления получают все большее распространение, т.к. по многим факторам являются более эффективными, нежели последовательные. И для их моделирования была разработана акторная модель, как своего рода потомок сетей Петри.

2.2.3 Акторная модель

Акторная модель применяется для моделирования параллельных вычислений и строится вокруг понятия «актор» (actor — актер, англ.) —

примитива параллельного исполнения.

Концептуально она определяет набор правил, согласно которому взаимодействуют компоненты системы — **акторы**. Акторы принимают *сообщения* и, основываясь на них, выполняют некоторые действия. Идея акторов имеет много общего с принципами Объектно-ориентированного программирования (ООП) с тем лишь различием (ограничением), что акторы являются независимыми друг от друга, не разделяют общего состояния, а их внутреннее состояние не может быть изменено действиями другого актора.

Важным замечанием является то, что несмотря на то, что несколько акторов одновременно могут работать параллельно, каждый актор вам по себе работает последовательно. Таким образом, если актору будут переданы, например, 3 сообщения, 3 ответных действия будут выполнены последовательно одно за другим. Чтобы выполнить эти действия параллельно, необходимо отправить сообщения трем разным акторам. Пример небольшой акторной системы представлен на Рисунке 4.

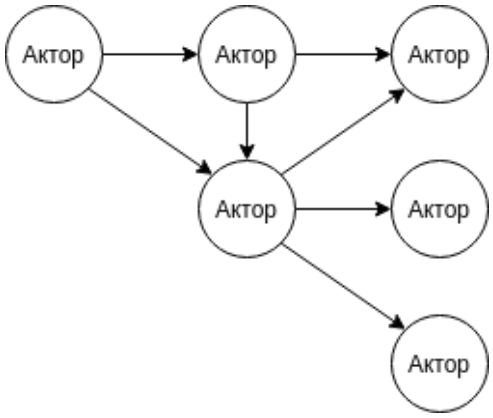

Рис. 4: Акторная система

Акторная модель находит широкое применение в прикладной разработке, а двумя самым известными ее реализациями являются язык Erlang и Akka — библиотека для языков JVM (Java, Scala и другие).

Erlang, где изначально появились акторные системы, превносит философию "позвольте программе сломаться"(let it crash). Это проявляется в том, что разработчику нет необходимости предугадывать все возможные поломки программы и действия в ответ на них. Erlang позволяет отдельным компонентам программы (акторам) «ломаться», но делает это таким образом, чтобы поломка была видима для вышестоящего актора и он может отреагировать на нее соответствующим образом (например, перезапуском актора или отправкой сообщения другому актору). Таким образом, поломки становятся обычным порядком вещей и не являются чем-то критичным для всей системы.

Каждый актор работает полностью независимо от остальных и его поведение или поломка полностью изолированы от всей остальной системы. При этом актор может даже не находиться на одной физической машине с остальной системой.

Все это позволяет строить масштабные и масштабируемые самовосстанавливающиеся (self-healing) системы. Платой за это является значительное концептуальное усложнение системы, сложность восприятия, сложность разработки и отладки.

2.2.4 Логика Хоара

Логика Хоара является формальной системой логических правил для доказательства корректности программ. Логика Хоара основана на т.н. «тройках Хоара» следующего вида

$$\{P\}C\{Q\}$$

где P является утверждением предусловия, C — командой, Q — утверждением постусловия. Т.е., если предусловие выполнено, то выполнение

команды сделает постусловие выполнимым [9].

Логика Хоара включает аксиомы и правила вывода для большинства конструкций среднего императивного языка программирования., например:

$$\frac{P' \to P, \{P\}S\{Q\}, Q \to Q'}{\{P'\}S\{Q'\}}$$

Условная конструкция:

$$\frac{\{B \wedge P\}S\{Q\}, \{\neg B \wedge P\}T\{Q\}}{\{P\}if\ B\ then\ S\ else\ T\ endif\{Q\}}$$

Цикл `while`:

$$\frac{\{B \wedge P\}S\{P\}}{\{P\}while\ B\ do\ S\ done\{\neg B \wedge P\}}$$

Правило композиции:

$$\frac{\{P\}S\{Q\}, \{Q\}T\{R\}}{\{P\}S\ ;\ T\{R\}}$$

Тем не менее, тройки Хоара подходят для представления императивных конструкций, и они не рассчитаны на применение в доказательстве функциональных программ, поэтому они приведены лишь для ознакомления.

2.2.5 Лямбда-исчисление

Лямбда-исчисление (lambda calculus, λ calculus) это формальная система записи выражений в математической логике. Она была введена в обращение математиком Алонцо Черчем (Alonzo Church) в 1930х годах и до сих пор является одновременно простой, при этом универсальной

Таблица 1: Нумерлы Черча

Число	Лямбда-выражение
0	$\lambda f.\ \lambda x.\ x$
1	$\lambda f.\ \lambda x.\ f\ x$
2	$\lambda f.\ \lambda x.\ f(f\ x)$
3	$\lambda f.\ \lambda x.\ f(f(f\ x))$
n	$\lambda f.\ \lambda x.\ f^n\ x$

моделью вычислений и формирует теоретическую модель для многих функциональных языков программирования [10].

В основе т.н. **«чистого» лямбда-исчисления** лежит простая нотация задания функций и их применения, а основные идеи — применение функции к аргументу и задание новой функции с помощью абстракции.

При этом такого понятия, как переменные и константы — нет и для того, чтобы представлять числа используется т.н. кодирование Черча (Church encoding). Представленные с помощью кодирования Черча целые числа называются **нумералами Черча** — функциями высшего порядка, количество применений которых к аргументу обозначает натуральное число. В Таблице 1 приведены примеры термов такого кодирования.

Поскольку число представляется количеством применений функции, то операции над числами схожи с операциями над степенями:

$$m+1 = f^{n+1}x = f(f^n x) = succ\ n\ f\ x = f(n\ f\ x) = \lambda n.\ \lambda f.\ \lambda x.\ f(n\ f\ x)$$

$$m+n = f^{m+n}x = f^m(f^n\ x) = plus\ m\ n\ f\ x = \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ f(n\ f\ x)$$

Логический же тип данных представляется т.н. Church booleans — функциями с двумя аргументами, где истина (True) обозначает выбор

первого аргумента, а ложь (False) — выбор второго аргумента:

$$true = \lambda a.\ \lambda b.\ a$$

$$false = \lambda a.\ \lambda b.\ b$$

Тезис Тьюринга-Черча утверждает, что любой вычислимый оператор может быть представлен в кодировке Черча. Благодаря этому, в дополнение к функциям, единственному «типу данных», мы способны выразить любую другую сущность через применение функций.

Лямбда-исчисление, расширенное введением описанных выше сущностей («встроенных» функций, переменных и констант) называется **прикладным лямбда-исчислением** (applied lambda calculus). Все описанное в дальнейшем будет рассаматриваться в рамках именно такого исчисления. Грамматика прикладного λ-исчисления описывается правилом со следующими тремя альтернативными цепочками символов:

$$\begin{aligned} \langle expr \rangle \quad ::= \ & \text{lambda}\ \langle variable \rangle\ .\ \langle expr \rangle \\ | \ & \langle expr \rangle\ \langle expr \rangle \\ | \ & \langle variable \rangle \end{aligned}$$

где `<expr>` — выражение, а `<variable>` — переменная.

Для начала можно рассмотреть простейший пример λ-терма:

$$\lambda x[x^2 - 2x + 5]$$

Оператор λ позволяет нам задать функцию, абстрагировавшись от значения x. Таким образом, мы можем задать выражение $x^2 - 2x + 5$, ожидающее значения x. Если мы, например, зададим значение $x = 2$, то осуществится *применение* функции к аргументу и будет получен ре-

зультат

$$(\lambda x[x^2 - 2x + 5])2 \to 2^2 - 2 * 2 + 5 = 4 - 4 + 5 = 5$$

Более формально, концепции лямбда-исчисления следующие.

В лямбда-исчислении присутствуют **переменные** (например, x), обозначающие значения.

Абстракция: $\lambda x.M$ — выражение, задающее функцию, где x — аргумент, M — тело функции. Важно заметить, что у этой функции нет названия, т.е. это «анонимная» функция. Пример выше:

$$\lambda x.x^2 - 2x + 5$$

Аппликация: $f\ a$ — применение функции к аргументу. Здесь f — функция, a — аргумент.

$$(\lambda x.x^2 - 2x + 5)2$$

β-редукция (так же известная, как β-конверсия) — правило, по которому производится подстановка, записывается следующим образом:

$$(\lambda x[M])N \to M[x := N]$$

И представляет из себя замену связанных вхождений x внутри тела функции M на аргумент N. Такой процесс так же называется нормализацией. Существуют и ненормализуемые термы, например $(\lambda x.\ xx)(\lambda x.\ xx)$ нормализуется в себя же.

Важным моментом является то, что аргумент всегда строго один. Но, чтобы задать функцию от, нескольких аргументов, необходимо восполь-

зоваться техникой *каррирования* (currying). Применим ее для следующей функции:

$$(x, y, z) \to x^2 + y^2 + z^2$$

Эта функция от трех аргументов принимает следующий вид в нотации лямбда-исчисления:

$$\lambda x.\lambda y.\lambda z.x^2 + y^2 + z^2$$

Последняя важная концепция, которую стоит рассмотреть, связана с математической логикой. Она заключается в наличии свободных и связанных переменных.

В математической логике вхождение переменной в формулу называется **связанным**, если оно находится в области действия квантора, использующего эту переменную. Иначе, вхождение переменной называется **свободным**.

В случае лямбда-исчисления абстракция предоставляет нам возможность связывать переменные в функции (формуле). Так, например в функцию

$$\lambda x.x^2 + y + 5$$

переменная y входит свободно, а переменная x — связанно. Важно заметить, что до тех пор, пока мы не связали y с конкретным значением, вычислить функцию мы не сможем.

Мы рассмотрели лишь несколько концепций лямбда-исчисления, но необходимо так же рассмотреть и его практическую составляющую в рамках данной работы. Для этого рассмотрим использование лямбда-исчисления в программировании.

Пусть существует некоторая нотация, которая позволяет нам записать следующие действия: конструирование списка (пары, если точнее), получение первого элемента получившейся конструкции и получение последней части конструкции (всех элементов, кроме первого, если быть точнее). Запишем эти команды на Листинге 2

Листинг 2: Действия над парой элементов

```
(cons p q)
> (p q)
(head (cons p q))
> p
(tail (cons p q))
> q
```

Так, с помощью операции *cons* мы задали упорядоченную пару (p, q) из элементов p и q. С помощью операции *head* получили первый элемент пары (т.н. «голову» списка), в данном случае — элемент p. С помощью *tail* получили последний элемент пары (т.н. «хвост» списка), в данном случае — элемент q.

Интересен тот факт, что эти операции, представляющие из себя простейшую программу, могут быть с легкостью перенесены на лямбда-нотацию в виде лямбда-абстракций:

$$CONS = \lambda a.\ \lambda b.\ \lambda f.\ f\ a\ b$$

$$HEAD = \lambda c.\ c(\lambda a.\ \lambda b.\ a)$$

$$TAIL = \lambda c.\ c(\lambda a.\ \lambda b.\ b)$$

По аналогии с записью `(head (cons a b))` выше, мы можем выпол-

нить лямбда-аппликацию, т.е. для некоторой пары (p, q) вычислить ее первый элемент следующим образом:

$$HEAD(CONS\ p\ q)$$

$$(\lambda c.\ c(\lambda a.\ \lambda b.\ a))(CONS\ p\ q)$$

$$CONS\ p\ q\ (\lambda a.\ \lambda b.\ a)$$

$$(\lambda a.\ \lambda b.\ \lambda f.\ f\ a\ b)\ p\ q(\lambda a.\ \lambda b.\ a)$$

$$(\lambda b.\ \lambda f.\ f\ p\ b)\ q(\lambda a.\ \lambda b.\ a)$$

$$(\lambda f.\ f\ p\ q)(\lambda a.\ \lambda b.\ a)$$

$$(\lambda a.\ \lambda b.\ a)\ p\ q$$

$$(\lambda b.\ p)\ q$$

$$p$$

То есть, путем применения лямбда-аппликации и бета-редукции мы по сути вычислили значение выражения, представляющее из себя простейшую программу на Листинге 2. Вычисление «хвоста» пары — `(tail (cons a b))` — очевидно и приводить его здесь не имеет смысла.

Важное наблюдение: для представления такой программы нам не понадобилось сохранять промежуточные вычисления, т.к. каждая «итерация» была самодостаточной и полностью описывала шаг процесса. Т.е., нам не понадобились никакие переменные, в которых мы бы сохраняли значения и которые мы бы модифицировали.

При рассмотрении предыдущих моделей вычисления (с помощью автоматов и машины Тьюринга, с помощью сетей Петри и акторов) мы видели с какой простотой и выразительностью можно моделировать вы-

числения, но большой проблемой было хранение промежуточных вычислений (т.н. «состояния программы» (program state) в программировании) в каждый момент вычисления.

Пример с лямбда-исчислением достаточно примитивен и при этом относительно «многословен», но он показывает «мощность» и выразительность лямбда-исчисления для моделирования программ.

Важнейшим же аспектом является **отсутствие состояния программы**. По сути мы «избавились от данных». В этом заключается преимущество этой модели — мы по-прежнему можем моделировать сложные вычисления (программы) без необходимости заботится о хранении промежуточных вычислений

Таким образом мы можем завершить поиск способа представления программы с помощью математической модели и остановиться на лямбда-исчислении, как на итоговом варианте.

2.3 Функциональное программирование

Функциональное программирование является парадигмой, в которой программы являются композициями функций. Такой подход существенно отличается от классического императивного программирования, где функции являются скорее сгруппированным набором последовательных инструкций.

Выражения в ФП возвращают значения, нежели изменяют состояние программы

$$y = f(g(h(x, z))) \leftrightarrow result = write(process(read(a, b)))$$

Концепции ФП базируются вокруг функциональной чистоты (functional purity), которая позволяет трактовать функции в математи-

ческом смысле — т.е. детерминированными, чистыми. Результат выполнения таких функций строго зависит от аргументов и не меняет общего состояния программы (отсутствует как таковой program state), не вызывает побочных эффектов, примеры которых были рассмотрены ранее.

Программы, написанные в функциональной парадигме как правило содержат меньше ошибок, проще в тестировании, т.к. всецело зависят от входных данных и могут быть формально верифицируемы.

Концепции функционального программирования следующие:

- функции высших порядков — функции могут принимать функции в качестве аргументов и возвращать функции в качестве выходных значений.

- каррирование (currying) — техника позволяющая выразить функцию с арностью аргументов больше 1 путем частичного применения функций с одним аргументом. Эта техника берет свое начало в лямбда-исчислении, и ранее была затронута в работе.

- чистые функции и отсутствие состояния программы — концепция, рассмотренная выше.

- рекурсия — аналог циклов в императивном стиле, тесно связанный с системой типов.

- хвостовая рекурсия и хвостовые вызовы функций — способ оптимизации работы программы,связанный с практическим выполнением функциональных программ процессором с последовательным выполнением инструкций.

- система типов — функции полностью определены своими сигнатурами, а процесс компиляции состоит в проверке типов. Таким об-

разом, компиляция программы тесно связана с ее формальной верфикацией и проверкой корректности. Алгебраические типы данных расширяют систему типов комплексными типами-произведениями (product type), например, парами и кортежами, которые были рассмотрены ранее (`cons`) и типами-суммами (sum type), например, объединениями.

2.4 Изоморфизм Карри-Говарда

Логика высказываний — раздел математики, занимающийся анализом структуры высказываний, так же называющийся пропозициональной (лат. propositio — предложение) логикой. Логика высказываний формализует представления о высказываниях и логических операциях над ними.

Классическое исчисление высказываний задается некоторым набором аксиом. Среди известных правил вывода можно указать т.н. modus ponens (MP): $\dfrac{A \quad A \to B}{B}$. Она обозначает истинность формулы в выводе при истинности двух формул в посылке.

При этом в логике высказываний существуют правила естественного вывода — они позволяют из уже известных выводимостей получать новые. Среди таких формул необходимо выделить следующие:

$$\frac{A \quad B}{A \wedge B}$$

$$\frac{A \wedge B}{A} \qquad \frac{A \wedge B}{B}$$

$$\frac{\begin{array}{c} A \\ B \end{array}}{A \implies B} \qquad \frac{A \implies B \quad A}{B}$$

Программы на функциональном языке могут рассматриваться точно так же, как математические функции. Изоморфизм Карри-Говарда

Таблица 2: Эквивалентные структурные элементы

Логическая система	**Язык программирования**
Высказывание	Тип
Доказательство высказывания P	Выражение типа P
Импликация $P \implies Q$	Функциональный тип $P \to Q$
Доказательство импликации $P \implies Q$	Функция типа $P \to Q$
Доказательство посылки P импликации $P \implies Q$	Аргумент функции $P \to Q$
Доказательство следствия Q импликации $P \implies Q$	Результат функции $P \to Q$
Конъюнкция $P \wedge Q$	Пара (tuple) $P \times Q$
Дизъюнкция $P \vee Q$	Тип суммы (union) $P + Q$
Аксиома	Примитивный тип

это мощный инструмент, на который можно опираться при разработке программ на типизированных функциональных языках.

Структурные элементы, приведенные в Таблице 2, благодаря соответствию, считаются эквивалентными.

Важно заметить, что соответствие не ограничивается приведенными в Таблице 2 структурами, как не ограничивается и логикой высказываний: логика высказываний второго порядка соответствует полиморфному лямбда-исчислению, а исчисление предикатов — лямбда-исчислению с зависимыми типами.

3 ПОСТРОЕНИЕ МАТЕМАТИЧЕСКОЙ МОДЕЛИ

3.1 Интерпретация лямбда-исчисления

Как было отмечено ранее, лямбда-исчисление является теоретической основной многих функциональных языков программирования, но необходимо разобраться для чего необходима эта теоретическая база.

Успех разработки программного обеспечения заключается в корректности последнего. Эта корректность может быть достигнута путем тестирования. При этом сложные программы представляют из себя «черный ящик» и их тестирование может быть сведено с проверке работы против некоторого ограниченного набора входных данных. Сама же программа внутри может быть слишком сложной для автоматизированного тестирования или написанной непригодным для такого тестирования способом. Значит, ошибки в ней не могут быть выявлены программным путем.

Тестирование императивных программ (написанных, например, на Java, C++ или Python), как правило сопровождается просмотром исходного кода программ — логика в них может быть сложной для восприятия, запутанной, с нарушенными абстракциями. При этом у императивной программы есть состояние (program state) и бывает невозможно вызвать процедуру дважды подряд с одними и теми же аргументами и ожидать одинаковый результат, т.к. процедура одновременно зависит от состояния и меняет его. Таким образом, тестирование сводится к проверке в стиле «*вызовет ли исключение этот набор входных данных? вызовет ли этот?*». Это достаточно низкоуровневое тестирование, которое скорее говорит о правильности реализации, но не доказывает наличие или отсутствие важных свойств программы. Эдсгер Дейкстра (Edsger

Dijkstra) высказывался следующим образом: «*Тестирование программы показывает наличие ошибок, а не их отсутствие*».

Также это значит, что ее свойства не могут быть выведены и доказаны автоматически. Что при это может сделать разработчик для доказательства корректности? Наиболее эффективным решением является использование **типизированного функционального программирования** [12].

Система типов языка программирования позволяет создавать корректное программное обеспечение, которое удовлетворяет поставленным требованиям. Типы как правило имеются у выражений, переменных, функций. При этом т.н. **соответствие Карри-Говарда** позволяет делать выводы о корректности программы по аналогии с математическими доказательствами. Типы в данном случае — высказывания, а переменных этих типов — доказательства этих высказываний.

Таким образом, если программа компилируется, т.е. проходит проверку типов, то это доказывает, что у программы имеются свойства, присущие этим типам. Тогда, для заданных входных данных программа вычисляет выходных данные. И эти выходные данные являются доказательством высказывания соответствующего типа. Благодаря этому, функциональными языками программирования можно оперировать основываясь на типах функций, нежели на конкретной их реализации.

Простой пример в данном случае — функция, принимающая полиморфный аргумент типа T и возвращающая значение того же типа T:

$$y \ : T = f(x \ : T)$$

В случае, если на T не накладывается никаких ограничений, единственно возможной реализацией является тождественное отображение (identity

function):

$$id(x) = x$$

Т.к. о T — типе x — ничего не известно заранее, то и никаких выводов о корректности функции (кроме отображения id) сделать нельзя.

Но, если T является конкретным типом, то он накладывает ограничения на действия, которые можно совершить над x. В таком случае все еще можно написать некорректную реализацию функции f, но она не пройдет проверку системой типов во время компиляции. Это существенно ограничивает количество некорректных реализаций функции.

В качестве примера можно рассмотреть язык Scala. Он позволяет писать программы в функциональной парадигме, но он пытается одновременно быть функциональным и императивным. Это способствует простоте его восприятия и удобстве работы с ним в рамках императивной парадигмы. Но функциональная его парадигма принесена в жертву.

Пусть есть функция, принимающая целочисленный аргумент и возвращающая целое число. Эта функция может иметь множество сторонних эффектов (side effects), таких как изменение глобальных переменных, печать в поток вывода, выбрасывание исключений. Компилятор в данном случае лишь проверяет типы входных и выходных данных и делает заключение о свойствах функции, основываясь на них. Но не гарантирует, что других свойств нет.

Это не дает возможность делать никаких выводов о поведении, основываясь на сигнатуре функции — всегда необходимо смотреть на ее реализацию и на реализацию тех функций, которые вызываются внутри. Как результат, абстракции не работают, а без абстракций сложно строить по-настоящему сложные вещи.

Это прямо противоположно тому, как происходят доказательства в

математике: если бы математик не мог опираться на меньшие теоремы (леммы) при доказательстве больших, и должен был бы каждый раз смотреть на доказательства этих лемм, то не было бы никакого смысла иметь леммы и прочие отдельные теоремы. Как результат, математика не была бы столь разнообразной и комплексной.

3.2 Scheme

Первым функциональным языком программирования (и вторым языком программирования в общем смысле, после FORTRAN) считается **Lisp** (LISt Processing) — язык, разработанный Джоном МакКарти (John McCarthy) в MIT в 1958 году. Задумывалась разработка системы программирования для символьных вычислений, а именно символьного дифференцирования.

Выражения на языке Lisp строятся с помощью т.н. S-выражений (S-expressions) — структур, обозначающих и операции и данные. В первоначальной реализации язык основывался не на лямбда-исчислении, а на теории Клини о «рекурсивных функциях первого порядка», несмотря на использование ключевого слова `LAMBDA`.

Язык Lisp остался в истории и стал основой для множества диалектов, многие из которых используются до сих пор, самыми известными из которых являются Scheme и Clojure.

Scheme является минималистичным диалектом языка Lisp, разработанный так же в MIT. Авторы языка — Гай Стил (Guy Steele) и Джеральд Сассман (Gerald Jay Sussman) — впервые представили его в своей работе «Scheme: An Interpreter for Extended Lambda Calculus» [3].

Семантика языка, в отличие от первой версии Lisp, сразу была осно-

вана на лямбда-исчислении, см. Листинг 3

$$(\lambda f.\ (\lambda x.\ f\ x\ a))$$

Листинг 3: Лямбда-абстракция на языке Scheme

```
(lambda (f)
    (lambda (x)
        (f x a)))
```

Благодаря этому Scheme можно даже считать имплементацией лямбда-исчисления, а большинство выкладок и формул, рассмотренных ранее могут быть с точностью до синтаксиса записаны и вычислены на языке Scheme.

Простота, минималистичность и сравнительное небольшой набор примитивов в ядре языка обеспечили ему будущее не только в академических кругах. Так, он встречается в качестве языка сценариев и расширений в графических приложениях, таких, как AutoCAD и GIMP.

У языка есть множество реализаций под разные нужды, например, Chicken Scheme, Guile, Chez Scheme и Racket.

В данной работе будет использоваться Scheme версии R6RS в реализации **Racket**, т.к. Racket является наиболее прикладной имплементацией, при этом поддерживает статически-проверяемые аннотации типов (Typed Racket) и поставляется в т.н. формате «batteries included», т.е. со средой разработки, богатой библиотекой и всем необходимым для работы в одном комплекте.

3.3 Правила вывода

В дальнейшей работе мы будем использовать язык Scheme в качестве основного инструмента, т.к. математическими моделями программ, рассматриваемых в данной работе является абстракции лямбда-исчисления, а язык Scheme, как было отмечено ранее, можно считать фактически программной реализацией лямбда-исчисления.

Рассмотрим следующую программу на диалекте Typed Racket языка Scheme, поддерживающем аннотирование типов:

Листинг 4: Конструирование объекта Point

```
(struct Point ([x :Int]
               [y :Int]))

(make-point x y)
> Point
```

На Листинге 4 представлена структура `struct Point`, описывающая точку с целочисленными координатами x и y. Так же существует функция `make-point`, «собирающая» точку по двум координатам. То есть, эта функция фактически отображает `Int` в `Point`. Уйдя от записи на языке Scheme мы можем представить эту же функцию, как правило вывода:

$$\frac{x \ : Int \quad y \ : Int}{makePoint \ x \ y : Point}$$

Здесь **посылка** — два целых числа, **вывод** — точка в пространстве. Как видно, отображение более, чем однозначное.

Теперь добавим 2 функции для получения координаты x и y этой точки. Пусть функции имеют названия `point-x` и `point-y`. Запишем их следующим образом на Листинге 5.

Листинг 5: Получение координат точки

```
(point-x (Point x y)
> x :Int
(point-y (Point x y)
> y :Int
```

Эти функции, как можно заметить, отображают объект типа Point в целое число. Следовательно, их тоже можно записать в виде правил вывода:

$$\frac{p\ : Point}{pointX\ p\ : Int} \qquad \frac{p\ : Point}{pointY\ p\ : Int}$$

Далее рассмотрим задание анонимной функции (лямбда-фукнции) таким же образом. На Листинге 6 представлено связывание анонимной функции с переменной f.

Листинг 6: Задание фукнции

```
(define f
    (lambda ([x :Int]) (* x 5)
```

Функция принимает целочисленное значение x и умножает его на число 5, возвращая результат. Таким образом, функция принимает Int и результатом умножения все так же остается Int. Тогда, можно записать ее с помощью следующего правила вывода:

$$\frac{\begin{array}{c} x\ : Int \\ x \times 5\ : Int \end{array}}{\lambda x.\ x \times 5\ : Int \to Int}$$

В посылке — целочисленный x, и целочисленный результат умножения $x \times 5$. В выводе — функция, отображающая целое число в целое число.

Применив эту фукнцию к числу, например, 8, получим значение 40, как показано на Листинге 7.

Листинг 7: Применение функции

```
(f 8)
> 40
```

Это тоже можно переписать в виде правила вывода, где посылкой будет отображение Int в Int и «левой» частью (т.е. аргументом) будет число 8 типа Int:

$$\frac{f \ : Int \to Int \quad 8 \ : Int}{f \ 8 \ : Int}$$

Теперь попробуем переписать правила вывода в обобщенном виде, заменив конкретные типы переменных в программах выше на абстрактные типы A и B. Point была «собрана» из двух целочисленных переменных, но в общем случае это может быть не так. Имеет смысл говорить о двух разных типах A и B, которые в рассмотренных выше примерах совпадали и являлись типами Int.

Правила вывода конкретной системы типов программ выше (слева) и общем виде (справа):

$$\frac{x \ : Int \quad y \ : Int}{makePoint \ x \ y : Point} \qquad \frac{a \ : A \quad b \ : B}{\langle a, b \rangle \ : A \times B}$$

$$\frac{p \ : Point}{pointX \ p \ : Int} \qquad \frac{p \ : A \times B}{fst \ p \ : A}$$

$$\frac{p \ : Point}{pointY \ p \ : Int} \qquad \frac{p \ : A \times B}{snd \ p \ : B}$$

$$\frac{\begin{array}{c} x\ :Int \\ x \times 5\ :Int \end{array}}{\lambda x.\ x \times 5\ :Int \to Int} \qquad \frac{\begin{array}{c} x\ :A \\ b\ :B \end{array}}{\lambda x.\ b\ :A \to B}$$

$$\frac{f\ :Int \to Int \quad 8\ :Int}{f\ 8\ :Int} \qquad \frac{f\ :A \to B \quad x\ :A}{f\ x\ :B}$$

Как можно заметить, формулы выше (правая колонка) представляют из себя типизированное лямбда-исчисление, с которым мы ранее сталкивались. При этом последние 2 формулы есть не что иное, как λ-абстракция и λ-аппликация. Мы вернемся к этим формулам позднее.

Можно сравнить бок-о-бок полученные правила вывода. Слева — конкретная система типов, в середине — та же система типов, но в общем виде (типизированное лямбда-исчисление), справа — правила естественного вывода в логике высказываний:

$$\frac{x\ :Int \quad y\ :Int}{makePoint\ x\ y : Point} \qquad \frac{a\ :A \quad b\ :B}{\langle a, b\rangle\ :A \times B} \qquad \frac{A \quad B}{A \wedge B}$$

$$\frac{p\ :Point}{pointX\ p\ :Int} \qquad \frac{p\ :A \times B}{fst\ p\ :A} \qquad \frac{A \wedge B}{A}$$

$$\frac{p\ :Point}{pointY\ p\ :Int} \qquad \frac{p\ :A \times B}{snd\ p\ :B} \qquad \frac{A \wedge B}{B}$$

$$\frac{\begin{array}{c} x\ :Int \\ x \times 5\ :Int \end{array}}{\lambda x.\ x \times 5\ :Int \to Int} \qquad \frac{\begin{array}{c} x\ :A \\ b\ :B \end{array}}{\lambda x.\ b\ :A \to B} \qquad \frac{\begin{array}{c} A \\ B \end{array}}{A \implies B}$$

$$\frac{f\ :Int \to Int \quad 8\ :Int}{f\ 8\ :Int} \qquad \frac{f\ :A \to B \quad x\ :A}{f\ x\ :B} \qquad \frac{A \implies B \quad A}{B}$$

С точностью до типов все эти формулы структурно одинаковы.

Получается, что существует связь между языком программирования и логической системой. Эта связь, а точнее изоморфизм между логической системой и типизированным исчислением, называется **соответствием Карри-Говарда** (Curry-Howard correspondence) [11] и была рассмотрена ранее в обхоре предметной области.

Это так же доказывает, что компилятор в процессе работы не просто транслирует программу с одного языка на другой, но и проверяет доказуемость программы (через доказуемость других выражений и функций). Если же программа не скомпилируется, значит присутствует незавершенное доказательство или теорема.

Таким образом, разрабатывая программу, программист автоматически доказывает ее корректность, приводя сначала точные типы выражений (формулируя высказывания), а затем приводя реализации этих типов в виде конкретных переменных (доказывая высказывания).

4 КОНСТРУИРОВАНИЕ КОМПИЛЯТОРА

Термин «*компиляция*» уже затрагивался ранее в работе. Под компиляцией подразумевается процесс трансформации программы, написанной на исходном (высокоуровневом) языке в эквивалентную программу на целевом (как правило, низкоуровневом) языке.

Эта задача обычно выполняется компилятором — программой, которая «понимает» синтаксис и низкоуровневое значение конструкций, а так же знает платформу, на которой будет исполнятся полученная программа.

Компиляция, как правило, выполняется в несколько этапов, т.н. фазы компиляции:

- лексический анализ — сканирование входного потока текста и преобразование текста в набор лексем (токены).

- синтаксический анализ (парсинг) — преобразование набора токенов с прошлого шага в виде синтаксического дерева. Порядок токенов сравнивается с грамматикой языка, и парсер проверяет их на синтаксическую корректность.

- семантический анализ — проверка корректности синтаксического дерева. Например, проверка типов, затронутая ранее в работе, проверка областей видимости переменных и проверка объявления переменных до использования.

- генерация промежуточного кода — преобразование программы на входном языке в некий промежуточный язык, работа с которым более эффективна.

- оптимизация промежуточного кода — промежуточное представление как правило может быть более эффективно оптимизировано и генерация целевого кода из него проще в реализации.

- кодогенерация — оптимизированное промежуточное представление преобразуется в целевой язык.

- оптимизация целевой программы — для более эффективного исполнения возможна последующая обработка сгенерированного кода.

Несмотря на то, что многие целевые архитектуры, для которых компилируются программы, реализованы аппаратно, существуют так же и программные их реализации. Самая известная программная реализация — виртуальная машина Java (Java Virtual Machine, JVM).

Следует также отметить, что компиляция изначально была процессом, выполняемым лишь раз перед запуском программы (ahead-of-time compilation, AOT), однако сейчас множество сред выполнения поддерживают динамическую компиляцию (just-in-time compilation, JIT), которая производится во время выполнения программы. Плюс этого подхода заключается в том, что во время выполнения возможно профилирование программы и становятся доступны знания о том, как именно выполняется программа. После этого некоторые фрагменты кода могут быть дополнительно скомиплированы для повышения эффективности. Этот процесс происходит дольше, но в результате программа может оказаться более производительной во время выполнения, чем ее статически-скомпилированный аналог.

В данной работе рассматривается процесс компиляции функциональных программ, которой отличается от компиляции классических императивных программ.

4.1 Рассахаривание программы

Мы ранее рассмотрели возможность оперирования программой в терминах математической логики. Тогда, по аналогии с эквивалентными преобразованиями логических формул, становится возможным записать эквивалентные преобразования программы на входном языке [2]

Опишем такие эквивалентные преобразования и приведем в примеры функций до и после преобразования. Т.к. компилятор агностичен к программам, которое он транслирует, описывать преобразование какой-то конкретной программы не имеет смысла: чтобы показать как можно больше преобразований, необходимо использовать как можно больше конструкций в одной программе, но это сильно «зашумляет» и увеличивает объем кода. Во-вторых, если показывать на одном примере, то необходимо акцентировать каждый раз внимание на каком-то конкретном преобразовании, в то время, как их может произойти несколько за один шаг.

Поэтому, в качестве примеров будут приведены небольшие функции и конструкции на языке Scheme, достаточно простые, чтобы без лишних объяснений продемонстрировать конкретное преобразование.

Начнем с замены объявления функции на связывание (binding) лямбда-абстракции с переменной. Правило замены выглядит следующим образом:

$$(define\ (f_{name}\ p_{formal1} \ldots p_{formalN})\ e_{body})$$

$$\equiv (define\ f_{name}\ (\lambda\ (p_{formal1} \ldots p_{formalN})\ e_{body}))$$

Пример этой замены в коде программы представлен на Листинге 8:

Листинг 8: Замена объявления функции

```
(define (quad x)
  (* x x)
=>
(define quad
  (lambda (x) (* x x)))
```

Здесь и далее на листингах *до стрелки* `=>` будет представлен исходный код, а *после* — преобразованный.

Преобразование условной конструкции с несколькими ветками и, ниже, с единственной веткой:

$$(cond\ [e_{test1}\ e_{action1}] \ldots [e_{testN}\ e_{actionN}])$$

$$\equiv (if\ e_{test1}\ e_{action1}\ (cond\ [e_{test2}\ e_{action2}] \ldots [e_{testN}\ e_{actionN}]))$$

$$(cond\ (else\ e_{action})) \equiv e_{action}$$

Применение этой конструкции представлено на примере вычисления кусочной функции *signum* на Листинге 9.

Листинг 9: Преобразование условной конструкции

```
(define (sign x)
  (cond
    ((< x 0) -1)
    ((= x 0)  0)
    (else     1)))
=>
(define (sign x)
  (if (< x 0)
```

```
    -1
    (if (= x 0)
        0
        1))))
```

Эквивалентная замена конструкции связывания на анонимную функцию:

$$(let\ ((v_{name1}\ e_{def1}) \ldots (v_{nameN}\ e_{defN}))\ e_{doby})$$

$$\equiv ((\lambda\ (v_{name1} \ldots v_{nameN})\ e_{body})\ e_{def1} \ldots e_{defN})$$

Листинг 10: Эквивалентная замена let

```
(let ((a (list 1 2 3))
      (b (list 4 5 6)))
  (cons a b))
=>
((lambda (a b) (cons a b))
  (list 1 2 3)
  (list 4 5 6))
```

На Листинге 10 изображен пример преобразования, при котором производится замена связывания нескольких переменных на применение их к лямбда-функции в качестве аргументов.

Преобразование `let*` в набор вложенных конструкций `let`:

$$(let*\ ([v_1\ e_1] \ldots [v_N\ e_N])\ e_{body})$$

$$\equiv (let\ ([v_1\ e_1])\ (let\ ([v_2\ e_2]) \ldots (let\ ([v_n\ e_n])\ e_{body}))$$

Листинг 11: Эквивалентная замена let*

```
(let* ((x 1)
       (y (- x 5)))
  (+ x y))
=>
(let ((x 1))
  (let ((y (- x 5)))
    (+ x y)))
```

На Листинге 11 изображена замена конструкции `let*`. Важный момент состоит в том, что у нас уже имеются правила замены конструкций `let`, которые должны быть применены следующим шагом, так что конструкция упрощается и дальше.

Наконец, последнее правило, которое стоит рассмотреть — замена `begin` — последовательности вычислений:

$$(begin\ e_1 \dots e_n) \equiv (let*\ ([t_1\ e_1] \dots [t_{n-1}\ e_{n-1}])\ e_n)$$

В данном случае дополнительно вводятся переменные $t_1 \dots t_n - 1$ для сохранения результатов промежуточных вычислений. Эта конструкция, хотя и входит в спецификацию языка, позволяет писать программы на Scheme в императивном стиле, но она точно таким же образом может быть выражена в виде `let*` — вложенных связываний переменных.

Существуют и другие правила замены, но они более тривиальны и приводить их здесь не имеет смысла.

Как можно заметить, описанный выше подход одновременно увеличивает объем кода программы, но при этом удаляет многие «более сложные» конструкции. Такой процесс называется трансформацией програм-

мы — эквивалентным преобразованием, не меняющим смысл, но упрощающим форму записи программы.

Т.н. «**синтаксический сахар**» — конструкции языка, являющиеся более удобной (компактной) формой записи уже имеющихся конструкций. Синтаксический сахар не вводит дополнительные возможности, но повышает удобство использования языка [13].

Соответственно, рассахаривание (desugaring) — обратный процесс, когда комплексные структуры языка постепенно меняются на более примитивные структуры. В итоге программа становится более объемной (с т.з. кода), но более простой с т.з. «содержания».

Язык Scheme в данном случае примечателен тем, что за его сложными структурами стоит сравнительно небольшой набор простых конструкций. Ядро языка — набора базовых конструкций — крайне мало. Значит, имеет смысл провести рассахаривание входной программы, т.к. производить дальнейшие манипуляции над программой будет проще, ввиду меньшей вариации синтаксиса.

Итогом рассахаривания является лишь небольшое по сравнению с исходным, подмножество языка. Все конструкции теперь могут быть выражены в следующем виде:

$$
\begin{array}{lll}
\langle prog \rangle & ::= & \langle def \rangle^* \ \langle exp \rangle^* \\
\langle def \rangle & ::= & (\text{define } \langle variable \rangle \ \langle exp \rangle) \\
\langle exp \rangle & ::= & \langle variable \rangle \\
 & | & \langle literal \rangle \\
 & | & \langle primitive \rangle \\
 & | & (\text{lambda } (\langle var \rangle^*) \ \langle exp \rangle) \\
 & | & (\text{set! } \langle var \rangle \ \langle exp \rangle)
\end{array}
$$

| (if ⟨*exp*⟩ ⟨*exp*⟩ ⟨*exp*⟩)

| (begin ⟨*exp*⟩*)

| (⟨*exp*⟩ ⟨*exp*⟩*)

Описания ⟨*variable*⟩, ⟨*literal*⟩, ⟨*primitive*⟩ опущены ради простоты.

Пример полноценной функции до и после рассахаривания представлен на Листинге 12 — здесь декларация функции вынесена наверх, сама функция определена через лямбда-выражение, а конструкция `cond` рассахарена в условную конструкцию.

Листинг 12: Вычисление факториала до и после рассахаривания

```
(define (fact n)
  (cond
    ((= n 0) 1)
    (else (* n (fact (- n 1))))))
=>
(define fact (void))
(set! fact
  (lambda (n)
    (if (= n 0)
        1
        (* n (fact (- n 1))))))
```

Интересный факт состоит в том, что и эти, приведенные выше, конструкции могут быть упрощены. Мы исследовали тот факт, что Scheme, как пример функционального языка программирования, имеет в основе лямбда-исчисление. Т.е., рассахаривание языка возможно вплоть до выражения всех конструкций в **прикладном лямбда вычислении**.

Можно считать лямбда-абстракцию главной частью получившегося ядра языка. Но, если рассахаривать язык вплоть до прикладного лямбда

исчисления, то правило замены лямбда-абстракции будет следующим:

$$(\lambda\ (v_1 \dots v_N)\ body) \equiv \lambda\ v_1.\ \lambda v_2. \dots \lambda\ v_N.\ body$$

Аппликация описанной выше лямбда-абстракции с N аргументами заменяется по следующему правилу:

$$(f\ arg_1 \dots arg_N) \equiv (\dots((f\ arg_1)\ arg_2) \dots arg_N)$$

Если применить все эти правила, то к этому моменту синтаксическим сахаром будут являться лишь числа и логические типы. То есть, к этому моменту программа будет представлена прикладным лямбда-исчислением. Но ее можно рассахарить до **чистого лямбда исчисления**. В этом могут помочь, например, нумералы Черча, рассмотренные в обзоре лямбда-исчисления и приведенные в Таблице 1.

По сути, 2 вышеописанных шага позволяют нам перейти от программирования к математической логике, при этом оставаясь в рамках аппарата Scheme.

Тем не менее, в данной работе не стоит цели «тотального рассахаривания», поэтому последние преобразования выполнены не будут — нас устраивает выделенное ранее ядро языка из ≈ 10 конструкций. Работу над ним мы и продолжим.

4.2 Компиляция динамических структур

Для рассмотрения следующего шага компиляции необходимо вернуться к лямба-выражениям и, конкретно, переменным в них. Как уже известно, лямбда-абстракция связывает символ в выражении таким образом, что

он становится заменяемым параметром, например, x в

$$\lambda x.x + 2$$

Такой символ называется **связанным** (входящим связанно). Но, если в выражении присутствует другой символ, например, y:

$$\lambda x.\ x * y + 2$$

то он называется **свободным** (входящим свободно). Поскольку не известно, что это за символ и какое у него значение, вычислить значение такого выражения нельзя.

Строго говоря, это применимо и к другим символам: $*$, $+$ и 2. В нашем случае, лишь потому, что мы используем прикладное лямбда-вычисление, нам известно, что значат эти символы. В случае программы эти символы, как правило, заданы в библиотеке, либо в самом языке.

Можно говорить о свободных переменных, как о переменных, заданных где-то «снаружи» выражения, в окружающем контексте, называемом **окружением** (environment). Строго говоря, окружение ϵ — это частичная функция, отображающая переменные var в их значения def:

$$\epsilon :\ var \rightharpoonup def$$

Окружением может быть большее выражение (в которое входит рассматриваемое), библиотека или сам язык.

Лямбда-выражения при этом так же делятся на 2 типа:

- Закрытые (closed) выражения — каждый символ в них связан своей лямбда-абстракцией. Другими словами, такие выражения самодо-

статочны и у них нет необходимости «заглядывать» в окружающий контекст.

- Открытое (open) выражение — в нем присутствуют несвязанные символы и нам необходима контекстуальная информация, если мы хотим вычислить значение.

Чтобы «закрыть» открытое выражение, необходимо предоставить окружение, которое связывало бы все переменные с конкретными значениями (числами, строками, другими функциями).

Замыкание (closure) лямбда-выражения это набор символов, определенных снаружи выражения (в окружении), которые определяют значения свободных символов лямбда-выражения, делая их, таким образом, связанными. Замыкание превращает открытое лямбда-выражение со свободными вхождениями переменных в закрытое.

В рассмотренном выше выражении $\lambda x.\ x * y + 2$ символ y свободен. Но, пусть существует следующее окружение ϵ:

$$\epsilon[+ \mapsto opertaion_+]$$
$$\epsilon[* \mapsto opertaion_*]$$
$$\epsilon[2 \mapsto number_2]$$
$$\epsilon[baz \mapsto 42]$$
$$\epsilon[y \mapsto 3]$$

Оно содержит значения всех неизвестных свободно входящих символов ($*$, y, $+$ и 2) и даже дополнительных символов (baz). Этот набор (за исключением лишнего baz) замыкает, т.е. закрывает лямбда-выражение, делая его вычислимым.

В языке Scheme поддерживаются лямбда-выражения (он из них состоит), но в большинстве других языков, как правило, императивных, такой поддержки нет. Следовательно, необходимо позволить использовать окружение при реализации анонимных функций, в какой бы язык мы не планировали компилировать Scheme.

Проблема здесь заключается в том, что большинство таких переменных определены на стеке и по завершении вызова они будут оттуда удалены. Тогда, чтобы лямбда выражение продолжило работать, необходимо «захватить» все свободные переменные из окружения и сохранить их таким образом (например, скопировать), чтобы в дальнейшем их можно было использовать, даже когда они будут удалены со стека.

С технической точки зрения замыкание может быть представлено структурой на Листинге 13,

Листинг 13: Структура замыкания

```
Closure {
  f: [ указатель на код лямбдафункции- ]
  e: [ указатель на окружение лямбдафункции- ]
}
```

которую необходимо использовать в программе вместо самой лямбда-функции.

На этом этапе компиляции необходимо заходить в анализируемые выражения, рекурсивно уходя «вглубь» и, дойдя до самого выложенного лямбда-выражения, необходимо сконструировать замыкание: проанализировать аргументы функции, вычислить свободные переменные и заменить обращение к свободным переменным на обращение к окружению.

Можно рассмотреть этот процесс на примере следующего лямбда-

выражения:

$$\lambda f.\ (\lambda x.\ f\ x\ a))$$

Оно может быть переписано на языке Scheme и представлено на Листинге 14

Листинг 14: Лямбда-выражения для ковертации замыкания

```
(lambda (f)
  (lambda (x)
    (f x a)))
```

Такое выражение будет конвертировано в набор замыканий, представленный на Рисунке 5.

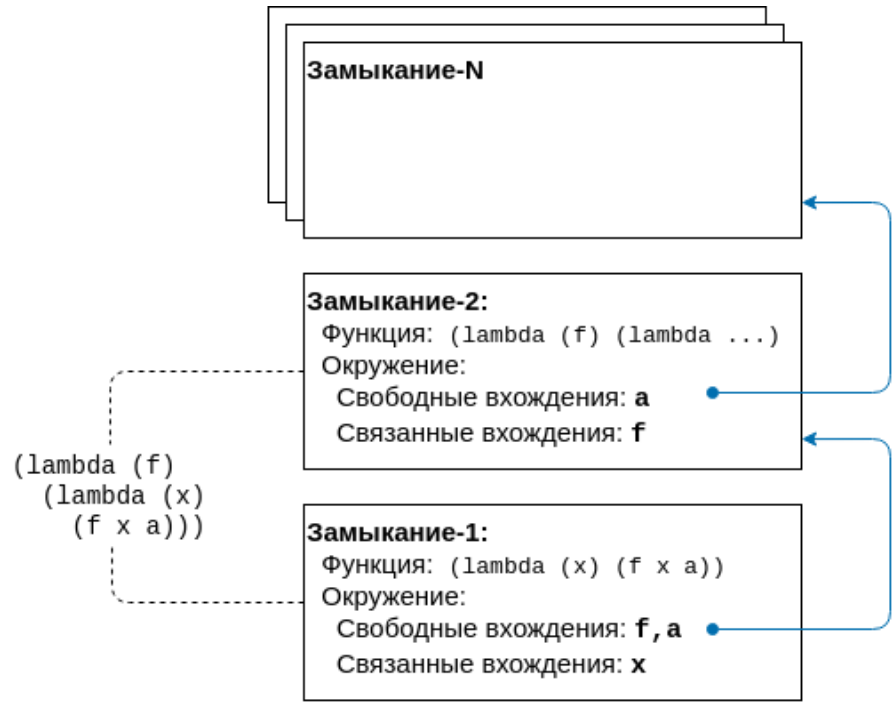

Рис. 5: Связанные замыкания

Здесь самое вложенное замыкание вместо конкретных значений `f` и `a` содержит указатель на замыкание уровнем выше. Переменная `x` в Замыкание-1 входит связанно.

Поднявшись на уровень выше можно заметить, что `f` связывается. Для разрешения всех переменных необходимо найти значение `a`, которого нет и на этом уровне. Тогда остается подниматься выше и выше, анализируя каждое объемлющее окружение. Если значение `a` не найдется и в «глобальном» окружении, то лямбда-выражение будет неразрешимым. Такое поведение вызовет ошибку компиляции программы, т.к. во время выполнения нам не хватит всех данных для работы.

Замену можно представить также в псевдокоде на Листинге 15: до замены было одно лишь лямбда-выражение, после замены лямбда начала принимать окружение, в котором находятся значения перменных `a` и `b`.

Листинг 15: Реализация лямбда-выражения через замыкание

```
lambda (x): x + a + b
=>
{
  lambda (env, x): x + env.a + env.b;
  env: {'a': a, 'b': b}
}
```

На этом второй шаг компиляции можно считать выполненным.

4.3 Генерация кода

Фаза синтеза — последний шаг процесса компиляции. К данному моменту, как правило, имеется уже некоторое промежуточное представление программы, анализируя которое можно распределять память, генерировать код и постобрабатывать получившуюся программу.

Промежуточный код обычно представлен направленным ациклическим графом (Directed acyclic graph, DAG). Идентификаторы, имена или

константы представлены листовыми вершинами, а операторы представлены промежуточными вершинами [1].

К такому графу может быть применено множество оптимизаций:

- удаление лишних инструкций — в случае, например, «транзитивных» присвоений, таких как `a <- b; b <- c;` переменная `b` оказывается лишней и может быть удалена без изменения смысла программы.

- удаление «мертвого» кода (dead code elimination, DCE).

  - удаление недостижимого кода — удаление блоков кода, куда из-за выстроенных языковых конструкций, никогда не будет передано управление, например, кода, следующего после конструкции `return` во многих языках программирования.

  - удаление «мертвых» переменных — переменных, в которые записываются значения, но откуда они никогда не считываются обратно [4].

- оптимизация графа потока управления (control flow graph, CFG) — оптимизации, в ходе которых граф потока управления приводится, например, в форму единственного присвоения переменных (Static single assignment form). Хотя, в случае функциональных языков обычно используется форма передачи продолжений (Continuation-passing style, CPS).

- упрощение арифметических выражений.

- упрощение операторов, например, замена умножений на битовые сдвиги, т.к. они выполняются эффективнее с точки зрения процессора [5].

Тем не менее, эта фаза компиляции зависит от целевого кода (target language), т.к. некоторые инструкции могут быть упрощены в таком случае, а другие, например, компиляция условных конструкций в ассемблерный код, будут, наоборот, усложнены. Следовательно, необходимо прежде всего разобраться с целевым кодом.

4.4 WebAssembly

В данной работе производится компиляция функционального языка в «переносимый код для виртуальной машины». Как было исследовано в главе «Выбор модели вычислений», виртуальная машина является самой распространенной на практике реализацией конечного автомата.

Виртуальная машина сочетает в себе плюсы компиляции и интерпретации одновременно: вместо конвертации исходного кода программы в набор инструкций конкретного процессора, программа компилируется в т.н. «байт-код» — промежуточный код, который далее будет исполняться не процессором, а виртуальной машина. Виртуальная же машина является уровнем абстракции, изолирующим программную часть от аппаратной таким образом, что исходная программа может быть скомпилирована единожды, а затем может выполняться на любой платформе без необходимости перекомпиляции. В то же время, разработка такой машина нетривиальна, т.к. она должна скрывать «под собой» особенности операционных систем, на которых запускается программа

Слоган «*напиши один раз, запускай где угодно*» (Write once — run anywhere, WORA) стал визитной карточкой одной из самых популярных виртуальных машин — Java Virual Machine, JVM.

Между тем, JVM — не единственная виртуальная машина. В 2017

году совместными усилиями компаний Mozilla, Google, Apple и Microsoft был представлен WebAssembly (сокращаемый до WASM) — стандарт портативного байт-кода для исполнения в браузере [14].

WASM можно рассматривать как некоторое промежуточное представление (intermediate representation, IR), похожее, например, на LLVM IR. Можно говорить о нем, как об ассемблере, исполняемом абстрактной виртуальной машиной. В браузерной виртуальной машине WebAssembly должен работать «рядом» с JavaScript и заниматься ресурсоемкими задачами, например, 3D-моделированием, дополненной и виртуальной реальностью (AR, VR) и прочими вычислениями.

Некоторое время спустя был представлен стандарт WebAssembly System Interface (WASI) для работы приложений вне браузера, как когда-то NodeJs, собранный на базе движка V8, позволил языку JavaScript стать «серверным» и даже «системным», но рассмотрение WASI выходит за рамки работы [15].

Простота, компилируемость, эффективность и безопасность делают WASM подходящей технологией для дальнейшего развития среды веб.

WebAssembly не является языком программирования в общем понимании этого термина, а является скорее целевой технологией в терминах компиляции. В случае известной среды LLVM (Low level virtual machine), WebAssembly является бекендом компилятора, т.е. LLVM может генерировать WebAssembly из практически любого языка.

Тем не менее, на данный момент (2019-2020 год) официально поддерживается только компиляция из C/C++ и Rust. Для остальных языков, например, Go, существует лишь экспериментальная поддержка. Функциональные же языки, такие, как Scheme, не поддерживаются вовсе из-за различия парадигм: LLVM работает в классической императивной парадигме.

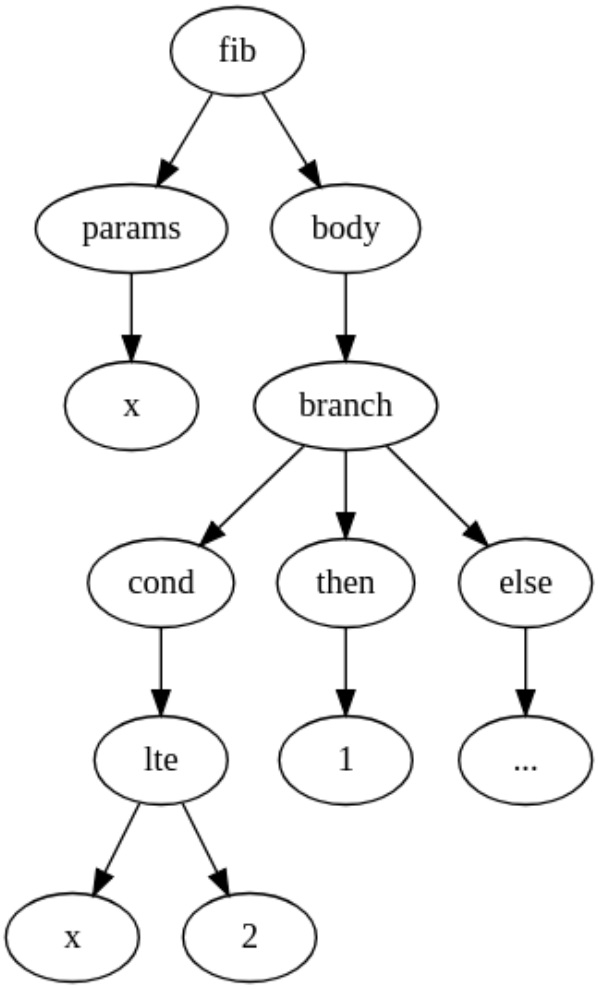

Рис. 6: Древовидное представление функции fibonacci

4.5 Программное решение

В данной работе в качестве целевой технологии мы будем генерировать текстовый формат WebAssembly, называемый WAT (WebAssembly text format).

Lisp был первым языком, использовавшим S-выражения (Symbolic expression) для записи конструкций языка, а WebAssembly можно считать самым современным их применением. Единица распространения

программ на WASM — модуль.

Модуль, как черный ящик, экспортирует наружу свои функции, которые могут быть вызваны из JavaScript. Модуль так же может импортировать в себя функции других модулей.

Модуль, экспортирующий функцию $id(x) = x$ представлен в WAT-формате на Листинге 16.

Листинг 16: Экспорт функции тождественного отображения из WASM-модуля

```
(module
    (func $id (param i32) (result i32)
        get_local 0
    )
    (export "identity" (func $id))
)
```

Тело функции `$id` состоит из всего лишь одной инструкции — получения на вход параметра с индексом 0. Сигнатура функции указывает на то, что функция принимает одно значение типа Int32 (`(param i32)`) и возвращает одно значение того же типа (`(result i32)` [16].

На Рисунке 6 изображено схематичное древовидное представление функции вычисления фибоначчи. Задача этого шага компиляции, как оговорилось выше, заключается в правильном переносе древовидной структуры на последовательный набор инструкций. На Листинге 17 представлена та же самая функция в формате WAT.

Листинг 17: Функция вычисления чисел Фибоначчи

```
(module
    (func $fib (param $n i32) (result i32)
        get_local $n
```

```
        i32.const 2
        i32.le_s
        (if (result i32)
            (then i32.const 1)
            (else get_local $n
                  i32.const 1
                  i32.sub
                  call $fib

                  get_local $n
                  i32.const 2
                  i32.sub
                  call $fib

                  i32.add)))
    (export "fibonacci" (func $fib))
)
```

Как следует из формулы вычисления числа Фибоначчи, каждо последующее число образовано из суммы предыдущего и предпредыдущего числа. Если же взглянуть на ветку `(else ...)` то становится понятна следующая особенность WASM: выполнение основано на простой **стековой машине**. То есть, чтобы вычислить предыдущее число Фибоначчи, необходимо поместить на стек текущее число с помощью `get_local $n` (к параметрам можно обращаться не только по индексам, но и по именам) и число 1. Затем операция `i32.sub` достанет 2 числа со стека, выполнит вычистание $(n - 1)$ и положит на стек результат вычитания.

Следующий вызов функции `fib` с помощью инструкции `call $fib`

произойдет с числом, лежащим на стеке в качестве аргумента вызова.

Поскольку сигнатура функции содержит `(result i32)`, то к окончанию работы функции на стеке должно остаться лишь одно целое число.

Работа с не числовыми данными немного сложнее, и организована через память. По умолчанию для работы WASM модуля выделяется некоторое линейное пространство памяти. При этом через эту память возможна работа с JavaScript, поскольку с т.з. JavaScript, память модуля представлена байтовым массивом `ArrayBuffer`.

Теперь необходимо лишь вызвать экспортированную функцию из JavaScript, т.к. WebAssembly выполняется в т.н. песочнице (sandbox) из соображений безопасности. Вызов функции представлен на Листинге 18

Листинг 18: Вызов экпортированной функции fibonacci WASM-модуля

```
WebAssembly.instantiateStreaming(fetch(WASM_FILE_NAME))
        .then(obj => {
            const { fibonacci } = obj.instance.exports;
            const result = fibonacci(9);
            console.log(result);
        });
```

Веб-сервер на Листинге 19 будет «раздавать» веб-страницу, скрипты и остальной статический контент, в том числе и WASM-модуль. Сервер может быть запущен следующей командой:

```
$ go run server.go
```

Листинг 19: Простейший веб-сервер для статического контента server.go

```
package main

import ("net/http")
```

```
func main() {
        mux := http.NewServeMux()
        mux.Handle("/", http.FileServer(http.Dir(".")))
        _ = http.ListenAndServe(":8080", mux)
}
```

Вызов JavaScript кода на Листинге 18 с полученной от веб-сервера страницы загрузит WASM-модуль с именем `"module.wasm"`, свяжет экспортированную функцию `"fibonacci"` с JavaScript-переменной и вызовет функцию с аргументом «9». В результате в поток вывода (лог консоли браузера) будет распечатан результат выполнения функции: $fib(9) = 34$.

4.6 Анализ результатов

На данный момент процесс компиляции можно считать завершенным. Осталось собрать воедино все компоненты проекта.

Листинг 20: Вычисление числа Фибоначчи

```
#lang racket
(define (fib n)
  (cond
    ((<= n 2) 1)
    (else (+ (fib (- n 1)) (fib (- n 2))))))
```

Итоговая схема решения представлена на Рисунке 8 и работает следующим образом

- Исходная программа на языке Scheme считывается и «рассахаривается». Исходной программой в качестве примера может выступать

программа вычисления числа Фибоначчи, представленная на Листинге 20.

- Далее динамические структуры программы конвертируются в статические с помощью замыканий лексического окружения.

- Далее по полученному линейному представлению программы генерируются WASM-инструкции, представленные на рассмотренном ранее Листинге 17.

- Затем программу в текстовом виде необходимо конвертировать в бинарный вид. Для этого используется программа `wat2wasm` из стандартного набора утилит WebAssembly Binary Tools (WABT).

```
$ wat2wasm module.wat -o module.wasm
```

- Полученный WASM-модуль будет инстанциирован при загрузке веб-страницы. Для этого необходимо настроить веб-сервер на «раздачу» статического контента. Это можно сделать с помощью веб-сервера на Листинге 19, так и с помощью стандартного модуля языка Python:

```
$ python -m http.server 8080
```

- После этого веб-страница с WASM-модулем доступны на порту 8080 текущего хоста.

После этого пользователь может открыть веб-страницу, ввести аргументы, а код с Листинга 18 вызовет эту функцию.

Пусть программа будет вызвана 3 раза с аргументами 5, 9, 10, 11. Тогда результат выполнения представлен на скриншоте на Рисунке 7.

Далее можно вручную протестировать работоспособность программы и корректность выдаваемых ею результатов. Эти результаты полностью сходятся с изначальным вариантом программы в функциональном стиле на языке Scheme.

То есть, программа, которая изначально была рассчитана на выполнение исключительно на настольном компьютере теперь выполняется на любом устройстве пользователя.

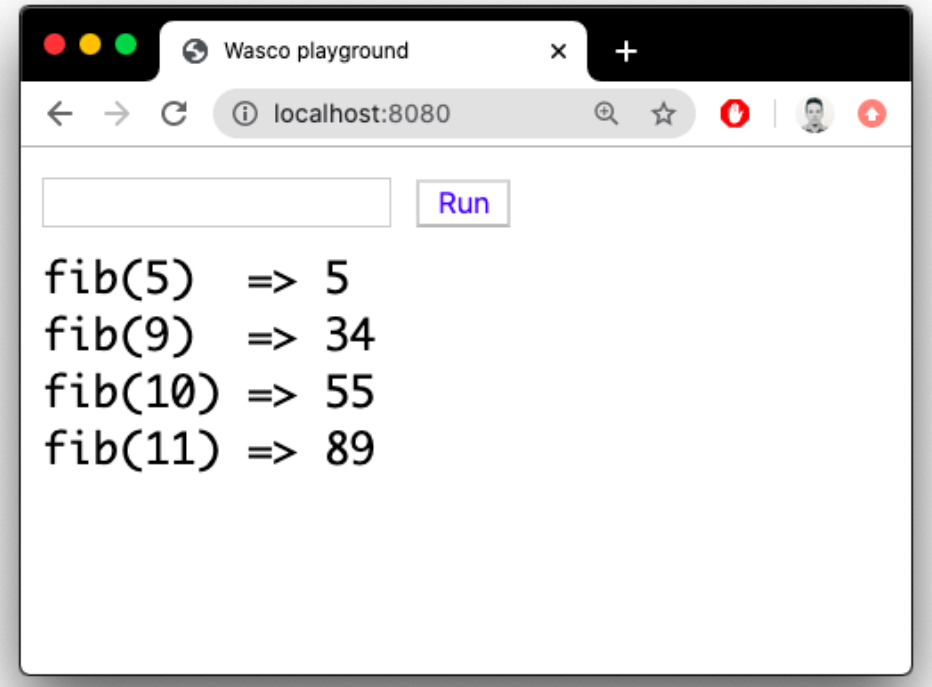

Рис. 7: Программа, разработанная для настольного компьютера выполняется в браузере на устройстве пользователя

Рис. 8: Схема работы решения

ЗАКЛЮЧЕНИЕ

В ходе данной работы были исследованы способы представления программ с помощью математических моделей. Выбранная математическая модель позволяет говорить о программе, как о композиции математических функций.

Была представлен способ моделирования программ, а затем был сделан переход от системы типов языка программирования к пропозициональной математической логике. Были исследованы способы трансформации программы по аналогии с эквивалентными преобразованиями логических формул.

Были исследованы подходы к компиляции функциональных языков программирования и самый подходящий из этих способов был программно реализован.

В качестве итогового решения был разработан компилятор, подтверждающий работоспособность решения и выполняющий компиляцию функционального языка Scheme в байт-код WebAssembly. Таким образом, программы, написанные на языке Scheme для настольных компьютеров могут быть переведены и исполнены в веб среде посредством браузера на любом устройстве с доступом в интернет.

Сформулированы рекомендации по улучшению реализации и более углубленному изучению темы. Дальнейшие исследования могут быть направлены в сторону оптимизации базового решения в сторону повышения эффективности, скорости и корректности.

Таким образом, цели и задачи, поставленные в работе можно считать выполненными.

Список литературы

1. А.Ахо, М.С.Лам, Р.Сети, Д.Ульман — Компиляторы: принципы, технологии и инструментарий — Вильямс, 2008 — 617 с.

2. Х.Абельсон, Д.Сассман — Структура и интерпретация компьютерных программ — 2006 — 520 с.

3. Д.Сассман, Г.Стил — Scheme: A Interpreter for Extended Lambda Calculus — 1998

4. Torben E.Mogensen — Basics of Compiler Design, anniversary edition — lulu, 2010 — 133 с.

5. S.Muchnick — Advanced Compiler Design and Implementation — Morgan Kaufmann, 1997 — 488 с.

6. Finite State Machines [Электронный ресурс] — Режим доступа: https://computationstructures.org/notes/fsms/notes.html — Дата обращения: 08.04.2020

7. Glynn Winskel — Formal Semantics of Programming Languages — The MIT Press, 1993 — 98 с.

8. Case Study: Actor Scheduler [Электронный ресурс] — Режим доступа: http://www.1024cores.net/home/scalable-architecture/case-study-actor-scheduler — Дата обращения: 22.03.2020

9. Cliff B.Jones, A.W.Roscoe, R. Wood — Reflections on the Work of C.A.R. Hoare — Springer, 2010 — 127 с.

10. The Lambda Calculus [Электронный ресурс] — Режим доступа:

https://plato.stanford.edu/entries/lambda-calculus — Дата обращения: 17.06.2019

11. Morten Sorensen, Pawel Urzyczyn — Lectures on the Curry-Howard Isomorphism. Studies in Logic and the Foundations of Mathematics — Elsevier Science, 2006 — 246 с.

12. Typed Functional Programming and Software Correctness [Электронный ресурс] — Режим доступа: https://dimjasevic.net/marko/2018/10/23/typed-functional-programming-and-software-correctness/ — Дата обращения: 10.05.2020

13. Tree transformations: Desugaring Scheme [Электронный ресурс] — Режим доступа: http://matt.might.net/articles/desugaring-scheme/ — Дата обращения: 21.12.2019

14. WebAssembly Introduction [Электронный ресурс] — Режим доступа: https://webassembly.github.io/spec/core/intro/introduction.html — Дата обращения: 10.06.2020

15. Standardizing WASI: A system interface to run WebAssembly outside the web [Электронный ресурс] — Режим доступа: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface — Дата обращения: 13.06.2020

16. Writing WebAssembly By Hand [Электронный ресурс] — Режим доступа: https://blog.scottlogic.com/2018/04/26/webassembly-by-hand.html — Дата обращения: 20.11.2019