КУРСОВАЯ РАБОТА

на тему: «Разработка вычислительной системы с механизмом самоопределения номинальных операций»

Направление: 09.03.03 Прикладная информатика

Направленность программы: «Прикладная информатика в психологии»

Аннотация

В данной курсовой работе с позиций реляционной биологии Н. Рашевского рассмотрены вопросы моделирования живых систем, в виде (M,R) системы, предложенной Р. Розеном. На основе этого рассмотрения поставлена цель разработки модели логического механизма с самоопределением бинарных номинальных операций и возможности его использования в качестве вычислителя нового типа. Для его программирования был использован язык C++. В результате в курсовой работе разработаны алгоритмы логического механизма и реализована его компьютерная программа. Предложенный механизм был затем включен в машину Поста для создания абстрактного вычислителя, обладающего некоторыми свойствами (M,R) систем, такими, как самоопределение и замкнутые петли внутренних причин. Доказана полнота системы вычислителя по Тьюрингу. Проведены эксперименты с вычислителем, показывающие, что полученная вычислительная система обладает высокой чувствительностью к начальным условиям и при этом содержит в результатах своей работы большое количество предельных циклов. Выдвинуто предположение, что эта вычислительная система может быть полезна в областях криптографии, клеточных автоматов и взаимодействия множества автономных агентов.

Оглавление

Введение 4

Глава 1. Моделирование механизма с самоопределением бинарных номинальных операций 6

Глава 2. Компьютерная реализация механизма с самоопределением бинарных номинальных операций 13

2.1. Процедуры работы механизма 13

2.2. Программирование алгоритмов механизма 16

2.3. Блок-схема программы механизма 19

2.4. Демонстрация результатов работы программы механизма 23

Глава 3. Разработка вычислительной системы с механизмом самоопределения бинарных номинальных операций 28

3.1. Общее представление о вычислителях 28

3.2. Машина Поста 30

3.3 Программирование машины Поста 31

3.4 Особенности языка программирования модифицированной машины Поста 36

Глава 4. Программирование вычислительной системы с механизмом самоопределения бинарных номинальных операций и доказательство её полноты по Тьюрингу 40

4.1. Доказательство полноты по Тьюрингу разрабатываемой вычислительной системы 40

4.2. Эксперименты с вычислительной системой 42

4.2.1. Первый эксперимент 45

4.2.2. Второй эксперимент 46

4.2.3. Третий эксперимент 47

4.2.4. Четвёртый эксперимент 47
4.2.5. Выводы из экспериментов 48
Заключение 50
Список источников 52
ПРИЛОЖЕНИЕ А 54
ПРИЛОЖЕНИЕ Б 56
ПРИЛОЖЕНИЕ В 62

Введение

Известно, что объяснение биологических процессов с точки зрения физики, как и в целом моделирование жизни, является трудной задачей (Шредингер, 2002). Те научные подходы, которые разработаны в физических и вычислительных задачах, дают сбой при описании работы такой структуры, как живая клетка. Абстрактная формализация живой системы в виде так называемой (M,R) системы была предложена в рамках так называемой реляционной биологии (Rashevsky 1961; Rosen, 1991). Процессы в (M,R) системе обладают свойством замыкания действующей причины, что делает систему сверхсложной и невычислимой в рамках машины Тьюринга (Rosen, 1991; Артеменков, 2016, 2022). В упрощенном виде в этой системе все внутренние функции самоопределяются внутри системы, что в целом пока непонятно как можно реализовать в больших технических моделях.

В настоящей работе рассматривается исследовательский подход представления (M,R) системы в виде упрощенного логического механизма с самостоятельным определением бинарных номинальных операций. Использование этого механизма в составе процессора вычислительной системы может быть интересным для получения новых исследовательских и практических вычислительных результатов.

Цель данной курсовой работы — представить двузначную логическую модель в виде простой вычислимой имитации (M,R) системы и разработать на этой основе программу-эмулятор процессора, концептуально основанного на машине Поста.

Чтобы достичь поставленной цели, необходимо решить следующие задачи:

1. Описать модель механизма с самоопределением бинарных номинальных операций.

2. Разработать алгоритмы описанного механизма и реализовать его компьютерную программу.
3. Разработать вычислительную систему с механизмом самоопределения бинарных номинальных операций.
4. Написать и протестировать программы для полученной вычислительной системы с механизмом самоопределения бинарных номинальных операций и доказать её полноту по Тьюрингу.

Данная курсовая работа состоит из введения, 4 глав, заключения, списка использованных источников и 3-х приложений.

Глава 1. Моделирование механизма с самоопределением бинарных номинальных операций

Разработку новых систем в науке разумно начинать с рассмотрения системных свойств и качеств с тем, чтобы определить те важные характеристики системы, которые будут характеризовать конкретную разработку. Известно, что системой обычно принято называть совокупность элементов, находящихся в отношениях и связях друг с другом, которая образует определённую целостность. Одним из важных факторов целостности является способность системы самостоятельно определять себя и свою организацию. Известно, что живые системы, в отличие от не живых, в известной мере, могут обладать особыми свойствами целостности. Вместе с тем создание таких систем по-настоящему не входит в задачи современной системотехники, в частности, потому что возможности построения таких систем пока недостаточно поняты наукой.

Со стороны физической науки одним из первых понять, что делает систему живой, пытался Э. Шрёдингер. Он пришёл к выводу, что такой её делает открытость, то есть непрерывный обмен веществом и энергией с окружающей средой. При этом исследование открытых систем оказывается много более трудным делом, поскольку физика работает в основном с закрытыми системами, не обменивающимися ничем с внешним миром. В результате дальнейшая формализация жизни оказалась невозможной ввиду того, что упрощенный физический подход оказался неприменим к живым существам (Шрёдингер, 2002).

Биология тоже столкнулась с трудностью объяснить жизнь на основе физикального подхода, согласно которому внутренние процессы живых организмов определяются материей, из которой состоит организм. Пытаясь понять основы механизмов жизни, биохимик Н. Рашевский предложил абстрагироваться от материальной составляющей и, в первую очередь, рассмотреть вопрос организации процессов в живой материи. Для этого

потребовалось взглянуть на возникающие особые отношения и связи в системе. Такой подход был назван реляционным. Согласно ему, все основные функции живого организма, такие как метаболизм, размножение и самовосстановление, причинно определяются в процессах внутри живой системы (Rashevsky, 1961).

Решая задачу, предложенную Н. Рашевским, его ученик математик Р. Розен предложил максимально абстрактную формализацию живой системы в виде так называемой (M,R) системы (Rosen, 1991). Эта система «в сжатом виде показывает организацию процессов метаболизма и восстановления в живой клетке», которые являются замкнутыми по действующей причине (Артеменков, 2016, с. 55).

Общая схема (M,R) системы показана на рисунке 1.1 ниже. Данный рисунок схемы представлен в работе, посвящённой попыткам моделирования (M,R) системы в алгебре процессов (Gatherer, Galpin, 2013). Узлы на рисунке связаны друг с другом с помощью двух видов стрелок. Полые стрелки обозначают процессы производства/преобразования клеточного материала, а стрелки с полными черными направляющими обеспечивают создание самих производящих механизмов. Таким образом, полые стрелки отвечают процессам, соответствующим материальной и формальной причинам. Полные стрелки обозначают действующие причины.

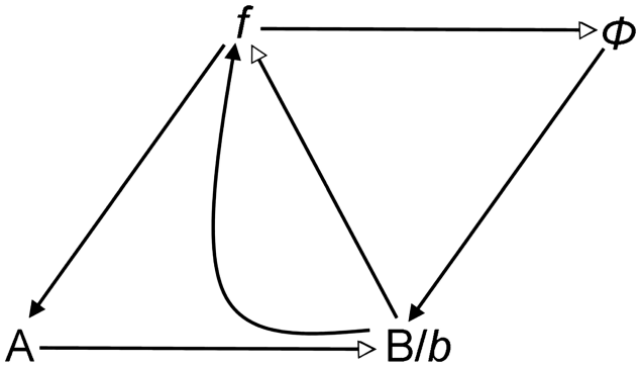

Рис. 1.1. Схема (M,R) системы

На рисунке А производит В, а *f* катализирует производство В из А. При этом Ф катализирует производство *f* из В, а *b* катализирует производство Ф из *f*. (M,R) система, как живая система, отличается от механизмов и машин, создаваемых человеком тем, что она замыкается по действующей причине, т. е. имеет возможность производить сама себя. В качестве минимального воплощения (M,R) системы в биологии рассматривается живая клетка. Конкретизацией особой системной организации живой клетки, в частности, является современная теория автопоэзиса (Матурана, Варела, 2001). Однако интересным является тот факт, что и более глобальные отношения между отдельными частями и процессами головного мозга также могут иметь вид (M,R) системы (Kercel, 2004).

Р. Розен предложил следующую формализацию (M,R) системы (Артеменков, 2016, с. 55):

$$F = \left\{\{\{F\}\}\right\}, F \to \{F\} \to \left\{\{F\}\right\} \to \left\{\{\{F\}\}\right\} \quad (1)$$

где F — гипермножество, то есть множество, содержащее в качестве своего элемента минимум одно множество. Это гипермножество помимо себя содержит все допустимые функциональные преобразования в системе (1).

Из приведённой формулы (1) видно, что (M,R) система описывается максимально обобщённо, что не позволяет сделать никаких выводов кроме одного: она рекуррентно определяется сама через себя. Формула (1) скорее описывает топологию системы, чем то, что можно поместить в компьютер в качестве алгоритма (доказательство этого факта приводится в приложении А). При этом стоит заметить, что в (M,R) системе между её частями присутствуют неоднозначные отношения, не позволяющие точно описать, что происходит в отдельных её частях в конкретный момент времени, не привлекая контекст остальных её частей — (M,R) система является контекстно-зависимой/чувствительной. Она несводима к своим компонентам и алгоритмически невычислима, поскольку не может быть оценена как функция

машиной Тьюринга. Если в реальных биологических сетях присутствуют (M,R)-подобные структуры, это говорит о том, что многие биологические сети будут невычислимыми, что имеет значение для тех областей системной биологии, которые используют компьютерное моделирование для прогнозирования (Gatherer, Galpin, 2013).

В связи с этим попытки моделирования (M,R) системы на компьютере не являются продуктивными. Например, (M,R) систему моделировали с помощью алгебры процессов Bio-PEPA, что приводило к тому, что система, основанная на случайных процессах, либо приходила к стабильному состоянию («умирала»), либо «жила» на протяжении 1000 тактов модели, после чего «умирала» (Gatherer, Galpin, 2013). Авторы этого исследования пришли к выводу, что полученные результаты могли бы служить опровержением теории Н. Рашевского и Р. Розена, однако при детальном рассмотрении оказывается, что некоторые аспекты (M,R) системы при моделировании не были учтены. В частности, рассмотренные авторами модели, основанные на лямбда-исчислении, являются Тьюринг-вычислимыми, так как их возможно реализовать на языке LISP, что вступает в противоречие с тезисом о невычислимости (M,R) системы.

Другие аспекты (M,R) системы, как то, откуда она берёт материал для работы, иногда опускаются, что приводит к неуверенности в том, что в результате получилась именно (M,R) система. Всё это перекликается с различием моделирования (как имитирования в модели механизмов системы) и симулирования. Симулирование, согласно приведённому исследованию, используется в системной биологии и предполагает воссоздание «внешних» признаков исследуемого объекта без его внутренней сложности. Симуляция может прогнозировать будущее, однако, если что-то пойдёт не так, невозможно будет понять, почему это произошло. На самом деле, внутренние процессы симуляции и реального объекта качественно различны, и их нельзя сравнивать напрямую. Симуляция упрощает исследуемую реальность, а имитационное

моделирование пытается передать внутренние и внешние отношения системы во всех их полноте. Р. Розен считал, что создать модель живого организма невозможно (Rosen, 1991; Cornish-Bowden et al., 2013).

Помимо алгебры процессов существуют модели, основанные на метаболических сетях, более приближенные к биологической реальности. Их авторы рассматривают моделирование всей (M,R) системы и отдельных её частей, приходя к выводу о неоднозначности этой возможности в том числе из-за терминологической трудности. Например, восстановление (репликация) в смоделированной системе получалась не совсем та, о которой говорил Р. Розен (Cárdenas, et al., 2010).

Вместе с описанным существует подход, предлагающий моделировать поведение системы, а не саму систему. Для этого использовали случайное блуждание по графам. Такой подход позволяет описывать поведение целых биологических сообществ и оценивать методами статистики вклад отдельного участника сообщества в общую динамику (Miranda, La Guardia, 2017).

Известные исследования, которые рассматривали моделирование (M,R) систем на современных компьютерах, приходили к тому, что в полной мере это сделать невозможно при учёте всех необходимых свойств и качеств. В связи с этим можно поставить вопрос не о моделировании, а о симуляции, которая может оказаться полезной для вычислений и выявления ранее неизвестных свойств математических объектов, например, чисел. Чтобы симулировать (M,R) систему в машине Тьюринга нужно задать неоднозначность возникающих в ней отношений и свести незамкнутость по действующей причине к минимуму. Зная, что программирование — это порождение операций из некоторого набора физических процессов, можно прийти к тому, что симуляция (M,R) системы должна быть самопрограммируемой настолько, насколько это возможно. Этого можно добиться, создав логический механизм, который будет принимать на вход числа и проходить как минимум через три отдельные процедуры с обратными связями, меняющими эти процедуры в

зависимости от результатов вычислений. Результат работы каждой из процедур таким образом будет не только передаваться на вход следующей, но и указывать на то, каким образом должны измениться другие процедуры.

Идея равенства количества состояний системы (чисел) количеству имеющихся в ней операций позволяет задать взаимно-однозначное соответствие между ними. Это позволит не только определять числа с помощью имеющихся операций, но и определять операции в зависимости от производимых чисел. Простейший вариант такого механизма можно реализовать для случая бинарной системы с двумя числами (в частности, 0 и 1) и двумя логическими номинальными операциями: повторения и отрицания. Переопределение системы по процессам может происходить дискретно по тактам во времени. Чтобы реальное соответствие между числами и операциями в модели не было заранее заданным и неизменным, это соответствие может определяться случайным образом. Такой механизм может быть назван механизмом с самостоятельным определением бинарных номинальных операций. Общая схема механизма изображена ниже на рис. 1.2.

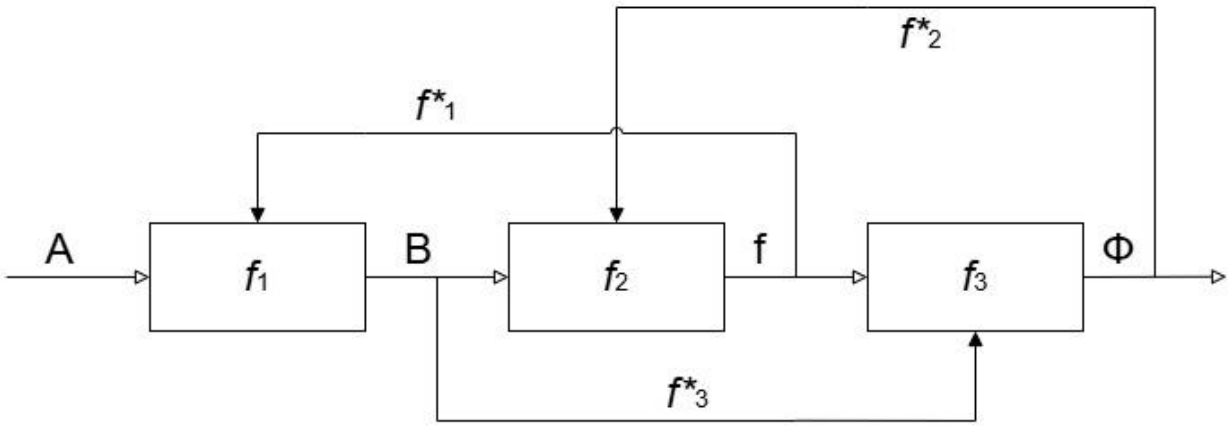

Рис. 1.2. Общая схема логического механизма с самоопределением бинарных номинальных операций

Блоки f_1, f_2 и f_3 на рис. 2 представляют три процедуры с номинальными операциями (повторение или отрицание). Горизонтальные стрелки обозначают вход, выход и переходы между блоками. На входах и выходах блоков подаются

числа 0 или 1. Вертикальные стрелки меняют номинальные функции блоков в зависимости от значений чисел на других блоках (f_1^*, f_2^* и f_3^* определяют новые преобразования в блоках f_1, f_2 и f_3).

Числа, поступающие на вход механизма, последовательно проходят через три блока. На основе выходных значений, получающихся в каждом блоке, преобразования в блоках меняют друг друга: первый блок меняет третий, третий меняет второй, второй меняет первый. Процесс работы механизма можно замкнуть, то есть его результат на выходе механизма подать на вход механизма. Учитывая, что здесь работа будет проводиться не с физическим материалом, а всего с двумя числами и двумя логическими функциями, то предлагаемую модель нельзя назвать полноценной симуляцией (M,R) системы, однако, как будет показано далее, такого рода модель можно встроить в уже известные вычислители, например, машину Поста, и получить необычные результаты.

В следующей главе рассматривается компьютерная реализация логического механизма с самоопределение бинарных номинальных операций.

Глава 2. Компьютерная реализация механизма с самоопределением бинарных номинальных операций

2.1. Процедуры работы механизма

Для более четкого описания работы предложенного логического механизма преобразуем схему на рисунке 1.2 к виду, представленному на рисунке 2.1.

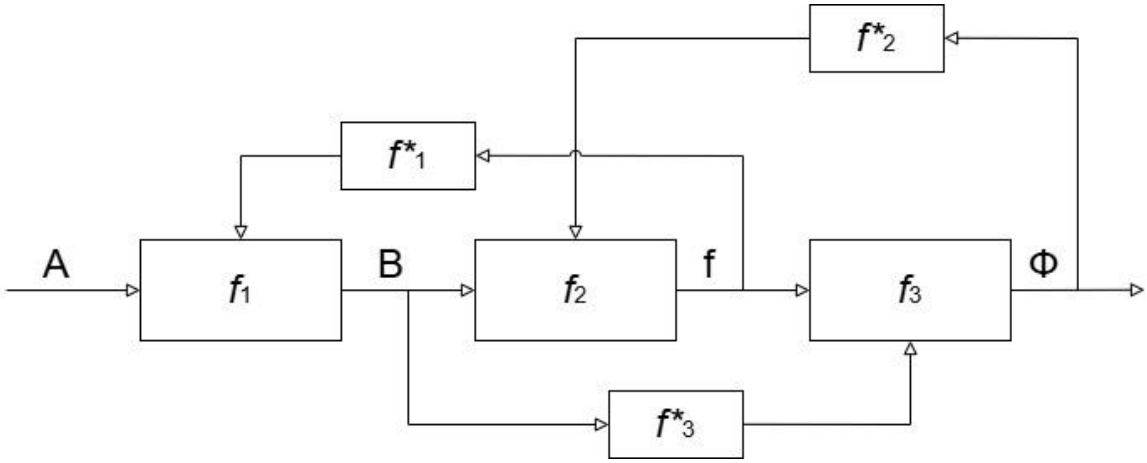

Рис. 2.1. Схема логического механизма с самоопределением бинарных номинальных операций с дополнительными блоками функций f_1^*, f_2^* и f_3^*

Горизонтальные стрелки, входящие на схеме в блоки f_1, f_2, f_3, обозначают постепенную передачу сигнала со входа А до выхода Ф. Функции f_1^*, f_2^* и f_3^* с помощью обратной или прямой связи определяют на схеме изменения функций f_1, f_2 и f_3 и представлены отдельными функциональными блоками, с выхода которых идут вертикальные стрелки, входящие в блоки f_1, f_2, f_3.

В выбранной бинарной логике оказывается возможным установить соответствие между числами и операциями, всех имеющихся в схеме функций. В таблице 2.1 представлен один из вариантов простых соотношений между двумя числами (0 и 1) и функциями f и f^* (повторение и отрицание). Представленные в таблице 2.1 соответствия могут меняться в зависимости от выбора конфигурации системы и при введении случайного изменения введенных соотношений.

Таблица 2.1. Определение функций и их соответствия числам.

Число	Функции	Обратные функции
0	$f_i(x) = x$	$f_i^*(0) = \neg x$
1	$f_i(x) = \neg x$	$f_i^*(1) = x$

Механизм запускается подачей на вход f_1 значения А (0 или 1), после чего, по мере выполнения соответствующих функций, осуществляются преобразования сигналов как внутри схемы, так и на выходе Ф. Рассмотрим вкратце процессы, происходящие в схеме на рисунке 2.1, с помощью имеющихся на схеме символических обозначений. Имеем следующие формулы для прямых и обратных преобразований.

$$\text{В} = f_1(A), \qquad \text{f} = f_2(B), \qquad \Phi = f_3(\text{f}) \tag{2}$$

$$f_1^*(\text{f}) = f_1, \qquad f_2^*(\Phi) = f_2, \qquad f_3^*(\text{В}) = f_3 \tag{3}$$

Используя формулы (2) и (3), а также соотношения в таблице 2.1, можно рассмотреть динамические изменения, происходящие с состояниями переменных в схеме 2.2 по тактам времени с учетом различных алгоритмов работы всех функций во времени. В частности, рассмотрим три варианта алгоритмического циклического действия функций на схеме 2.1.

1. Сначала функции *f* проходят полный цикл обработки данных (аргументов) по формулам (2), а затем функции *f** проходят полный цикл изменения друг друга по формулам (3), после чего всё повторяется вновь. Это соответствует циклу по процедуре (4).

$$f_1 \to f_2 \to f_3 \to f_3^* \to f_1^* \to f_2^* \to \tag{4}$$

2. В цикле каждая из функций *f** проходит изменения по формулам (3), и затем происходит последовательная работа функций *f*. Это соответствует циклу по процедуре (5).

$$f_1^* \to f_3^* \to f_2^* \to f_1 \to f_2 \to f_3 \to \quad (5)$$

3. Функции поочерёдно передают друг другу выходы и изменяются своими выходами после каждой передачи. Это соответствует циклу по процедуре (6).

$$f_1 \to f_3^* \to f_2 \to f_1^* \to f_3 \to f_2^* \to \quad (6)$$

Первый вариант требует предустановки изначальных выходных значений функций *f*,* что определяет функции *f*. После их установки на вход f_1 подаётся число, и механизм запускается. Второй вариант требует предустановки выходов функций *f*, затем происходят все дальнейшие изменения. В третьем варианте можно ограничиться установками первой функции, изначальных состояний и входного значения.

Можно видеть, что при зацикливании процессов по формулам (4) и (5) они в целом с точностью до фазового сдвига по тройке функций образуют одинаковые циклы. Итогом становятся два основных варианта работы механизма:

1. Сначала проходят данные, а потом меняются функции.
2. В цикле обработка данных и изменения функций смешаны.

Следует отметить, что важной особенностью рассмотренного механизма является то, что он сам в зависимости от функциональных вычислений переопределяет свои функции. Но следует заметить, что способ этого переопределения кроме схемной структуры жестко задан извне соответствием функций, представленным в таблице 2.1. Ясно, что соотношение соответствия функций тоже можно менять. В самом механизме эти изменения можно делать

с использованием вероятностного определения соответствий в таблице 2.1. В этом случае данный логический механизм становится стохастическим.

Согласно таблице 2.1 было принято, что единица соответствует функции отрицания. Так можно было провести ассоциацию от заявленных отношений к алгебре в двоичной системе счисления. Добавление случайности разрушает это допущение. Наиболее рационально менять соответствия случайным образом после каждого периода циклов, так как, если менять его после каждого шага внутри цикла, то система может полностью потерять упорядоченность и выродится в генератор случайных чисел.

2.2. Программирование алгоритмов механизма

Программа логического механизма была реализована на языке программирования C++ (компилятор: x86_64-win32-seh-rev0, Built by MinGW-Builds project 14.2.0, стандарт языка: C++17). Код был написан в редакторе кода Notepad++ версии 8.7.2 (x64).

Язык C++, на котором написана программа механизма, является компилируемым, имеет статическую типизацию, поддерживает функциональную, процедурную и объектную парадигмы программирования. Программа написана в процедурной парадигме. Сам механизм описан в процедуре henkamono(), функции, не возвращающей ничего.

Процедура принимает на вход входное значение для механизма, изначальные состояния его функций, значение «Использовать ли цикличность?» (подача на вход А первой функции сигнала с выхода третьей Ф до тех пор, пока состояние функций *f* и *f**, их выходов и входа не повторится), два числа, отвечающих за конфигурацию и вероятность того, что единица будет отрицающим значением. Все необходимые переменные объявляются и инициализируются, то есть устанавливаются, внутри процедуры.

После этого программа входит в бесконечный цикл while(true), в котором происходит исполнение работы логического механизма, а именно проход

входного значения через функции и их изменения. На каждой итерации цикла состояния функций и входное значение записываются следующим элементом в массив истории состояний. Выход из цикла осуществляется, когда в массиве истории состояний обнаруживаются два одинаковых значения — это необходимо для избежания бесконечности работы программы в детерминированном случае.

Программа работает в трёх режимах, условно обозначенных: 01, 10 и 11. При этом первые три режима соответствуют циклическим процедурам (4-6), а четвертый режим не является циклическим.

1. 01 — режим по процедуре (4) – сначала входные данные проходят через функции *f*, а потом эти функции изменяются.
2. 10 — режим по процедуре (5) – сначала изменяются функции *f* (здесь необходимо изначально задавать значения на выходах функций *f*), а только потом через них проводятся входные данные.
3. 11 — режим по процедуре (6) – функции *f* изменяются после обработки данных каждой из них.

Процесс работы программы, симулирующей работу логического механизма, проходит циклически, где итерация цикла работы механизма — это шесть тактов его работы. Каждый такт — это отработка одного из шести функциональных блоков, представленных на схеме на рисунке 2.1. В каждом функциональном блоке «лежит число», ноль или единица, отвечающая за то, какая функция будет применена к вошедшему в него значению. Нулю и единице соответствуют функции повторения и отрицания, как показано в таблице 2.1, однако на каждой итерации работы механизма соответствие может быть установлено случайным образом (например, с вероятностью 0.5 ноль может стать отрицающим). Выходные значения функциональных блоков f_1, f_2, f_3, показанных на схеме (рис. 2.1), является содержимым блоков с обратными функциями, то есть блоков f_1^*, f_2^*, f_3^* (их такты работы на блок-

схеме ниже обозначены, как W) с той же схемы, выходные значения которых уже применяются к содержимому блоков $f_1,\ f_2,\ f_3$.

При запуске программа требует ввести:

1. Состояния функций $f_1,\ f_2,\ f_3$.
2. Состояния выходов функций, что определят обратные функции $f_1^*,\ f_2^*,\ f_3^*$.
3. Две цифры, идентифицирующие режим работы.
4. Значение, подаваемое на вход механизма.
5. Управляющий параметр, определяющий, будет ли работа механизма зациклена, подачей выхода третьей функции (Ф) на вход первой функции и вновь проходом до тех пор, пока состояние механизма не повторится.
6. Вероятность того, что данной итерации цикла работы механизма единица будет отрицающим значением.

Каждый такт работы логического механизма может быть, как изменением и передачей входного значения функциональными блоками (на блок-схеме ниже обозначается Q с индексом),

Значения, идентифицирующие применяемые функции *f* и обратные функции *f** (содержимое шести блоков) вместе с входным значением хранятся в массиве истории состояний системы. После одной итерации цикла последнее состояние функций, их выходов и выходное значение Ф, которое может быть подано на вход А первой функции, записывается в массив истории состояний. Во всех трёх режимах (01, 10 и 11) в случае повторения состояния механизма (A, B, f, Ф, $f_1^*,\ f_3^*,\ f_2^*$) программа выходит из цикла (это необходимо для избежания бесконечного повторения одних и тех же уже пройденных состояний). В программе заложена также возможность задавать вероятность того, что единица в выходе функции будет соответствовать отрицанию. Это

вводит недетерминированность в работу программы (соответствие чисел операциям меняется каждый проход цикла).

Стоит заметить, что логический механизм записан в процедуре, принимающей начальные условия A, B, f, Ф, f_1^*, f_3^*, f_2^* и управляющее параметры, потому возможно включение логического механизма в иные программные и вычислительные системы, что будет использовано в дальнейшем.

Начальные условия B, f, Ф, f_1^*, f_3^*, f_2^*, представленные семёркой чисел кратко будут обозначены следующим образом:

1. $I_1 = 000000$
2. $I_2 = 001001$
3. $I_3 = 101101$

2.3. Блок-схема программы механизма

Блок-схема программы представлена ниже на рисунке 2.2. Ниже приводится описание отдельных блоков программы, представленных на блок-схеме.

Блок «Начало». Этот блок означает, что программа запускается и просит у пользователя ввести следующие значения: входное значение (ноль или один), первое число номера режима работы (далее «конфигурация») (левая цифра), второе число конфигурации (правая цифра), вероятность того, что единица будет отрицающим значением, зацикливать ли программу (ноль — нет или один — да). В случае отказа от зацикливания, отработка функций и их изменение произойдёт всего один раз (пройдёт всего одна итерация цикла, что будет описано ниже).

Блок «Принять на вход значение». Этот блок означает, что механизм принимает на вход либо ноль, либо единицу.

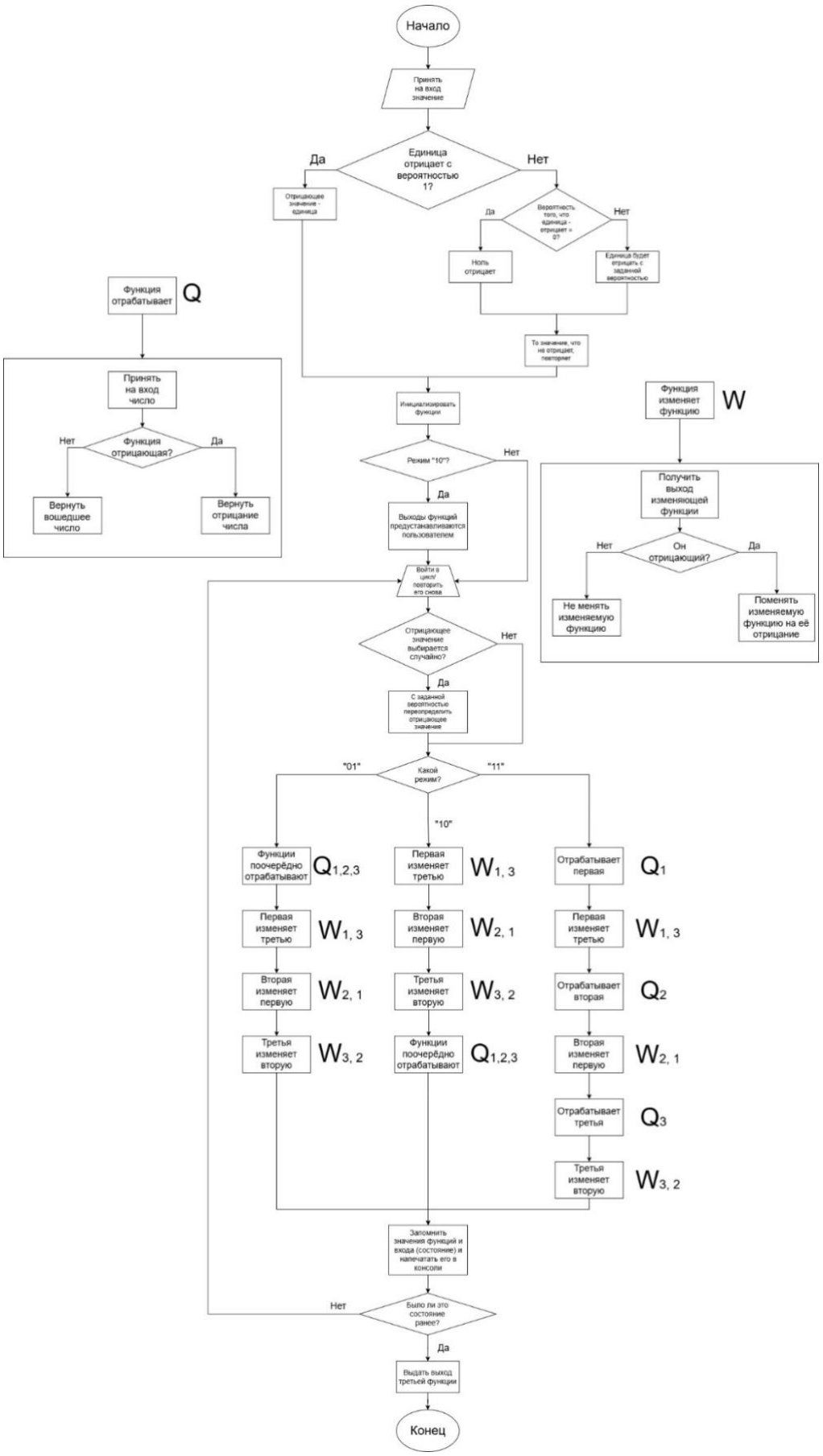

Рис. 2.2. Блок-схема логического механизма (в циклическом исполнении)

Блок «Единица отрицает с вероятностью 1? и ветви от этого блока». Этот блок означает, что программа просит ввести вероятность p того, что единице в соответствие будет поставлено отрицание. Если будет введена единичная (в данной реализации 100) вероятность, то единица станет отрицающим значением (единице будет соответствовать отрицание). В противном случае возможно два варианта: «вероятность того, что единица будет соответствовать отрицанию равна нулю» и «вероятность того, что единица будет соответствовать отрицанию не равна нулю». В первом случае ноль будет отрицанием, а во втором на каждой итерации цикла работы механизма (описан далее) единица будет становиться отрицающим значением с вероятностью, введённой пользователем (с вероятностью $1 - p$ отрицающим значением будет становиться ноль).

Блок «Инициализировать функции». Этот блок означает, что программа устанавливает значения трёх функций, равными либо нулю, либо единице. Эти значения берутся извне, то есть задаются либо пользователем, либо «средой» (в зависимости от реализации логического механизма).

Блок «Режим «10»? и ветви этого блока. Этот блок определяет, что в случае конфигурации, равной «10» (режим 10), изначальные значения выходов функций предустанавливаются пользователем так же, как и значения самих функций.

Блок «Войти в цикл/повторить его снова». Этот блок означает точку начала цикла и/или очередной его итерации. Всё, что последует ниже выполняется с этого места программы и повторяется до тех пор, пока не будет выполнено условие прерывания цикла, описанное в одном из нижних блоков.

Блок «Отрицающее значение выбирается случайно?» и ветви этого блока. Этот блок означает, что если вероятность того, что единица соответствует отрицанию, равна нулю или единице, то отрицанию соответствует либо ноль (если вероятность ноль), либо единица (если

вероятность единица). Если же вероятность не равна нулю или единице, то единица становится отрицающим значением с заданной ранее вероятностью.

Блок «Какая конфигурация?» Этот блок означает ветвление в зависимости от полученной конфигурации. Возможны четыре варианта:

1. «01» содержит в себе поочерёдное изменение функций ($Q_{1,2,3}$) и затем их поочерёдную отработку (W_1, W_2, W_3).
2. «10» содержит в себе сначала последовательно отрабатывающие функции (W_1, W_2, W_3), а затем последовательное изменение функций друг другом ($Q_{1,2,3}$).
3. «11» содержит в себе попеременную последовательную отработку и изменение функций ($W_1, Q_1, W_2, Q_2, W_3, Q_3$).

Блоки W_n и Q_n будут подробно рассмотрены далее.

Блок «Функция отрабатывает (Q_n)». Этот блок означает, что функция под номером $n \in \{1, 2, 3\}$ принимает на вход значение, а дальше возможно два варианта: если эта функция представлена отрицающим значением, то пришедшее на вход значение отрицается (ноль превращается в единицу и передаётся дальше, а единица — в ноль и передаётся дальше), если же функция представлена повторяющим значением, то вход просто передаётся дальше без изменений. Индекс обозначает, какая функция отрабатывает, то есть обрабатывает/видоизменяет пришедшее в неё значение. Индекс *n* обозначает номер функции, к которой относится блок (если индексы перечисляются, то это означает последовательную отработку функций). Декомпозиция общего вида блока (*Q* без индекса) отработки функции расположена на схеме (рис. 2.2) слева вверху.

Блок «Функция изменяет функцию ($W_{n,m}$)». Этот блок означает, что функция под номером $n \in \{1, 2, 3\}$ изменяет значением своего выхода функцию под номером $m \in \{1, 2, 3\} \backslash \{n\}$, а дальше возможно два варианта:

если выход изменяющей функции представлен отрицающим значением, то значение изменяемой функции отрицается (ноль становится единицей, а единица — нулём), если же выход изменяющей функции представлен повторяющим значением, то значение изменяемой функции остаётся без изменений. Индекс *n* обозначает, какая функция изменяет другую функцию, чей индекс — *m*. Декомпозиция общего вида блока (*W* без индекса) изменения функции расположена на схеме справа вверху.

Блок «Запомнить значение функций и входа (состояние) и напечатать его в консоли». Этот блок означает, что результат, конечное состояние системы (A, B, f, Φ, f_1^*, f_3^*, f_2^*) на каждой итерации цикла (после прохода данных и изменений функций), запоминается в массиве, хранящем семёрки чисел: входное значение, три состояния функций и три выхода функций (A, B, f, Φ, f_1^*, f_3^*, f_2^*).

Блок «Было ли это состояние ранее?» и ветви этого блока. Этот блок означает, что, если в массиве состояний встретятся два одинаковых списка состояний, то есть два идентичных массива с семью числами, описывающими состояние системы, то цикл прервётся, иначе будет проделана следующая итерация. Это нужно для избегания бесконечных циклов.

Блок «Выдать выход третьей функции». Этот блок означает, что конечным результатом работы логического механизма (в тот момент, когда он повторит одно из пройденных состояний и цикл прервётся) является выходное значение третьей функции.

Вывод результатов работы программы осуществляется в консоль в виде числовой последовательности A, B, f, Φ, f_1^*, f_3^*, f_2^* , полностью описывающей состояние системы.

2.4. Демонстрация результатов работы программы механизма

Динамика работы реализованной программы механизма в трёх режимах работы («01», «10», «11»), объяснённые ранее в разделе 2.2, может быть

показана в виде последовательности потактовых изменений массива состояний механизма, характеризующихся списком из семи значений 0 или 1. Соответствующие таблицы состояний удобно представить графически, обозначив единичные значения черным цветом, а нулевые – белым.

На рисунке 2.3 представлены цепочки неповторяющихся состояний трёх режимов при входе ноль. Цепочки неповторяющихся состояний такого рода наиболее часто встречаются при работе механизма.

Рис. 2.3. Сравнение трёх режимов работы механизма при входном сигнале 0 и начальном состоянии I_2 в детерминированном случае

На рисунке 2.4 можно видеть работу механизма при тех же начальных условиях, но при входном сигнале «единица». В результате длина цепочек неповторяющихся состояний осталась прежней. Эксперименты демонстрируют, что ярких изменений в длине цепочек не происходит при изменении входного сигнала.

На рисунке 2.5 представлены те же три режима работы при входном сигнале «ноль», но в стохастическом случае с вероятностью отрицающей единицы, равной 0.5. Полученный результат демонстрирует, что цепочки неповторяющихся состояний могут быть, как очень короткими (так получилось в режиме «01»), так и достаточно длинными (в режимах «10» и «11»). Это можно объяснить тем, что поведение системы становится хаотическим и есть возможность чуть дольше избегать повторений.

Рис. 2.4. Сравнение трёх режимов работы механизма при входном сигнале 1 и начальном состоянии I_2 в детерминированном случае

Рис. 2.5. Сравнение трёх режимов работы механизма при входном сигнале 0 и начальном состоянии I_2 в стохастическом случае с вероятностью отрицающей единицы 0.5

На рисунке 2.6 можно видеть то же, что и на рисунке 2.5, но с входным сигналом 1. Цепочка неповторяющихся состояний в этот раз для режима «01» оказалась длиннее.

В детерминированном случае единица на входе позволяет системе пройти больше итераций, избегая повторений состояния («01» при входе 1 прошла на 1 итерацию больше, чем при входе 0). Стохастические же аналоги

проходят до повторения состояния гораздо больше итераций (за исключением режима 01 при входе ноль).

№	Режим: 01, состояние: I_2							Режим: 10, состояние: I_2							Режим: 11, состояние: I_2						
	A	B	f^*_1	f	f^*_2	Ф	f^*_3	A	B	f^*_1	f	f^*_2	Ф	f^*_3	A	B	f^*_1	f	f^*_2	Ф	f^*_3
1	■	■	■		■			■		■		■	■		■				■	■	■
2		■		■	■	■				■	■	■	■		■	■	■		■	■	
3		■	■				■	■	■								■		■	■	
4	■		■				■		■	■		■				■	■	■		■	
5	■			■	■	■		■	■	■				■						■	
6			■	■	■	■	■	■				■				■	■	■		■	
7	■	■								■	■	■	■								
8		■		■	■	■															

Рис. 2.6. Сравнение трёх режимов работы механизма при входном сигнале 1 и начальном состоянии I_2 в стохастическом случае с вероятностью отрицающей единицы 0.5

В среднем детерминированные системы перед повторением уже пройденного состояния успевают выполнить 8-9 итераций, не повторив пройденное состояние, а стохастические (с вероятностью того, что единица соответствует отрицанию, 0.5), если повезёт, 9-10 итераций. Случаи с отрицающим нулём, как показывают эксперименты, не так интересны, потому сюда не включены.

Наиболее интересно то, что в стохастических системах (вероятность отрицающей единицы равна 0.5) иногда возникают цепочки длиной в 11-15 неповторяющихся итераций, они приведены ниже на рисунке 2.7.

Из проведённых экспериментов можно сделать вывод, что наиболее длинные циклы у систем варианта «10» с вероятностью 0.5 и единицей на входе — они единственные, кому удалось преодолеть девять итераций и не повторить одно из уже пройденных состояний.

Полученные таблицы напоминают результат работы одномерного темпорального клеточного автомата, однако в отличие от классических одномерных клеточных автоматов, где состояние каждой клетки определяется

суммой состояний её соседей, рассмотренный логический механизм представляет собой последовательное изменений ячеек справа налево при проходе «инициирующего значения» слева направо, повторяющееся циклически.

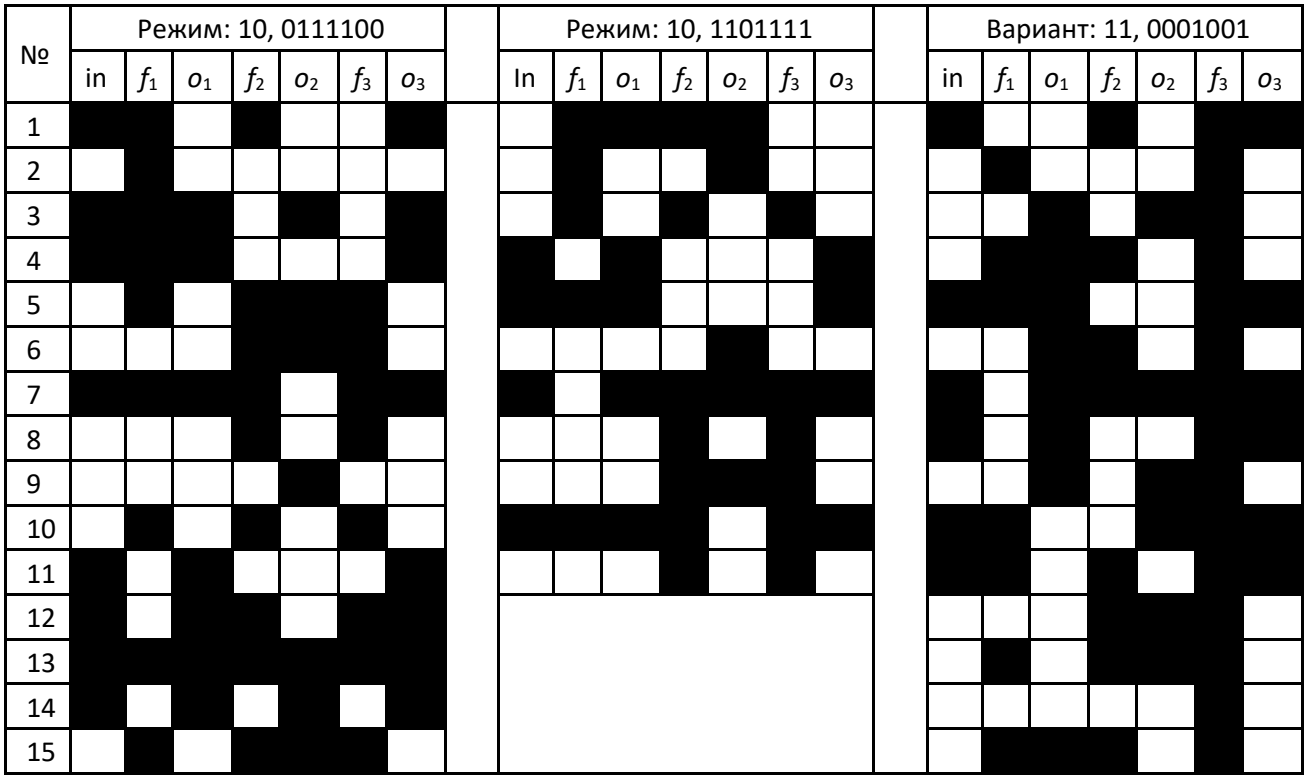

Рис. 2.7. Представление трёх наиболее примечательных длинных цепочек неповторяющихся состояний, возникающих в режимах 10 и 11

Полученная система может быть использована для получения ограниченных последовательностей псевдослучайных чисел, что, потенциально, могут иметь нетривиальное распределение при удлинении цепочек итераций (Gravner, Liu, 2022).

В целом система очень чувствительна к начальным условиям — малейшее их изменение может привести к сильным изменения в поведении, в частности резком увеличению количества итераций без повторений пройденных состояний.

Следующая глава посвящена вопросу включения полученного логического механизма в машину Поста.

Глава 3. Разработка вычислительной системы с механизмом самоопределения бинарных номинальных операций

3.1. Общее представление о вычислителях

Разработанный логический механизм позволяет ввести самоопределение операций в вычислительную систему, в которую он будет включен. Все известные человечеству компьютеры, вычислители, разделяют программу и данные, с которыми она работает. Сама структура вычислителя при этом остаётся всегда неизменной. Даже в нейронных сетях единственное, что меняется, — это веса связей между нейронами, а не сама архитектура сети (Системы искусственного интеллекта, 2023). Включение логического механизма в вычислительную систему позволит задавать управляющие параметры и изначальные состояния механизма в данных, с которыми она будет работать. Тогда при изменении данных будет меняться и её, вычислительной системы, поведение, операции, таким образом данные и вычислительное устройство будут неразрывно связаны. Определяя поведение системы через программу, вызывающую те или иные её действия, можно тем самым связать не только данные и устройство, но и программу. Тем самым программа, вычислитель и данные образуют неразрывное целое, чего ранее не наблюдалось в истории вычислительной техники.

Перед тем, как включить разработанный логический механизм, в машину Поста, были рассмотрены наиболее интересные случаи абстрактных вычислителей, в том числе и превосходящих машину Тьюринга.

Концепция абстрактных вычислителей появилась не для решения проблем моделирования (M,R) систем, рассмотренных ранее, а для решения иных вопросов, таких, как проблема остановки, описанная Тьюрингом (Хокинг, 2022). Проблема остановки заключается в вопросе: «Возможно ли по описанию алгоритма и входным данным определить, остановится ли исполнение алгоритма?» В машине Тьюринга получить ответ на этот вопрос невозможно (Хокинг, 2022). Потому были предложены концепции иных

вычислительных устройств, способных выполнять гипервычисления, невычислимые в машине Тьюринга.

Одним из таких устройств является машина Зенона (Potgieter, 2006). Каждый её n-й шаг требует 2^{-n} времени, то есть она способна за конечное время выполнить бесконечное число итераций. Очевидно, что создание такой машины в реальности вряд ли возможно, потому она не подходит для реализации включения логического механизма в неё.

Другой идеей является O-машина Тьюринга, представляющая собой машину Тьюринга с модулем «Оракула», внутреннее устройство которого неизвестно (Copeland, Proudfoot, 1999). Суть таких «вычислений с Оракулом» заключается в том, что за одно обращение «Оракул» угадывает решение проблемы остановки для конкретного алгоритма, после чего машине остаётся только проверить его. По ясным причинам этот способ тоже не подходит для включения в него логического механизма.

Существует также идея, предложенная Хавой Зингельман, о том, что бесконечно эволюционирующие рекуррентные нейронные сети способны проводить гипервычисления, однако эта вычислительная система нереализуема на современном уровне развития технологий, потому она не подходит для включения в неё логического механизма (Rodrigues et al., 2001).

Таким образом, все существующие на данный момент вычислительные модели, превосходящие машину Тьюринга, так или иначе, сводятся к временным бесконечностям. В случае «вычислений с Оракулом» используется модуль, находящий правильный ответ на вопрос об остановке за одно обращение, то есть бесконечно малый промежуток времени. В случае же иных альтернативных моделей требуется провести бесконечное количество итераций, и желательно за конечное время, чтобы результат имел смысл, то есть модели гипервычислений требуют, с точки зрения машины Тьюринга, алгоритма, выполняющегося за бесконечное число итераций. Также эти

системы невозможно реализовать на современных компьютерах, в виду того, что они представляют собой системы, отличные от машины Тьюринга. Потому смысл имеет включать логический механизм только в вычислительные системы, эквивалентные машине Тьюринга.

3.2. Машина Поста

Существует несколько вычислительных систем, пригодных для включения в них логического механизма. Машина Тьюринга, являющаяся «эталоном» в вопросе проверки вычислимости, слишком сложна в понимании её настройки и её работы, ввиду большого количества правил, описанных Аланом Тьюрингом (Хокинг, 2022). Её вариациями являются машина Минского, имеющая несколько лент (Минский, 1971), что не уменьшает её сложности. Существуют также нормальные алгоритмы Маркова (Марков, 1954), однако трудно представить, как поместить в них логический механизм.

Помимо различных модификаций машины Тьюринга, алгоритмов Маркова и гипервычислительных систем, рассмотренных в предыдущем разделе, существует вычислительное устройство, называющееся машиной Поста, предложенной Эмилем Постом в 1936 году. Эта машина, как и машина Тьюринга, имеет бесконечную ленту и считыватель, способный ходить по ней. Это абстрактное вычислительное устройство управляется программой, написанной на языке программирования, состоящим из операторов, приведённых в таблице 3.1 (Успенский, 1988).

Ввиду более простого описания, эта машина более проста для понимания и программирования, потому для экспериментов с логическим механизмом в вычислительных системах была выбрана именно она.

Таблица 3.1. Операторы языка программирования машины Поста

Оператор языка машины Поста	**Его действие**
V j	Поставить метку, перейти к j-й строке программы.
X j	Стереть метку, перейти к j-й строке.
← j	Сдвинуться влево, перейти к j-й строке.
→ j	Сдвинуться вправо, перейти к j-й строке.
? j1; j2	Если в ячейке нет метки, то перейти к j1-й строке программы, иначе перейти к j2-й строке
!	Конец программы («стоп», останов).

3.3 Программирование машины Поста

В машину Поста можно включить разработанный логический механизм только одним путём — заменить им её считыватель. Тогда такая модифицированная машина Поста будет описываться следующим образом: логический механизм с самоопределением операций (далее просто «считыватель») может видоизменять любую ячейку ленты, что содержит в себе нули и единицы. Логический механизм может, ссылаясь на ячейки по их номерам (адресам), брать из них и записывать в них значения входа, функций, их выходных значений и выхода. Параметры, определяющие режим работы считывателя, хранятся в двух ячейках. Значение, определяющее будет ли механизм замыкать процесс своей работы также хранится в одной из ячеек ленты. Все эти значения принимаются механизмом для инициализации его параметров из ячеек с указанными номерами, что позволяет при определённой настройке держать все параметры работы считывателя в одной ячейке.

Язык программирования модифицированной машины Поста имеет следующий синтаксис на каждой строке:

оператор операнд1 операнд2.

Оператор и операнды отделены друг от друга одним пробелом. Если в начале строки стоит пробел, то строка игнорируется. Язык чувствителен к регистру.

Операнды — это адреса/номера ячеек на ленте, то есть числа от нуля (0) до двухсот пятидесяти пяти (255). В случае, если число будет указано минус один (-1), то значением операнда полагается значение, сохранённое в отдельной ячейке (не на ленте), которое равно заранее определённому адресу ячейки (по умолчанию это значение равно нулю, далее использование этого операнда будет называться «взятием/записью значения по адресу»). Если же оператором являются block, label, do и to, то единственным операндом является названием блока или метки, которое может быть представлено строкой произвольной длины, не содержащей в себе пробела и управляющих символов.

Машина исполняет программу построчно, выполняя действие, указанное на каждой строчке. При этом сами программы безусловно цикличны, то есть, когда будет выполнена последняя строчка, программа начнёт выполнятся с нулевой строчки. И так до тех пор, пока по каким-то причинам не будет исполнена строчка с оператором конца программы (owari, яп. 終わり — конец). Возможно так же при запуске программы начинать исполнение с указанной строчки через использование оператора hajimaru (яп. 始まる — начало). В случае, если операндом является адрес ячейки, то минус единица в данном контексте означает взятие номера ячейки из переменной, в которой хранится адрес, на который указывает считыватель (рассмотренное ранее «взятие по адресу»). Ещё одной отличительной особенностью языка

программирования модифицированной машины Поста является то, что возможен переход по меткам и исполнение блоков, объявленных ниже в программе. Это возможно, благодаря двум чтениям программы (на первом размечаются метки и блоки).

Полный список операторов языка программирования машины представлен в таблице, приведённой в приложении Б (каждый оператор сопровождён примером применения и описанием его действия). Полный код полученного вычислителя приводится в приложении В.

Программно машина реализована на языке C++ (компилятор: x86_64-win32-seh-rev0, Built by MinGW-Builds project 14.2.0, стандарт языка: C++17). Код был написан в редакторе кода Notepad++ версии 8.7.2 (x64). Программа, реализующая машину, начинает своё исполнение с функции main(), в которой происходит первое чтение файла с кодом на языке машины, в котором определяется с какой строки начинать исполнение (оператор начала hajimaru) и присутствует ли оператор остановки owari (если его нет, программа на языке машины гарантированно никогда не остановится, как как при достижении последней строчки кода исполнение начинается с нулевой и так до бесконечности). После первого чтения файл начинает считываться повторно. В этот раз каждая строчка разбивается на оператор и два операнда, после чего отправляется в функцию interpret_line(), которая в зависимости от операнда либо выполняет прописанное действие (оно может быть вынесено в отдельную функцию в программе), либо возвращает ненулевой код ошибки, который «подхватывается» в функции main(), и машина останавливается с указанием причины остановки. Принцип работы программы иллюстрирует блок-схема, представленная ниже (рисунок 3.1).

Ниже представлено краткое описание блоков программы, интерпретирующей файл с кодом модифицированной машины Поста.

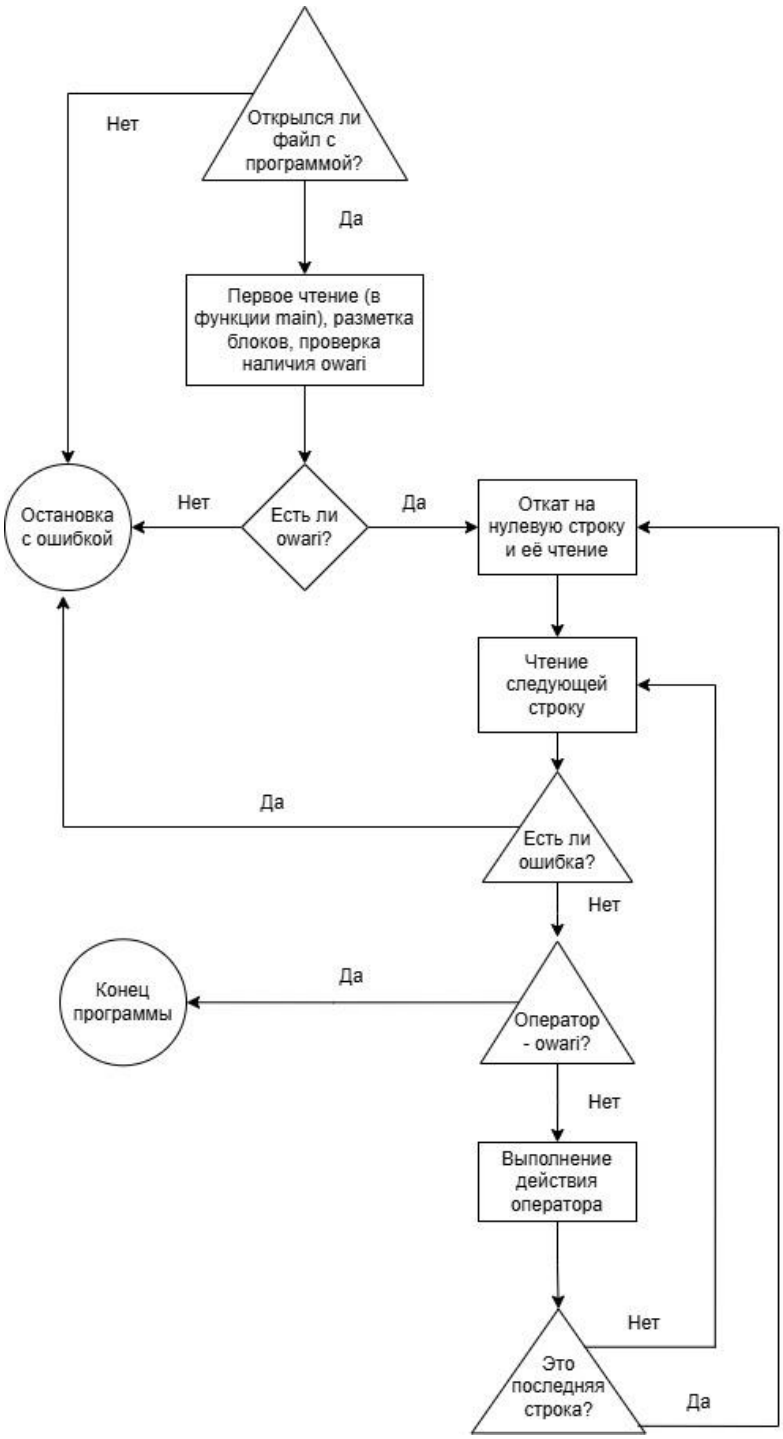

Рис. 3.1. Блок-схема программы, интерпретирующей файл с кодом модифицированной машины Поста

Блок «Открылся ли файл с программой?» и его ветви. Этот блок проверяется, существует ли файл с кодом, который будет интерпретироваться машиной. В случае его отсутствия программа, реализующая машину, выдаст ошибку.

Блок «Первое чтение (в функции main), разметка блоков, проверка наличия owari». Этот блок отвечает за первое чтение программы, на котором размечаются блоки, метки и проверяется наличие оператора остановки.

Блок «Есть ли owari?» и его ветви. Этот блок означает, что если оператора остановки нет, программа не запустится, а если есть, то её считывание начнётся с нулевой строки.

Блок «Откат на нулевую строку» и её чтение. Этот блок отвечает за начало второго чтения программы, в процесс которого программа будет исполняться построчно.

Блок «Чтение следующей строки». Этот блок переход к следующей строке программы машины.

Блок «Оператор – owari?» и его ветви. Этот блок означает, что если мы встретили оператор остановки, то программа закончится (блок **Конец программы**).

Блок «Выполнение действия оператора». Этот блок означает исполнение строки программы.

Блок «Это последняя строка?» и его ветви. Этот блок означает, что если мы достигли последней строки программы, то чтение начинается заново, с нулевой строки. Если это не последняя строка программы, то выполняется переход на следующую строку, то есть переход к блоку **«Чтение следующей строки»**.

Новый считыватель в этой модифицированной машине Поста представлен в качестве оператора henkamono, который работает с двумя числами, номерами (адресами) ячеек. Его управляющие параметры, такие как состояния функции, их выходов, и значение, которое подаётся на вход представляются в качестве параметров, определяемых, использованием соответствующих операторов языка программирования машины Поста.

3.4 Особенности языка программирования модифицированной машины Поста

Разработанный язык программирования модифицированной машины Поста имеет много особенностей и нюансов, о которых подробно написано ниже. Адрес (оператор: addr, операнд: -1) — номер ячейки, на которую указывает считыватель (変 化 物). При применении считывателя адрес изменяется на номер ячейки, в которую считыватель запишет обработанные данные. Перемещение (動く), являясь синтаксическим «сахаром», реализует применение считывателя к двум ячейкам с состояниями функций 000 (каждая цифра с отдельной ячейки) и вероятностью 100 и потому также изменяет значение адреса на номер ячейки, переданной первым операндом, однако значение адреса не меняет.

Факты, требующие особого внимания при использовании языка:

- Оператор goto позволяет перейти на любую метку, даже если строка с её объявлением которой находится ниже в программе. Возможно описывать вложенные блоки. Например, следующая программа выведет в консоль ноль:

```
block q2

block q1

kaku 0 0

break q1

break q2

do q1

owari
```

Однако такие конструкции лучше избегать, чтобы не усложнять код. В приведённом примере при вызове q2 вместо q1 программа просто завершится, ничего не выведя в консоль, так как вложенный блок не был вызван.

Следующая программа выведет ноль дважды:

```
block q2

block q1

kaku 0 0

break q1

do q1

break q2

do q2

owari
```

Это связано с технической реализацией логики блоков, которую можно понять, просмотрев исходный, код, поставляемый вместе с инструкцией и exe-файлом интерпретатора. Лучше не вызывать блоки, которых нет и не использовать block и break не по назначению и поодиночке.

- Поведение интерпретатора оказалось весьма сложным именно в связи с большой вариативностью считывателя (変化物). Общее количество различных циклов считывателя при разных начальных условиях составило 1024 штук при наличии числа шагов в каждом цикле от 2 до 9. Среднее число шагов в циклах 4.97, медианное значение 5. И это без учёта вероятности, которая позволяет перескакивать между разными цепочками состояний, позволяя порождать, сколь угодно длинные

цепочки шагов — главное избегать повторений состояний функций, их выходов и входного значения в них.

- Необходимо следить за значениями операндов, так как не везде реализована проверка на допустимость введённых/указанных значений. В идеале в коде должны встречаться только минус единица и числа от нуля до 256. Следует учитывать, что в представленной реализации на ленте всего 256 ячеек!

- Если в программе встретится оператор, не указанный в таблице операторов, приведённой выше, то строчка будет проигнорирована с выводом в консоль сообщения о неизвестном операторе с указанием номера строки, на которой он встретился.

- Очень важно помнить, что **по умолчанию все настройки, ссылки и адрес инициализированы, как нули** — программисту придётся в каждой программе указывать, откуда берутся значения для настроек считывателя!

- Первым операндом всегда должна быть какая-нибудь переменная (включая массивы).

- Не следует пытаться сложить число со строкой и/или провести другие подобные абсурдные операции — необходимо следить за типами операндов!

- Не рекомендуется называть переменные числами!

Язык модифицированной машины Поста, в которой её считыватель заменён разработанным логическим механизмом не подходит для создания программ, используемых в коммерческой или промышленной сфере, однако он подходит для проведения вычислительных экспериментов и проверки тех или иных концепций. Следующая глава посвящена экспериментам, связанным с

модифицированной машиной Поста и доказательству её полноты по Тьюрингу, там же будут рассмотрены особенности работы с её считывателем.

Глава 4. Программирование в вычислительной системе с механизмом самоопределения бинарных номинальных операций и доказательство её полноты по Тьюрингу

4.1. Доказательство полноты по Тьюрингу разрабатываемой вычислительной системы

Перед проведением экспериментов с полученной системой имеет смысл доказать её полноту по Тьюрингу, то есть возможность запрограммировать в ней алгоритм, реализующий любую вычислимую функцию (Хокинг, 2022). Доказательство этого основывается на факте эквивалентности уже рассмотренной ранее машины Поста и машины Тьюринга (Успенский, 1988). Возможность выполнить операторами полученной вычислительной системы то, что выполняется операторами машины Поста, станет доказательством их, этим машин, эквивалентности, а следовательно, и эквивалентности созданной вычислительной системы машине Тьюринга, что и докажет её полноту по Тьюрингу. Полный список операторов вычислительной системы приводится в приложении Б. В таблице 4.1 установлено соответствие между операторами машины Поста и её модифицированного аналога, доказывающее эквивалентность этих машин.

Операторы hitotsu и zero отвечают за постановку на ленту единицы и нуля соответственно. Без этих операторов вычислительная система работать не будет, так как они отвечают за создание изначальных условий на ленте. Возможно использовать и логический механизм для выполнения тех же операций. Для этого достаточно обеспечить такие его настройки (управляющие параметры и начальное состояние), чтобы проходящее через него значение не изменялось (один из вариантов таких настроек: все нули). Иначе говоря, разработанная вычислительная система избыточна по отношению к машине Тьюринга.

Таблица 4.1. Переход от машины Поста к операторам вычислительной системы ($j \wedge j1 \wedge j2 \in N: \forall n \in \mathbb{N} + \{0\}$) без использования логического механизма

Машина Поста	**Выч. система**	**Действие**
V j	hitotsu -1 goto j	Поставить метку, перейти к j-й строке программы.
X j	zero -1 goto j	Стереть метку, перейти к j-й строке.
← j	<- goto j	Сдвинуться влево, перейти к j-й строке.
→ j	->\ goto j	Сдвинуться вправо, перейти к j-й строке.
? j1; j2	bunkiten -1 0 goto j1 goto j2	Если в ячейке нет метки, то перейти к j1-й строке программы, иначе перейти к j2-й строке
!	owari	Конец программы («стоп», останов).

Стоит заметить необычную особенность, возникающую при настройке начального состояния логического механизма при включении в машину Поста. В главе два, в разделе 2.2, были описаны три режима, номера которых для удобства были приведены в двоичной системе. При автономной работе

логического механизма задание номера режима работы, его идентификатора, производится либо человеком, либо специально разработанной программной средой. В случае включения логического механизма в машину Поста в качестве считывателя возможно возникновение трудностей с определением режима работы. Если все управляющие параметры и начальное состояние будет браться из ячеек ленты, то возможна такая ситуация, что все ячейки, из которых берутся значения, окажутся содержащими нули. В таком случае может оказаться, что идентификатор режима будет равен «00». Чтобы избежать сбоя в работе программы, в рассматриваемой программной реализации усовершенствованной машины Поста при значении «00» логический механизм, выступающий в роли её считывателя, не будет проводить никаких операций с входным значением, сразу копируя его в целевую ячейку.

4.2. Эксперименты с вычислительной системой

Написание программ для полученного вычислителя сопряжено с рядом трудностей:

1. **Низкая выразительность языка.** Сложные и многоуровневые абстракции, несмотря на тьюринговскую полноту языка потребуют несколько тысяч строк кода, состоящих из одного оператора. Такую программу написать обычному человеку не под силу.
2. **Непредсказуемость или сложность прогнозирования поведения программы.** Принято, что программы пишутся ради достижения какой-нибудь цели. В рассматриваемой системе случайность и сложные петли внутренних причинно-следственных отношений не позволяют человеку написать сложную целенаправленную программу.
3. **Бесконечные циклы.** Этот пункт вытекает из предыдущего — условный оператор сравнения значений двух ячеек является единственной возможностью обеспечить нелинейность поведения

программы. Вместе с предыдущими пунктами мы получаем очень маленькие возможности для управления поведением. Может оказаться так, что условие остановки никогда не выполнится, и программа будет работать в бесконечном цикле, порождая странные структуры на ленте модифицированной машины Поста.

В связи с этим можно допустить, что один эксперимент, задействующий все, или почти все, возможности машины, окажется показательным. Критерии «показательности» для этого эксперимента следующие:

1. Задействовать всю ленту, а не только её часть.
2. Использовать считыватель.
3. Избежать бесконечного цикла.

Следствием из перечисленных критериев является следующая постановка задачи для эксперимента:

1. Установить в первых ячейках ленты начальные условия и значения управляющих параметров.
2. Использовать считыватель для перемещения значения из нулевой ячейки в следующую ячейку после ячеек начальных условий и управляющих параметров.
3. Сдвинуть указатели ячеек условий и параметров на единицу и переместить в следующую за полученными ячейками значение из первой ячейки.
4. Повторять эту процедуру до тех пор, пока значение нулевой и 256-й ячейки не совпадёт.

Этот алгоритм реализует программа, построчно представления в таблице 4.1, в которой объясняется каждый этап работы программы.

Таблица 4.1. Программа с проходом ленты и её объяснение

Часть программы	**Объяснение части программы**
addr 0	Установить значение указателя (адреса), равным нулю (первая ячейка ленты)
prob 100	Вероятность отрицающей единицы равна 1
hitotsu 0 hitotsu 1 hitotsu 2 hitotsu 3	В первых четырёх ячейках ленты установлены единицы.
label cycle	Метка начала цикла
-> cycle -1	Сдвиг указателя на 1 вправо и взятие из ячейки с соответствующим номером значения для управляющего параметра зацикленности работы логического механизм (считывателя). В данном случае то, что указатель равен единице, указывает на вторую ячейку ленты, в которой лежит единица.
-> conf1 -1	Сдвиг указателя на 1 вправо и взятие первой цифры значения режима работы логического механизма.
-> conf2 -1	То же для второй цифры.
-> f1 -1 -> o1 -1	Сдвиг указателя на 1 вправо, установление значения f_1, сдвиг указателя вправо и установление обратной f_1^*.
-> f2 -1	То же самое, но для f_2 и её обратной.

Часть программы	Объяснение части программы
-> o2 -1	
-> f3 -1 ->o3 -1 ->	То же самое, но для f_3 и её обратной.
henkamono -1 0	Использование считывателя. Взятие значения из нулевой ячейки ленты (первая), пропуск её через считыватель и помещение полученного результата в ячейку, на которую указывает указатель (в первый раз это десятая ячейка, то есть под номером девять).
kaku 0 255	Вывод на экран полученного состояния всей ленты.
bunkiten 0 255	Условный оператор. Если нулевая ячейка равна ячейке 255 (256-я), то выполниться следующая строчка, иначе последующая.
owari	Конец работы программы.
goto cycle	Переход на метку начала цикла. Это и обеспечивает зацикленность программы.

4.2.1. Первый эксперимент

Результат работы программы представлен на снимке экрана (рисунок 4.1).

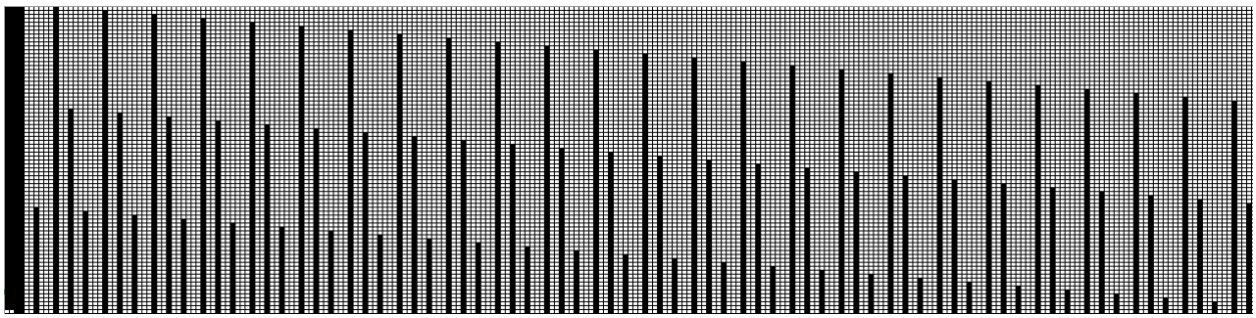

Рис. 4.1. 77 построчно выведенных состояний ленты для случая задания единиц в первых трех ячейках

Мы можем видеть то, как количество единиц на ленте, с каждой строчкой нового состояния, возрастает до тех пор, пока в ячейке 255 не окажется единица (это ограничение нужно было для того, чтобы сохранить уверенность в конечности времени работы программы). Полученный результат распечатывания состояния ленты на каждой итерации цикла работы программы можно визуально описать, как тройки пиков из единиц (чёрные клетки), причём каждая правая тройка ниже, стоящей рядом с ней левой. Такое убывание высоты можно исследовать. Всего было пройдено 77 состояний ленты до того, как условие остановки сработало.

4.2.2. Второй эксперимент

Попробуем постепенно убирать единицы из ячеек 1, 2 и 3. Результат без единицы в ячейке 3 представлен на рисунке 4.2.

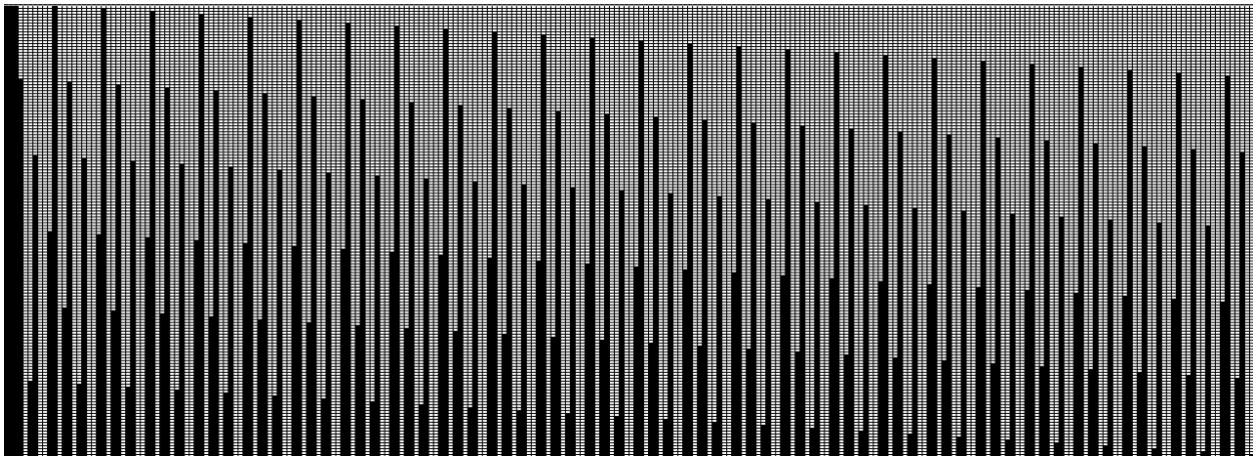

Рис. 4.2. Вывод состояний ленты, но в начале единицы в ячейках 0, 1 и 2

При единицах в ячейках 0, 1 и 2, и с нулями в остальных ячейках ленты в качестве начального условия, мы получаем результат, в котором программа

изменила ленту 153 раза до тех пор, пока не выполнилось условие остановки. Осмотр результата не выявил никаких значимых изменений.

4.2.3. Третий эксперимент

Отказ постановки единицы не только в ячейке 3, но и в ячейке 2 дал иной результат, приведённый на рисунке 4.3.

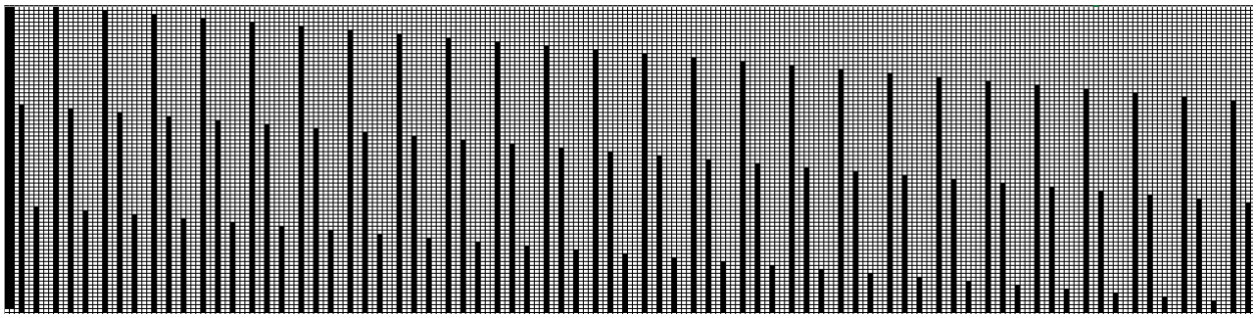

Рис. 4.3. Вывод состояний ленты, но в начале единицы в ячейках 0 и 1

Можно заметить, что результаты, представленные на рисунках 4.1 и 4.3 отличаются — после отказа от единицы в ячейке 2 (третья ячейка) можно заметить, что «стройный ряд единиц первых четырёх ячеек» с рисунка 4.1 рассыпался — теперь можно видеть, что третья ячейка (ячейка 2 на рисунке 4.3) всегда нулевая. Пройдено было 77 состояний ленты до остановки.

4.2.4. Четвёртый эксперимент

Что же будет, если единица будет стоять только в нулевой (первой ячейке)? Результат представлен на рисунке 4.4.

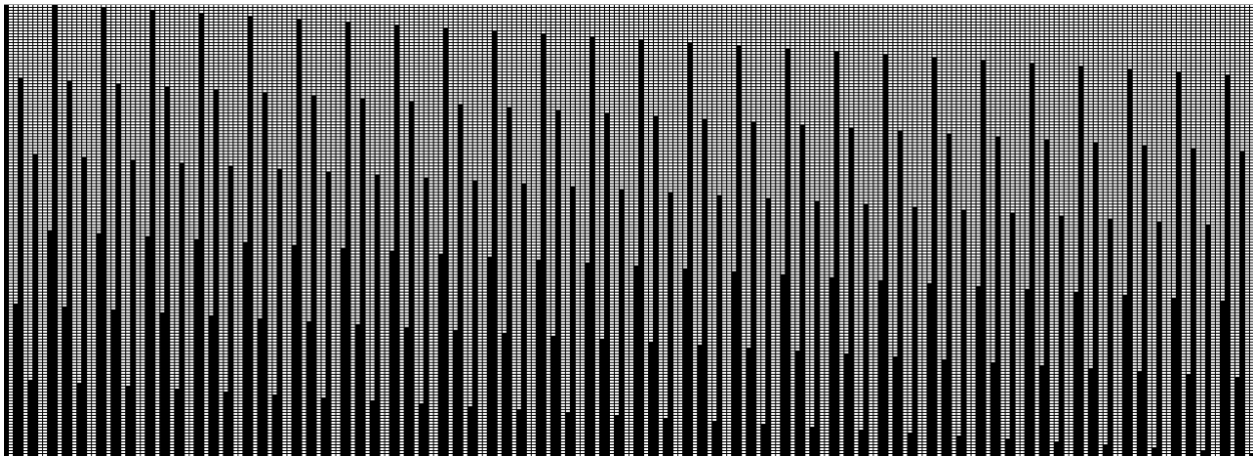

Рис. 4.4. Вывод состояний ленты, когда только в нулевой ячейке единица

В этот раз 153 пройденных состояния, что является двухкратным увеличением количества пройденных состояний ленты до выполнения условия остановки. Картина с «убывающими тройками пиков» всё так же отчётливо видна.

Общим итогом для проведённых экспериментов может являться следующее заключение: полученная вычислительная система, несмотря на свою трудность в применении именно к самим вычислениям, позволяет организовывать структуры, в которых можно выявлять закономерности.

Эксперименты с вероятностями не приведены ввиду трудности интерпретации полученных результатов.

4.2.5. Выводы из экспериментов

Из проведённых экспериментов можно сделать следующие выводы:

1. **Разработанная вычислительная система сложна для целенаправленного программирования.** Чтобы реализовать описанные результаты с лентой, пришлось составить длинный алгоритм, причём заранее не было очевидно, остановится он или нет. Полученные результаты сложно где-либо применить (если это вообще возможно).
2. **Система позволяет генерировать числовые последовательности со сложными внутренними отношениями.** Одной из ключевых особенностей разработанного вычислителя является то, что получаемые в процессе его работы числовые последовательности могут описываться не только статичным алгоритмом, генерирующим числа из некоего начального условия, но и самими этими числами, определяющими вид алгоритма их генерации. Порождаемые данные оказывают влияние на порождающий их процесс, что создаёт в процессе работы вычислителя замкнутые причинные отношения, петли обратной связи. «Программа и данные становятся единым

целым» (в рассмотренном случае поведение оператора henkamono было не постоянным в процессе работы программы).

3. **Результаты работы вычислителя могут открывать ранее неизвестные закономерности, связанные с числами.** Несмотря на ограниченность вычислимостью по Тьюрингу, разработанный вычислитель потенциально способен помочь открыть ранее неизвестные закономерности с использованием изменения внутренних отношений в числовых последовательностях.

Общим итогом проведённых экспериментов можно назвать то, что данный вычислитель не подходит для вычислений в привычном понимании этого слова, однако его использование может помочь в иных областях математики. Примером такой области может быть криптография. Возможно переводить текстовую информацию в двоичный вид (например, перевести в кодировку UTF-8, а потом полученные коды символов в двоичную систему счисления), после чего рассмотреть полученную строку, как ленту (ограничение на 256 тогда снимается) и применить вычислитель к ней с уже ранее рассмотренной (или подобной ей) программой. Полученную строку можно передавать по открытым каналам связи. Чем длиннее послание, тем больше состояний строки будет пройдено алгоритмом, следовательно, тем сложнее будет подобрать подходящее состояние, так как применение к полученному шифротекст зашифровывающей программы вернёт тот же шифротекст, являющийся состоянием, с которого она начнёт и на которое же натолкнётся при работе. Расшифровкой может заниматься программа, находящаяся у адресата сообщения. Причём она же может служить шифрующей программой для другого текста. Таким образом, при соответствующих доработках возможно использовать полученный вычислитель в качестве нового алгоритма асимметричного шифрования.

Заключение

Цель, представить двузначную логическую модель в виде имитации (M,R) системы и разработать на этой основе программу-эмулятор процессора, концептуально основанного на машине Поста, достигнута.

Поставленные задачи решены следующим образом.

1. Представлена и описана модель механизма с самоопределением бинарных номинальных операций.
2. Разработаны алгоритмы описанного механизма и реализована его компьютерная программа. Показано, что разработанный механизм в детерминированном виде обнаруживает 1024 различных циклов работы с длинной от 2 до 9 шагов в одном цикле.
3. На основе машины Поста разработана система, вычислитель которой построен на основе механизма самоопределения бинарных номинальных операций и обладает некоторыми свойствами (M,R) систем, такими, как самоопределение и замкнутые петли внутренних причин.
4. Для полученной вычислительной системы реализованы программы с механизмом самоопределения бинарных номинальных операций, приведены результаты работы системы и доказана её полнота системы по Тьюрингу. Эксперименты с вычислителем показали, что он обладает высокой чувствительностью к начальным условиям и при этом содержит в результатах своей работы большое количество предельных циклов.

Достоинством данной работы является то, что в ней предложен новый подход имитации (M,R) системы в виде простого логического механизма. Перспективой данной работы может являться усложнение предложенного вычислителя (в рамках машины Поста) и его использование в криптографии для создания новых алгоритмов шифрования и дешифровки. Предполагается

также, что такого рода самоопределяемые алгоритмы могут оказать помощь в обеспечении более сложного автономного взаимодействия в клеточных автоматах и агентских искусственных системах, в частности, используемых для беспилотников. Возможно также соединять несколько экземпляров описанного логического механизма в сети, входы и выходы которых будут соединены (в том числе и через функциональные блоки), что может открыть новые возможности в области самоопределяемых вычислений.

Список использованных источников

1. Артеменков С.Л. Аспекты моделирования и особые свойства сложных систем. *Моделирование и анализ данных*, 2016, №1, С. 47-59. DOI: 10.17759/mda.04.
2. Артеменков С.Л. Онтологический и эпистемологический аспекты моделирования: модельное отношение и адиафорные системы. *Моделирование и анализ данных*. 2022. Том 12. № 4. С. 5–24. DOI: 10.17759/mda.2022120401.
3. Марков А.А. Теория алгорифмов, Тр. МИАН СССР, 42, Изд-во АН СССР, М.–Л., 1954, С. 3–375.
4. Матурана У., Варела Ф. Древо познания. Перевод с англ. Ю.А. Данилова. – М.: Прогресс-Традиция, 2001. – 224 с.
5. Минский М. Вычисления и автоматы. — М.: Мир, 1971. — 360 с.
6. Системы искусственного интеллекта : учебник и практикум для вузов / М.В. Воронов, В.И. Пименов, И.А. Небаев — 2-е изд., перераб. и доп. — Москва : Издательство Юрайт, 2023 — 267 с.
7. Успенский В.А. Машина Поста. М.: Наука, 1988. 96 с.
8. Хокинг С. Бог создал целые числа: математические открытия, изменившие историю. М.: АСТ, 2022. 816 с.
9. Шредингер Э. Что такое жизнь? Физический аспект живой клетки. — Москва-Ижевск: НИЦ «Регулярная и хаотическая динамика», 2002. 92 с.
10. Cárdenas M.L., Letelier J.C., Gutiérrez C., Cornish-Bowden A. and Soto-Andrade J. Closure to efficient causation, computability and artificial life. *Journal of Theoretical Biology*, 2010, 263, 1, 79. DOI: 10.1016/j.jtbi.2009.11.010].
11. Copeland B.J., Proudfoot D. Alan Turing’s Forgotten Ideas in Computer Science, *Scientific American*, 1999, vol. 280, issue 4, pp. 98-103. DOI: 10.1038/scientificamerican0499-98

12. Cornish-Bowden A., Piedrafita G., Morán F., Cárdenas M.L. & Montero F. Simulating a model of metabolic closure. *Biological Theory*, 2013, 8 (4): 383-390. DOI:10.1007/s13752-013-0132-0
13. Gatherer D., Galpin V. Rosen's (M,R) system in process algebra. *Systems Biology*, 2013, 7:128. https://doi.org/10.1186/1752-0509-7-128.
14. Gravner J., Liu X. One-dimensional cellular automata with random rules: longest temporal period of a periodic solution. *Electron. J. Probab*. 27 (2022), article no. 25, 1–23. ISSN: 1083-6489 https://doi.org/10.1214/22-EJP744
15. Kercel S.W. The Endogenous Brain. *Journal of Integrative Neuroscience*. 2004. Vol. 3 (1). P. 61–84.
16. Miranda P.J., La Guardia G. On a relational theory of biological systems: a natural model for complex biological behavior. arXiv preprint arXiv:1705.10394, 2017. DOI:10.48550/arXiv.1705.10394
17. Potgieter P.H. Zeno machines and hypercomputation, *Theoretical Computer Science*, Volume 358, Issue 1, 2006, Pages 23-33, ISSN 0304-3975, https://doi.org/10.1016/j.tcs.2005.11.040. (https://www.sciencedirect.com/science/article/pii/S0304397505009011)
18. Rashevsky N. Mathematical principles in biology and their applications. Springfield, Illinois: Charles Thomas, publisher, 1961. 128 p.
19. Rodrigues, P., Costa, J.F., Siegelmann, H.T. (2001). Verifying Properties of Neural Networks. In: Mira, J., Prieto, A. (eds) Connectionist Models of Neurons, Learning Processes, and Artificial Intelligence. IWANN 2001. Lecture Notes in Computer Science, vol 2084. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45720-8_19
20. Rosen R. Life Itself: A Comprehensive Inquiry into the Nature, Origin, and Fabrication of Life. NY.: Columbia University Press, 1991. 285 p.

ПРИЛОЖЕНИЕ А

Доказательство, что F — топология

Доказательство того, что формула Роберта Розена описывает топологию некоего пространства (что множество F является топологией), будет основываться на проверке верности двух утверждений о топологии, относительно формулы (1), вновь, для удобства, приведённой далее:

$$F = \left\{\left\{\{F\}\right\}\right\}, F \to \{F\} \to \left\{\{F\}\right\} \to \left\{\left\{\{F\}\right\}\right\} \quad (1)$$

Топологией на множестве X называется любая система τ его подмножеств G (то есть некое множество подмножеств), для которой верны два утверждения (Воронов, 2023):

1. Множество X и пустое множество $\emptyset$ принадлежат τ.
2. Сумма $\bigcup_\alpha G_\alpha$ любого (в том числе бесконечного) и пересечение $\bigcap_{k=1}^{n} G_k$ любого конечного числа множеств из τ принадлежит τ.

Первое утверждение, очевидно, истинно, так как множество F определяется, как вложенный элемент в три последовательно вложенных друг в друга вложенных множеств. Наличие в F пустого множества гарантируется тем, что оно содержится в любом множестве в качестве подмножества. Таким образом первое утверждение верно.

Проверим второе утверждение. В общем виде F после подстановки себя же в формулу даст нам себя же, на счётно-бесконечно глубоком вложенном уровне. Таким образом, ввиду идемпотентности операции объединения/суммы множеств ($A \cup A = A$), мы получим всё то же F , а оно является своим тривиальным подмножеством ($A \subseteq A$). Пересечение любого конечного числа подмножеств F даст нам либо F в случае пересечения с собой же, либо —

пустое множество в случае пресечения с пустым множеством. Оба этих случая принадлежат F. Отсюда верно и второе утверждение.

Таким образом, мы имеем тривиальную топологию $F = \{F, \emptyset\}$, что и требовалось доказать.

Стоит заметить, что тривиальность описанного топологического пространства не влечёт за собой тривиальность рассматриваемых (M,R) систем.

Доказательство, что полное описание F требует бесконечности символов

Этот факт тривиален и доказывается подстановкой F в формулу (1):

$$F = \{\{\{F\}\}\} \Longrightarrow F = \{\{\{\{\{\{\dots\}\}\}\}\}\} \tag{A.1}$$

Формула (A.1) при полном описании потребует записи бесконечного количества скобок. Стоит заметить, что такой эффект проявится и при одинарной вложенности:

$$F = \{F\} \Longrightarrow F = \{\{\{\{\{\{\dots\}\}\}\}\}\} \tag{A.2}$$

Однако, несмотря на то, что топология сохранится, из получившейся системы (свёрнутого описания) нельзя будет построить (M,R) систему в описанном виде (полученный результат скорее будет говорить о том, что есть некая самоизменяющаяся система, то есть это будет ещё более высокий уровень обобщения/абстракции).

ПРИЛОЖЕНИЕ Б

Таблица Б.1. Операторы языка программирования улучшенной машины Поста с лентой на 256 ячеек

Оператор	Пример применения	Действие
hajimaru (始まる — начало)	hajimaru	При запуске программа начинает исполняться со строчки, на которой расположен этот оператор (затем этот оператор игнорируется даже в случае использования безусловной цикличности). Если этого оператора в программе нет, то программа начинает исполняться с нулевой строчки.
owari (終わり — конец)	owari	При его исполнении программа заканчивается. Необходим для того, чтобы программа закончилась.
<-	<-	Уменьшает значение адреса на единицу. Если адрес, когда встретился этот оператор, равен нулю, то после применения он будет равен 256.
->	->	Увеличивает значение адреса на единицу. Если адрес, когда встретился этот оператор, равен 256, то после применения он будет равен нулю.
addr	addr 249	Делает адрес, равный указанному значению. Если значение указано больше 256, либо меньше нуля, то значением станет число, появившееся после учёта цикличности ленты.
inaddr	inaddr	Запрашивает ввод адреса у пользователя. В остальном делает всё то же, что и addr.
goto	goto 17 goto label1 goto intvar	Переход на указанную строчку программы. Учитывайте, что нумерация начинается с нуля и не следует переходить на строчки, которых нет — если у вас в программе

Оператор	Пример применения	Действие
		последняя строчка седьмая (нумерация учитывает нулевую), то бессмысленно пытаться перейти на, например, тринадцатую, которой нет. Можно переходить на номера строк, записанные в целочисленных переменных. Также позволяет переходить на метки. В совокупности с bunkiten позволяет реализовывать циклические конструкции.
addrwokaku (addr を書く — написать адрес)	addrwokaku	Вывод в консоль, чему равен адрес.
mojiwokaku (文字を書く — написать символ)	mojiwokaku	Вывод в консоль символа, чей ASCII-код равен значению адреса.
kyouki (狂気 — безумие)	kyouki	Переход на случайную строчку программы. Действует так же, как и goto, но строка, на которую делается переход выбирается случайно (не беспокойтесь — выход за количество строчек не произойдёт). **Внимание: перед исполнением этого оператора должен быть инициализирован генератор случайных чисел через применение оператора “rand”!**
		Пустая строка, которая игнорируется при исполнении (не используйте пробел в начале строки для создания строки комментария!).

Оператор	Пример применения	Действие
;	; 温子は狂気なよね。	Комментарий.
loop	Loop	Адрес становится равным числу количества пройденных программой циклов. Изначально программа прошла ноль циклов, однако, каждый раз доходя до последней строчки, не встретив owari, она возвращается в начало (нулевая строка), при этом счётчик циклов увеличивается на единицу. Оператор loop приравнивает значение адреса количеству пройдённых циклов с учётом замкнутости ленты.
f1	f1 147	Указывает, из какой ячейки берётся значение для первой функции в считывателе.
f2	f2 37	То же, что и f1, но для второй функции.
f3	f3 54	То же, что и f1, но для третьей функции.
o1	o1 147	Указывает, в какую ячейку будет записан выход первой функции.
o2	o2 27	То же, что и o1, но для второй функции.
o3	o3 17	То же, что и o1, но для третьей функции.
prob	prob 27	Вероятность того, что в считывателе единица будет соответствовать отрицанию. При вводе любого целого числа (за исключением минус единицы) берёт остаток от деления его абсолютного значения на 101 и записывается в качестве вероятности. При значении, равным -1, берёт остаток от деления значения адреса на 101 и записывает в качестве

Оператор	Пример применения	Действие
		вероятности. Лучше указывайте число от нуля до сотни.
cycle	cycle 217	Берёт значение из ячейки с указанным номером (минус один — по адресу) и, в зависимости от полученного значения, определяет, будет ли считыватель циклично подавать себе на вход выходное значение, пока не достигнет устойчивого состояния. Ноль — нет, единица — да.
conf1	conf1 3	Берёт один разряд (ноль/один) из указанной ячейки для составления конфигурации (левый знак — «x1»). Если операнд — минус единица, то берёт по адресу.
conf2	conf2 56	То же, что и conf1, но для правого знака («1x»).
conf	conf 7	Эквивалентно случаю, когда для conf1 и conf2 указана одна и та же ячейка. Если в ячейке лежит единица, то конфигурация будет «11», иначе — «00».
kaku (書く — написать)	kaku 23 75	Выводит в консоль значения ячеек ленты из указанного диапазона. Настоятельно рекомендуем следить за тем, чтобы второй операнд был больше первого. Минус один (-1) означает взятие значения по адресу. Если оба операнда равны, то выведена будет только одна ячейка.
bunkiten (分岐点 — поворотный момент)	bunkiten 0 76	Сравнивает значения в двух ячейках. Если они равны, выполняет следующую строчку программы, а если нет — пропускает/перескакивает следующую строчку. Если оба операнда

Оператор	Пример применения	Действие
		равны, то следующая строчка не будет пропущена. Минус единица (-1) указывает ячейку по адресу.
henkamono (変化物 — изменитель)	henkamono 88 44	Считыватель берёт значение из ячейки, чей номер передан вторым операндом, обрабатывает его, согласно своим настройкам, и записывает результат в ячейку, чей номер передан первым операндом. Минус единица означает ячейку по адресу. **Обратите внимание, что этот оператор делает адрес, равным первому операнду!**
ugoku (動く — переместить)	ugoku 93 127	Копирует значение из ячейки, чей номер передан вторым операндом в ячейку, чей номер передан вторым операндом. Минус единица означает взятие значения по адресу. В случае равенства операндов ничего не изменится (ячейка скопируется сама в себя). Оператор является синтаксическим сахаром и может быть реализован через применение считывателя с функциями 000 и вероятностью 100 (либо применение считывателя в режиме работы «00»). **Обратите внимание, что этот оператор делает адрес, равным первому операнду!**
label	label tape7	Объявление метки с именем, переданным первым операндом, которое не должно содержать пробелов и управляющих символов. Запоминает номер строчки, на которой объявлена метка.

Оператор	Пример применения	Действие
block	block q1	Начало блока с именем, переданным первым операндом (ограничения на имя такие же, как для label). Всё, что идёт после block до break игнорируется, если не был исполнен оператор do, вызывающий исполнение блока.
break	break q1	Конец блока. Всё, что после break, уже не игнорируется и исполняется, как обычно.
do	do q1	Вызов исполнения блока с именем, переданным первым операндом. После исполнения кода, расположенного между block и break, интерпретатор возвращается к строке, с которой был совершён вызов блока (т.е. исполнен do) и продолжает исполнять программу со следующей строчки. Примечательно то, что блок может быть вызван из любой точки программы, даже до того, как он был объявлен, так как исполнение происходит на втором чтении, после того, как в первом чтении блоки уже были размечены.
zero (ゼロ)	zero 13	Записать ноль в ячейку, чей номер передан первым операндом. Минус единица — записать по адресу.
hitotsu (一つ)	hitotsu 7	Записать единицу в ячейку, чей номер передан первым операндом. Минус единица — записать по адресу.

ПРИЛОЖЕНИЕ В

Полный код вычислителя (без неупомянутых расширенных возможностей):

```
#include <array>
#include <fstream>
#include <string>
#include <sstream>
#include <map>
#include "windows.h"
#include <cmath>
#include <iomanip>
#include <iostream>
#include <vector>
#include <algorithm> // Для std::find_if
#include <cstdlib>  // Для функции rand() и srand()
#include <ctime>    // Для функции time()
using namespace std;
std::array<int, 257> tape = {};
int f1 = 0;
int f2 = 0;
int f3 = 0;
int cyc = 0;
int conf1 = 0;
int conf2 = 0;
int prob = 100;
int output = 0;
int addr = 0;
int inaddr = 0;
int cycles = 0; // Сколько прошло циклов программы
// Метки, номера строк, блоки
std::map<std::string, int> labels; // Хранилище меток
std::map<std::string, int> blocks; // Хранилище блоков
std::map<std::string, int> dos; // Хранилище строк исполнения
std::map<std::string, bool> inblocks; // Хранилище переменных "находимся ли мы в блоке"
std::map<std::string, int> vars; // Хранилище меток
std::map<std::string, std::vector<std::string>> sarrs;
bool watchblock = false;
bool jikanwomiru = false;
clock_t start;
bool var_messages = true;
int& lineNumber = vars["lineNumber"];
std::vector<std::string>& program = sarrs["program"];
int _negation(int input) {return !input;}
int tautology(int input) {return input;}
void henkamono(int input_tape, int f1, int f2, int f3, int cyc, int conf1, int conf2, int prob, int output_tape) {
  int input = tape[input_tape];
        int output1;
        int output2;
        int output3;
        int negation_value;
        int not_neg_v;
        if (prob == 100) {
                negation_value = 1;
        } else if (prob == 0) {
                negation_value = 0;
        }
        not_neg_v = !negation_value;
  vector<int> functions = {f1, f2, f3};
  vector<vector<int>> history; // Для хранения истории состояний
        if ((conf1 != 0) && (conf2 != 0)) {
                output1 = f1;
                output2 = f2;
                output3 = f3;
        }
  while (true) {

                if (prob > 0 && prob < 100) {
                        // Генерация случайного числа от 0 до 99
                        int randomNumber = std::rand() % 100;
                        if (randomNumber < prob) {negation_value = 1;}
                        else {negation_value = 0;}
                        not_neg_v = !negation_value;
                }
                if ((conf1 == 0) || (conf2 == 0)) {
                        tape[output_tape] = tape[input_tape];
                        break;
                } else if ((conf1 == 0) || (conf2 == 1)) {
                        output1 = (functions[0] == negation_value) ? _negation(input) : tautology(input);
                        output2 = (functions[1] == negation_value) ? _negation(output1) : tautology(output1);
                        output3 = (functions[2] == negation_value) ? _negation(output2) : tautology(output2);
                        tape[f1] = output1;
                        tape[f2] = output2;
                        tape[f3] = output3;
                        if (output3 == negation_value) {
                                functions[1] = (functions[1] == not_neg_v) ? negation_value : not_neg_v; // Тавтология <-> отрицание
                        }
                        if (output2 == negation_value) {
                                functions[0] = (functions[0] == not_neg_v) ? negation_value : not_neg_v; // Тавтология <-> отрицание
                        }
                        if (output1 == negation_value) {
```

```
                                        functions[2] = (functions[2] == not_neg_v) ? negation_value : not_neg_v; // Тавтология <-> отрицание
                                }
                        } else if ((conf1 == 1) || (conf2 == 0)) {
                                if (output3 == negation_value) {
                                        functions[1] = (functions[1] == not_neg_v) ? negation_value : not_neg_v; // Тавтология <-> отрицание
                                }
                                if (output2 == negation_value) {
                                        functions[0] = (functions[0] == not_neg_v) ? negation_value : not_neg_v; // Тавтология <-> отрицание
                                }
                                if (output1 == negation_value) {
                                        functions[2] = (functions[2] == not_neg_v) ? negation_value : not_neg_v; // Тавтология <-> отрицание
                                }
                                output1 = (functions[0] == negation_value) ? _negation(input) : tautology(input);
                                output2 = (functions[1] == negation_value) ? _negation(output1) : tautology(output1);
                                output3 = (functions[2] == negation_value) ? _negation(output2) : tautology(output2);
                                tape[f1] = output1;
                                tape[f2] = output2;
                                tape[f3] = output3;
                        } else if ((conf1 == 1) || (conf2 == 1)) {
                                output1 = (functions[0] == negation_value) ? _negation(input) : tautology(input);
                                if (output3 == negation_value) {
                                        functions[1] = (functions[1] == not_neg_v) ? negation_value : not_neg_v; // Тавтология <-> отрицание
                                }
                                output2 = (functions[1] == negation_value) ? _negation(output1) : tautology(output1);
                                if (output2 == negation_value) {
                                        functions[0] = (functions[0] == not_neg_v) ? negation_value : not_neg_v; // Тавтология <-> отрицание
                                }
                                output3 = (functions[2] == negation_value) ? _negation(output2) : tautology(output2);
                                if (output1 == negation_value) {
                                        functions[2] = (functions[2] == not_neg_v) ? negation_value : not_neg_v; // Тавтология <-> отрицание
                                }
                                tape[f1] = output1;
                                tape[f2] = output2;
                                tape[f3] = output3;
                        }
    vector<int> currentState = {functions[0], functions[1], functions[2], input};
    auto it = std::find_if(history.begin(), history.end(), [&](const vector<int>& state) {return state == currentState;});
    if (it != history.end()) {break;}
    history.push_back(currentState);
    if (cyc == 1) {input = output3;}
                        tape[output_tape] = output3;
  }
}
void printtape(int a1, int a2){
                std::vector<int> tapeprint;
                for (int i = a1; i <= a2; i++) {
                        tapeprint.push_back(tape[i]);
                }
                // Вывод значений на печать
  for (const auto& value : tapeprint) {
    std::cerr << value << " ";
  }
  std::cerr << std::endl;
}
int interpretline(string progline) {
                std::istringstream iss(progline);
  std::string operation, label, block, varname, varname2, type, varname3;
                iss >> operation;
                std::string stv;
                int a1 = 0;
                int a2 = 0;
                int value = 0;
                char cvalue = 0;
                float fvalue = 0;
                double dvalue = 0;
                if (watchblock == false) {
                        if ((operation == "label")) {;}
                        else if ((operation == "do") || (operation == "break") || (operation == "block")) {
                                iss >> block;
                                if (operation == "do") {
                                        inblocks[block] = true;
                                        dos[block] = lineNumber;
                                        lineNumber = blocks[block];
                                } else if (operation == "break") {
                                        if (inblocks[block] == true) {
                                                lineNumber = dos[block];
                                                inblocks[block] = false;
                                        }
                                } else if (operation == "block") {
                                        watchblock = true;
                                }
                        } else if ((operation == "ugoku") || (operation == "henkamono") || (operation == "bunkiten") || (operation == "conf") || (operation == "conf1") ||
                        (operation == "conf2") || (operation == "kaku") || (operation == "cycle") || (operation == "zero") || (operation == "hitotsu") || (operation == "f1") ||
                        (operation == "f2") || (operation == "f3") || (operation == "addr") || (operation == "addrwokaku") || (operation == "mojiwokaku") || (operation == "<-") ||
                        (operation == "->") || (operation == "") || (operation == ";") ||  (operation == "owari") || (operation == "prob") || (operation == "hajimaru") ||
                        (operation == "inaddr") || (operation == "loop") || (operation == "kyouki")) {
                                iss >> a1 >> a2;
                                if (operation == "ugoku") {
                                        if ((a1 == -1) && (a2 != -1)) {
                                                tape[addr] = tape[a2];
                                        } else if ((a1 != -1) && (a2 == -1)) {
                                                addr = a1;
                                                tape[a1] = tape[addr];
                                        } else if ((a1 == -1) && (a2 == -1)) {
                                                tape[addr] = tape[addr];
                                        } else {
                                                addr = a1;
                                                tape[a1] = tape[a2];
```

```
		}
} else if (operation == "henkamono") {
		if ((a1 == -1) && (a2 != -1)) {
			henkamono(a2, f1, f2, f3, cyc, conf1, conf2, prob, addr);
		} else if ((a1 != -1) && (a2 == -1)) {
			addr = a1;
			henkamono(addr, f1, f2, f3, cyc, conf1, conf2, prob, a1);
		} else if ((a1 == -1) && (a2 == -1)) {
			henkamono(addr, f1, f2, f3, cyc, conf1, conf2, prob, addr);
		} else {
			addr = a1;
			henkamono(a2, f1, f2, f3, cyc, conf1, conf2, prob, a1);
		}
} else if (operation == "bunkiten") {
		if ((a1 == -1) && (a2 != -1)) {
			if (tape[addr] == tape[a2]) {lineNumber += 0;}
			else {lineNumber += 1;}
		} else if ((a1 != -1) && (a2 == -1)) {
			if (tape[a1] == tape[addr]) {lineNumber += 0;}
			else {lineNumber += 1;}
		} else if ((a1 == -1) && (a2 == -1)) {
			if (tape[addr] == tape[addr]) {lineNumber += 0;}
			else {lineNumber += 1;}
		} else {
			if (tape[a1] == tape[a2]) {lineNumber += 0;}
			else {lineNumber += 1;}
		}
} else if (operation == "conf") {
		if ((a1 == -1) && (a2 != -1)) {
			conf1 = tape[addr];
			conf2 = tape[a2];
		} else if ((a1 != -1) && (a2 == -1)) {
			conf1 = tape[a1];
			conf2 = tape[addr];
		} else if ((a1 == -1) && (a2 == -1)) {
			conf1 = tape[addr];
			conf2 = tape[addr];
		} else {
			conf1 = tape[a1];
			conf2 = tape[a2];
		}
} else if (operation == "kaku") {
		if ((a1 == -1) && (a2 != -1)) {
			printtape(addr, a2);
		} else if ((a1 != -1) && (a2 == -1)) {
			printtape(a1, addr);
		} else if ((a1 == -1) && (a2 == -1)) {
			printtape(addr, addr);
		} else {
			printtape(a1, a2);
		}
} else if (operation == "cycle") {
		if (a1 == -1) {
			cyc = addr;
		} else {
			cyc = a1;
		}
} else if (operation == "conf1") {
		if (a1 == -1) {
			conf1 = tape[addr];
		} else {
			conf1 = tape[a1];
		}
} else if (operation == "conf2") {
		if (a1 == -1) {
			conf2 = tape[addr];
		} else {
			conf2 = tape[a1];
		}
} else if (operation == "zero") {
		if (a1 == -1) {
			tape[addr] = 0;
		} else {
			tape[a1] = 0;
		}
} else if (operation == "hitotsu") {
		if (a1 == -1) {
			tape[addr] = 1;
		} else {
			tape[a1] = 1;
		}
} else if (operation == "f1") {
		if (a1 == -1) {
			f1 = tape[addr];
		} else {
			f1 = tape[a1];
		}
} else if (operation == "f2") {
		if (a1 == -1) {
			f2 = tape[addr];
		} else {
			f2 = tape[a1];
		}
} else if (operation == "f3") {
		if (a1 == -1) {
			f3 = tape[addr];
		} else {
			f3 = tape[a1];
```

```
                                        }
                                } else if (operation == "addr") {
                                        if (a1 < 0) {
                                                addr = 257 - std::abs(a1);
                                                // Если addr меньше 0, то применяем ту же логику, что и для -258 и ниже
                                                while (addr < 0) {addr = 257 - std::abs(addr);}
                                        } else {addr = a1 % 257;}
                                } else if (operation == "addrwokaku") {std::cerr << addr << std::endl;}
                                else if (operation == "mojiwokaku") {std::cerr << char(addr);}
                                else if (operation == "hajimaru") {;}
                                else if (operation == "->") {
                                        if (addr == 256) {addr = 0;}
                                        else {addr++;}
                                } else if (operation == "<-") {
                                        if (addr == 0) {addr = 256;}
                                        else {addr--;}
                                }
                                else if (operation == "owari") {return 2;}
                                else if (operation == "prob") {
                                        if (a1 == -1) {prob = addr % 101;}
                                        else {prob = std::abs(a1) % 101;}
                                }
                                else if (operation == ";") {lineNumber += 0;}
                                else if (operation == "") {lineNumber += 0;}
                                else if (operation == "kyouki") {lineNumber = std::rand() % (lineNumber+1);}
                                else if (operation == "inaddr") {
                                        std::cin >> inaddr;
                                        if (inaddr < 0) {
                                                addr = 257 - std::abs(inaddr);
                                                // Если addr меньше 0, то применяем ту же логику, что и для -258 и ниже
                                                while (addr < 0) {addr = 257 - std::abs(addr);}
                                        } else {addr = inaddr % 257;}
                                }
                                else if (operation == "loop") {
                                        if (cycles > 256) {addr = cycles % 257;}
                                        else {addr = cycles;}
                                }
                                else {return 1;}
                        }
                } else {
                        if (operation == "break") {watchblock = false;}
                        else if (operation == "*/") {watchblock = false;}
                }
                return 0;
}
int main(int argc, char* argv[]) {
                lineNumber = 0;
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " <filename>" << std::endl;
                        std::cerr << "Kikkago - Quine is so easy in this language!" << std::endl;
                        std::cerr << "Version 1.0.3" << std::endl;
        return 1;
    }
    // Получаем имя файла из аргументов командной строки
    std::string filename = argv[1];
    std::ifstream file(filename);
    if (!file.is_open()) {
        std::cerr << "File not found." << std::endl;
        return 1;
    }
    std::string line;
                int lineNum = 0; // Для номера строки
                int hajimaru = 0;
                vars["cycles"] = 0; // Сколько прошло циклов программы записывается в техническую переменную
                bool dec_owari = false;
                while (std::getline(file, line)) {
                        std::istringstream lstr(line);
                        std::string declare, block, label, varname;
                        std::string stv;
                        int value = 0;
                        char cvalue = 0;
                        float fvalue = 0;
                        double dvalue = 0;

                        lstr >> declare;
                        if (declare == "jikannohajimaru") {
                                start = clock();
                                jikanwomiru = true;
                        } else if (declare == "label") {
                                lstr >> label;
                                if (isnotdeclared(label)) {labels[label] = lineNum;}
                                else return 3;
                        } else if (declare == "block"){
                                lstr >> block;
                                blocks[block] = lineNum;
                        } else if (declare == "hajimaru") {hajimaru = lineNum;}
                        else if (declare == "owari") {
                                dec_owari = true;
                        } else if (declare == "rand") { // Инициализация генератора случайных чисел
                                lstr >> value;
                                if (value == 0) {
                                        std::srand(static_cast<unsigned int>(std::time(0)));
                                } else {
                                        std::srand(value);
                                }
                        } else if (declare == "nsm") {
                                var_messages = false;
                        }
```

```
    program.push_back(line);
                lineNum++;
    }
    file.close();

        if (dec_owari == false) {
                std::cerr << "The 'owari' operator was never written down! Your program will never stop!" << std::endl;
                return 1;
        }

    lineNumber = hajimaru;
        int next_lineNumber = lineNumber + 1;
        char choice;
    while (lineNumber < program.size()) {
                int code = interpretline(program[lineNumber]);

                if (code == 0) {
                        ;
                } else if (code == 2) {
                        if (var_messages == true) {
                                std::cerr << "The program " << filename << " has completed successfully!" << std::endl;
                                if (jikanwomiru == true) {
                                        jikanwomiru = false;
                                        clock_t end = clock();
                                        double seconds = (double)(end - start) / CLOCKS_PER_SEC;
                                        std::cerr << "Seconds: " << seconds << std::endl;
                                }
                        }
                        return 0;
                } else {
                        if (code == 1) {
                                std::cerr << "Unknown operation in line!" << std::endl;
                        } else if (code == 3) {
                                std::cerr << "Declaration of existing variable (var, label or block) or array in line!" << std::endl;
                        } else if (code == 4) {
                                std::cerr << "Division by zero in line!" << std::endl;
                        } else if (code == 5) {
                                std::cerr << "Operand error!"<< std::endl;
                                std::cerr << "The type of the operand is incorrect, or a variable was not passed as the first operand!" << std::endl;
                        }
                        std::cerr << "" << std::endl;
                        std::cerr << "The line on which the error occurred:" << std::endl;
                        std::cerr << "" << std::endl;
                        std::cerr << "Line "<< lineNumber << ": " << program[lineNumber] << std::endl;
                        std::cerr << "" << std::endl;
                        do {
                                std::cout << "Ignore the error (the line will be skipped, even bigger errors are possible!)? (y/n):";
                                std::cin >> choice;
                                if (choice == 'y' || choice == 'Y') {
                                        std::cerr << "The program will continue!" << std::endl;
                                        std::cerr << "" << std::endl;
                                        break;
                                } else if (choice == 'n' || choice == 'N') {
                                        std::cerr << "The program stops!" << std::endl;
                                        return code;
                                } else {
                                        std::cerr << "Invalid input. Please enter 'y' or 'n'." << std::endl;
                                        std::cerr << "" << std::endl;
                                }
                        } while ((choice != 'y') || (choice != 'Y') || (choice != 'n') || (choice != 'N'));
                }
        lineNumber++;
                next_lineNumber = lineNumber + 1;
                if (next_lineNumber > program.size()) {
                        lineNumber = 0;
                        cycles += 1;
                        vars["cycles"] = cycles;
                }
    }
    return 0;
}
```