|  |
| --- |
| Министерство образования Российской Федерации  Государственное образовательное учреждение  **Московский авиационный институт**  **Курсовая работа**  по предмету  «Фундаментальная информатика»  **Построение алгоритмических моделей**  **на примере моделей Тьюринга и Маркова**  Выполнил:  студент гр. М8О-108Б-23 **Романов В.М.**  Проверил:  Преподаватель, каф. 806 **Севастьянов В.С.**  **Москва 2023** |

Содержание:

Введение……………………………………………………………………........2-3

1. Машина Тьюринга…………………………………….…………………….4-14

1.1. Теоретическая часть, связанная с МТ, и тесты..……..…………….4-6

1.2. Демонстрация сделанной МТ и описание команд.………...……..7-13

1.3. Сложностная оценка и выводы к машине Тьюринга….……………14

2. Диаграмма Тьюринга (ДТ)………………………………………………...15-26

2.1. Теоретическая часть, связанная с ДТ, и тесты…………………..15-16

2.2. Диаграмма main machine..………………………………………...18-19

2.3. Диаграммы is\_zero и is\_one….....…………………………………20-21

2.4. Диаграммы count и inverted\_count..………………………………22-23

2.5. Диаграммы two\_to\_eight и inverted\_two\_to\_eight….……………24-26

2.6. Сложностная оценка диаграммы Тьюринга и выводы..……………27

3. Нормальные алгоритмы Маркова (НАМ)…..…………………………….28-37

3.1. Теоретическая часть, связанная с НАМ………………………….28-29

3.2. Демонстрация сделанного НАМ…………………………………30-35

3.3. Сложностная оценка сделанного НАМ и выводы………………….36

Заключение……………………………………………………………………….37

Введение.

Перед тем, как перейти к практической части моей курсовой работы, необходимо ознакомится во введении с её теоретической частью.

Итак, что же такое алгоритмы и алгоритмические модели? Алгоритм —это точно заданная последовательность правил, указывающая, каким образом можно за конечное число шагов получить выходное сообщение определенного вида, используя заданное исходное сообщение. Всё было бы замечательно с этим определением, если бы не одно «но»: оно не является формальным. Для более строгой трактовки определения необходимы так называемые алгоритмические модели. Среди них выделяют: рекурсивные функции (понятие алгоритма связывается с вычислениями и числовыми функциями), машины Тьюринга (алгоритм представляется как описание процесса работы некоторой машины, способной выполнять лишь небольшое число весьма простых операций), нормальные алгоритмы Маркова (алгоритмы описываются как преобразования слов в произвольных алфавитах). В моей курсовой работе будут фигурировать лишь машины Тьюринга и нормальные алгоритмы Маркова. После такого небольшого вступления можно перейти к описанию целей и задач этой работы.

Цель: проиллюстрировать определения алгоритма путём построения алгоритмических моделей Тьюринга и Маркова.

Задачи: (взял цели из лабораторных работ)

1. составить программу машины Тьюринга в четвёрках, выполняющую заданное действие над словами, записанными на ленте;
2. разработать диаграмму Тьюринга решения задачи с использованием стандартных машин (r, l, R, L, K, a) и вспомогательных машин, определяемых задачей.
3. разработать нормальный алгоритм Маркова, обменивающий местами два троичных числа, разделённых знаком "^".
4. обобщить полученную информацию и сделать соответствующий вывод.

Для выполнения поставленных задач мне необходимы:

* эмулятор машины Тьюринга в четвёрках (ссылка на него: <file:///C:/Users/vadim/Downloads/l05-2011_jstu4-2.3/jstu4-2.3/jstu4/jstu4.html?lang=ru>)
* диаграммер для работы с диаграммами Тьюринга («VirtualTuringMachine.exe»)
* эмулятор для нормальных алгоритмов Маркова (<file:///C:/Users/vadim/OneDrive/Рабочий%20стол/nam/index.html>)
* текстовый редактор Microsoft Word (не хочется мне использовать Latex)
* мой репозиторий на GitHub, куда были выложены все описанные ниже алгоритмы(https://github.com/vadimromanov05/Laboratory-works/tree/master)

В качестве списка литературы могу представить лишь один пункт:

* С. С. Гайсаpян, В. Е. Зайцев «Курс информатики», Москва, Издательство Вузовская книга, 2013 г.

На этом моё вводная часть курсовой работы подходит к концу. Пришло время приступить к рассмотрению самих алгоритмических моделей.

1. Машина Тьюринга.

1.1. Теоретическая часть, связанная с МТ

Пока я не начал описывать сделанную машину Тьюринга, необходимо дать ей более точное определение для простоты восприятия материала. Итак, Машиной Тьюринга называется упорядоченная четверка объектов T = (A, Q, P, q0), где T - символ МТ, A - конечное множество букв (рабочий алфавит), Q -конечное множество символов (имен состояний), q0 - имя начального состояния, P - множество упорядоченных четверок (q, a, v , q ′), q ,q ′∈ Q, a ∈A∪{λ}, v ∈ {l, r } ∪ A ∪ {λ} (программа), определяющее три функции: функцию выхода Fl: Q∗̄ A→ ̄A (̄A =A∪{λ}),функцию переходов Ft :Q∗ ̄ A → Q, и функцию движения головки Fv : Q ∗ ̄A → {l, r, s} (символ s означает, что головка неподвижна).

Мне это определение и самому было не понятно, пока я не сделал пятую лабораторную работу. В переводе на русский язык можно сказать, что каждая четвёрка состоит из:

1. начального состояния (то, в котором находится головка до выполнения следующей команды)
2. начальной буквы, расположенной там, где находится головка
3. команды, которую должна сделать машина Тьюринга. Это может быть:

- переход на одну ячейку влево;

- переход на одну ячейку вправо;

- замена начальной буквы на другую;

- ничего не делать, просто в пункте 4 сменить состояние (смысла в такой команде нет, поэтому её не упоминают).

4) состояния, в которое должна перейти головка.

Небольшая вставка: если в четвёрке не описано какое-нибудь действие (команда или смена состояние), то это приведёт к зацикливанию программы и всё будет очень плохо.

Например: 01,0,0,01 – у вас просто будет постоянно ставится ноль, и головка никогда не сдвинется с места.

Подводя итоги теоретической части, нужно сказать, что на вот этих «четвёрках» и построено определение алгоритма по Тьюрингу. С их помощью и задаются правила, по которым можно определить за конечное число шагов необходимое нам преобразование сообщения. Определение алгоритмов по Тьюрингу настолько фундаментально, что с его помощью можно описать всё, что только возможно представить в виде алгоритма.

А теперь пришло время продемонстрировать работу моей машины Тьюринга. Её задача заключалась в нормированном переводе введённого числа из девятеричной системы в троичную. Под нормированным переводом подразумевалось начальное положение головки справа от входного сообщения, сохранение исходного сообщения и запись нового числа справа от исходного сообщения через один пробел.

Сначала я постараюсь описать основополагающую информацию для выполнения данной лабораторной работы.

Примером такой информации является таблица перевода цифр из девятеричной системы в троичную.

|  |  |
| --- | --- |
| Девятеричная система счисления | Троичная система счисления |
| 0 | 00 |
| 1 | 01 |
| 2 | 02 |
| 3 | 10 |
| 4 | 11 |
| 5 | 12 |
| 6 | 20 |
| 7 | 21 |
| 8 | 22 |

Так можно сделать не со всеми системами счисления. Такое упрощение работы связано с тем, что основание большей системы счисления является результатом возведения в степень основания меньшей системы счисления. А количество цифр в меньшей системе счисления равно показателю данного возведения в степень.

Кроме того, при переводе из девятеричной системы счисления в троичную необходимо избавиться от незначащих нулей: сначала проигнорировать их в начальном сообщении, а после нужно создать особые команды для перевода чисел 1 и 2 в троичную систему счисления, если они стоят в старшем разряде исходного числа (там тоже надо убрать незначащий ноль).

Не стоит забывать и про нормированность. Чтобы её соблюдать, пришлось ввести некоторые дополнительные команды, о которых я расскажу позднее.

Перед тем, как показывать работу программы добавлю сода несколько тестов для машины.

|  |  |
| --- | --- |
| Входное сообщение | Выходное сообщение |
| 0123456780 | 10210111220212200 |
| 000123 | 10210 |
| 876543210 | 222120121110020100 |
| 111 | 10101 |
| n \* '0', n in N | 0 |
| 806 | 220020 |
| 010302 | 100100011 |

Таким образом, мы получим формальное определение алгоритма через алгоритмическую модель Тьюринга, где командами или состояниями я буду называть правила, связанные между собой и предназначенные для выполнения поставленной перед ними задачи.

1.2. Демонстрация сделанной МТ и описание команд

Так как вставить всю машину целиком и так подойти под необходимый объём курсовой будет слишком просто, я решил демонстрировать команды лишь для одной-двух цифр для экономии пространства. Кроме того, было бы проще показывать состояния не по номерам, как это обычно делается, а по названиям, которые я сам им дал. Так будет гораздо проще разобраться в работе программы.

Итак, начнём с команды первого перемещения головки в начало

00, ,<,run\_to\_begin (< означает сдвиг головки на ячейку влево)

run\_to\_begin,0,<,run\_to\_begin run\_to\_begin,1,<,run\_to\_begin run\_to\_begin,2,<,run\_to\_begin run\_to\_begin,3,<,run\_to\_begin run\_to\_begin,4,<,run\_to\_begin run\_to\_begin,5,<,run\_to\_begin run\_to\_begin,6,<,run\_to\_begin run\_to\_begin,7,<,run\_to\_begin run\_to\_begin,8,<,run\_to\_begin run\_to\_begin, ,>,run\_to\_digits

Под 00 подразумевается начальное состояние головки машины Тьюринга. Дальше команда run\_to\_begin осуществляет перевод головки в начало слова, пока не дойдёт до пробела. В этом случае она делает шаг вправо (к первой цифре) и переходит в новое состояние: run\_to\_digits.

Далее идёт как раз команда run\_to\_digits.

74, ,>,run\_to\_digits

run\_to\_digits,0,>,run\_to\_digits

run\_to\_digits,1,=,copy\_first (= означает отсутствие действий для головки)

run\_to\_digits,2,=,copy\_first

run\_to\_digits,3,=,copy\_first

run\_to\_digits,4,=,copy\_first

run\_to\_digits,5,=,copy\_first

run\_to\_digits,6,=,copy\_first

run\_to\_digits,7,=,copy\_first

run\_to\_digits,8,=,copy\_first к этой команде я вернусь позднее

run\_to\_digits, ,>,fin\_place\_zero (> - это сдвиг головки на ячейку вправо)

Команду run\_to\_digits я добавил уже незадолго до сдачи лабораторной работы как раз для работы с самой первой цифрой входного числа.

Немного остановлюсь на команде fin\_place\_zero. Она нужна для случая, если входного слова просто нет. В таком случае просто ставится ноль.

75, ,>,fin\_place\_zero

fin\_place\_zero, ,0,run\_to\_fin - это уже команда для завершения программы

Как и обещал, возвращаюсь к команде copy\_first\_one.

65, ,>,copy\_first

copy\_first,0,>,copy

copy\_first,1, ,jump\_first\_one

copy\_first,2, ,jump\_first\_two

copy\_first,3, ,jump\_three

copy\_first,4, ,jump\_four

copy\_first,5, ,jump\_five

copy\_first,6, ,jump\_six

copy\_first,7, ,jump\_seven

copy\_first,8, ,jump\_eight

copy\_first, ,>,run\_to\_fin

(цифра или пробел на третьей позиции означают его постановку на рабочую ячейку)

Данная программа нужна для запоминания первой цифры числа и её стирания (это необходимо, чтобы мы не обработали её ещё раз при следующей итерации программы). Если она все цифры уже обработала и встречает пробел, то программа начинает своё завершение.

После запоминания цифры работает команда jump\_(цифра) или jump\_first\_(цифра). Она нужна для перехода вправо через пробел, который мы создали стиранием цифры.

66, ,>,jump\_first\_one

jump\_first\_one, ,>,copy\_first\_one

Как я ранее упоминал, слово first означает лишь первую цифру числа. Пока что все команды выглядят одинаково.

Далее нам необходимо добраться до конца первого слова с помощью команды copy\_(цифра) или copy\_first\_(цифра)

67, ,>,copy\_first\_one

copy\_first\_one,0,>,copy\_first\_one

copy\_first\_one,1,>,copy\_first\_one

copy\_first\_one,2,>,copy\_first\_one

copy\_first\_one,3,>,copy\_first\_one

copy\_first\_one,4,>,copy\_first\_one

copy\_first\_one,5,>,copy\_first\_one

copy\_first\_one,6,>,copy\_first\_one

copy\_first\_one,7,>,copy\_first\_one

copy\_first\_one,8,>,copy\_first\_one

copy\_first\_one, ,>,run\_first\_one

Достигнув цели, мы передаём эстафету команде run\_(цифра) или run\_first\_(цифра), которая доводит головку до конца нового числа или останавливает головку в двух пробелах от первого числа, если мы ничего ещё не записали.

68, ,>,run\_first\_one

run\_first\_one,0,>,run\_first\_one

run\_first\_one,1,>,run\_first\_one

run\_first\_one,2,>,run\_first\_one

run\_first\_one,3,>,run\_first\_one

run\_first\_one,4,>,run\_first\_one

run\_first\_one,5,>,run\_first\_one

run\_first\_one,6,>,run\_first\_one

run\_first\_one,7,>,run\_first\_one

run\_first\_one,8,>,run\_first\_one

run\_first\_one, ,1,come\_back\_one

Начиная с этой команды, для разных чисел в разном расположении уже начинаются различия. В вышеописанном случае с первой единицей всё просто: справа через один пробел от входного сообщения мы ставим единицу в троичной системе счисления и передаём управление команде come\_back\_(цифра), о которой я расскажу позднее.

В случае с другими цифрами, начиная с двух, всё будет по-другому.

60, ,>,run\_eight

run\_eight,0,>,run\_eight

run\_eight,1,>,run\_eight

run\_eight,2,>,run\_eight

run\_eight,3,>,run\_eight

run\_eight,4,>,run\_eight

run\_eight,5,>,run\_eight

run\_eight,6,>,run\_eight

run\_eight,7,>,run\_eight

run\_eight,8,>,run\_eight

run\_eight, ,2,place\_eight

61, ,>,place\_eight

place\_eight,2,>,place\_eight

place\_eight, ,2,come\_back\_eight

Здесь уже нужна вспомогательная команда place\_(цифра), которая сместит головку ещё на ячейку вправо, поставит ещё одну цифру в троичной системе счисления и также передаст управление команде come\_back\_(цифра). Стоит заметить, что для значащего нуля всё будет так же, как и для восьмёрки.

Следующее состояние нужно для возвращения головки в начало нового слова. С этого момента команды уже унифицированы.

62, ,<,come\_back\_eight

come\_back\_eight,0,<,come\_back\_eight

come\_back\_eight,1,<,come\_back\_eight

come\_back\_eight,2,<,come\_back\_eight

come\_back\_eight,3,<,come\_back\_eight

come\_back\_eight,4,<,come\_back\_eight

come\_back\_eight,5,<,come\_back\_eight

come\_back\_eight,6,<,come\_back\_eight

come\_back\_eight,7,<,come\_back\_eight

come\_back\_eight,8,<,come\_back\_eight

come\_back\_eight, ,<,new\_come\_back\_eight

new\_come\_back\_(цифра) необходима для перехода к стёртой нами ранее цифре и её возвращения на место в начальном числе.

07, ,<,new\_come\_back\_zero

new\_come\_back\_zero,0,<,new\_come\_back\_zero

new\_come\_back\_zero,1,<,new\_come\_back\_zero

new\_come\_back\_zero,2,<,new\_come\_back\_zero

new\_come\_back\_zero,3,<,new\_come\_back\_zero

new\_come\_back\_zero,4,<,new\_come\_back\_zero

new\_come\_back\_zero,5,<,new\_come\_back\_zero

new\_come\_back\_zero,6,<,new\_come\_back\_zero

new\_come\_back\_zero,7,<,new\_come\_back\_zero

new\_come\_back\_zero,8,<,new\_come\_back\_zero

new\_come\_back\_zero, ,0,jump\_2\_zero

Дальше с помощью команды jump\_2\_(цифра) мы переходим к новому числу и начинаем весь цикл заново, пока не закончатся незадействованные цифры в начальном числе.

08, ,>,jump\_2\_zero

jump\_2\_zero,0,>,copy

Теперь стоит вспомнить про состояние run\_to\_fin, о котором я рассказывал ранее. Оно необходимо для соблюдения нормирования вычислений. Головка должна оказаться справа от созданной мною строки.

72, ,>,run\_to\_fin

run\_to\_fin,0,>,run\_to\_fin

run\_to\_fin,1,>,run\_to\_fin

run\_to\_fin,2,>,run\_to\_fin

run\_to\_fin,3,>,run\_to\_fin

run\_to\_fin,4,>,run\_to\_fin

run\_to\_fin,5,>,run\_to\_fin

run\_to\_fin,6,>,run\_to\_fin

run\_to\_fin,7,>,run\_to\_fin

run\_to\_fin,8,>,run\_to\_fin

run\_to\_fin, ,=,fin

Далее идёт переход к конечному состояние в лице команды fin и завершение работы программы. (# означает переход к конечному состоянию)

73, ,#,fin

fin, ,#,fin

1.3. Сложностная оценка и выводы к машине Тьюринга

Для проведения сложностной оценки здесь и далее я буду предоставлять таблицы с входными сообщениями разной длины и временем выполнения программы. На их основе сложность алгоритма можно будет определить, не углубляясь далеко в дебри матанализа. Все вычисления времени будут примерными, так как пользоваться я буду обычным секундомером и округлять значения до целого. Такая погрешность компенсируется обильностью тестов.

|  |  |
| --- | --- |
| Входное сообщение | Время выполнения программы, с |
| 1 | 1 |
| 11 | 2,5 |
| 111 | 4 |
| 1111 | 6 |
| 11111 | 9 |
| 111111 | 12 |
| 1111111 | 15 |

Как можно убедится из примеров, при увеличении длины слова в 2 раза время выполнения возрастает больше чем в 2, но меньше чем в 4 раза, из чего можно сделать вывод, что сложность алгоритма составляет O(n\*log(n)). Учитывая специфику программ на машине Тьюринга, сделать сложность линейной для моей задачи нельзя, ровно, как и логарифмическую сложность.

Что касается выбранного мною способа решения, то для меня он оказался наиболее простым в исполнении (а именно посимвольный перенос данных из входного слова в ответ). Принципиально более продвинутых методов решения даже сам Зайцев В.Е. (с логарифмической сложностью) предлагать не стал, так как посчитал это слишком сложным для нас. Да их, собственно, и нет. Можно было только изменить состояния для перемещения головки, что ничего бы толком не изменило. На этом работа над машиной Тьюринга завершена

2. Диаграмма Тьюринга

2.1. Теоретическая часть, связанная с диаграммами Тьюринга, и тесты

Стандартные машины Тьюринга имеют ряд существенных недостатков. Из тех, что я обнаружил при выполнение лабораторной работы, могу выделить следующие:

1. Приходится прописывать состояния для перемещения головки для каждой буквы отдельно, что очень замедляет работу.
2. Даже моя простая программа имеет длину более тысячи строк, и в формате tu4 воспринять её практически невозможно даже мне. Чтобы этого избежать, приходилось давать состояниям длинные имена, что тоже не ускоряет и не упрощает работу.
3. Кроме того, я боюсь даже представить, насколько сложно было бы копировать слова в формате tu4. Если бы мне пришлось этим заняться, программа увеличилась бы ещё раза в полтора.

Чтобы решить эти проблемы и придать машине Тьюринга «человеческое лицо», были придуманы так называемые диаграммы Тьюринга.

Диаграммы Тьюринга представляют одни МТ через другие, более простые МТ иным, визуально-топологическим способом, причём, как будет показано далее, этот способ не менее строг и полон, нежели "обычные" МТ. Так, машина, копирующая на ленте записанное на ней слово, может быть представлена через МТ, которые ищут начало слова на ленте, конец слова на ленте, копируют одну из букв слова и т. д. Эти более простые МТ в свою очередь могут быть представлены через еще более простые МТ и т. д. Такой нисходящий процесс представления МТ через более простые МТ должен обязательно оборваться, так как рано или поздно мы сведем описание каждой из рассматриваемых МТ к элементарным действиям, введенным при определении МТ. При этом рассматриваемая МТ будет описана через элементарные МТ, т. е. такие, которые уже нельзя описать через более простые МТ, так как каждая из них выполняет всего одно элементарное действие и останавливается.

Эти элементарные МТ решают описанные мною проблемы 1) и 3). А топографический формат записи и возможность разбивать программу на части ликвидируют второй недостаток «обычных» машин Тьюринга.

Как раз на элементарных МТ стоит остановиться поподробнее.

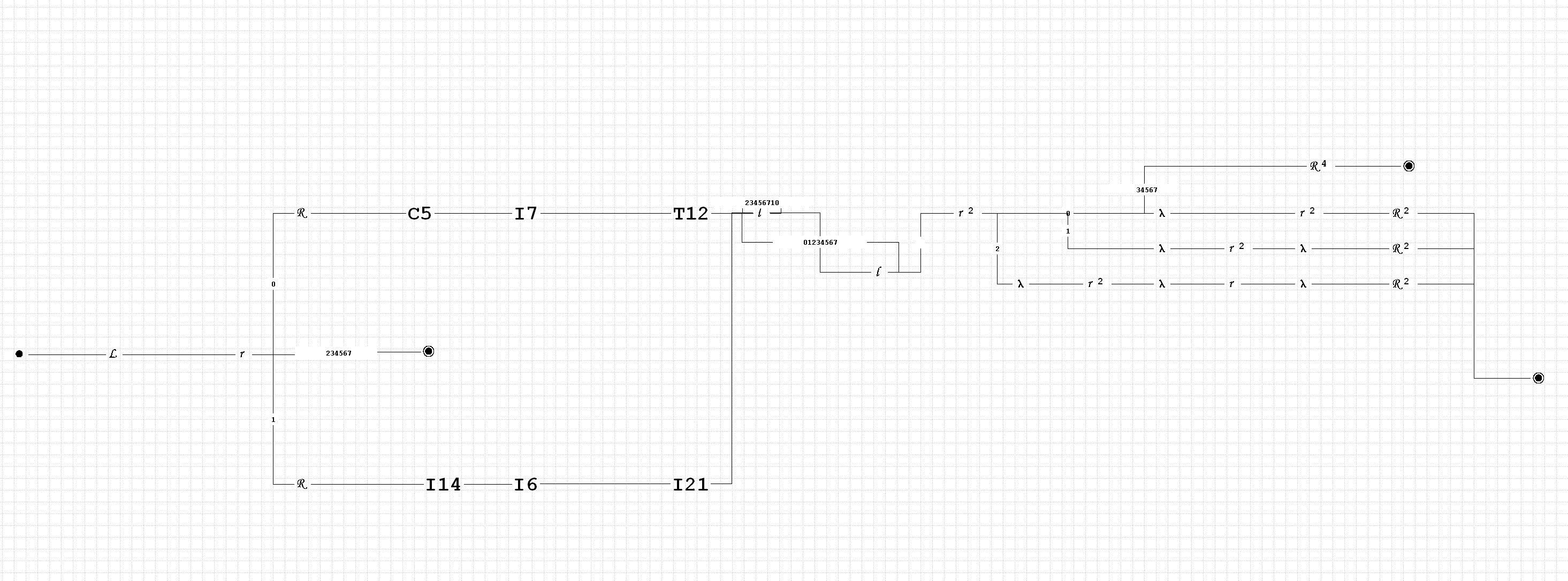
1. Машина сдвига на одну ячейку влево (обозначается l)
2. Машина сдвига на одну ячейку вправо (обозначается r)
3. Машина сдвига влево до конца слова (обозначается L)
4. Машина сдвига вправо до конца слова (обозначается R)
5. Машина копирования слова (обозначается K)
6. Машина постановки символа (по умолчанию λ)

На этих машинах как раз и основана моя диаграмма Тьюринга, которую я продемонстрирую далее. Её задачей являлось восстановление целого числа в восьмеричной системе счисления по дополнительному коду. Дополнительный код — это способ записи числа, в котором старший разряд является знаковым. Если значение старшего разряда равно 0, то это значит, что в остальных разрядах записано положительное двоичное число, совпадающее с прямым кодом. А если нет, то цифры в числе нужно заменить (1 на 0, 0 на 1).

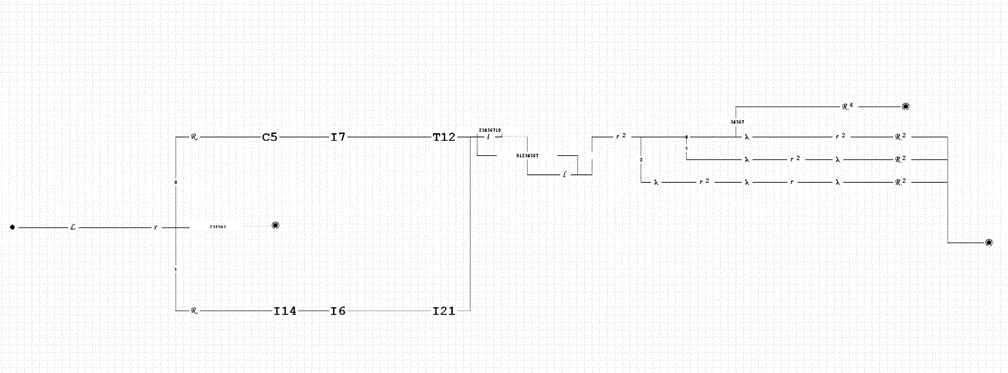
А пока стоит ещё остановиться на тестах.

|  |  |
| --- | --- |
| Входные данные | Выходные данные |
| 0\_000\_001\_010\_011\_100\_101\_110\_111 | 1234567 |
| 1\_111\_110\_101\_100\_011\_010\_001\_000 | 1234567 |
| 0\_000 | 0 |
| 1\_111 | 0 |
| 0\_111 или 00\_111 | 7 |
| 0 или 00 или 000 | 0 |

2.2. Диаграмма main machine



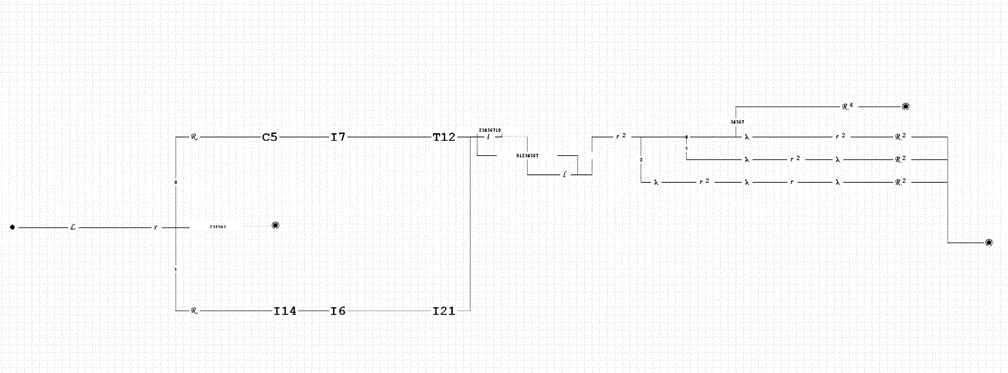
Так она выглядит целиком. Если честно, в таком формате я ничего не вижу, поэтому имеет смысл продемонстрировать её по частям.



Вот так мы хотя бы можем увидеть, как мы начинаем работу программы с левой стороны. Головка доходит до начала слова, а затем переходит к первой цифре, которая определяет знак числа. В зависимости от того, какая цифра там оказалась мы передаём управления разным подмашинам. В случае с нулём у нас работают is\_zero, two\_to\_eight и count. В случае с единицей – is\_one, inverted\_two\_to\_eight и inverted\_count. Смысл работы у обеих ветвей одинаковый: сначала идёт проверка достаточность поставленных в начальном числе цифр: если их не хватает, слева дописываются нули, потом идёт на незначащие нули, и только после этого мы осуществляем перевод из двоичной системы счисления в восьмеричную по разным таблицам соответствия, которые я приведу позднее.

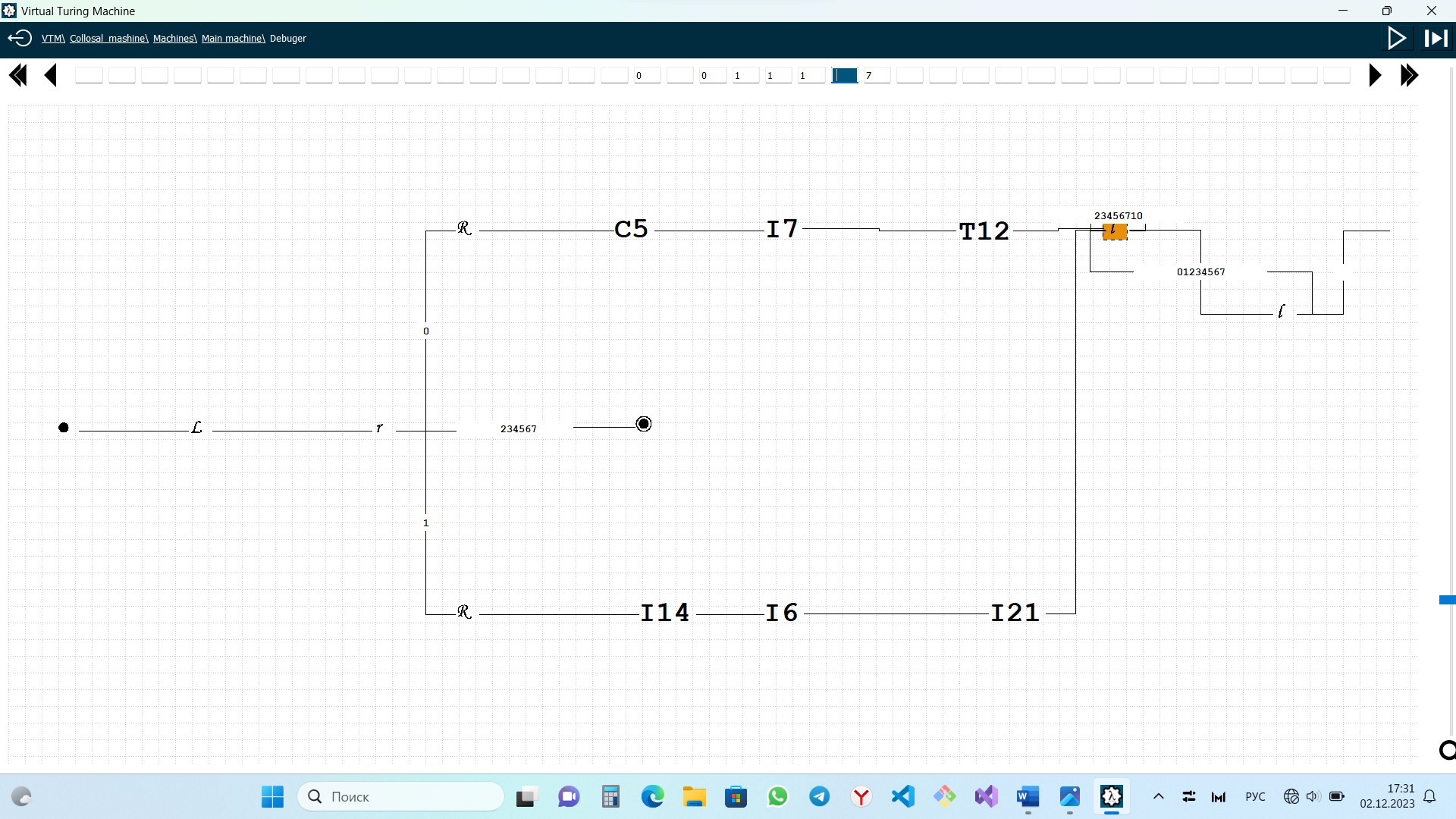
Если кто-то решил проверить мою смекалку и введёт любую другую цифру, кроме нуля и единицы, то программа просто остановится, хотя надо было просто её зациклить (это я уже потом узнал, но переделывать ничего не стал).

По окончании работы моих подмашин ветви диаграммы объединяются и дальше происходит следующее:



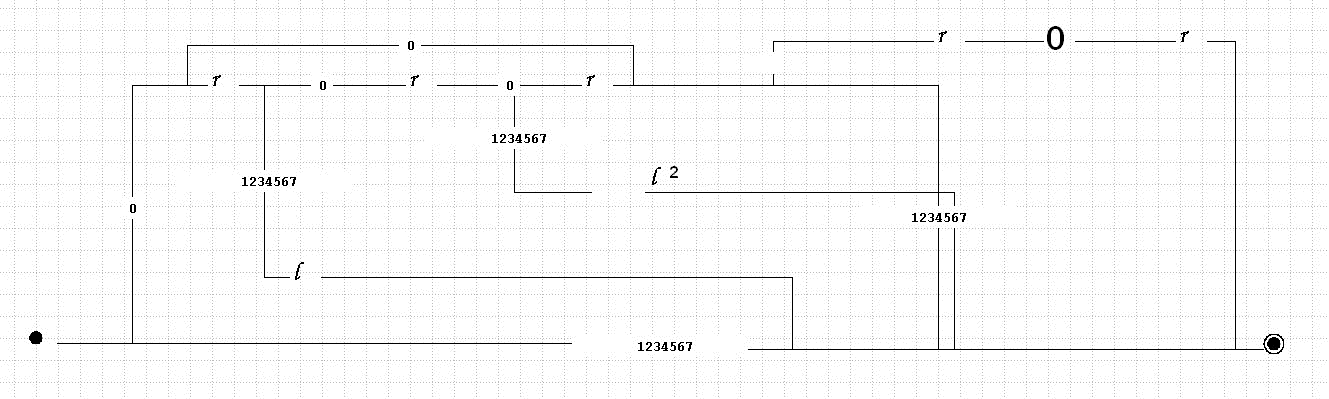
На самом деле здесь просто собирается весь оставленный предыдущими функции мусор. Попробую объяснить понятнее, как это происходит.

Вот так выглядит лента после выполнения всех подмашин:



Почему на ленте появляется ноль – это вопрос уже к подмашинам, о которых я расскажу позднее. Он необходим для обозначения количества нулей или единиц, которые я поставил слева от числа, чтобы его количество цифр дополнить до вида: 3n + 1, где n – натуральное число. Эти цифры и сам ноль (хотя там может быть единица и двойка необходимо убрать для нормированности вычислений). Для этого головка перемещается в начало входного сообщения и ещё одним прыжком доходит до левого нуля, считывает, а после и стирает его. В зависимости от числа головка смещается вправо и либо ничего не стирает, а просто идёт в конец второго слова и завершает работу программы, либо перед этим стирает одну или две первых цифры входного сообщения. На этом работа основной диаграммы и всей программы в целом подходит к концу. Пришло время перейти к вспомогательным диаграммам, которые я вскользь уже упоминал.

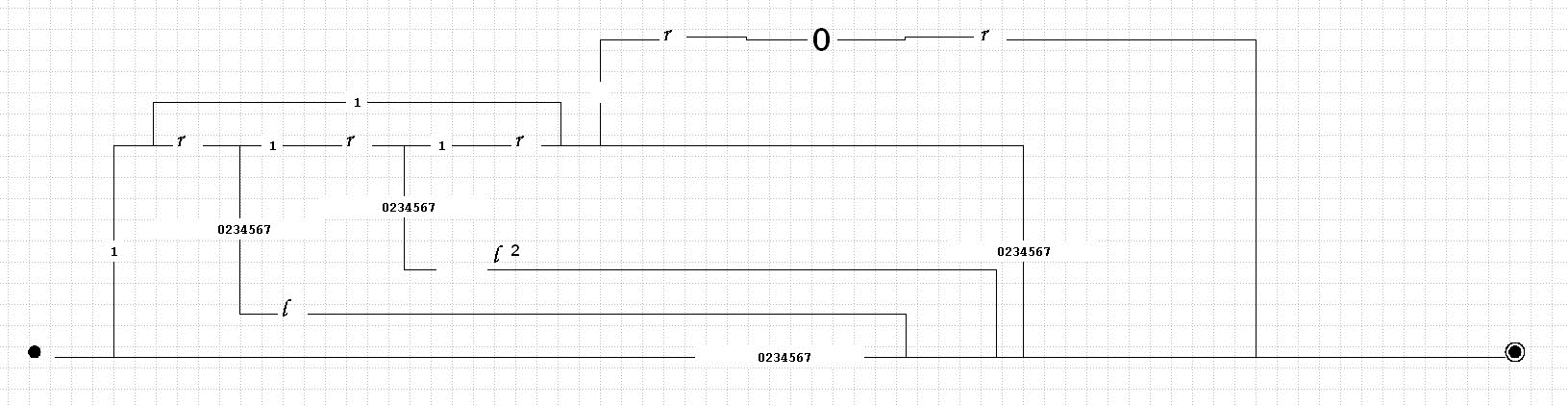
2.3. Диаграммы is\_zero и is\_one.

Свои разъяснения я начну с диаграммы is\_zero. Вот и она сама: 

Её работа начинается сразу после обработки первой цифры входного числа в случае, если это ноль. Сразу после старта я заставляю подмашину завершить работу, если головка видит цифру, отличную от нуля (но это завершение работы не имеет нужды, так как мы всё равно такой цифры не увидим из-за ветвления, описанного в main machine).

В программе в любом случае всё хорошо, и поэтому головка делает шаг вправо и проверяет, является ли следущее число нулём. Если это не так, то головка возвращается в ячейку, где она была в начале выполнения подмашины и завершает её работу. Если она всё-таки видит ноль, то делается ещё шаг вправо и «условный оператор» повторяет свою работу. Если же мы снова видим ноль, то программа повторяет описанные выше действия до тех пор, пока не найдёт хоть одно ненулевое число или пробел. Так мы решаем проблему с незначащими нулями: они просто игнорируются программой во время работы следующих подмашин. Если же на ленту были введены только нули, то программа после нахождения пробела делает ещё шаг вправо, ставит на ленте 0, делает ещё шаг вправо и завершает работу. Дополнительным шагом в конце мы не даём работать следующим подмашинам и доводим программу до очистки мусора в неизменном виде.

В случае с диаграммой is\_one принцип работы примерно тот же:



Эта подмашина выполняет те же самые операции в случае, если самой первой цифрой в числе была единица.

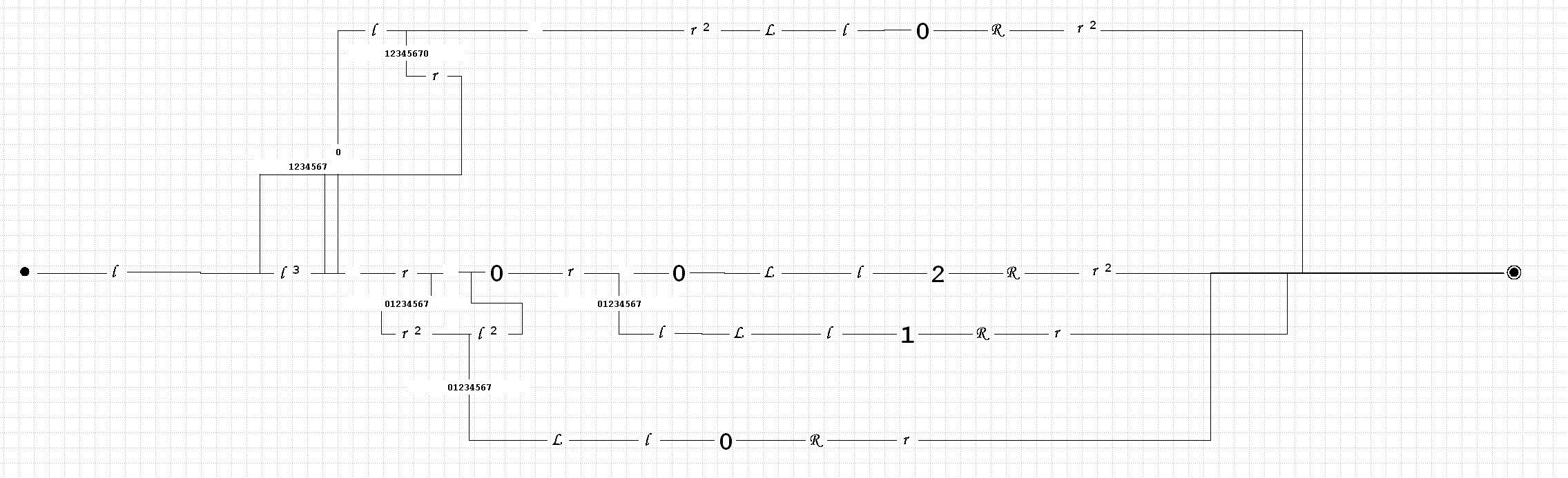
В качестве вывода к этим подмашинам могу лишь сказать, что они были нужны исключительно как защитные программы, целью которых было увеличение выживаемости моей программы при прохождении тестов. Они закрывают собой очень большую «брешь» в виде возможного наличия незначащих нулей во входном числе, связанное с инициативой лаборанта.

На этом описание работы данных подмашин завершено.

2.4. Диаграммы count и inverted\_count

Стоило разместить эти диаграммы в самом начале, так как они работают первыми, но я думаю, что всё и так будет понятно.

Начну я с подмашины count. Она предназначена для добавления недостающих нулей в том случае, если в введённом числе количество цифр не равно 3n + 1, где n -натуральное число.

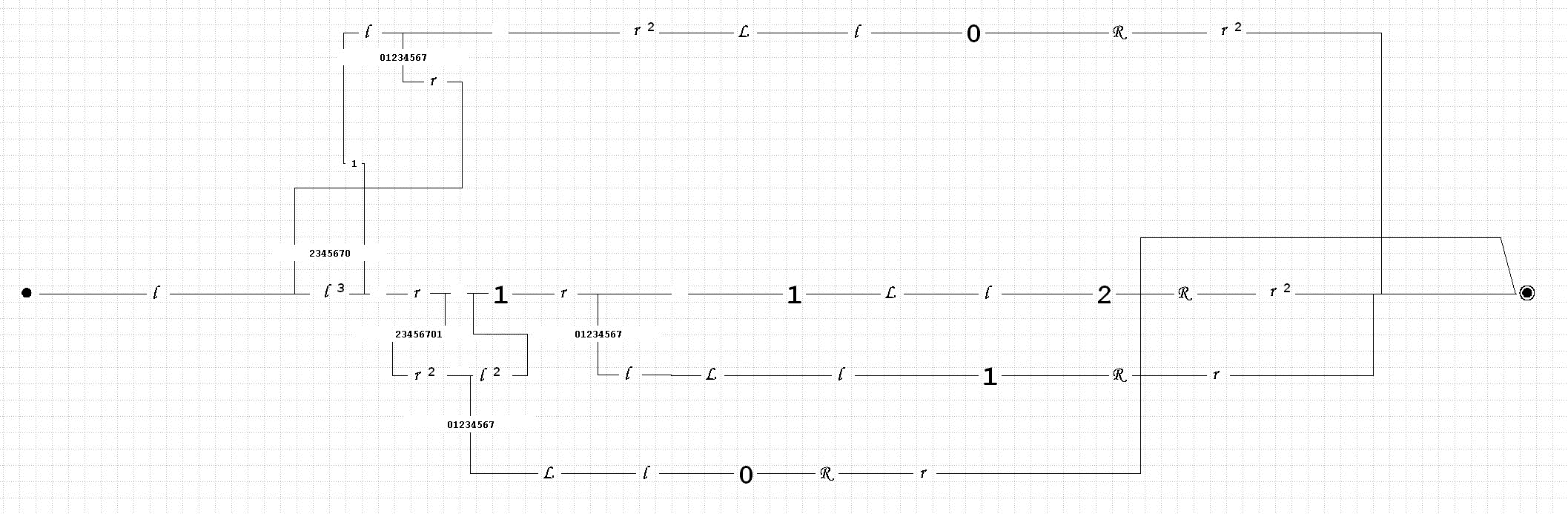


В начале работы диаграммы мы находимся справа от введённого слова. Головка делает шаг влево, а после начинает двигаться влево уже на 3 ячейки сразу до тех пор, пока не встретит пробел. В случае встречи с нулём присходит следущее: головка делает шаг влево и проверяет, что записано слева от нуля. Если там записана какая-то цифра, то делается обратно шаг вправо, и прыжки через 3 ячейки продолжаются. Если же слева от нуля пробел, то это значит, что число соответсттвует всем критериям и ничего добавлять не нужно. В этом случае головка делает два шага вправо, потом доходит до начала входного слова, делает ещё шаг влево и ставит 0 – количество нулей, которое мы добавили слева от числа. Потом головка делает два шага вправо и идёт до конца введённого числа, после чего передаёт управление машине is\_zero.

Если же цифр оказалось недостаточно, то головка рано или поздно дойдёт до пробела, после чего она будет делать шаги вправо, пока не дойдёт до близжайшего нуля. В зависимости от количества шагов будет поставлено соответствующее количество нулей и слева от них записано их количество (1 или 2). После этого головка завершит работу подмашины сразу после прихода в конец начального числа.

Теперь стоит сказать пару слов о самом нижнем ответвлении диаграммы. В случае, если мы после встречи с пробелом в двух ячейках справа найдём какое-либо число, то это тоже будет считаться как отсутствие необходимости ставить лишние нули, и головка просто повторит работу самой верхней ветки.

С машиной inverted\_count ситуация аналогичная, только здесь мы уже работаем с дополнительным кодом.



Эти две машины также отвечают за защиту программы от неверного ввода данных. Полезной нагрузки они особенно не несут, а лишь позволяют диаграмме пережить лишние пару тестов.

Объяснение работы диаграмм count и inverted\_count подошло к концу.

2.5. Диаграммы two\_to\_eight и inverted\_two\_to\_eight

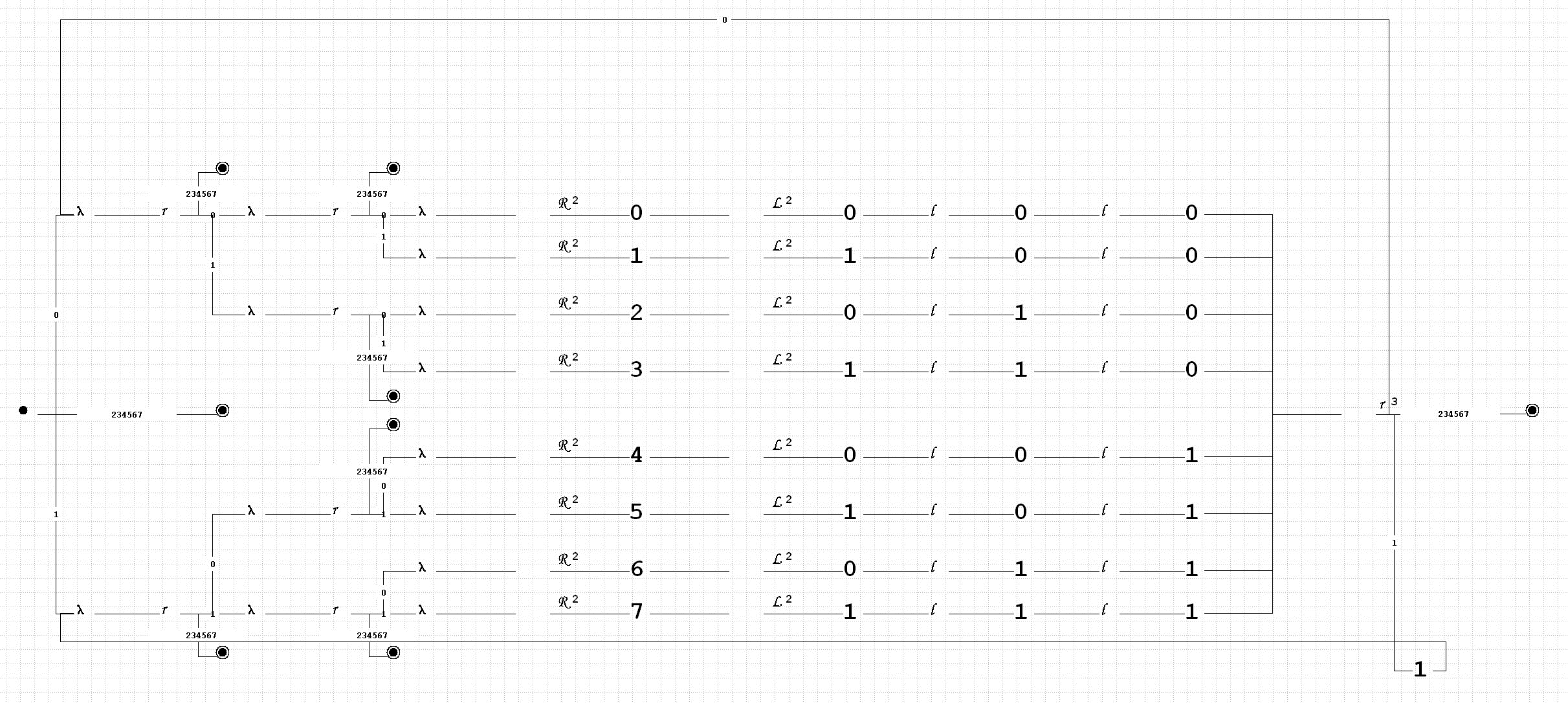
Пришло перейти к самым важным подмашинам в этой программе – машинам перевода числа из двоичной системы счисления в восьмеричную по прямому и дополнительному коду.

Как обычно, принцип работы диаграммы буду объяснять для случая с первой цифрой ноль в числе. В случае с единицей изменится только таблица соответствия.

Таблица соответствия для прямого кода:

|  |  |
| --- | --- |
| Двоичная система счисления | Восьмеричная система счисления |
| 000 | 0 |
| 001 | 1 |
| 010 | 2 |
| 011 | 3 |
| 100 | 4 |
| 101 | 5 |
| 110 | 6 |
| 111 | 7 |

Сама диаграмма:



С виду диаграмма большая, но на самом деле операции здесь однотипные. Поведение головки я объясню на примере нуля (значащего). В начале работы подмашины головка стоит на первой цифре из трёх, которые описывают восьмеричное число. В случае, если это ноль, головка запоминает этот факт, стирает цифру и делает шаг вправо. Если и вторая цифра ноль, головка запоминает и это, стирает цифру и шагает вправо к третьей цифре. Только в случае, если третье число – это ноль, головка после стирания последней цифры и прихода в конец вторгого слова (если его нет, головка остановится через пробел от входного слова) поставит туда ноль. Иначе, в данном случае, будет поставлена единица.

После постановки цифры в конец вторгого слова головка вернётся туда, где она ранее стёрла цифры и вернёт их на место. После этого делается три шага вправо и процесс повторяется снова, пока цифры не кончатся. В случае, если они закончились, работа подмашины сразу же завершается.

С диаграммой inverted\_two\_to\_eight ситуация аналогичная:

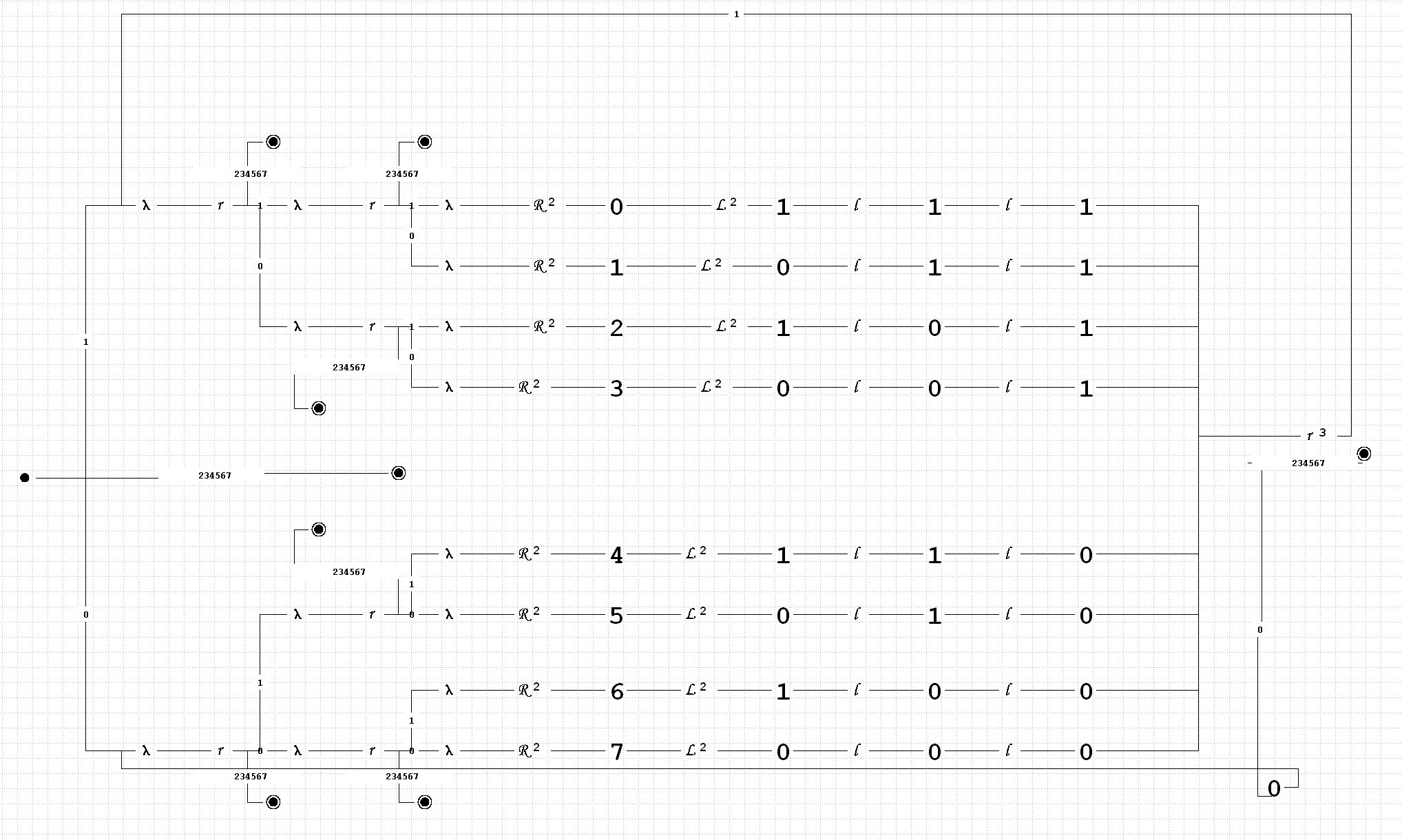


Таблица соответствия для дополнительного кода:

|  |  |
| --- | --- |
| Двоичная система счисления | Восьмеричная система счисления |
| 111 | 0 |
| 110 | 1 |
| 101 | 2 |
| 100 | 3 |
| 011 | 4 |
| 010 | 5 |
| 001 | 6 |
| 000 | 7 |

Если честно, я не имею ни малейшего представления о том, как можно было бы упростить работу данных подмашин. Кажется, алгоритма проще и быстрее придумать невозможно (я имею в виду алгоритм для числа любой длины). Поэтому я могу со спокойной совестью перейти к оценке сложности данной диаграммы Тьюринга.

2.6. Сложностная оценка диаграммы Тьюринга и выводы

Как и в случае с машиной Тьюринга сложность я буду оценивать на основе данных измерений зависимости скорости выполнения программы от длины строки.

|  |  |
| --- | --- |
| Входные данные | Время выполнения, с |
| 0\_111 | 4,5 |
| 1\_010\_100 | 7 |
| 0\_100\_011\_010 | 9,5 |
| 1\_010\_101\_000\_111 | 13 |
| 0\_100\_011\_010\_001\_101 | 16,5 |
| 1\_010\_101\_000\_111\_011\_101 | 21 |
| 0\_100\_011\_010\_001\_101\_000\_111 | 25 |

По данным таблице видно, что сложность алгоритма колеблется около O(n). Я думаю, что для диаграммы Тьюринга это неплохой результат. Да, я не использовал защитные диаграммы в анализе: все числа в полном порядке. Это усложнило бы определение сложности алгоритма из-за неоднотипности задач. То же игнорирование незначащих нулей сильно ускоряло бы работу программы, а их добавление, наоборот, тормозило бы нахождение результата. Ни о каком справедливом ответе и речи бы не шло в данном случае.

Стоит отметить, что работа с диаграммами очень сильно ускорило создание алгоритма, хотя он и сложнее предыдущего. Я потратил на целый день меньше, так что слова о «человеческом лице» диаграмм Тьюринга себя оаправдали.

На этом демонстрация определения алгоритмов через алгоритмические модели диаграмм Тьюринга заверешено.

3. Нормальные алгоритмы Маркова (НАМ)

3.1. Теоретическая часть, связанная с НАМ

Нормальный алгоритм Маркова (НАМ) представляет собой упорядоченный набор правил-продукций — пар слов (цепочек знаков, в том числе пустых цепочек длины 0), соединенных между собой символами → или 7 →. Каждая продукция представляет собой формулу замены части входного слова, совпадающей с левой частью формулы, на ее правую часть. И левая, и правая части продукций могут быть пустыми: либо выполняется безусловная подстановка правой части, либо удаляется часть исходного слова. Однако поскольку пустое слово присутствует и слева, и справа от каждой буквы преобразуемого слова, то подстановка с пустой левой частью зацикливается, и соответствующий алгоритм неприменим ни к какому входному слову. Если удается применить какую-то формулу подстановки, заменив вхождение ее левой части в исходном слове на правую часть, происходит возврат в начало алгоритма, и снова ищутся вхождения левой части первой продукции в измененное слово. Если же какую-то продукцию не удалось применить, проверяется следующая за ней, и так далее. Процесс выполнения нормального алгоритма заканчивается в одном из двух случаев: либо все формулы оказались неприменимыми, т. е . в обрабатываемом слове нет вхождений левой части ни одной формулы подстановки; либо только что применилась так называемая терминальная (завершающая) продукция, в которой правую и левую часть разделяет символ 7 →. Терминальных продукций в одном НАМ может быть несколько.

Стоит ещё упомянуть и порядок обработки правил.

1. если применимо несколько правил, то берется правило, которое встречается в описании алгоритма первым;

2. если правило применимо в нескольких местах обрабатываемого слова, то выбирается самое левое из этих мест.

Существует два простых достаточных признака применимости НАМ ко всем входным словам:

1. левые части всех продукций непустые, а в правых частях нет букв, входящих в левые части;

2. в каждом правиле правая часть короче левой.

Необходимость в этих правилах и признаках возникла из-за отсутствия в нормальных алгоритмах Маркова головки, курсора или ещё чего-нибудь, что обозначало бы расположение рабочей ячейки, как в машинах и диаграммах Тьюринга. В результате этого, реализация алгоритмов Маркова очень отличается от Тьюринга даже на уровне идеи, что отразилось и на моей работе. Даже формат вывода отличается (в случае с Марковым он может быть ненормированным, а значит мне не нужно ничего копировать)

Итак, моей задачей было создание нормального алгоритма Маркова, меняющего местами два троичных числа, разделённых знаком “^”.

Как обычно, перед осмотром программы нужно прописать пару тестов.

|  |  |
| --- | --- |
| Входные данные | Выходные данные |
| 0^1 | 1^0 |
| 12^210 | 210^12 |
| 12^10 | 10^12 |
| 1221^2112 | 2112^1221 |
| 111^000 | 0^111 |
| 000^000 | 0^0 |
| 212^012 | 12^212 |

3.2. Демонстрация сделанного НАМ

На этот раз алгоритм у меня получился небольшой, поэтому я продемонстрирую его целиком, постепенно объясняя смысл работы каждого блока правил.

Но сначала хотелось бы объяснить вкратце, что этот алгоритм делает:

1. Удаляются все пробелы, которые может поставить проверяющий, чтобы сломать программу.
2. Ставится \* перед первой цифрой
3. Первая цифра обозначается \* с двух сторон (\*1\*01)
4. Начинается перенос вправо первой цифры при помощи \*
5. Как только мы доходим до конца второго числа, после него ставим # и переносим нашу цифру через неё, но уже со знаками !
6. При помощи ! с двух сторон от цифр возвращаемся в начало первого числа и берём следующую цифру
7. Повторяем пункты 3-6 до тех пор, пока все цифры первого числа не окажутся справа от #
8. После этого переносим знак ^ вправо и ставим его вместо # при помощи символа $
9. Завершаем работу программы

А теперь демонстрирую уже сам алгоритм с краткими пояснениями

λ->

удаление всех пробелов (защитный блок на случай, если кто-то неправильно введёт числа)

\*0\*^->^\*0\* перенос цифры через домик

\*1\*^->^\*1\*

\*2\*^->^\*2\*

\*0\*0->0\*0\* перенос цифры вправо

\*0\*1->1\*0\*

\*0\*2->2\*0\*

\*1\*0->0\*1\*

\*1\*1->1\*1\*

\*1\*2->2\*1\*

\*2\*0->0\*2\*

\*2\*1->1\*2\*

\*2\*2->2\*2\*

0!0!->!0!0 перенос моего "курсора" влево

0!1!->!0!1

0!2!->!0!2

1!0!->!1!0

1!1!->!1!1

1!2!->!1!2

2!0!->!2!0

2!1!->!2!1

2!2!->!2!2

\*0\*#0->#0$0$ перенос цифры в начало скопированного числа

\*0\*#1->#1$0$

\*0\*#2->#2$0$

\*1\*#0->#0$1$

\*1\*#1->#1$1$

\*1\*#2->#2$1$

\*2\*#0->#0$2$

\*2\*#1->#1$2$

\*2\*#2->#2$2$

$0$0->0$0$ перенос цифры вправо в скопированном числе

$0$1->1$0$

$0$2->2$0$

$1$0->0$1$

$1$1->1$1$

$1$2->2$1$

$2$0->0$2$

$2$1->1$2$

$2$2->2$2$

$0$->!0! установка цифры в конец скопированного числа и подготовка к движению влево

$1$->!1!

$2$->!2!

\*0\*->#!0! установка самой первой цифры в скопированном числе

\*1\*->#!1!

\*2\*->#!2!

0#!0!->!0!#0 переход "курсора" во второе число

0#!1!->!0!#1

0#!2!->!0!#2

1#!0!->!1!#0

1#!1!->!1!#1

1#!2!->!1!#2

2#!0!->!2!#0

2#!1!->!2!#1

2#!2!->!2!#2

0^!0!->!0!^0 переход "курсора влево" в первое число

1^!0!->!1!^0

2^!0!->!2!^0

0^!1!->!0!^1

1^!1!->!1!^1

2^!1!->!2!^1

0^!2!->!0!^2

1^!2!->!1!^2

2^!2!->!2!^2

$0^#->.0^ удаление лишних символов в конце работы алгоритма, завершение работы

$1^#->.1^

$2^#->.2^

$0^0->0$0^ перемещение домика вправо к началу скопированного числа

$0^1->0$1^

$0^2->0$2^

$1^0->1$0^

$1^1->1$1^

$1^2->1$2^

$2^0->2$0^

$2^1->2$1^

$2^2->2$2^

^!0!0->^!0! удаление незначащих нулей во втором числе

^!0!1->^!1!

^!0!2->^!2!

^!0!->$0^ начало переноса домика после окончания копирования первого числа

^!1!->$1^

^!2!->$2^

\*00->\*0 удаление незначащих нулей в первом числе

\*01->\*1

\*02->\*2

\*0->\*0\* пометка звёздочками цифры первого числа, подлежащей копированию

\*1->\*1\*

\*2->\*2\*

!0!->\*0\* окончание перехода в начало первого числа, начало копирования следующей цифры

!1!->\*1\*

!2!->\*2\*

->\* установка звёздочки перед первой цифрой

Думаю, теперь стоит обратить внимание на сложность данного алгоритма.

3.3. Сложностная оценка сделанного НАМ и выводы

Уже не вижу смысла рассказывать о том, как я буду оценивать сложность алгоритма. Скажу лишь то, что я не буду ставить в числа незначащие нули и пробелы, чтобы обеспечить хоть какую-нибудь точность вычислений. Стоит просто предоставить таблицу.

|  |  |
| --- | --- |
| Входные данные | Время выполнения, с |
| 1^2 | 2 |
| 11^22 | 4 |
| 111^222 | 6,5 |
| 1111^2222 | 9,5 |
| 11111^22222 | 13 |
| 111111^222222 | 17,5 |
| 1111111^2222222 | 22 |

В данном алгоритме Маркова сложность оказалась O(n\*log(n)). Считаю, что для посимвольного переноса цифр это хороший результат, также, как и в машине Тьюринга. Стоит отметить, что при выполнении данной работы простые методы отладки программы не помогали. Я не мог просто взять и усложнить работу алгоритма в проблемном месте, чтобы избежать ошибок. Приходилось по много раз переписывать этот крошечный набор правил с нуля, а ближе к концу работы ещё нужно было частенько менять их местами, ведь в нормальных алгоритмах Маркова порядок следования правил очень важен.

На этом демонстрация определения через НАМ завершена. Пришло время делать окончательные выводы в этой курсовой работе.

Заключение

Итак, в этой курсовой работе я проиллюстрировал определения алгоритма путём построения алгоритмических моделей Тьюринга и Маркова. Как оказалось, это определение каждый раз отличалось.

В случае с машиной Тьюринга алгоритмом называлась совокупность состояний, в которых описаны конкретные действия, переходя через которые головка выполняла определённые операции, приводящие программу к решению поставленной задачи.

В диаграммах Тьюринга вместо состояний головка ориентировалась на последовательность элементарных машин Тьюринга и сделанных программистом подмашин, соединённых между собой определённым образом.

В нормальных алгоритмах Маркова результат работы достигался путём последовательного выполнения правил, которые постепенно преобразовывали входную строку нужным нам образом.

К этим умозаключениям я пришёл во время выполнения задач курсовой работы. Все они были мною успешно проделаны.

На этом мои идеи по добавлению воды в курсовую закончились. По объёму план даже слегка перевыполнил, требования все соблюдены. Надеюсь, эта работа проверку пройдёт.