ПОЛНОТА КОМАНДЫ MOV ПО ТЬЮРИНГУ

БЕРЕЗОВСКАЯ ЕЛИЗАВЕТА

324 ГРУППА

ВВЕДЕНИЕ

Препринт Stephen Dolan **mov is Turing-complete (2013)**

URL: **https://web.archive.org/web/20210214020524/https://stedolan.net/research/mov.pdf**

Стивен Долан (Кембриджский университет) занимается теорией и практикой формальных методов программирования, исследует типовые системы, семантику языков и фундаментальные свойства вычислений.

В 2017 году стал лауреатом премии CPHC/BCS за выдающуюся диссертацию благодаря своей работе «Алгебраическое подтипирование».

ПОЛНОТА ПО ТЬЮРИНГУ

Вычислимая функция — математическая функция, значение которой может быть получено посредством эффективной процедуры — алгоритма.

Полнота по Тьюрингу – характеристика исполнителя, означающая возможность реализовать на нём любую вычислимую функцию.

Впервые термин «полнота по Тьюрингу» был упомянут в работе «Вычислимость рекурсивных функций» (1963) - Дж.Шепердсон и Х.Стерджис.

Исполнитель обладает **свойством полноты по Тьюрингу**, если на нём можно реализовать любой алгоритм.

ОСНОВНАЯ ИДЕЯ – ЭМУЛЯЦИЯ МАШИНЫ ТЬЮРИНГА

Главная идея: построить в памяти структуры, эквивалентные компонентам Тьюринг-машины и реализовать все операции только через команду **mov**.

Элементы машины, которые нам будет необходимо промоделировать:

1. Лента – бесконечный массив ячеек.

2. Головка – указатель, последовательно перемещающийся по ячейкам вправо или влево, который может менять её.

3. Алфавит символов – конечное множество допустимых символов.

4. Состояния машины – конечное множество состояний, в котором может находиться машина.

5. Таблица переходов – которая определяет поведение машины в зависимости от текущего состояния и символа, прочитанного головкой.

ОСОБЕННОСТИ ЭМУЛИРУЕМОЙ МАШИНЫ ТЬЮРИНГА

1. Полубесконечная лента, которая расширяется вправо.

a	b	c	d	e	…

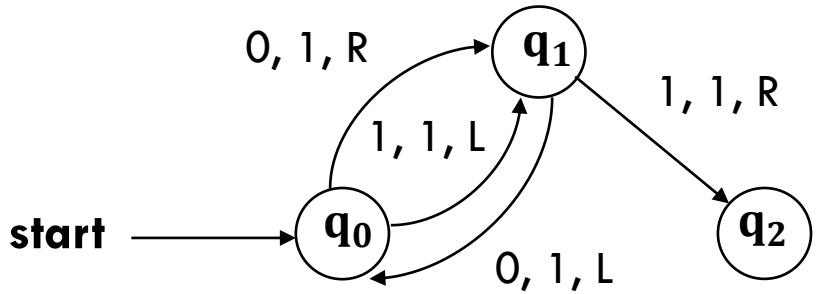
2. Головка машины всегда сдвигается либо влево, либо вправо, нет опции «остаться на месте».

3. Завершение программы реализовано нестандартно.

Программа машины Тьюринга в виде таблицы

	0	**1**
$\mathbf{q_0}$	1, R, q_1	1, L, q_1
$\mathbf{q_1}$	1, L, q_0	1, R, !

В виде графа



ИСПОЛЬЗУЕМЫЙ ИНСТРУМЕНТАРИЙ

Архитектура – на базе x86:

1. Память:

- Используется прямая адресация
- Важны не значения в памяти, а адреса, по которым мы к ним обращаемся

2. Регистры

Инструменты:

1. Три формы команды mov:

- mov R_{dest}, c ; запись константы
- mov R_{dest}, $[R_{src} + R_{offset}]$; чтение из памяти
- mov $[R_{dest} + R_{offset}]$, R_{src} ; запись в память

2. Безусловный переход **jmp** для циклического выполнения алгоритма машины Тьюринга.

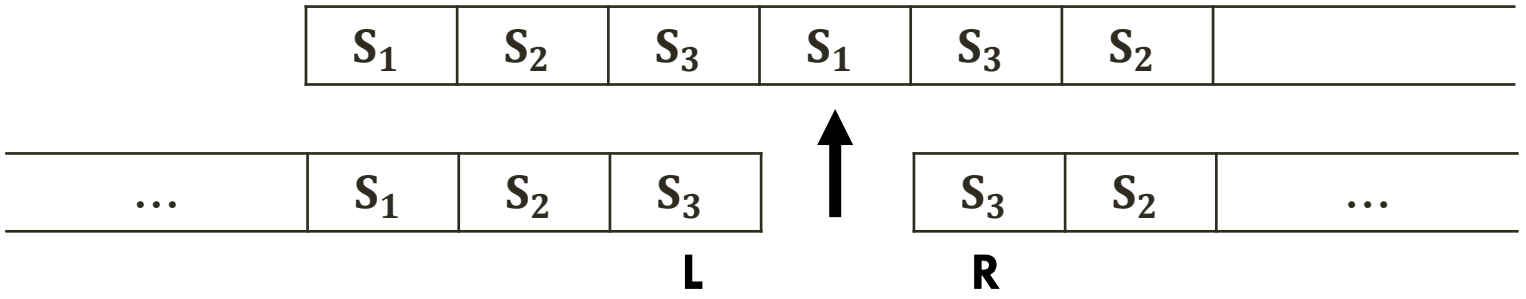
СТРУКТУРЫ ДАННЫХ В ПАМЯТИ

1. Символы: алфавит машины Тьюринга состоит из символов $\boldsymbol{\sigma_0}$, $\boldsymbol{\sigma_1}$, …, $\boldsymbol{\sigma_n}$. Каждому символу соответствует фиксированный адрес в памяти:

- Символ $\boldsymbol{\sigma_0}$ → адрес $\mathbf{S_1}$
- Символ $\boldsymbol{\sigma_1}$ → адрес $\mathbf{S_2}$

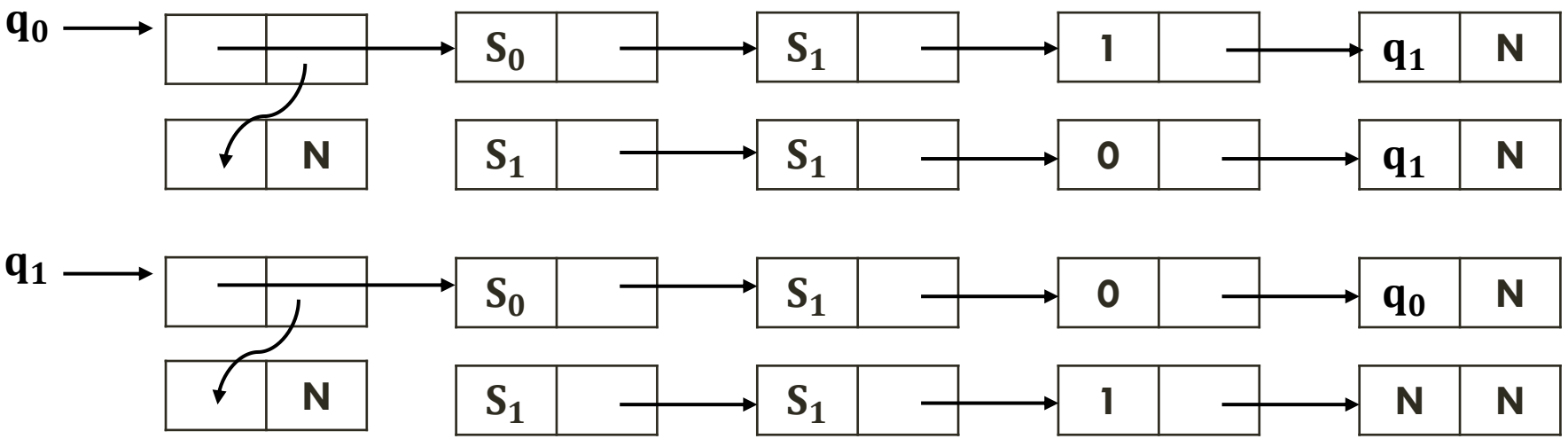
  …
- Символ $\boldsymbol{\sigma_n}$ → адрес $\mathbf{S_{n+1}}$

2. Лента: два связных списка и текущая позиция на нём



СТРУКТУРЫ ДАННЫХ В ПАМЯТИ

3. Состояния: связные списки



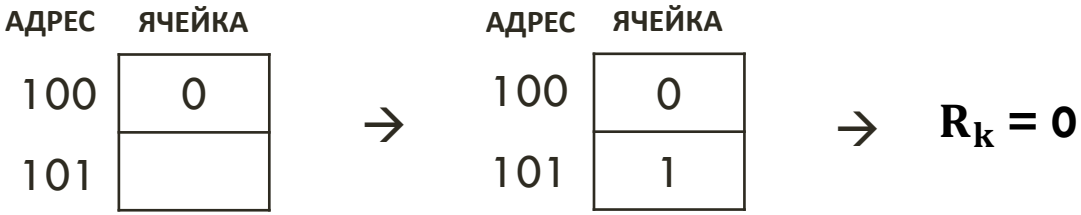
СРАВНЕНИЕ БЕЗ УСЛОВНЫХ ОПЕРАТОРОВ

mov $[R_i]$, 0

mov $[R_j]$, 1

mov R_k, $[R_i]$

1.



→ → $\mathbf{R_k = 0}$

2.

→ → $\mathbf{R_k = 1}$

Обозначения:
$\mathbf{R_i}$– регистр, который хранит адрес памяти или константы
$\mathbf{[R_i]}$ – значения ячейки памяти по адресу, хранящемуся в $\mathbf{R_i}$

Результат:

- $\mathbf{R_k = 0}$, если адреса разные
- $\mathbf{R_k = 1}$, если адреса одинаковые

УСЛОВНЫЕ ВЫРАЖЕНИЯ БЕЗ УСЛОВНЫХ ОПЕРАТОРОВ

Пытаемся реализовать: if R_i != R_j then R_l := B else R_l := A

mov [N], R_A

mov [N + 1], R_B

mov R_l, [N + R_k]

$$R_k = \begin{cases} 0, & R_i \mathrel{!}= R_j \\ 1, & R_i = R_j \end{cases} \rightarrow R_l = \begin{cases} A, & R_i \mathrel{!}= R_j \\ B, & R_i = R_j \end{cases}$$

Обозначения:
N – адрес ячейки памяти
[N] – значения ячейки памяти по адресу, хранящемуся в **N**

N	A
N + 1	B

Результат:

- Если $\mathbf{R_k = 0}$, то вычисляется адрес **N + 0 = N**, и в $\mathbf{R_l}$ попадает то, что находиться в памяти по этому адресу – **A**
- Если $\mathbf{R_k = 1}$, то в $\mathbf{R_l}$ попадает то, что находиться в памяти уже по адресу **N + 1** – **B**

СИМУЛЯЦИЯ ОДНОГО ШАГА МАШИНЫ ТЬЮРИНГА

Определим назначение регистров в памяти:

T – текущее состояние

S – указатель на текущую ячейку ленты (адрес)

L – список, представляющий часть ленты слева от текущей позиции

R – список, представляющий часть ленты справа от текущей позиции

N – адрес пустого списка (NULL), служащий маркером конца списка

Начальные состояние регистров: **T** $= \mathbf{q_0}$ **S** = T1 **L** = **N** **R** = T2

N – тот самый пустой список, служащий маркером
$\mathbf{q_0}$ - адрес списка переходов начального состояния

T1 – первая ячейка ленты (текущая позиция)
T2 – правая часть ленты: начинается со второй ячейки

АЛГОРИТМ

1. Проверка соответствия текущего перехода

2. Выполнение перехода

3. Перемещение по ленте

4. Добавление текущего символа в стек **L** или **R**

5. Отмена движения при несоответствии

6. Извлечение нового **S** и обновление **L**/**R**

7. Поиск следующего перехода

8. Проверка на остановку

9. Реализация завершения и цикла

1. ПРОВЕРКА СООТВЕТСТВИЯ ТЕКУЩЕГО ПЕРЕХОДА

```
mov X, [T]          ; достаем переход
mov X, [X]          ; достаем символ, на который реагирует переход
mov Y, [S]          ; достаем текущий символ на ленте
mov [Y], 0          ; записываем 0 по адресу текущего символа на ленте
mov [X], 1
mov M, [Y]          ; снова читаем значение по адресу текущего символа
```

Обозначения:
T – текущее состояние
S – указатель на текущую ячейку ленты (адрес)
M – флаг совпадения

Результат:

- **M = 1**, если символы совпадают
- **M = 0**, если символы разные

2. ВЫПОЛНЕНИЕ ПЕРЕХОДА

```
mov X, [T]          ; достаем переход
mov X, [X + 1]      ; пропускаем символ, на который реагирует переход
mov X, [X]          ; загружаем новый символ
mov Y, [S]          ; загружаем старый символ
mov [N], Y          ; выбираем между X и Y
mov [N + 1], X
mov Z, [N + M]
mov [S], Z          ; записываем выбранный символ обратно в текущую ячейку ленты
```

Обозначения:
T – текущее состояние
S – указатель на текущую ячейку ленты (адрес)
M – флаг совпадения

Результат:

- Если **M = 1** → переход проходит и в **S** записывается новый символ
- Если **M = 0** → символ остается старым

3. ПЕРЕМЕЩЕНИЕ ПО ЛЕНТЕ

```
mov D, [T]          ; достаем переход
mov D, [D + 1]      ; пропускаем символ, на который реагирует переход
mov D, [D + 1]      ; пропускаем новый символ
mov D, [D]          ; загружаем направление
```

Обозначения:
D – направление
T – текущее состояние

Результат:

- Если **D = 0** → смещаемся влево
- Если **D = 1** → смещаемся вправо

4. ДОБАВЛЕНИЕ ТЕКУЩЕГО СИМВОЛА В СТЕК L ИЛИ R

```
mov [N], R          ; выберем новое значение для [S + 1]
mov [N+1], L
mov X, [N + D]
mov [S + 1], X      ; установим ссылку на соответствующий стек
mov [N], L          ; выберем новое значение для L
mov [N + 1], S
mov L, [N + D]
mov [N], S          ; выберем новое значение для R
mov [N + 1], R
mov R, [N + D]
```

Обозначения:
D – направление
S – указатель на текущую ячейку ленты (адрес)

Пояснения: Если направо (**D = 1**) → добавляем старое значение **S** в **L**.

Если налево (**D = 0**) → добавляем старое значение **S** в **R** и берем новый **S** из **L**.

5. ОТМЕНА ДВИЖЕНИЯ ПРИ НЕСООТВЕТСТВИИ

```
mov [N], 1          ; установим X = not D
mov [N+1], 0
mov X, [N + D]
mov [N], X          ; выберем между D и X
mov [N+1], D
mov D, [N + M]
```

Обозначения:
D – направление
M – флаг совпадения

Пояснения:

- Если **M = 1**, это значит, что переход совпал и мы продолжаем двигаться дальше
- Если **M = 0** – переход не совпал и необходимо откатиться назад

6. ИЗВЛЕЧЕНИЕ НОВОГО S И ОБНОВЛЕНИЕ L/R

```
mov [N], L          ; выберем новое значение для S
mov [N+1], R
mov S, [N + D]      ; выбирает: если D = 0 → S = L, если D = 1 → S = R
mov X, [S + 1]
mov [N], X          ; выберем новое значение для L
mov [N+1], R
mov L, [N + D]
mov [N], R          ; выберем новое значение для R
mov [N+1], X
mov R, [N + D]
```

Обозначения:
S – указатель на текущую ячейку ленты (адрес)
D – направление

Пояснение:

Извлекаем новый текущий элемент **S** из стека (соседний) и обновляем головы списков.

7. ПОИСК СЛЕДУЮЩЕГО ПЕРЕХОДА

```
mov X, [T + 1]      ; достаем следующий переход в текущем состоянии
mov Y, [T]          ; достаем текущий переход
mov Y, [Y + 1]      ; пропускаем символ, на который реагирует переход
mov Y, [Y + 1]      ; пропускаем новый символ
mov Y, [Y + 1]      ; пропускаем направление
mov Y, [Y]          ; загружаем список состояний
mov [N], X          ; выбираем следующий переход
mov [N + 1], Y
mov T, [N + M]
```

Обозначения:
T – текущее состояние
M – флаг совпадения

Пояснение:

Выбираем новую **T** на основе **M**: если **M = 1** – переход сработал, переходим в новое состояние, если **M** = **0** – продолжаем со следующего перехода в том же состоянии.

8. ПРОВЕРКА НА ОСТАНОВКУ

```
mov X, [T]        ; сохраняем изначальное значение T в X
mov [N], 0        ; записываем 0 по адресу N
mov [T], 1
mov H, [N]        ; читаем значение по адресу N
mov [T], X        ; восстанавливаем T
```

Обозначения:
T – текущее состояние
H – флаг остановки

Пояснение:

Если **T = N –** это означает, что либо список переходов закончился, либо новое состояние не имеет переходов – машина должна остановиться.

9. РЕАЛИЗАЦИЯ ЗАВЕРШЕНИЯ И ЦИКЛА

```
mov [N], 0          ; выбираем между 0 и N
mov [N+1], N
mov X, [N + H]
mov X, [X]          ; загружаем 0 или N
jmp start
```

Обозначения:
N – специальный адрес: маркер конца списка
H – флаг остановки

Пояснение:

Программа выбирает адрес **N** вместо адреса начала программы – это означает завершение работы. Таким образом, если **H = 0** → продолжение цикла – **jmp start**, если **H = 1** → завершение программы.

ЗАКЛЮЧЕНИЕ

Потенциальные практические плюсы использования такой архитектуры:

1. Упрощение декодера инструкций → снижение энергопотребления

2. Отсутствие АЛУ и сложных функциональных блоков → больше места под кэш и память

3. Упрощение архитектуры → более простая реализация и производство

4. Сохранение вычислительной универсальности при минимальном наборе инструкций

Мои выводы:

С одной инструкцией архитектура проще, дешевле и потенциально надёжнее, но сложность смещается в программную часть: для обычной логики и ветвлений нужен сложный код и организация данных. Вычислительная мощность при этом остаётся прежней – mov по-прежнему может описать любую вычислимую программу.

ЛИТЕРАТУРА

1. Полнота по Тьюрингу // Википедия. URL: https://ru.wikipedia.org/wiki/Полнота_по_Тьюрингу

2. Вычислимая функция // Википедия. URL: https://ru.wikipedia.org/wiki/Вычислимая_функция

3. Shepherdson J.C., Sturgis H.E. Computability of Recursive Functions // Journal of the ACM. 1963. URL: https://www.cs.cmu.edu/~15455/resources/ShepherdsonSturgis1963.pdf