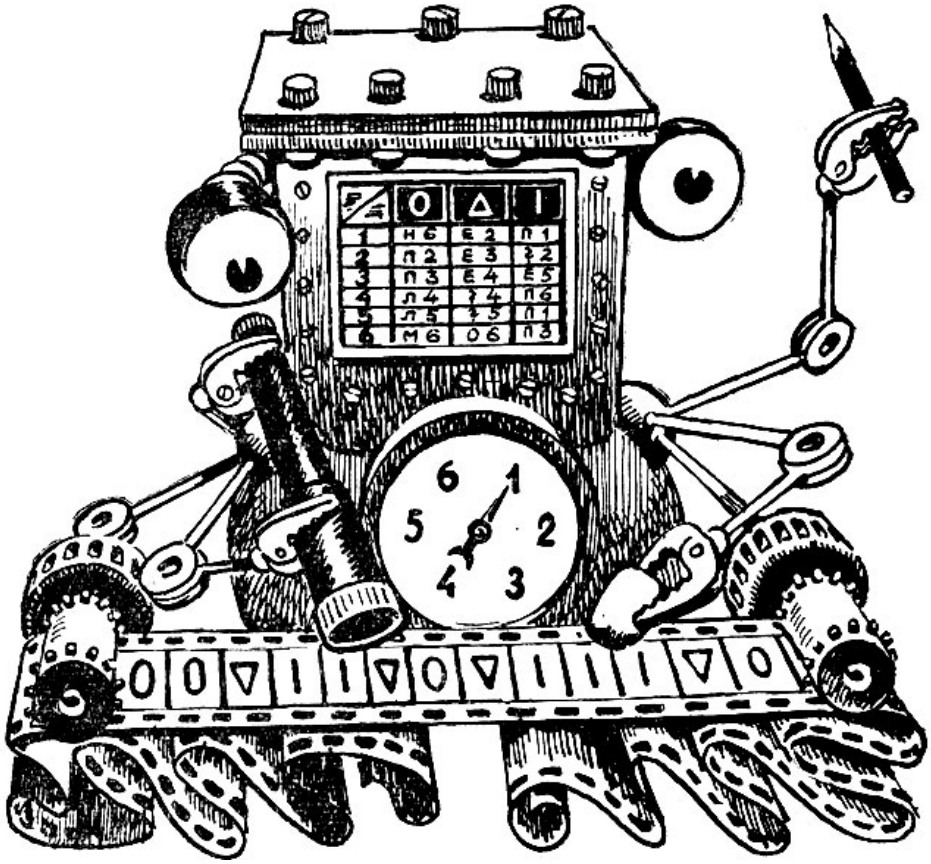


# Інструкція до емулятору машини Тьюрінга.

Булак А.С.

КА-04 (2020)

Київ 2021

Зміст:

**1.Про мову емулятора:**
1. Для чого він та чому він такий?
2. Структура команди.
3. Структура програми.
4. Макроси та де вони макро-.
5. Коментарі.

**2.Консольна версія:**
1. Як використовувати консольну версію.
2. Як працює debugger.

**3.Графічна версія:**
1. Що треба знайти, щоб уникнути помилок.
2. Загальні підходи до проектування GUI.
3. Про структуру GUI.
4. Hotkeys.

**4.Додатки:**
Порт з Оніщенка.
Приклад композиції машин

## 1.1. Для чого він та чому він такий?

Щоб розуміти якесь рішення, треба розуміти його природу, причини з яких воно склалося. Ось і ми, у цій коротенький брошурі, підемо шляхом хоробрих архітекторів: будемо стикатися з проблемами та вирішувати їх, зазвичай, найпростішим шляхом.

Всі знають що таке машина Тьюрінга, сьогодні весь ІПСА користується або емуляторами, що не мають достатньо розвинутих аналізаторів, або таким, де команди задаються таблицею. Особисто мені не подобається писати у таблицях, це не дуже інтуїтивно зрозуміло та вимагає використання мишки, навіть при наявності hotkeys. Емулятор Оніщенка хороший, але має купу фатальнх недоліків: крім нестабільності, через яку можна втратити роботу декількох годин, він не має

зручного формату зберігання та, що найважливіше, не має коментарів, через що вже після написання 20 рядків, программа стає темним та страшним лісом. Я вирішив взяти ідею Оніщенка, та переписати його так, щоб мені самому було приємно робити домашні та індивідуальні завдання.

Значною мірою на емулятор вплинула ідея, використання його на UNIX-подібних операційних системах. Як не дивно, я не знайшов жодного такого емулятора, не впевен що вони взагалі є. Як всі ми знаємо - до ОС Windows світ розмовляв з комп'ютерами не жестами, повертаючись у початок людської історії, а словами. Звідси ще одна вимога для емулятора - можливість роботи з терміналу.

Згадавши ті 3 рядки, що я писав на ассемблері, та мовою C, я одразу визначився з дизайном мови, отже знання цих мов і редактору Vim значно полегшить розуміння.

### 1.2. Структура команди.

Ідея виду команд взята з Оніщенка, але, треба зауважити, що емулятор Оніщенка не вміє маніпулювати іменами довше 6 символів, та не вміє працювати з пробілами, що нестерпно погано.

Зформулюємо загальний вид команди:

```
state1, word1 -> state2, word2, direction
```

Всі state1, state2 мають складатися тільки з латинських літер верхнього та нижнього регістру, цифр та символу нижнього підкреслення "_".

У word1, word2 також допустимі латинські літери верхнього та нижнього регистру, але тільки такі спецсимволи: $+, -, /, *, =, :, \hat{}, \#, !, ?, \&$.

Не мають обмежень щодо довжини (не гарантую більше 256 символів).

"direction"може мати тільки такі значення $\{R, r; L, l; S, s\}$.

Отже:

| Тип імені | Множина допустимих символів |
|---|---|
| state1, state2 | латинські літери верхнього та нижнього регистру, числа та символ нижнього підкреслення "_". |
| word1, word2 | латинські літери верхнього та нижнього регистру, числа, спецсимволи: $+, -, /, *, =, :, \hat{}, \#, !, ?, \&$. |
| direction | $\{R, r; L, l; S, s\}$ |

### 1.3. Структура програми.

Отже, через необхідність роботи з терміналом, нам треба спроектувати мову так, щоб ми мали можливість зберігати у одному текстовому файлі вхідні данні та програму. У той же час, ці дві речі не мають прямого, суто синтаксичного зв'язку, тобто вони мають бути якось відділені один від одного у окремих областях файла.

Хорошим прикладом рішення такої проблеми є мови ассемблерів, що зберігають в одному текстовому файлі текст програми та вхідні дані у окремих "секціях". Це задовольняє нашим вимогам.

Будь-яка програма на мові ассемблерів має особливий секційний вид, що і дозволяє їй зберігати всі ці данні впорядковано, використовуючи лише можливості

таких мінімалістичних, навіть бідних мов, як ассемблери. Будь-який ассемблер має три види секцій. Наша мова, потребує всього два. Перша секція буде традиційно називатися :

```
section .data
```

Вона, відповідно до своєї назви, буде зберігати у собі дані. Уважні читачі зададуться питанням: а як можна зберегти нескінченну стрічку, яка розділена на клітинки та має одну особливу, на якій стоїть курсор (зчитувальна голівка)?

Цим питанням задався і я. Я захотів щоб це було красиво, очевидно та просто.

Заради елегантності, роздільником клітинок був обраний пропуск (" ").

Задавання курсора повинно виділятися та бути завжди помітним. На мою думку, краще за всі інші символи в тексті видно ("|").

Традиційно склалося, що термінали мають ширину у 70-80 знакомісць (треба сказати, що графічний інтерфейс з частини 3 має 200+ знаків у input рядку), чого повинно бути достатньо для невеликої (а може і великої) програми.

Ще одне цікаве питання - пустий символ. Для наочності ним був обраний символ "lambda", слово латинськими маленькими літерами.

Що ж ми спроектували? Ось приклад реальної секції:

```
section .data
|1| 0 lambda 1 1 1 0 *
```

Не найприємніше у житті, але й життя не завжди цукерка.

Другою традиційною секцію таких мов є :

```
section .text
```

Що, відповідно до назви, є секцією тексту програми.

Важливо зауважити, що серед *.data* обирається остання, а всі *.text* зливаються в одну. Кінцем секції вважається початок іншої або кінець файла.

Розібралися з секціями, а питання таки залишилися - як почати та закінчити программу? Оберемо початковим станом *"start"*, а кінцевим *"end"*.

Приклад простої програми, що ставить на стрічку одну одиницю:

```
section .data
|lambda|

section .text
start, lambda -> end, 1, s
```

### 1.4. Макроси та де вони макро-.

Всі хто це читатиме вже, знайомі з мовою C++, деякі з читачів, на щастя, знайомі з мовою C, виходить, всі ми чули про макроси. C++ має дуже багато механізмів, частина з яких була створена для того, щоб замінити макроси (наприклад, inline func), в C потужність макросів демонструється у будь-якій більш-менш серйозній програмі.

Макроси нашої мови мають той же механізм роботи, що макропідстановки мови C, тобто заміняють один шматок тексту на інший.

Продемонструємо синтаксис макроса:

```
#define >> ->
```

Аргументи можуть мати довільну довжину. Макрос завжди починається зі слова $\#define$, та між двома словами має бути один пропуск. Макросам все-одно що в них, вони сприймають будь-які символи. Після першого пробілу в тексті частини що "підставляється" можуть будти пропуски, але я не рекомендую так робити.

Область дії макросів - лише section .text.

Продемонструємо справжню силу макросів на прикладі програми xor:

```
#define :: _ZnU11_
#define >> ->

section .data
|1| 0

section .text
start,0->start,0,r ;проходимо вправо від двох бітів.
start,1->start,1,r ;
start,lambda->xor::start,lambda,s

;xor::start
xor::start, lambda >> xor::transit1, lambda, l
xor::transit1, 0 >> xor::nub1_0, lambda, l
xor::transit1, 1 >> xor::nub1_1, lambda, l
xor::nub1_0, 0 >> xor::ret, 0, r
xor::nub1_0, 1 >> xor::ret, 1, r
xor::nub1_1, 0 >> xor::ret, 1, r
xor::nub1_1, 1 >> xor::ret, 0, r
;xor::ret

xor::ret, lambda -> end, lambda, s
```

У прикладі макроси дають замінити стандартний оператор переходу ”->”, та використати для всіх рідний оператор дозволу області видимості ”::”, що не може бути дозволенним у нашій мові.

Я пропоную використовувати нечитаємі комбінації такого виду, парадуючи "маклінг"компіляторів C++, для зменшення ймовірності співпадіння з іншими станами, що не використовують наш макрос.

Таким чином, макроси та структура нашої программи дозволяє нам писати окремі модулі-секції з оговореними стандартними точками входу та виходу, своїми простірами імен.

### 1.5. Коментарі.

Як ви могли помітити у попередній главі, мова має коментарі. Вони, згідно з ассемблерними традиціями, мають однорядковий синтаксис та починаються з символу ”;”.

Наче це все.

**Увага:**

Через різниці в кодуванні тексту, файли з однієї ОС можуть некоректно оброблятися на іншій.

### 2.1. Як використовувати консольну версію.

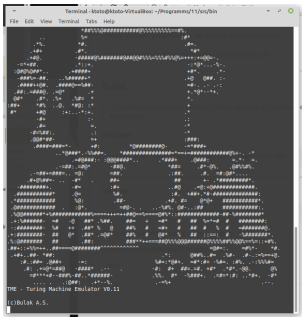
Цей розділ здебільшого написаний для Linux та Mac юзерів.

Після оформлення программи згідно правилам попереднього розділу, треба викликати емулятор з відповідними аргументами. Ось пояснення до їх використання:

"Визначаючим" аргументом є найперший з них.

Якщо першого аргументу взагалі немає, тобто пишемо "./TME" або "*start* TME.exe", то програма виведе інформацію про свою версію та невелику інструкцію англійською.

Якщо "-v" (виходить "./TME -v"), ми можемо дізнатися про версію емілятора та побачити лого.



Якщо перший аргумент це шлях до файлу, ми маємо право вказати "непозиційні аргументи", список яких наведено нижче. Якщо ж ми не додаємо непозиційні, це рівносильно "./TME ../example.txt -g -a -e".

```
example@example-laptop:$ ./TME $path to programm$/programm.tme $KEYS$
```

$KEYS$ = -g, -a, -e, -l, -d

Де:

| | |
|---|---|
| -g | генерація бд команд і інших тимчасових файлів |
| -a | аналіз команд, що підказує, можливо, потрібні команди та вказує на фатальні помилки |
| -e | запускає емулятор, після емуляції буде створений файл $*\_out.txt$ , з вихідним рядком |
| -l | замінює всі входження lambda на пробіли |
| -d | debugger |

### 2.2. Як працює debugger.

Окремо хочеться розповісти про debugger.

Після застосування команди

```
example@example-laptop:$ ./TME $path to programm$/programm.tme -d
```

Пройде генерація, аналіз та емулятор запуститься у debug режимі.

Кожен крок він буде зупинятись, очікуючі вашого вводу. Показувати поточний стан, всю стрічку та поточний символ.

Для продовження роботи дебагера необхідно задати кількість вільних ітерацій цілим числом, після яких він зупиниться та знов передасть вам управління.

Якщо кількість ітерацій перевищила кількість необхідних для виконання програми - емулятор штатно завершиться.

## 3.1. Що треба знати, щоб уникнути помилок.

(GUI - Graphical User Interface)

GUI додатку написаний за допомогою фреймворка Qt, що накладає деякі обов'язки на мене та додає деякі незручності користувачу.

По-перше, дотримуючись ліцензії, я повинен лінкувати додаток динамічно, це ті сами .dll файли, які насправді є бібліотеками, які додаток "підгружає"динамічно, у процесі виконання програми.

Для роботи програми необхідно ніяк не змінювати структуру папки з архіву, в якому поставляється exe файл.

Я дуже рекомендую створити ярлик для цього .exe файла та покласти його у зручне для вас місце.

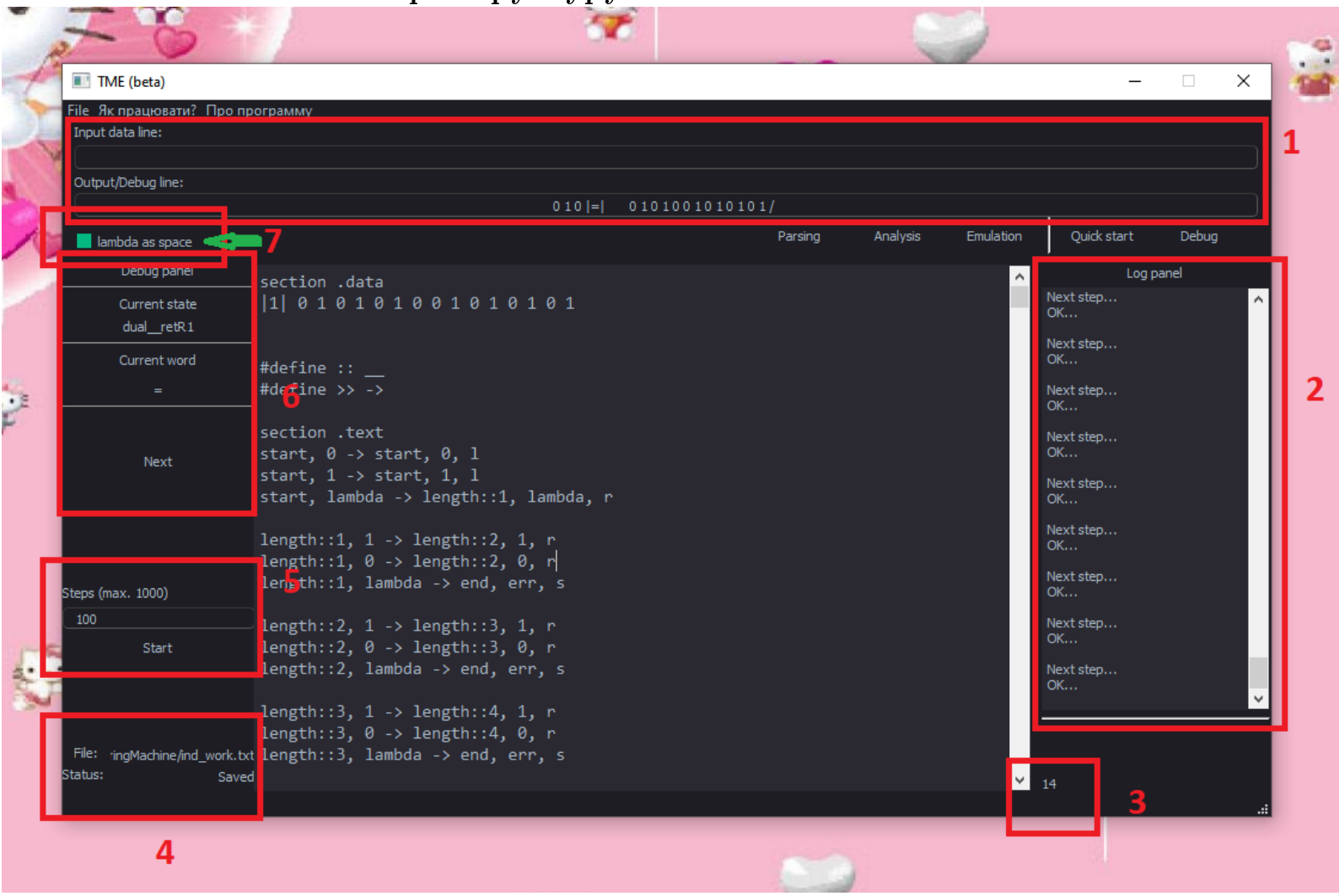
По-друге, я використовую тему Google Material Dark, що може бути незручним, при використанні, наприклад, під рідним нам, палким Кримським сонцем. Щоб відключити тему, треба знайти в executable path файл MaterialDark.qss та якось змінити його назву, наприклад на MaterialDark1.qss. Після цього програма штатно застосує білу тему.

## 3.2. Загальні підходи до проектування GUI.

Будь-яка людська робота, особливо творча, важлива з великої кількості причин. Працюючи на емуляторі Оніщенка, я декілька разів втрачав свої програми. Отже, треба від цього захиститися!

Заради збереження нашого часу та нервів, при роботі з графічним інтерфейсом ми обов'язково повинні спочатку відкрити файл, чи створити його. Це запобігає втраті коду при виникненні непередбачених помилок у емуляторі.

### 3.3. Про структуру GUI.




1. Рядки вводу даних та їх виводу.
2. Панель логів.
3. Покажчик номеру рядка, на якому зараз стоїть курсор.
4. Панель стану файла. Показує його назву та стан.
5. Autodebug panel.
6. Debug panel.
7. Checkbox, що заміняє lambda на ” ”, при наступному виводі.

Важливо приділити увагу 1,5,6 блокам:

(1) має два інпута, перший з яких, при запуску емулятора, просто додається зверху файла з підписом "section.data".

(5) працює тільки у дебаг-режимі, який вмикається кнопкою зверху зліва "debug". Та проводить автоматичні ітерації. Задля безпечної роботи програми, має ліміт у 1000 штук. Не вважаю це проблемою: натиснути 20 разів - це не 20 000.

(6) Дебаг панель, що під часть дебаг-режиму вказує на поточні слово та стан. Кнопка Next робить одну ітерацію, при натисненні.

Quick start - запускає парсинг та емуляцію.

### 3.4. Hotkeys.

Дуже пості та інтуїтивно зрозумілі комбінації, що значно спрощують роботу та дозволяють забути про мишку.

| | |
|---|---|
| Ctrl+S | Зберігає відкритий файл. |
| Ctrl+O | Ініціалізує відкриття файла. |
| Ctrl+N | Ініціалізує створення файла. |
| Ctrl+Shift+X | Натиская Quick start. |
| Ctrl+Tab | аналізує рядок, на якому курсор<br>та доповнює його до форми ", − >, , \n".<br>аналог Tab з емулятора Дікарева. |
| Ctrl+Space | Робить 1 ітерацію у debug режимі. |

**Приємної роботи!**

## Порт з Оніщенка.

```
#define ,: ,lambda:
#define ,, ,lambda,
#define q0 start
#define ! end
#define * star
#define : ->
section .data
|lambda|

section .text
q0,:q1,g,r
q1,:q2,e,r
q2,:q3,o,r
q3,:q4,r,r
q4,:q5,g,r
q5,:q6,e,r
q6,:!,,r
```

**Приклад композиції машин.**

Красивим і простим трюком є написання композиції машин Тьюрінга, де машини виступають підпрограмами з власними "просторами імен", що реалізовані за допомогою макросів. Таким чином, уникнемо перетину внутрішніх алфавітів:

```
#define :: _ppp_

section .data
|lambda| a a b

section .text
start, lambda -> m1::start, lambda, s ; старт

m1::start, lambda -> m1::start, lambda, r ;робота першої машини
m1::start, a -> m1::a, a, r
m1::a, a -> m1::a, a, r
m1::a, b -> m1::a, a, r
::a, lambda -> m2::start, lambda, l

m2::start, a -> m2::start, b, l ;робота другої машини

m2::start, lambda -> end, lambda, s ;кінець
```