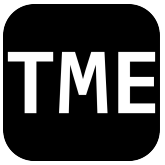


Інструкція
до емулятора машини Тьюрінга

Булак А.С.

Зміст

Вступ

Мета цієї інструкції — опис можливостей TME, покрокове введення читача у використання емулятора (версії 2.0.0b). Контакти для зв'язку можна знайти у кінці інструкції.

Інструкція складається з декількох розділів, кожен з яких об'єднує близькі за темою глави:

Розділ нуль коротко описує варіати поставки емулятора, додаткові ресурси.

Перший розділ описує мову емулятора, загальну структуру цієї формальної [1] мови, її обмеження, пояснюється логіка її проектування. З метою навчити користувача вирішувати виникаючі проблеми, коротко описується внутрішній устрій емулятора.

Другий розділ розповідає про графічну версію програми. Наводяться описи інтерфейсу, розповідається про загальні підходи до його проектування, ексклюзивні можливості програми.

Також в інструкції є додатки, що містять приклади програм, додаткові матеріали, мої контакти.

Обов'язковим до вивчення є перший розділ. Користувачів графічного додатка може зацікавити розділ номер два, користувачів консольного — нульовий. Корисно переглянути додатки.

Досвід роботи з різними інтегрованими середовищами розробки (IDE) значно полегшить розуміння викладеного, як і знання мов C, будь-якої з сімейства Assembler, та текстового редактора Vim.

Якщо сподобається емулятор — можете залишити зірочку у моєму репозиторії.

[1] Важливий термін, до якого я буду звертатися ще багато разів. Уникаючи громіздких математичних формулювань, можна сказати так: Формальна мова — це множина слів (у нашому випадку — рядків), що сформовані з множини алфавіту, за множиною правил.

Розділ 0

Дистрибуція

Наразі існує дві версії програми:

- Консольна версія програми розповсюджується у вигляді вихідних текстів мовою C++, а отже, якщо ви вирішите користуватися нею — вам доведеться зібрати її самостійно. На щастя це нескладний процес, опис якого можна знайти у репозиторії проекту.
- Графічна версія, що розповсюджується у виді вже зібраних бінарних файлів та вихідних текстів, інструкцію до збирання яких також можна знайти у репозиторії.

Приклади написаних програм можна знайти тут.

Вважаю що без короткого опису інтерфейсу консольного додатку інструкція вийшла б неповною. Отже, якщо ви використовуєте версію з графічним інтерфейсом — ви можете одразу перейти до розділу 1 ;)

0.1 Опис інтерфейсу консольного додатку

Для початку роботи, за аналогією з будь-яким іншим компілятором, програму, що написана мовою з 1, треба зберегти у текстовий файл.

Опції емулятора

Для вибору режиму роботи емулятора передбачені аргументи командного рядка. «Визначаючим» аргументом є найперший з них.

Якщо аргументів не задано, то програма виведе інформацію про свою версію, та невелику довідку англійською.

Виклик з ключем `-v` вкаже поточну версію програми і покаже невеликий малюнок ☺.

Якщо перший аргумент це шлях до файлу, ми маємо вказати «непозиційні аргументи», список яких наведено нижче. Виклик без непозиційних аргументів, рівносильний виклику `./TME ../example.txt -g -a -e`

```
Windows:
TME.exe $path to the programm file$ $KEYS$

UNIX-like:
./TME $path to the programm file$ $KEYS$
```

$KEYS$ $\subset$ {-g, -a, -e, -l, -d}
Де:

-g обробка вхідного тексту, генерація зрозумілого машині представлення
-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 у результуючій стрічці
-d debugger

Після виконання програми в поточній директорії буде створено

- службовий файл (.db),
- файл логів (_log.txt)
- файл, що містить результат, тобто копію стрічки машини після виконання програми (_out.txt)

0.2 Консольний debugger

Після розставлення точок зупину (читайте розділ 1!), можна запустити дебагер командою

```
Windows:
TME.exe $path to the programm file$ -d

UNIX-like:
./TME $path to the programm file$ -d
```

Після генерації і аналізу запуститься дебагер, та буде виконувати програму у по-рядковому режимі з першого ж рядка. Щоб виконувати програму по рядку, треба натискати «Enter». Щоб перейти до наступної точки зупинки — введіть непорожню послідовність символів, і натисніть «Enter».

У випадку помилки між точками зупину программа сама зупиниться та покаже вміст машини та її параметри.

Розділ 1

Мовне питання

1.1 Шлях

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

На початку другого семестру ви повинні були познайомитися з концепцією машини Тьюрінга. На жаль, на відміну від теорії, реальність не можна придумати; у моєму другому семестрі існувало декілька емуляторів, кожен з яких мене чимось не влаштовував.

Оніщенко [1] «лається» на пробіли, а таблиці виглядають громісткими (чесно кажучі, я навіть не намагався в них писати — страшно), і жоден з них не працював на UNIX-системах[2].

Існує ще декілька моїх вимог, які жоден емулятор не міг задовольнити. По-перше, необхідні коментарі. Через те, що наша формальна мова є регулярною — рядки мало відрізняються один від одного, і вже за 20 рядків програма стає темним та страшним лісом. Треба віддати належне табличним емуляторам, вони це вміють, але не у тій формі, яку я хочу. Друга вимога — хороший, приємний дебагер, схожий на один з тих що є в популярних IDE [3] . Третя і остання — зрозумілий формат зберігання програми. Це вирішує декілька серйозних проблем водночас: легко зберігати та передавати [4] , легко правити у будь-якому редакторі, якщо вам не подобається мій.

[1] Я поважаю його працю, але я змушений відмовляти вас від використання цього емулятора. Програма має «витоки пам'яті» та помилки часу виконання, що викликає цікавий букет емоцій при його «зхлопуванні» без збереження вашої кількагодинної праці.

[2] У вузькому сенсі терміну — загальна назва для Linux, MacOS.

[3] IDE — інтегрована середа розробки. VS, Eclips, PyCharm — такі приклади.

[4] Емулятор побудований навколо ідеї, що жодна робота не повинна пропасти. Все повинно зберігатися у постійній пам'яті.

1.2 Структура команди

Мені сподобалася форма команди з Оніщенка, вона має наступний вигляд:

```
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 і цифри, але тільки такі спецсимволи: $+, -, /, *, =, :, \hat{}, \#, !, ?, \&, >, <, \%, \_$ (без коми!).

Команда не має обмежень щодо довжини (не варто перевіряти), не має обмежень щодо пробілів між ідентифікаторами.
«direction» може мати тільки такі значення $\{R, r; L, l; S, s\}$.

Отже:

«Тип сутності»	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 Вхідний рядок

Згідно з теорією, машина має нескінченну у дві сторони стрічку, розділену на клітинки. Вважаючи всі клітинки, окрім визначених нами, порожніми, звузимо нашу задачу до скінченного рядка.

Вважатимемо, що я вирішив розділяти клітинки стрічки пробілом « », бо це найбільша клавіша на клавіатурі.

Залишається ще два питання: пустий символ, бо пробіл вже зайнятий, та текстове подання курсора. Пустим символом я обрав «lambda», латинецею нижнього регістру; курсором буде «обрамлення» з символів «|» з двох сторін від обраної клітинки. Цей символ зазвичай знаходиться над клавішею Enter (натисніть Shift+«\»).

Наведу приклад вхідного рядка:

```
|1| 0 lambda 1 1 1 0 *
```

Для отримання пустих клітинок з боків від введених даних не потрібно писати їх руками. При спробі здвинути курсор на поки неіснуючі пусті клітинки програма сама їх створить.

1.4 Структура програми

Як ми вирішили у главі 1.1, нашою метою є створення мови, яка б мала зручне текстове подання.

Так як машина Тьюрінга має два різні по своїй сутності типи вхідних даних (множину команд та вхідний рядок), треба знайти спосіб вмістити все в один текстовий файл.

На порятунок приходить секційний стиль мов асемблера. Асемблери теж зберігають все в одному файлі, роблячи це в окремих секціях, для чого використовують «відокремлюючі» директиви [5] :

```
section .data
section .text
```

Пристосуємо їх до нашої формальної мови. Відповідно до назви, секція .data буде зберігати в собі вхідні дані, тоді як .text — текст програми, тобто множину команд.

Важливо зауважити, що під час аналізу тексту програми серед .data обирається остання, а всі .text зливаються в одну. Кінцем секції вважається початок іншої або кінець файла.

Машина має два особливих стани — початку та кінця програми. Їх ми назвемо «start» та «end» відповідно.

Приклад простої програми, що ставить на стрічку одну одиницю:

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

1.5 Макроси та де вони макро-

Всі читачі, які добре знайомі з мовою C, знають яку велику роль грають макроси у швидкості та зручності написання програми.

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 з директиви `#define`, між директивою та її двома аргументами має бути по одному або більше пробілу. Макросам все-одно що в них, вони сприймають будь-які символи. Після першого пробілу, в тексті частини що «підставляється» (у другому аргументі), можуть будти пропуски, але так робити не рекомендовано.

[5] Деякі читачі можуть згадати ще третю директиву визначення секції, але вона нам ні до чого.

Область дії макросів - лише section .text.

Макроси нашої формальної мови здібні до «зациклювання». Макроси працюють тільки нижче рядка їх оголошення; макроси підставляються у послідовності їх оголошення.

Продемонструємо роботу 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 заміняють стандартний оператор переходу «->» на «>>», та дають змогу використати «::» подібно до просторів імен мови C++.

Я пропоную використовувати нечитаємі комбінації літер та чисел для макросів, які ви використовуєте як оператор області видимості «::» .

Таким чином, макроси та структура нашої програми дозволяє нам писати окремі модулі-секції з обумовленими точками входу та виходу, своїми просторами імен.

1.6 Коментарі

Як ви могли помітити у попередній главі, мова має коментарі. Вони мають однорядковий синтаксис та починаються з символу «;», **працюють всюди крім секції даних**.

Згідно з ассемблерними традиціями, такі коментарі ставлять через табуляцію від рядка програми. Зазвичай, коментарі вирівнюють до одного рівня, ось приклад:

```
section .text
start, lambda ->  world, Hello, r          ; printing Hello
world, lambda -> name, World, r            ; printing World
name, lambda -> exclamation, %username%, r ; printing username
exclamation, lambda -> end, !<3, r         ; printing exclamation
                                           ; sign and ending
```

1.7 Breakpoints

Всі дебагери, що я бачив у таких емуляторах, не використовували точки зупину, тому я був вимушений вигадати їм текстовий аналог.

Вийшло так:

```
section .text
start, lambda ->  world, Hello, r
world, lambda -> name, World, r            ;#d
```

«;#d» — коментар позначаючий точку зупину.

Я не рекомендую залишати коментарі після точок зупинки, хоча така можливість і існує.

1.8 Про двигун і колеса

Як часто буває, для ефективного користування механізмом потрібно мати уявлення про його устрій, внутрішні процеси.

Роботу нашого емулятора можна розкласти на три окремі режими, що можуть комбінуватися різним чином (Рис 1.1).

Першим і найважливішим є режим обробки вхідного тексту (Parsing). Після отримання вхідного коду, емулятор відділяє секцію даних у тимчасовий файл «datasection.tmp» в поточній директорії. Одразу після цього емулятор записує множину команд у таблицю спеціального виду, яку зберігає у файл з розширенням «.db» (db від англ. database — база даних). Перед складанням таблиці команд, підставляються всі макроси.

Аналіз є необов'язковим етапом. На вхід він отримує таблицю команд, яку згенеровано за режиму «Parsing». Взагалі, режим задумувався як статичний аналізатор користувацької програми, але сьогоднішні його можливості найскромніші.

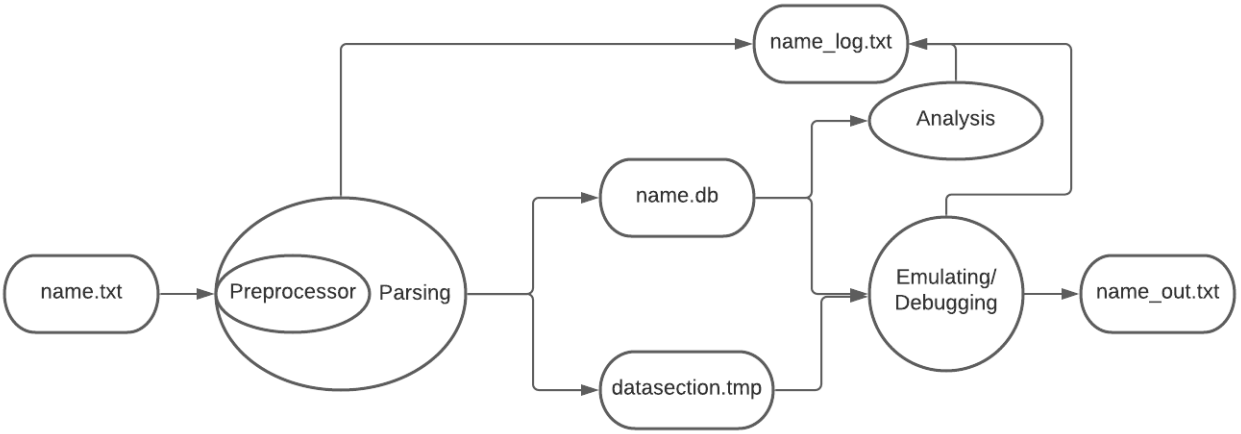


Рис. 1.1: Схематичне зображення «тракту» емулятора.

Емуляція і відладка по своїй суті відрізняються тільки способом виконання одних і тих же дій. У цьому режимі створюється об'єкт машини, у який завантажується вхідна стрічка, яка в процесі виконання програми обробляється на основі таблиці команд. Вихідна стрічка записується у файл, назва якого закінчується на «_out.txt»; звісно, тільки у випадку успішного виконання.

Розділ 2

Графічний додаток

2.1 Загальні підходи.

Будь-яка людська робота важлива. Працюючи на емуляторі Оніщенка, я декілька разів втрачав свої програми. Отже, треба від цього захиститися.

При запуску програми на екрані з'явиться пропозиція (навіть вимога!) відкрити або створити файл. Пропустити цю дію неможливо, а зроблено це заради збереження вашого часу та нервів — це запобігає втраті коду при виникненні непередбачених помилок у емуляторі, що може статися з будь-якою програмою.

Коли щось йде не так, добре мати генезис цієї проблеми. Наприклад, всі сучасні операційні системи мають «системний журнал», туди записується кожен рух користувацьких програм та всі події, які відносяться до операційної системи. Такий журнал буде вести в ході своєї роботи і емулятор. Цей прийом називають логуванням (англ. logging), а файл у якому записано події називають файлом «логів».

2.2 Опис інтерфейсу

Скріншот програми є на сторінці 13.

Майже всі елементи інтерфейсу підписані, сподіваюся буде легко зорієнтуватися по тексту.

Почнемо з найбільшого елемента — поля редагування тексту. Нажаль, Qt framework, на якому написано емулятор, не має простих інструментів додавання колонки нумерації рядків. Звісно, залишатися без нумерації — це залишатися без засобів відладки програми. Результатом цих обставин стала Vim-style нумерація, яку можна побачити справа знизу, під панеллю логів. Поле показує номер рядка на якому стоїть курсор. Сама панель логів не потребує представлення.

Перейдемо до двох довгих смуг зверху вікна — «Input data line» та «Output/Debug line», до чекбоксів [1] поряд. «Output/Debug line» — поле у якому

[1] (від англ. check box) флагова кнопка, галочка.

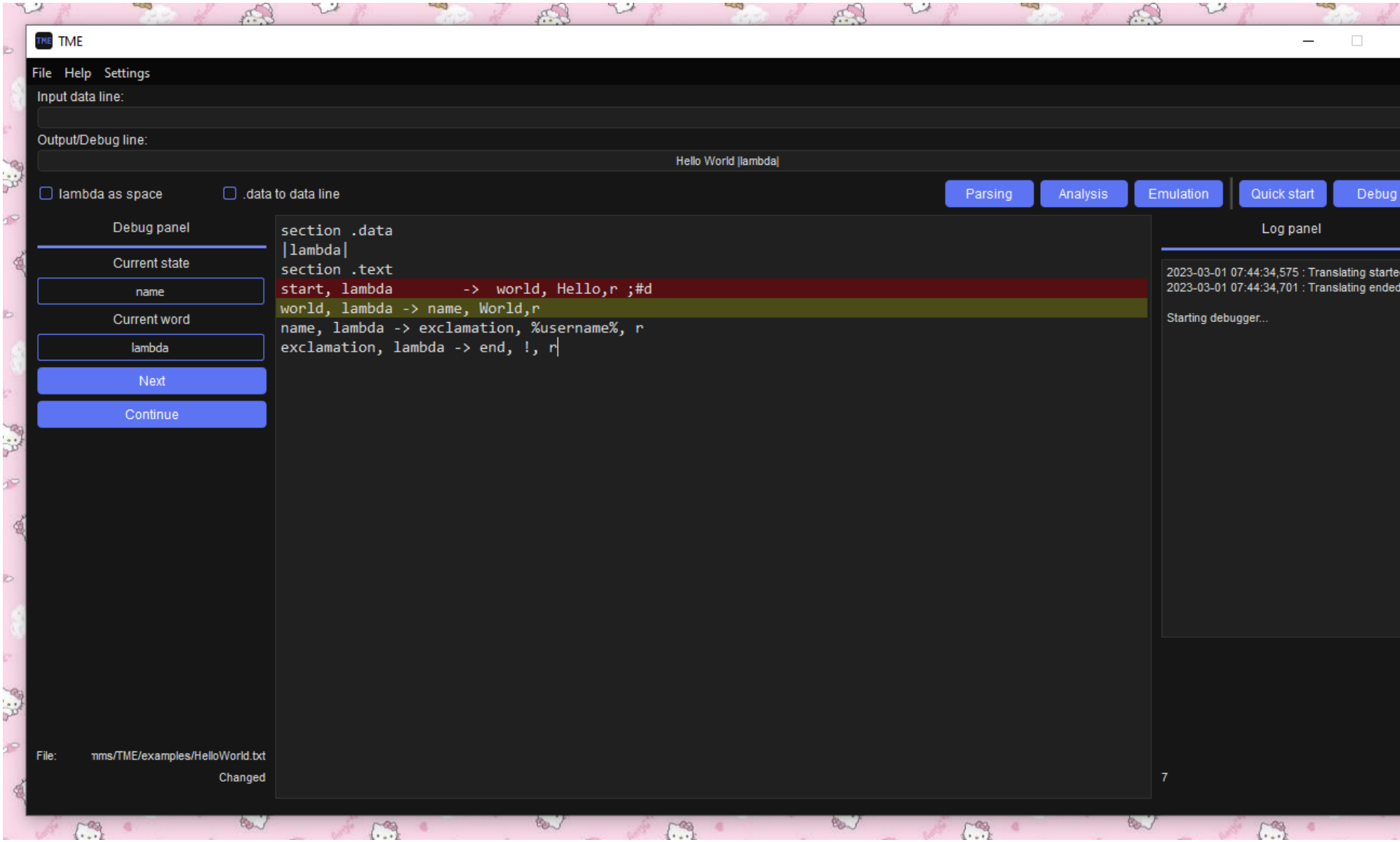
показуються результати роботи або поточний стан стрічки машини під час процесу відладки. Як на мене, «lambda» у великому вихідному рядку іноді дезорієнтує, тому я створив чекбокс «lambda as space», що замінює всі входження «lambda» на « » у «Output/Debug line». Чекбокс починає працювати при наступному виводі даних.

Поясню про «Input data line», ідея цього рядка — тримати вхідну стрічку перед очима користувача, та візуально відокремлювати від секції тексту. Це поле нерозривне у своєму використанні з чекбоксом «.data to data line». При натисканні на вимкнений чекбокс, емулятор порівнює перший рядок файла з рядком «section .data», якщо вони збігаються — секція «виймається», її вміст записується у «Input data line». При відтисканні кнопки відбувається зворотній ефект, тобто секція дописується зверху файла, навіть коли вона порожня. Спробуйте, інакше не зрозумієте!

Нижній лівий кут має ідентифікатор стану файлу («Saved», «Changed»), та показує n символів з кінця абсолютного шляху до файла.

Про режими роботи емулятора я згадував у главі 1.8, кнопки зправа зверху — це втілення режимів «у металі» з однією надбудовою: з'явилася кнопка «Quick start», Quick start=Parsing+Emulation.

Верхнє меню складається з декількох вкладок. Вкладка «File» виконана стандартно для текстового редактора, «Help» носять інформаційний характер. Рекомендую подивитись вкладку «Settings», там багато цікавого і корисного.


TME
File Help Settings
Input data line:
Output/Debug line:
Hello World |lambda|
lambda as space
.data to data line
Parsing
Analysis
Emulation
Quick start
Debug
Debug panel
Current state
name
Current word
lambda
Next
Continue
section .data
|lambda|
section .text
start, lambda -> world, Hello,r ;#d
world, lambda -> name, World,r
name, lambda -> exclamation, %username%, r
exclamation, lambda -> end, !, r
Log panel
2023-03-01 07:44:34,575 : Translating started...
2023-03-01 07:44:34,701 : Translating ended.
Starting debugger...
File: nms/TME/examples/HelloWorld.txt
Changed
7

2.3 Debugger

Скріншот програми є на сторінці вище.

Суть роботи відладника в тому, щоб дати користувачу виконувати покроково проблемні відрізки програми. Для цього використуються breakpoints, вони ж «точки зупину». Залишаючи їх, користувач просить дебагер зупинитися на конкретному моменті виконання та чекати подальших вказівок. Крім того, дебагер дозволяє у будь-який момент часу дізнатися параметри машини та подивитися її стрічку.

Після натискання на кнопку «Debug», дебаг-панель вмикається, дебагер робить один крок та зупиняється, чекає наступних інструкцій. Debug panel має дві кнопки «Next» та «Continue». Перша робить один крок машини, інша змушує відладник йти до наступної точки зупину, помилки або кінця програми. У всіх трьох випадках він поводиться однаково — показує в панелі свої останні параметри. Звісно, кнопки «Next» та «Continue» працюють тільки у режимі дебагу.

У режимі дебагера точки зупину підсвічуються червоним, поточний рядок — жовтим.

Краще спробувати зараз!

2.4 Гарячі клавіші

Комбінація	Дія
Ctrl+N	Ініціалізує створення файла.
Ctrl+O	Ініціалізує відкриття файла.
Ctrl+S	Зберігає відкритий файл.
Ctrl+Shift+X	Quick start.
Ctrl+Tab	Аналізує рядок, на якому стоїть курсор та доповнює його до форми ",—>, , \n". Аналог «Tab» з емулятора Дікарева.
F5	Робить 1 ітерацію у debug режимі.
Ctrl+D	Ставить точку зупину на рядку курсора.

Приємної роботи!

Повідомляйте про помилки в інструкції та програмі на пошту, вказану у додатках. Якщо мене ще не відрахували та я ще не в армії — буду вельми радий почути відгуки.

Якщо ви щось не зрозуміли — можливо я просто погано пояснив. Пишіть питання **по тексту** інструкції на пошту.

Додатки

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. Уникаємо 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
m1::a, lambda -> m2::start, lambda, l

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

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

Будьте почутими

Кожен може відправити питання чи пропозиції мені на пошту у будь-якому виді. Знайшли помилку в тексті – буду радий її виправити.

На всі питання по запуску або збірці проекту, особливо на MacOS, я намагатимусь відповісти.

Зворотній зв'язок

Product site:
kaifolog.github.io/TME-website/
Email:
a.kaifolog@gmail.com

Github repository:
github.com/Kaifolog/TME
Github releases:
github.com/Kaifolog/TME/releases