Інструкція до емулятору машини Тьюрінга.

Зміст:

Дуже важливо: Графічна версія може трохи відрізнятися від консольної, зміни впроваджуються більш рідко.

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 вся ІПСА користується або емуляторами, що не мають достатньо розвинутих аналізаторів, або таким, де команди задаються таблицею. Мені не подобається писати у таблицях, це неінтуїтивно та неочевидно. Емулятор Оніщенка хороший, але має купу фатальнх недоліків, крім нестабільності, через яку можна втратити роботу декількох годин, він не має зручного формату збері-

гання та, найважливіше, не має коментарів, через що вже після написання 20 рядків, программа становиться 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, для яких емуляторів, якщо вони існують, я не зміг знайти. Як всі ми знаємо - до ОС Windows світ розмовляв з компьютерами не жестами, повертаючись у початок людської історії, а словами. Звідси ще одна вимога для емулятора - можливість роботи з командним рядком.

Пам'ятаючи ті 3 рядки, що я писав на ассемблері, та мовою Сі, я зразу визначився з дизайном мови, отже знання цих мов буде значним плюсом для розуміння.

2. Структура команди.

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

Зформулюємо загальний вид команди:

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

Всі state1, state2 мають складатися тільки з латинських літер верхнього та нижнього регистру, 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 $\{R, r; L, l; S, s\}$.

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 такого рішення 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 ассемблера має особливий секційний вид, що і дозволяє їй заберігати всі ці данні впорядковано, використовуючи лише можливості таких мінімалістичних, можна навіть сказати бідних мов, як ассемблери. Ассемблер має три види секцій, наша мова, виходячи з наших потреб, буде мати всього дві. Перша секція буде традиційно називатися :

```
section .data
```

Та відповідно до своєї назви буде зберігати у собі дані. Уважні читачи зададуться питанням: а як можна у тексті, зберегти нескінченну стрічку, яка розділена

на клітинки та має одну, особливу (!), на якій стоїть курсор (зчитувальна голівка)?

Цим питанням задався і я, а т.я. працюю я для себе, а не на роботі, я захотів щоб це було красиво, очевидно та просто.

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

Задавання курсора повинно виділятися. На мою думку, краще за всі інщі символи в тексті видно ("|").

Традиційно склалося, що термінали мають ширину у 70-80 знакомісць (треба сказати, що графічний інтерфейс з глави 7 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 обран символ *lambda*, слово латинськими маленькими буквами.

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
```

4. Макроси та де вони макро-.

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

Знаючи про це, я їх зробив, до того ж це дуже просто, та недорого, порівнюючи з іншими аспектами аналізу вхідного тексту.

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.

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 стандартній оператор переходу "->", та використати для всіх рідний оператор дозволу області видимості "::", що не може бути дозволенним у нашій мові.

Автор пропонує 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 стандартними точками входу та виходу, своїми простірами імен.

5. Коментарі.

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

Наче це все.

6. Як використовувати консольну версію.

Ця глава здебільшого написана для Linux та Mac юзерів.

Після оформлення программи згідно правилам попередніх глав, треба написати

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

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

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

-d debugger

-v version

7. Гайд для версії з графічним інтерфейсом.

Будь-яка людська робота, особливо творча, важлива для суспільства 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 зпочатку відкрити файл, чи створити його. Це запобігає втраті коду при виникненні помилок у емуляторі.

Емулятор має гарячі клавіші, список яких можна знайти у "Як працювати?"

Ще важливо, емулятор завжди відкриває текст файла у головному полі, але існує поле input, що дозволяє писати section .data, виокремлюючи її з тексту, та поряд з виводом/дебагером.

Потрібно знати, що всередени, емулятор просто зберігає input як секцію даних, а відкриває все у головному полі.

Додано checkbox, що дозволяє замінити *lambda* у вихідному тексті на пропуск (" ").

8. 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
```

9. 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
```

Приємної роботи!
Булак А.С. КА-04 2021. (Відраховано.)