

ЯЗЫК ПРОГРАММИРОВАНИЯ
«Хризантемный язык» (Kikkago)
「菊花語」
РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Научная группа:
Аматэрасу-о-миками
Сергей Львович
Вера Алексеевна
Кошка Юко
Ивакава Ацуко

2024 年

*Посвящается памяти Ады Лавлейс, Алана Тьюринга, Эмиля Поста и
Кэтлин Бут*

Содержание:

1. Общая информация	3
2. Низкий стиль: лента	4
3. Низкий стиль: считыватель	6
4. Kikago: общие принципы	7
5. Kikago: таблица команд/операторов	8
6. Объяснения и рекомендации	18
7. Доказательство полноты по Тьюрингу и перспективы применения	20
8. Использование	22

Общая информация

Этот странный язык программирования изначально был создан в качестве эмулятора странного процессора, но затем авторам захотелось вспомнить про великий и ужасный Naikugo (он же «Язык Хайку/Хайку-язык» или же просто 俳句語). Хайку-го, в отличие от Кикка-го (С каждым разом название всё лучше и лучше!) имел аж целое пустое множество документации, и операторы его применялись отдельно к разным типам данных, что создавало сложность в создании мнемонической записи языка, который сам по себе должен был быть мнемонической записью (его величество Ассемблер).

В итоге мы приняли решение сделать следующее: язык позволяет писать код в низком и высоком стилях. Низкий стиль — это процессор, то есть считыватель и лента на 256 ячеек нулей и единиц. Высокий стиль — это работа с переменными, которые хранятся в отдельных массивах. Поэтому сначала будет дано описание низкого стиля, а затем и высокого.

Низкий стиль: лента

Язык низкого стиля состоит из двух элементов: считыватель и 256-битная лента, каждая ячейка которой может содержать в себе либо ноль, либо единицу. При этом лента процессора замкнута, то есть после 256-й ячейки следует нулевая, а перед нулевой располагается 256-я.

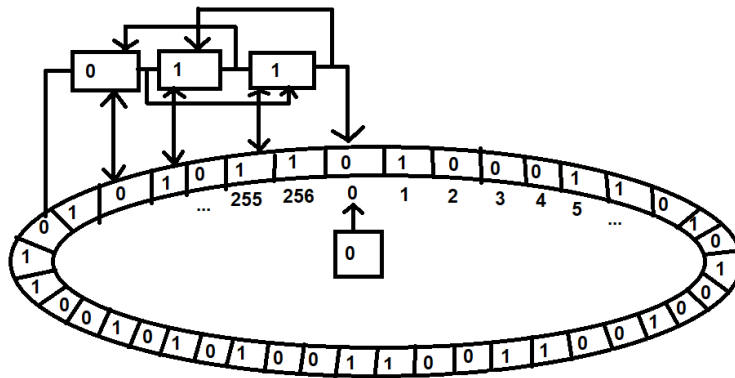


Схема языка низкого стиля

Считыватель может двигаться по ленте влево и вправо. При достижении граничных значений (ноль и 256) считыватель, если продолжит свой путь, переместится на противоположный конец. Если сдвинуться влево от нуля, то адрес станет равным 256. Если с 256 сдвинуться вправо, то будет совершён переход на нулевой адрес. На этом описание ленты заканчивается.

Низкий стиль: считыватель

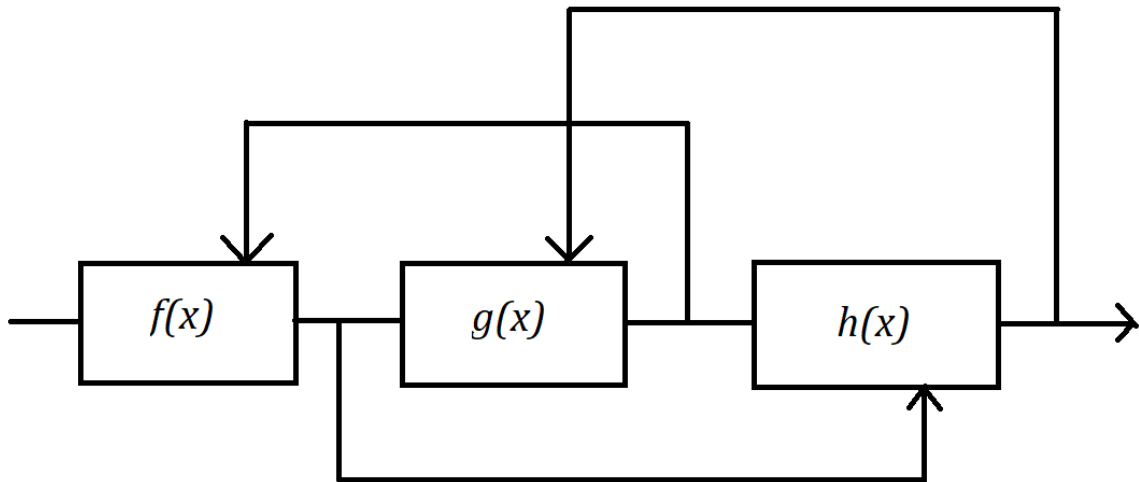


Схема считывателя

Считыватель на ленте — это абстрактное вычислительное устройство, представленное на схеме выше представляет собой последовательное применение трёх функций к входным данным, а затем изменение этих функций друг другом на основе их выходных результатов.

У нас есть три логические функции: $f(x), g(x), h(x)$. Каждая из функций может быть, как тавтологией, так и отрицанием:

$$f(x) = x$$

$$f(x) = \text{not}(x)$$

Каждой из двух возможностей соответствует либо ноль, либо единица. Изменение одной функцией другой функцией можно записать, как применение изменяющей функции к изменяемой. Например, $f(x)$ изменяет $g(x)$ можно записать, как $f(g(x))$. Изменение же на основе выходных результатов возможно записать через применение к изменяемой функции такой функции, что какой соответствует выход изменяющей функции. Запишем в общем виде соответствия:

$$a \rightarrow q(x) = x$$

$$b \rightarrow q(x) = \text{not}(x)$$

$$a, b \in \{0, 1\}, a \neq b$$

Тогда применение некоей функции $f(x)$ к $g(x)$ на основе выходного результата $f(x)$ будет записываться следующим образом:

$$f(c) = d, c \wedge d \in \{0, 1\}$$

$$d \rightarrow w(x) = (q(x) = x) \vee (q(x) = \text{not}(x))$$

$$f(c) = d, d(g(x)) = w(g(x))$$

c — входные данные в $f(x)$.

Применение $f(x)$ к $g(x)$ на основе выходных данных $f(x)$ будет записывать в дальнейшем так:

$$f * g$$

Тогда можно следующим образом описать работу вычислительного устройства со схемы выше следующим образом:

x – входные данные.

y – выходные данные.

$$y = h(g(f(x)))$$

$$h * g$$

$$g * f$$

$$f * h$$

В случае, если $x = y$, мы получаем замыкание, а если они не равны, мы на вход всегда подаём одно и то же. Устройство прекращает свою работу, если состояния функций, которые можно описать одним числом (либо 0, либо 1) и входное значение одинаковые.

Рассмотрим ситуацию, когда на вход всегда подаём ноль, единица соответствует отрицанию, а изначально функции описываются строкой 010. Мы получим следующее:

Текущее состояние функций: 010

Выходы: 0 1 1

Текущее состояние функций: 100

Выходы: 1 1 1

Текущее состояние функций: 011

Выходы: 0 1 0

Текущее состояние функций: 111

Выходы: 1 0 1

Следующим состоянием будет 100, а на входе будет ноль, потому цикл завершается.

Однако это можно не ограничиваться таким поведением этой системы (режим работы «01»).

Можно сначала проводить изменения функций (тогда их выходы полагаются на первой итерации равными их изначальным состояниям), а только потом проход данных через них (режим работы «10»).

Либо же можно пропустить данные через первую функцию, первая изменит третью, затем данные пройдут через вторую функцию, вторая изменит первую, а потом данные пройдут через третью функцию, которая затем изменит вторую (режим работы «11»).

Можно также сделать соответствие отрицание единице или нулю, основанным на вероятности.

Режим работы «00» просто копирует значение из одной ячейки в другую.

Kikkago: общие принципы

Программы на Kikkago (яп. 菊花語 — язык цветов хризантем; далее просто язык) состоят из конечного числа строчек, имеющих следующий синтаксис:

оператор операнд1 операнд2

Оператор и операнды отделены друг от друга одним пробелом. Если в начале строки стоит пробел, то строка игнорируется. Язык чувствителен к регистру.

Для низкого стиля в общем случае операнды — это адреса/номера ячеек на ленте, то есть числа от нуля (0) до двухсот пятидесяти шести (256). В случае, если число будет указано минус один (−1), то значением операнда полагается значение, сохранённое в отдельной ячейке (не на ленте), которое равно заранее определённом адресу ячейки (по умолчанию это значение равно нулю, далее мы будем называть это «взятием/записью значения по адресу»). Если же оператором являются `block`, `label`, `do` и `to`, то единственным операндом является названием блока или метки, которое может быть представлено строкой произвольной длины, не содержащей в себе пробела и управляющих символов.

Интерпретатор языка исполняет программу построчно, выполняя действие, указанное на каждой строчке. При этом сами программы безусловно цикличны, то есть, когда будет выполнена последняя строчка, программа начнёт выполняться с нулевой строчки. И так до тех пор, пока по каким-то причинам не будет исполнена строчка с оператором конца программы (`owari`, яп. 終わり — конец). Возможно так же при запуске программы начинать исполнение с указанной строчки через использование оператора `hajimaru` (яп. 始まる — начало). В случае, если операндом является адрес ячейки, то минус единица в данном контексте означает взятие номера ячейки из переменной, в которой хранится адрес, на который указывает считыватель (то самое «взятие по адресу»).

Для высокого стиля в общем случае операнды — это имена переменных, хранящихся в соответствующих массивах.

Kikkago: таблица команд/операторов

После рассмотрения общих принципов работы мы приведём таблицу всех операторов ассемблера с примером их применения и объяснением, что они делают.

Оператор	Пример применения	Действие
Команды низкого стиля		
hajimaru (始まる — начало)	hajimaru	При запуске программа начинает исполняться со строки, на которой расположен этот оператор (затем этот оператор игнорируется даже в случае использования безусловной цикличности). Если этого оператора в программе нет, то программа начинает исполняться с нулевой строки.
owari (終わり — конец)	owari	При его исполнении программа заканчивается. Необходим для того, чтобы программа закончилась.
<-	<-	Уменьшает значение адреса на единицу. Если адрес, когда встретился этот оператор, равен нулю, то после применения он будет равен 256.
->	->	Увеличивает значение адреса на единицу. Если адрес, когда встретился этот оператор, равен 256, то после применения он будет равен нулю.
addr	addr 249	Делает адрес, равный указанному значению. Если значение указано больше 256, либо меньше нуля, то значением станет число, появившееся после учёта цикличности ленты.
inaddr	inaddr	Запрашивает ввод адреса у пользователя. В остальном делает всё то же, что и addr.
goto	goto 17	Переход на указанную строчку программы. Учитывайте, что нумерация начинается с нуля и не следует переходить на строчки, которых нет — если у вас в программе последняя строчка седьмая (нумерация учитывает нулевую), то бессмысленно пытаться перейти на, например, тринадцатую, которой нет.
addrwokaku (addr を書く — написать адрес)	addrwokaku	Вывод в консоль, чему равен адрес.
mojiwokaku (文字を書く — написать символ)	mojiwokaku	Вывод в консоль символа, чей ASCII-код равен значению адреса.
kyouki (狂気 — безумие)	kyouki	Переход на случайную строчку программы. Действует так же, как и goto, но строка, на которую делает переход выбирается случайно (не беспокойтесь — выход за количество строчек не произойдёт). Внимание: перед исполнением этого оператора должен быть инициализирован генератор

		случайных чисел через применение оператора “rand”!
		Пустая строка, которая игнорируется при исполнении (не используйте пробел в начале строки для создания строки комментария!).
;	; 温子は狂気なです。	Комментарий.
loop	loop	Адрес становится равным числу количества пройденных программой циклов. Изначально программа прошла ноль циклов, однако, каждый раз доходя до последней строчки, не встретив owarі, она возвращается в начало (нулевая строка), при этом счётчик циклов увеличивается на единицу. Оператор loop приравнивает значение адреса количеству пройденных циклов с учётом замкнутости ленты.
f1	f1 147	Указывает, из какой ячейки берётся значение для первой функции в считывателе.
f2	f2 37	То же, что и f1, но для второй функции.
f3	f3 54	То же, что и f1, но для третьей функции.
prob	prob 27	Вероятность того, что в считывателе единица будет соответствовать отрицанию. При вводе любого целого числа (за исключением минус единицы) берёт остаток от деления его абсолютного значения на 101 и записывается в качестве вероятности. При значении, равным –1, берёт остаток от деления значения адреса на 101 и записывает в качестве вероятности. Лучше указывайте число от нуля до сотни.
cycle	cycle 217	Берёт значение из ячейки с указанным номером (минус один — по адресу) и, в зависимости от полученного значения, определяет, будет ли считыватель циклично подавать себе на вход выходное значение, пока не достигнет устойчивого состояния. Ноль — нет, единица — да.
conf1	conf1 3	Берёт один разряд (ноль/один) из указанной ячейки для составления конфигурации (левый знак — «x1»). Если операнд — минус единица, то берёт по адресу.
conf2	conf2 56	То же, что и conf1, но для правого знака («1x»).
conf	conf 7	Эквивалентно случаю, когда для conf1 и conf2 указана одна и та же ячейка. Если в ячейке лежит единица, то

		конфигурация будет «11», иначе — «00».
kaku (書く — написать)	kaku 23 75	Выводит в консоль значения ячеек ленты из указанного диапазона. Настоятельно рекомендуем следить за тем, чтобы второй операнд был больше первого. Минус один (−1) означает взятие значения по адресу. Если оба операнда равны, то выведена будет только одна ячейка.
bunkiten (分岐点 — поворотный момент)	bunkiten 0 76	Сравнивает значения в двух ячейках. Если они равны, выполняет следующую строчку программы, а если нет — пропускает/перескакивает следующую строчку. Если оба операнда равны, то следующая строчка не будет пропущена. Минус единица (−1) указывает ячейку по адресу.
henkamono (変化物 — изменитель)	henkamono 88 44	Считыватель берёт значение из ячейки, чей номер передан вторым операндом, обрабатывает его, согласно своим настройкам, и записывает результат в ячейку, чей номер передан первым операндом. Минус единица означает ячейку по адресу. Обратите внимание, что этот оператор делает адрес, равным первому операнду!
ugoku (動く — переместить)	ugoku 93 127	Копирует значение из ячейки, чей номер передан вторым операндом в ячейку, чей номер передан вторым операндом. Минус единица означает взятие значения по адресу. В случае равенства операндов ничего не изменится (ячейка скопируется сама в себя). Оператор является синтаксическим сахаром и может быть реализован через применение считывателя с функциями 000 и вероятностью 100 (либо применение считывателя в режиме работы «00»). Обратите внимание, что этот оператор делает адрес, равным первому операнду!
label	label tape7	Объявление метки с именем, переданным первым операндом, которое не должно содержать пробелов и управляющих символов. Запоминает номер строчки, на которой объявлена метка.
to	to tape7	Переход на метку с именем, переданным первым операндом. В совокупности с bunkiten позволяет реализовывать циклические конструкции.

block	block q1	Начало блока с именем, переданным первым операндом (ограничения на имя такие же, как для label). Всё, что идёт после block до break игнорируется, если не был исполнен оператор do, вызывающий исполнение блока.
break	break q1	Конец блока. Всё, что после break, уже не игнорируется и выполняется, как обычно.
do	do q1	Вызов исполнения блока с именем, переданным первым операндом. После исполнения кода, расположенного между block и break, интерпретатор возвращается к строке, с которой был совершён вызов блока (т.е. исполнен do) и продолжает исполнять программу со следующей строчки. Примечательно то, что блок может быть вызван из любой точки программы, даже до того, как он был объявлен, так как исполнение происходит на втором чтении, после того, как в первом чтении блоки уже были размечены.
zero (ゼロ)	zero 13	Записать ноль в ячейку, чей номер передан первым операндом. Минус единица — записать по адресу.
hitotsu (一つ)	hitotsu 7	Записать единицу в ячейку, чей номер передан первым операндом. Минус единица — записать по адресу.
Команды высокого стиля		
int	int a 5	Объявляет переменную типа int с указанным через пробел именем, которое не должно содержать пробелов, и указанным значением. Переменные разных типов не могут иметь одно и то же имя и нельзя переобъявлять уже существующую переменную! (массивы тоже являются переменными!) По умолчанию существует техническая переменная “cycles”, в которой записывается, сколько прошло циклов программы!
double	double pi 3.1415	То же, что и int, но для double. Переменные разных типов не могут иметь одно и то же имя! Технические переменные “pi” и “euler” хранят число Пи и число Эйлера соответственно. Не меняйте их!
float	float euler 2.78	То же, что и int, но для float. Переменные разных типов не могут иметь одно и то же имя и нельзя переобъявлять уже существующую переменную!

char	char and &	То же, что и int, но для char. Переменные разных типов не могут иметь одно и то же имя и нельзя переобъявлять уже существующую переменную!
string	string my_first_string ahaha	То же, что и int, но для string. Переменные разных типов не могут иметь одно и то же имя и нельзя переобъявлять уже существующую переменную!
bool	bool kawaii_ne 2173	То же, что и int, но для bool. Всё, что не ноль , делает значение переменной истинной. При bool kawaii_ne 2173 переменная будет истинной, а при bool kawaii_ne 0 переменная будет ложной. Переменные разных типов не могут иметь одно и то же имя и нельзя переобъявлять уже существующую переменную!
inInt inFlo inDou inCha inStr inBoo	inInt a inFlo b inDou c inCha d inStr e inBoo f	Запрашивают ввод от пользователя переменных типов int, float, double, char, string и bool соответственно. В остальном это всё то же объявление переменной, но с запросом значений через ввод в консоли, а не прямое указание в программе. Технические переменные “pi” и “euler” хранят число Пи и число Эйлера соответственно. Не меняйте их!
print	print I_LOVE_FLOWERS!!!	Выведет в консоль переменную любого из типов. Позволяет выводить массивы.
sum	sum b z sum b 2.3 sum b -2	Складывает два операнда, записывая результат сложения в первый операнд. Приводит тип второго операнда к типу первого операнда. Второй операнд может быть передан числом. Слева должна быть переменная!
mult	mult b z mult b 2.3 mult b -2	То же, что и sum, но умножает. Слева должна быть переменная!
sub	sub b z sub b 2.3 sub b -2	То же, что и sum, но вычитает. Слева должна быть переменная!
div	div b z div b 2.3 div b -2	То же, что и sum, но делит. Не делите на ноль! Слева должна быть переменная!
pow	pow a q pow a 3 pow a -3	Возведение первого операнда в степень второго операнда с записью результата в первый операнд. Первый операнд может быть любого типа. Второй операнд может быть передан, как числом, так и переменной,

		и должен быть типа int (в случае несовпадения типа приводится к типу int). Результат приводится к типу первого операнда.
sqrt	sqrt a	Взятие квадратного корня из неотрицательной переменной типа double.
<	< a b < a 1000 < b 0.17	Не пропускает следующую строчку, если левый операнд меньше правого. Слева должна быть переменная!
>	> a b > a 1000 > b 0.17	Не пропускает следующую строчку, если левый операнд больше правого. Слева должна быть переменная!
<=	<= a b <= a 1000 <= b 0.17	Не пропускает следующую строчку, если левый операнд меньше или равен правому. Слева должна быть переменная!
>=	>= a b >= a 1000 >= b 0.17	Не пропускает следующую строчку, если левый операнд больше или равен правому. Слева должна быть переменная!
==	== a b == a 1000 == b 0.17	Не пропускает следующую строчку, если левый операнд равен правому. Слева должна быть переменная!
!=	!= a b != a 1000 != b 0.17	Не пропускает следующую строчку, если левый операнд не равен правому. Слева должна быть переменная!
factor	factor qq	Вычисляет факториал от переменной, записывая в неё результат.
cnk	cnk n k cnk n 9 cnk n 6.7	Вычисляет число сочетаний из первого операнда по второй операнда с записью результата в первый операнд. Можно передавать переменные любого числового типа. Результат запишется в первый операнд с сохранением типа переменной, что является первым операндом. Вторым операндом можно передавать любое число (его тип будет приведён к int).
sin	sin n	Вычисляет синус от операнда. Результат приводится к типу операнда, потому рекомендуем использовать переменную типа double, чтобы сохранить точность.
cos	cos n	Вычисляет косинус от операнда. Результат приводится к типу операнда, потому рекомендуем использовать переменную типа double, чтобы сохранить точность.
exp	exp n	Вычисляет экспоненту от операнда. Результат приводится к типу операнда, потому рекомендуем использовать переменную типа double, чтобы сохранить точность.

ln	ln n	Вычисляет натуральный логарифм от операнда. Результат приводится к типу операнда, потому рекомендуем использовать переменную типа double, чтобы сохранить точность.
equal	equal a b equal v 10 equal c -1.2	Сделает первый операнд, который обязательно должен быть переменной, равным второму операнду, который может быть, как переменной, так и числом. Тип первого операнда сохраняется.
prec	prec 75	Устанавливает, сколько знаков после запятой будет отображаться у числа с плавающей запятой. Сработает, если объявить в любом месте программы, так как выполняется на первом чтении, когда размечаются блоки.
run	run stringLOL	Запуск программы на языке Kikkago, то есть файла, чьё имя сохранено в указанной строке. Например: string lol kek.kikka run lol Запустит программу, записанную в файле kek.kikka
mod	mod a b mod a 5 mod w 7.12	Берёт остаток от деления первого операнда на второй. Переменные могут быть любого типа, но для вычислений будет использоваться только целая часть числа, то есть точно будет нарушена, если переменные не целочисленного типа. Второй операнд может быть задан просто числом. Результат записывается в первый (левый) операнд с сохранением его типа.
tint tfloat tdouble tchar tbool tstring	tint a 7 tfloat b 2.71 tdouble c 3.14 tchar d @ tbool e 0 tstring f 菊の花は咲いて	То же, что и простое объявление переменных, однако переменные объявляются на первом чтении, то есть доступны из любой точки программы. Технические переменные типа double "pi" и "euler" хранят число Пи и число Эйлера соответственно. Не меняйте их!
ToInt ToFloat ToDouble	ToInt intvar q ToFloat floatvar w ToDouble doublevar c	Записывает в первый операнд значение второго с приведением типа. Позволяет переделывать строки в числа.
wasurete (яп. 忘れて — забыть)	wasurete intvar wasurete floatttt wasurete double123 wasurete charcharchar wasurete strrrrringLOL	Удаляет указанную переменную из памяти. Удалять можно только переменные типов int, float, double, char, bool, string. Блоки и метки удалять нельзя.

array	<pre>array a int a_int array a1 int 2173 array a2 float a_flo array a3 float 2.71 array a4 double a_dou array a5 double 3.1415 array a6 char a array a7 bool q1 array a8 string aw</pre>	<p>Массивы объявляются следующим образом сразу с добавлением в них нулевого элемента:</p> <pre>array <имя массива> <тип> <значение></pre> <p>Можно объявлять массивы типов: int, float, double, char, bool, string.</p> <p>Для числовых (int, float, double) и строкового (string) массивов качестве значения можно передавать, как переменную, так и сразу значение.</p> <p>Стоит помнить, что помещённое в массив значение приведётся к типу массива (но не пытайтесь помещать в массив чисел строки — это плохо кончится).</p>
set	<pre>set arrayName index varName set arrayName 17 varName</pre>	<p>Этот оператор записывает значение переменной в элемент массива с указанным номером и имеет следующий синтаксис:</p> <pre>set <имя массива> <номер элемента> <переменная></pre> <p>Массивы чисел и строк приводят тип операнда и помещают его в себя.</p>
get	<pre>get arrayLOL index varName get arrayName 17 varName</pre>	<p>Всё то же самое, что и set, но вместо записи значения переменной в элемент с индексом, get, наоборот, получает значение из элемента по индексу, записывая его в переменную.</p> <p>Обратите внимание на две вещи:</p> <ul style="list-style-type: none"> • синтаксис этой команды повторяет синтаксис предыдущей; • результат здесь записывается в третий операнд, а не в первый! <p>Приведение типов в данном операторе отсутствует!</p>
append	<pre>append array123 my_var</pre>	<p>Добавляет значение переменной (второй операнд) в конец массива (первый операнд). Массивы чисел (int, float, double) и строк (string) приводят добавляемое значение к типу данных массива, но массивы булевых переменных и символов приведением типов не обладают.</p>
pop	<pre>pop array 123 my_var</pre>	<p>Помещает значение последнего элемента массива в переменную, удаляя последний элемент из массива. Только массивы чисел (int, float, double) приводят типы данных.</p>
length	<pre>length intvar arrrrr</pre>	<p>Записывает в первый операнд (обязательно целочисленного типа!) длину указанного массива.</p>
mean	<pre>mean dvar doublearr</pre>	<p>Считает среднее арифметическое double-массива (второй операнд) и записывает в первый операнд (double).</p>

cumsum	cumsum dvar doublearr	Считает кумулятивную сумму double-массива (второй операнд) и записывает в первый операнд (double), то есть позволяет записать сумму всех элементов массива.
rand	rand a1 rand 0 rand -21 rand 73	Инициализирует генератор случайных чисел. В качестве операнда можно передавать, как целочисленную переменную, так и целое число напрямую. Если операнд равен нулю, то зерном генерации станет текущее время, приведённое к беззнаковому целому, то есть при каждом запуске программы вы будете получать разные случайные числа. Если операнд не будет равен нулю, то при каждом запуске программы вы будете получать одни и те же случайные числа. Внимание: исполнение этого оператора необходимо перед исполнением всех операторов, связанных со случайными числами/величинами/вероятностью.
uniform	uniform q w e uniform q 2.3 7.4	В первый операнд записывается число из равномерного распределения с параметрами второго и третьего операндов, которые являются границами отрезка, из которого берётся число (левая и правая границы соответственно). Второй операнд должен быть меньше третьего. В качестве второго и третьего операндов можно передавать, как переменные типа double, так и числа. В качестве первого операнда должна идти переменная типа double.
normal	normal q w e normal q 0 0.25	Запишет в первый операнд число из нормального распределения. Все операнды должны быть типа double. Среднее (второй операнд) и стандартное отклонение (третий операнд) могут быть переданы числами. Третий операнд должен быть больше нуля.
bernoulli	bernoulli a b bernoulli a 0.7	Число из распределения Бернулли (0 или 1) записывается в первый операнд. Вероятность того, что мы получим единицу (второй операнд), можно передавать переменной типа double или числом напрямую. Вероятность принадлежит отрезку от нуля до единицы.
poisson	poisson a lambda poisson a 2	Записывает в первый операнд число из распределения Пуассона. Вторым

		операндом идёт параметр Лямбда, что должен быть типа double и больше нуля.
dispersion	dispersion dvar doublearr	Считает несмещённую дисперсию double-массива (второй операнд) и записывает в первый операнд (double).
write	write file123 arrSTRRRR	Записывает массив строк в файл (каждый элемент массива с новой строки). Первым операндом идёт строковая переменная, в которой находится имя файла, включая расширение, а вторым операндом идёт имя строкового массива.
read	read strarr fileKek	Записывает в строковый массив (первый операнд) содержимое файла (имя в строковой переменной вторым операндом).
jikannohajimaru (яп. 時間の始まり — начало времени) jikannoowari (яп. 時間の終わり — конец времени)	jikannohajimaru ; какой-нибудь код jikannoowari Или: jikannohajimaru ; какой-нибудь код owari	Jikannohajimaru срабатывает при первом чтении и начинает считать время, а jikannoowari завершает подсчёт времени выполнения кода, расположенного между этими двумя операторами. Если jikannoowari не встретился, то owari выполняет его функцию. В результате работы этих двух операторов в консоль выведется время (в секундах) работы кода, заключённого между ними. Безусловная цикличность программы не учитывается оператором jikannoowari, поэтому используйте owari для подсчёта времени работы безусловно циклического кода!

Объяснения и рекомендации

Адрес (`addr, -1`) — номер ячейки, на которую указывает считыватель (変化物). При применении считывателя адрес изменяется на номер ячейки, в которую считыватель запишет обработанные данные. Перемещение (動 <), являясь синтаксическим сахаром, реализует применение считывателя к двум ячейкам с состояниями функций 000 (каждая цифра с отдельной ячейки) и вероятностью 100 и потому также изменяет значение адреса на номер ячейки, переданной первым операндом, однако значение адреса не меняет.

Рекомендации и факты:

- Оператор `to` позволяет перейти только на ту метку, строка с объявлением которой была исполнена ранее, то есть оператор `label` может быть расположен, например ниже, чем `to`, но, если по каким-то причинам он был исполнен, то `to` позволит перейти на метку. Не рекомендуем нарушать принцип использования этих двух операторов.
- Возможно описывать вложенные блоки. Например следующая программа выведет в консоль ноль:

```
block q2  
  
block q1  
  
kaku 0 0  
  
break q1  
  
break q2  
  
do q1  
  
owari
```

Однако мы рекомендуем очень внимательно подходить к таким конструкциям и по мере сил избегать их. В приведённом примере при вызове `q2` вместо `q1` программа просто завершится, ничего не выведя в консоль, так как вложенный блок не был вызван.

Следующая программа выведет ноль дважды:

```
block q2  
  
block q1  
  
kaku 0 0  
  
break q1  
  
do q1  
  
break q2  
  
do q2  
  
owari
```

Это связано с технической реализацией логики блоков, которую можно понять, просмотрев исходный, код поставляемый вместе с этой инструкцией и `exe`-файлом интерпретатора. Не рекомендуем вызывать блоки, которых нет, а также использовать `block` и `break` не по назначению и поодиночке.

- Поведение интерпретатора может быть очень и очень сложным. Один только вывод всех возможных состояний считывателя (変化物) занял 86 листов а4. И это без учёта вероятности, которая позволяет перескакивать между разными цепочками состояний, позволяя порождать, сколь угодно длинные цепочки — главное избегать повторений состояний функций и входного значения в них.
- При использовании низкого стиля рекомендуем следить за значениями операндов, так как не везде реализована проверка на допустимость введённых/указанных значений. В идеале в коде низкого стиля должны встречаться только минус единица и числа от нуля до 256. Учитывайте, что в ленте всего 256 ячеек!
- Если в программе встретится оператор, не указанный в таблице операторов, приведённой выше, то строка будет проигнорирована с выводом в консоль сообщения о неизвестном операторе с указанием номера строки, на которой он встретился.
- Помните: **по умолчанию все настройки, ссылки и адрес инициализированы, как нули** — вам придётся в каждой программе указывать, откуда берутся значения для настроек считывателя!
- Первым операндом всегда должна быть какая-нибудь переменная (включая массивы).
- Не пытайтесь сложить число со строкой и/или провести другие подобные абсурдные операции — следите за типами операндов!
- Не называйте переменные числами!

Это все рекомендации, которые первыми приходят в голову. Всего описать невозможно, так как тогда мы получим противоречие (теорема Гёделя). Общая рекомендация может быть описана следующим списком:

1. Соблюдайте описанные правила.
2. Избегайте вложенных меточных, циклических и блочных конструкций.
3. Старайтесь писать код, не создавая неоднозначных ситуаций, в которых Вы не уверены.
4. Все операнды в низком стиле должны принадлежать следующему списку:
 - Минус единица (-1).
 - Числа от нуля до 256 ($0 \leq n \leq 256$).
5. Избегайте использования оператора `goto`, если это возможно.
6. Разрабатывая алгоритмы/программы, сначала запишите всё на листочке и проверьте код в уме. Грамотно спроектированный код гораздо проще читать и отлаживать!
7. В высоком стиле следите за типами и помните, что первым операндом всегда должна быть какая-нибудь переменная.
8. Не называйте переменные числами!

Эти рекомендации обусловлены опытом разработки языка Naikugo (俳句語) и написания на нём программ. Следуя этим рекомендациям, Вы сможете писать код, который легко читать и отлаживать.

Доказательство полноты по Тьюрингу и перспективы применения

Этот раздел посвящён чистой теории — если Вас интересует только практический аспект, то Вы можете перейти сразу к следующему разделу. Доказательство полноты по Тьюрингу описываемого процессора (низкий стиль) и его ассемблера основывается на следующих фактах:

- Любой вычислитель, на котором возможно реализовать машину Тьюринга, является полным по Тьюрингу, то есть на нём возможно реализовать любую вычислимую функцию.
- Машина Поста эквивалентна машине Тьюринга, являясь одной из тавтологий алгоритма.
- Вычислитель, на котором возможно реализовать другой вычислитель, который полон по Тьюрингу, сам является полным по Тьюрингу.

Основываясь на высказанных фактах, можно заключить, что возможность реализовать операторы машины Поста в терминах ассемблера процессора (низкий стиль) будет доказательством полноты процессора по Тьюрингу. Таблица перехода от машины Поста к ассемблеру приведена ниже ($j \wedge j1 \wedge j2 \in N: \forall n \in N \rightarrow 0 \leq n < \infty$).

Машина Поста	Ассемблер процессора	Действие
V j	hitotsu -1 goto j	Поставить метку, перейти к j-й строке программы.
X j	zero -1 goto j	Стереть метку, перейти к j-й строке.
← j	<- goto j	Сдвинуться влево, перейти к j-й строке.
→ j	-> goto j	Сдвинуться вправо, перейти к j-й строке.
? j1; j2	zero 0 bunkiten -1 0 goto j1 goto j2	Если в ячейке нет метки, то перейти к j1-й строке программы, иначе перейти к j2-й строке
!	owari	Конец программы («стоп», останов).

Из таблицы видно, что полнота по Тьюрингу гарантируется лишь небольшим подмножеством языка программирования, то есть теоретически возможно описать любую вычислимую функцию, даже не прибегая к изменителю и вероятностям.

Перспективы использования процессора по прямому назначению весьма туманны, однако можно рассматривать исключительно теоретические основы. Например, возможно проводить эмпирические тесты тех или иных алгоритмов, проводить проверки гипотез, моделировать сложные системы, связанные с вероятностями и самоопределением внутренних процессов. Ассемблер позволяет, не прибегая к высоким уровням абстракции, связать программу и данные так, чтобы сама программа и данные взаимно изменяли друг друга, образуя неразрывное целое. Использование стохастических возможностей считывателя и оператора случайного перехода по строкам программы позволяет задать неопределённое поведение процессора, которое спрогнозировать невозможно. Для вычислений эти возможности не подходят, чего нельзя сказать о простейших стохастических алгоритмических моделях. Наверное, возможно развить теорию стохастических алгоритмов, и тогда этот процессор можно использовать в качестве готовой среды для проверки и формализации положений этой гипотетической теории.

В конце концов Вы можете развивать этот язык программирования (в частности процессор), модифицируя исходный код, добавляя новые операторы и возможности. Эти язык и процессор могут стать основой для целого семейства архитектур и языков.

Возможно затронуть также и темпоралогические аспекты. Программу в процессе её исполнения можно рассматривать в качестве замкнутой временной линии, тогда путь по программе

(последовательность исполнения строк) описывается мировой линией в двумерном пространстве-времени, где пространственное измерение — программа, а временное — порядок исполнения её строк. Тогда возможны точки схождения и расхождения этих мировых линий. Так, например, все строки с оператором `bunkiten` будут точками дивергенции и конвергенции (в зависимости от того, какие последовательности исполнения строк мы рассматриваем). То же верно и для строк с оператором `kyouki`. Это теорию можно развить.

Доказательство полноты по Тьюрингу не процессора (низкий стиль), а самого интерпретатора (высокий стиль) предоставляется читателю в качестве нетрудного упражнения.

Использование

Исполняемый файл (kikka.exe) — это интерпретатор, который принимает kikka-файлы. Вообще интерпретатор принимает любой текстовый файл, в котором записан код, однако рекомендуется использовать только kikka-файлы, так как в новых версиях интерпретатора и других его архитектурах расширение файла может оказаться значимым.

kikka.xml — это файл с описанием подсветки синтаксиса для редактора кода Notepad++. Вы можете открыть kikka-файлы в Notepad++, после чего перейти в раздел «Синтаксисы» и в выпадающем списке выбрать «Пользовательский синтаксис», после чего импортировать синтаксис из kikka.xml. Тогда у Вас появится подсветка синтаксиса.

pdf-файл — это файл с руководством, который Вы сейчас читаете.

kikka.cpp — исходный код интерпретатора на языке C++.

Сами эти файлы располагаются в папке Kikkago, которую следует распаковать в удобное для Вас место.

Если вы просто нажмёте на kikka.exe, то увидите мерцание консоли. Правильный запуск и эксплуатации интерпретатора может быть представлена следующим порядком действий (после ввода каждой команды нужно нажимать Enter, чтобы она выполнялась):

1. Откройте консоль. Это можно сделать либо через ярлыки, либо нажав сочетание WIN+R (клавиша с символом Windows и английская R). В открывшемся окне введите cmd и нажмите Enter.
2. Перейдите в директорию с kikka.exe, написав команду:

```
cd C:\путь к файлу
```

Например, если вы распаковали папку в корень диска C, то команда будет следующей:

```
cd C:\Kikkago
```

3. Проверьте, что всё работает, введя и исполнив:

```
kikka
```

Если Вы всё сделали правильно, то консоль напишет:

```
Usage: kikka <filename>
```

```
IBN-KIKKA-24 - Interpreted assembler? For what?
```

```
Version 1.0.2
```

4. После этого Вы можете запустить какую-нибудь из программу. Для этого Вам нужно написать kikka и через пробел название файла с расширением. Синтаксис такой:

```
kikka название.kikka
```


Либо, если в папке:

```
kikka название/название.kikka
```

Например, запуск ascii.kikka вместе с результатом выглядит так:


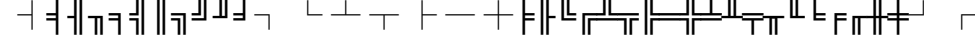
```
C:\Kikkago>kikka programs/ascii.kikka
```



☺ ♡ ♥ ♦ ♣



 ▼ !"#\$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefg

 hijklmnopqrstuvwxyz{|}~△АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзий

 клмноп  | 

  рстуфхцчшщъыьэюяЁёЄėİĩŸŸ°·√ №. ▯ ■ The program programs/ascii.kikka has

 completed successfully!

C:\Kikkago>

Теперь Вы можете создавать текстовые файлы, переименовывать их, заменяя расширение txt на kikka, и писать в них код, который Вы уже знаете, как запускать.

Теперь Вы знаете всё, что нужно для успешного использования процессора. Если Вы смогли запустить `ascii.kikka` и настроить подсветку синтаксиса в Notepad++, то Вы показываете уровень компьютерной грамотности, более высокий, чем у большинства пользователей, а это означает, что перед Вами открываются бескрайние дали — Вы можете придумывать разные стохастические алгоритмы, разные просто обычные алгоритмы. Вы можете, если знаете C++, создавать свои языки на основе ассемблера этого процессора, либо вообще создавать свои архитектуры.

Если же всё, описанное выше, покажется Вам бесполезным, то вспомните, что люди создают картины, пишут музыку... убивают друг друга... Создание очередного странного языка программирования — занятие явно более полезное, чем кошмар, который устраивает человечество в войнах.

Желаем удачи в прогулке по миру странных алгоритмов, что возможно реализовать на этом языке!