Машина исполнения компьютерных алгоритмов – архитектура математической модели с центральным процессором и неограниченными лучами данных.

Д. Л. Гуринович

16 марта 2021 года

УДК 510.51

Содержание

Аннотация

Рассматривается модель исполнения алгоритмов на основе «центрального процессора» и конечного количества «лучей данных», которые начинаются от процессора. Данная модель подобна современным вычислительным системам и позволяет использовать накопленный в ИТ опыт операций с данными. Одной из строк (частью «входных» данных) при этом оказывается сам алгоритм (соответствующий программный текст) – расположенный на одном из лучей данных.

Решен в принципиальном отношении вопрос масштабирования современных вычислительных систем до использования ими оперативной памяти неограниченных размеров при ограниченной разрядности процессора.

1 Практика вычислений и модель «Машина исполнения компьютерных алгоритмов»

I. Проблема масштабирования современных вычислительных систем, имеющих принципиальные ограничения на размер доступной им памяти

Было бы замечательно сослаться на современные вычислительные системы с их центральным процессором (одним процессором – для простоты) и памятью – взяв эти вычислительные системы в качестве образца для аналогичной математической модели исполнения алгоритмов.

Но – не получается так просто. Дело в том, что для математической модели нам необходима возможность оперировать с памятью любых размеров – без каких либо ограничений. А все имеющиеся (известные мне, по крайней мере) практические вычислительные системы имеют принципиальные ограничения на размер памяти, с которой они способны работать как со строками.

Память бывает разная – оперативная, на жёстком диске, в Интернете и т.п. варианты, которые зачастую переходят один в другой при рассмотрении с точки зрения разных потребителей данных из одного источника. То, что для одного является «облаком», для другого – данные с жёсткого диска сервера, а для третьего – оперативная память.

Но везде присутствует принципиальное ограничение на размер доступной для использования вычислительной системой памяти.

До смены 32-разрядных процессоров на 36-разрядные невозможно было использовать оперативную память более 4 Гб. Да, имеется технология PAE (physical address extension), но это тоже разрядность на аппаратном уровне, пусть и отдельно от процессора – к тому же предельный размер для отдельных задач (включая строковые функции) всё равно остаётся ограничен разрядностью процессора.

На жёстких дисках с ростом их объёма приходится увеличивать размер секторов – потому что количество секторов, с которыми может работать компьютер – ограничено. И самый неочевидный пример – сеть (Интернет и локальные сети). Разберём его чуть подробней.

Во-первых, ограничено количество доступных IP-адресов. Именно из-за этого происходит постепенный переход с интернет протокола IPv4 к IPv6, но и там количество адресов ограничено.

Во-вторых, возможное расширение памяти при помощи указания пути, вроде: \\BUH\Users\Buhgalter - тоже не решает вопрос в принципиальном отношении, потому что размер этой адресной строки тоже ограничен на практике, а вместе с ним ограничена и возможность адресации различающихся блоков памяти.

И, к тому же, нам нужна такая память, с которой можно работать непосредственно используя её в строковых функциях, а не получая её многоступенчатым способом перед тем, как произвести нужные нам строковые манипуляции со строкой ограниченного размера.

Ведь если нет простой модели операций со строками, то нет никакого практического смысла в теории для строк произвольного размера, потому что тогда в реальности нет возможности непосредственного оперирования со строками произвольного размера. Тогда есть лишь такие вычислительные системы и алгоритмы, возможности которых заранее ограничены некоторым фиксированным

размером доступной им памяти в плане непосредственных операций со строками (словами).

Но, мы заранее можем утверждать, что в реальности нет никаких принципиальных ограничений для вычислительных систем относительно размеров строк для их строковых операций. Потому что у нас есть простая и масштабируемая на бесконечность (но практически совершенно не разработанная – в сравнении с современными вычислительными системами) модель исполнения алгоритмов – Машина Тьюринга.

Поэтому, у нас заведомо (на основе опыта и интуиции, но пока не на основе факта реализации) должен найтись способ для совершения строковых операций со строками любого размера. Добиться этого можно разными способами – в том числе и при помощи многоступенчатых «путей», возможности наращивания IPv-числа, но всё это должно делаться «за кулисами» простых строковых операций, которые мы исследуем теоретически, и применяем на практике.

Но любой из возможных способов работы со строками произвольного размера должен быть реализован хотя бы на уровне проекта (который можно довести и до практической реализации) и обоснован (на основании известных свойств реального мира). И только тогда мы сможет утверждать, что у нас есть необходимая нам модель, которая сможет стать стандартной интерпретацией для теории компьютерных строк. В разбираемом здесь случае речь идёт про Машину исполнения компьютерных алгоритмов в качестве стандартной интерпретации для теории компьютерных строк.

II. Машина исполнения компьютерных алгоритмов (МКА)

У нас есть (пока не до конца обоснованная) Машина исполнения компьютерных алгоритмов (МКА) с «лучами данных», каждый из которых имеет своё имя и с центральным процессором, расположенным в начале всех «лучей данных». Один из лучей данных – назовём его «$Program$» - содержит текст исполняемой программы.

Ещё один из лучей данных имеет имя «$i_{Program}$» и содержит номер позиции на луче данных $Program$, с которого начинается текст текущей исполняемой команды.

Технически мы можем менять символы в строке $Program$ в процессе работы программы, но условимся этого не делать – для простоты рассмотрения. Всё равно любые такие «изменяющие себя» программы можно переписать с использованием неизменного «интерпретатора» на луче данных $Program$ и «интерпретируемой программы» на каком-нибудь другом луче данных $Program_2$, например.

И количество аргументов и переменных надо считать ограниченным – иначе наше представление о сходящихся в одной точке лучах данных начнёт существенно расходиться с нашими возможностями в реальности. Что создаст ложные выводы в теории о возможной скорости исполнения программы. Любые конечные данные можно разместить хоть на одном луче данных, поэтому саму возможность вычисления мы никак не ограничиваем тем, что ограничиваем количество аргументов и переменных (лучей данных). А в отношении корректного расчета времени, ограничение на количество лучей данных – правильное, в отличие от отсутствия такого ограничения. Будем считать, что имеются только те аргументы и переменные, имена которых используются в неизменной (в процессе работы программы) программе на луче данных $Program$.

Снижается ли «общность рассуждений» по сравнению с Машиной Тьюринга из-за ограничения количества лучей данных? Нет – потому что для Машины Тьюринга хватило бы всего двух лучей данных, чтобы охватить все доступные для Машины Тьюринга варианты работы. А зависимость от программы есть и у Машины Тьюринга – просто никто не рассматривал (или мне это неизвестно) каким именно образом записан и используется алгоритм конкретной Машиной Тьюринга. Есть подозрение, что чем больше размер программы, тем больше должна быть разрядность у «процессора» (или что там вместо процессора?) Машины Тьюринга для исполнения этой программы.

Кстати, модель должна опираться на свойства реального мира, а не умалчивать о практических способах реализации тех или иных своих свойств. Машина Тьюринга очень ясно описывает способ чтения и записи данных, но нет такой же практичности в описании работы с программой. Может, программа тоже записана на чем-то вроде «игрушечной железной дороги» и внутри большой тележки Тьюринга ездит маленький паровозик по конечной игрушечной железной дороге от одной команды программы к другой?

Но ведь тогда чем больше в программе имеется переходов (условных и безусловных) к разным номерам или меткам команд – тем больше должны быть размеры этих номеров или меток. А это значит, что при росте количества таких меток должны расти и расстояния между «шпалами» – чтобы эти большие метки там помещались. Вот такой аналог роста разрядности при усложнении программы.

Будем считать, что разрядность процессора достаточна для того, чтобы «в один ход» (как обычный компьютер с его ограниченной возможностью использовать память) оперировать любыми именами лучей данных и любой командой программы, когда эта команда помещена в некоторый регистр процессора. Поэтому в командах программы мы никогда не будем использовать константы произвольного размера – так как такое использование могло бы вывести нас за рамки разрядности процессора. Подробности будут чуть ниже.

В силу ограниченного количества лучей данных мы не можем считать нашу Машину исполнения компьютерных алгоритмов универсальной для любых алгоритмов. Но для конкретного алгоритма (и для любого фиксированного конечного списка разных алгоритмов) мы всегда можем построить соответствующую их требованиям по количеству переменных Машину с нужной разрядностью. И в рамках этих алгоритмов данная Машина сможет оперировать с данными любого размера – для входных аргументов и используемых переменных при выполнении соответствующей программы.

Программный текст состоит из «строк текста» (они же – «команды»), каждая из которых заканчивается точкой с запятой и переводом строки. Однако в тексте статьи ниже мы будем в некоторых случаях заменять символ перевода строки на пробел – иначе программный текст занимает слишком много места в тексте данной статьи. И точка с запятой при такой сжатой записи очень помогает визуально разделять команды между собой. Но подразумевается, что после точки с запятой всегда стоит перевод строки, а не пробел.

Вот перечень всех возможных «базовых» команд, из которых может быть составлен любой программный текст:

1) $[next_1]$;

Данная команда является меткой, и переход к ней происходит по следующей (пункт 2) команде. Сама команда $[next_1]$; не исполняется, но сразу передаёт управление команде, которая расположена вслед за ней.

Квадратные скобки не могут находиться нигде, кроме как в начале (первый символ команды) и конце (сразу перед точкой с запятой) данной команды. Будем считать, что имя метки (то, что расположено между квадратными скобками) может быть записано только буквами, цифрами, знаком подчёркивания (или переходом к записи субскриптом и обратно) и дефисом после подчёркивания – как и имя любого луча данных. Имя не должно начинаться с цифры. Дефис используется в именах в некоторых языках программирования, кстати – в Windows PowerShell, например. Но там его используют не после подчёркивания.

Условимся, что подчёркивания в имени не могут идти подряд и не могут быть в начале или конце имени. Данные правила позволяют однозначно отличать отрицательный индекс внутри имени от знака вычитания – который не используется в стандартном языке SI, но может применяться в других используемых далее языках программирования.

2) $goto\, x_1$;

Команда перехода к исполнению программы, начиная с метки, значение которой (без квадратных скобок, точки с запятой и перевода строки) записано на луче данных с именем «x_1».

Полную строку для метки (с квадратными скобками, точкой с запятой и переводом строки) обо-

значим тут x_2. Тогда результатом данной команды явится запись на луч данных $i_{Program}$ результата следующей функции из Теории компьютерных строк:

$\mathrm{find}(Program, x_2, 0) + \mathrm{len}(x_2)$, когда искомая метка найдена.

Напомню, что первый аргумент ($Program$) в функции $find$ – это «обыскиваемая» строка. Второй аргумент (x_2) – искомая строка. Третий аргумент – сколько пропускаем символов с начала $Program$ перед началом поиска. А результатом функции $find$ является номер первой позиции в $Program$, начиная с которой имеется подстрока, обозначенная тут переменной x_2. Или ноль, если подстрока, обозначаемая как x_2, не найдена. Слагаемое $\mathrm{len}(x_2)$ служит для того, чтобы получить первую позицию сразу после строки метки в программе – ведь именно с этой позиции должна начинаться новая текущая команда.

Команду $goto\, x_1$ для случая, когда нужной метки нет, будем считать ничего-не-делающей. В этом случае происходит только обычный переход алгоритма к исполнению следующей за $goto\, x_1$ команды. Тогда команда $goto\, x_1$ оказывается аналогична команде $switch$, которая есть в разных языках программирования. В команде $switch$ есть «подразделы» $case$, которые выполняются в случае, если исполнено указанное в них условие. У нас разделы $case$ такие: места программы после меток. А условие исполнения – совпадение названия метки со значением переменной в $goto\, x_1$.

3) .;

Команда остановки – на этой команде выполнение программы завершается.

4.1) $\mathrm{x}_1() = \ldots$;

Команда присваивания «дописыванием строки». Данная команда дописывает в конец строки, которая записана на луче данных с именем «x_1», ту строку, значение которой будет получено справа от оператора равенства. Там вместо многоточия должен быть записан некоторый базовый способ вычисления с аргументами.

Результат данной команды совпадает с результатом функции конкатенации из теории:

$x_1 \cdot x_2$,

где x_2 является значением, которое будет получено справа от оператора равенства.

Если хоть одна из строк, указанных слева или справа от знака равенства, окажется бесконечной, то эта операция никогда не закончится. Дальше подобные оговорки будут подразумеваться без их явной записи.

И конкретизируем, что означает, что на луче данных записана строка «строка», например. А означает это то, что в ячейках с 1-й по 6-ю включительно расположены символы «с», «т», «р», «о», «к», «а» соответственно. А 7-я ячейка является пустой, или, что то же самое – с символом $\ominus$.

На луче данных записана пустая строка, если её первая ячейка является пустой (символ $\ominus$). И любая строка, которая записана на луче данных, никак не зависит от того, что записано в ячейках после первой же пустой ячейки на этом луче данных. Всё, что после 1-й пустой ячейки – можно считать незначимым «шумом» или «мусором».

4.2) $\mathrm{x}_1(0) = \ldots$;

Команда присваивания «заменой». Данная команда записывает на луч данных с именем «x_1»

ту строку, значение которой будет получено справа от оператора равенства. При этом прежнее значение на данном луче данных полностью исчезает.

4.3) $\mathrm{x}_1(x_2) = \ldots$;

Если значение на луче данных с именем «x_2» равно 0 (нулю), то смотри предыдущий пункт. В противном случае x_2 должно быть числом и данная команда для не-нуля является командой присваивания «вставкой». Результат, который будет получен на луче данных x_1, совпадает с результатом следующей функции теории:

$\mathrm{Ins}(x_1, x_2, x_3)$.

Где x_3 является значением, которое будет получено справа от оператора равенства и «вставлено» на место x_2 и далее в строку x_1 – с заменой прежних символов новыми.

Но место x_2 должно не выходить за пределы размеров строки x_1, в противном случае результат будет совпадать с конкатенацией:

$x_1 \cdot x_3$

Способ присваивания «вставкой» - не моё изобретение. В языке VBA, например, есть оператор $Mid\$(\ldots,\ldots,\ldots)$, работающий вот так (пример из книги «Access 2000. Руководство разработчика. Том 1», Кен Гетц, Пол Литвин, Майк Гилберт):

$strvalue =$ «$I like\, you$»

$Mid\$(strvalue, 3, 4) =$ «$love$»

$Debug.Print\, strvalue$

Этот код напечатает в окне отладки строку «I love you».

Можно по-разному решить, как работает команда, если на месте числа x_2 стоит не число. Можно считать, например, что программа прекращает свою работу. Можно даже условиться, что для таких случаев у нас есть луч данных с именем «Error», на котором записывается какое-то «сообщение» при подобном «аварийном завершении».

Если у нас возникнет необходимость разбирать случаи некорректной работы программы, то мы вернемся к этому вопросу. А до этого будем писать «Аварийное завершение» в комментарии к ситуации, которая не должна возникнуть при корректно составленной программе и корректных входных данных.

Теперь перечислим команды, которые дают значение для правой части команды присваивания (замена многоточию в пунктах 4):

5) $\mathrm{len}(x_1)$;

Длина строки – это на 1 меньше, чем номер первой пустой ячейки на луче данных. Напоминаю (из теории компьютерных строк), что нумерация символов в строке начинается с 1.

6) $\mathrm{Chr}(x_1)$;

Результат работы команды совпадает с результатом одноимённой функции теории, когда значением на луче данных x_1 является число. И далее совпадения названий функций Машины с функциями теории – когда они есть – не являются случайными.

Чтоб не заглядывать в теорию, напомню, что функция $\mathrm{Chr}(x_1)$ возвращает символ, который в «алфавите» имеет номер x_1. При этом нумерация начинается с нуля. Если номер оказывается

больше максимального для алфавита, то результатом функции будет пустой символ (он же – пустая строка) $\ominus$.

Расширим немного логику теории в отношении этой функции. А именно – будем считать, что если x_1 содержит не число (включая случай бесконечной строки), то результатом функции тоже будет $\ominus$. Такое расширение никак не противоречит теории, так в теории есть логика только для числовых аргументов, а нам практически удобно его расширить:

Достаточно взять начало строки x_1 такого размера, который равен размеру номера самого последнего не пустого символа из алфавита плюс один, чтобы выяснить результат функции $\mathrm{Chr}(x_1)$. Потому что всё сверх этого размера может дать только $\ominus$ в качестве результата, как очевидно из расширения теоретической логики функции $\mathrm{Chr}(x_1)$. И размер начала аргумента, который достаточен для выяснения результата функции, будем называть значимым размером аргумента – если он есть, конечно, у функции. Вот у $\mathrm{Chr}(x_1)$ он заведомо есть.

7.1) $\mathrm{Value}(x_1)$;

Просто значение, полученное с луча данных с именем «x_1». В «обычном» программировании вместо данной конструкции было бы записано просто x_1. Но технически удобней разбирать текст команды присваивания, если справа всегда записан аналог функции, а имена аргументов находятся между скобками. А хотелось бы иметь удобную для лексического разбора (и программно, и при исследовании в теории) структуру команд.

7.2) $\mathrm{from}(x_1, x_2)$;

То же, что функция $\mathrm{end}(x_1, x_2)$ из теории компьютерных строк.

Не могу использовать служебное слово «end» в языке программирования аналогично тому, как использовал его в теории, потому что в программировании со словом «end» тесно связаны другие конструкции и использование. Слово «from» используется в языке запросов SQL, но по смыслу это не слишком далеко от работы со строками.

Для функции получения «хвоста» желательно имя, которое подразумевает меньшую обработку и сохранение основной части строки (это «хвост» - где может быть любое количество символов), чем самое «общее» имя «str». Но я не нашёл специализированной функции для получения «хвоста» строки, с позиции i включительно, в известных мне языках программирования. Есть перегруженные функции, работа которых зависит от количества аргументов и название, поэтому, не несёт смысла «меньшей переработки», чем «str». Например – Mid (VBA), substring ($C\#$), slice (JavaScript). Поэтому остановился на «from».

7.3) $\mathrm{str}(x_1, x_2, x_3)$;

Напоминаю, что результатом будет строка, чей 1-й символ совпадёт с x_2-м символом из строки x_1, 2-й – с $x_2 + 1$-м символом и так далее x_3 раза, но лишь, если, и до тех пор, пока символы в x_1 не кончились. И ещё считаем $\mathrm{str}(x_1, 0, x_3) = \mathrm{str}(x_1, x_2, 0) = \ominus$.

8) $\mathrm{Comp}(x_1, x_2)$;

Напоминаю, что результатом будет число 0, если $x_1 = x_2$. Результатом будет число 1, если $x_1 < x_2$. Результатом будет число 2, если $x_1 > x_2$. Где знаки (не)равенств понимаются обычным в программировании строк образом. Сначала сравниваются первые символы между собой, если они

равны – то вторые и т.д. При этом $\ominus < x$, если $x \neq \ominus$.

9) $\mathrm{Next}(x_1)$;

Аналог функции увеличения на 1 из теории:

x_1'

10) $\mathrm{Previous}(x_1)$;

Уменьшение на 1, если $x_1 > 0$. Иначе результатом будет ноль.

На этом все возможные в программном тексте команды исчерпаны.

Теперь рассмотрим комментарии, рассмотрим обозначения для чисел «по смыслу» и привычную запись чисел, рассмотрим не используемую в тексте программы «команду» (псевдокоманду), и четыре предопределённых неизменных луча данных, которые одновременно являются ещё и регистрами процессора. Про регистры пойдёт речь при обсуждении масштабирования Машины на лучи данных с бесконечной памятью:

11.1) // Комментарий

Всё, что расположено от двух слешей включительно до символа перевода строки является комментарием. И мы считаем, что этой части строки нет в программе. Нет либо целиком (с символом перевода строки) – если в начале строки нет какой-либо команды, либо перевод строки всё же есть – в противном случае.

11.2) /* Комментарий */

А это уже многострочный комментарий, который мы считаем полностью отсутствующим в программе вместе с символами перевода строки, которые в нём оказались. Перевода строки сразу после данного комментария тоже нет (так считаем) если этот комментарий с самого своего начала занимает все строки текста (разделённые переводом строки), в которых присутствует.

11.3) / # Число, которое что-то обозначает # /

Иногда вместо числа (вместо имени луча данных, на котором записано число, точнее) нам удобней записать смысл, которому оно соответствует. В этом случае мы пишем «комментарий» с этим смыслом, заключая его в «скобки» вида

/ # $<-$ Открывающая скобка # / <- Закрывающая скобка

Разумеется, такая запись подразумевает, что на самом деле вместо этой записи в программе есть последовательность настоящих команд, в результате которых данное число попадает на некоторый луч данных (отличающийся от всех остальных в программе). Конечно, используя такую запись, мы не до конца готовим текст программы для Машины исполнения компьютерных алгоритмов, но для рассмотрения логики алгоритма и понимания возможности его создания нам зачастую удобней скрыть подобными сокращениями второстепенные подробности. Например:

$\mathrm{x}_1(0) =$ / # Результат работы $\mathrm{Comp}(a, b)$ при $a > b$ # /;

Может быть переписано так:

$\mathrm{x}_1(0) = \mathrm{Next}(0)$;

$\mathrm{x}_1(0) = \mathrm{Next}(x_1)$;

Как видим, запись с / # Результат работы $\mathrm{Comp}(a, b)$ при $a > b$ # / лучше проясняет, какую цель преследует присваивание лучу данных x_1 числа 2. А как именно решается эта простая вспо-

могательная задача – второстепенный вопрос, ответ на который мы всегда доделаем при необходимости.

11.4) Десятичная запись чисел

Договоримся (из технологических соображений работы с числами), что числа записываются на лучах данных в десятичном виде, но в обратном порядке (младший разряд единиц располагается в 1-й ячейке и т.д.). Из-за такой договорённости представление их строкой противоречит нашим привычкам, и немного сбивает с толку. Можно было бы использовать запись

/ # Десятичное число 321 # /

Но условимся, что запись числа можно делать без всяких скобок, и понимать его в обычном смысле:

321 – три сотни, два десятка и одна единица.

А раскрыть запись

$\mathrm{x}_1(0) = 321;$

Можно так, например:

$\mathrm{tmp}(0) = \mathrm{Next}(0);$

$\mathrm{tmp}(0) = \mathrm{Next}(tmp);$

...

$\mathrm{tmp}(0) = \mathrm{Next}(tmp);$ // Здесь на луче данных tmp уже записан номер цифры 1 в алфавите символов

$\mathrm{x}_1(0) = \mathrm{Chr}(tmp);$ // На луче данных x_1 записано «1»

$\mathrm{tmp}(0) = \mathrm{Next}(tmp);$

$\mathrm{x}_1() = \mathrm{Chr}(tmp);$ // На луче данных x_1 записано «12»

$\mathrm{tmp}(0) = \mathrm{Next}(tmp);$

$\mathrm{x}_1() = \mathrm{Chr}(tmp);$ // На луче данных x_1 записано «123»

11.5) $\mathrm{Const}(\ldots)$ и $\mathrm{Const}(\ldots);$

Между открывающей и закрывающей скобками в конструкции с символом перевода строки в конце:

$\mathrm{Const}(some\ string);$

можно ставить любые символы, кроме символа перевода строки. В том числе можно ставить скобки. Признаком конца записи этой псевдокоманды является последовательность символов:

);

Включая символ перевода строки, который стоит в конце предыдущего абзаца, и «виден» как окончание записи символов на предыдущей строке. Данная конструкция может стоять только в качестве правой части команды присваивания, разумеется.

А конструкция $\mathrm{Const}(\ldots)$ может стоять вместо имени луча данных везде, где может стоять имя луча данных. Но вместо многоточия в ней могут быть только читаемые знаки (которые имеют графическое представление, и записываются друг за другом в строке), исключая закрывающую круглую скобку.

Что означает эта псевдокоманда (в обеих формах), станет ясно после следующих предопределённых лучей данных:

11.6) $ChrEnter$

Имя предопределённого регистра и луча данных одновременно, который содержит строку из символа перевода строки.

С помощь него и последней псевдокоманды можно записать в переменной любой текст (в том числе и с переводом строки) следующим образом:

$\mathrm{x}(0) = \mathrm{Const}($ Мама мыла раму $,);$

$\mathrm{x}() = \mathrm{Value}(ChrEnter);$

$\mathrm{x}() = \mathrm{Const}($ Папа чинил самолёт $.);$

Теперь особенный луч данных и одновременно – константа:

11.7) 0

Это не только константа ноль из теории, но и имя регистра и луча данных одновременно, значением которого является ноль.

На месте аргументов команд в программном тексте всегда записаны только имена лучей данных, включая специальное имя 0 (ноль).

При помощи последовательности команд присваивания с командами $\mathrm{Next}(i)$ и $\mathrm{Chr}(i)$ можно получить любые символы алфавита.

А с помощью команд вида:

$\mathrm{x}(0) = \mathrm{Chr}(i_1);$

$\mathrm{x}() = \mathrm{Chr}(i_2);$

...

Можно записать на луч данных любую строку (последовательность символов алфавита).

Именно таким образом и представлена в программном тексте любая псевдокоманда вида

$\mathrm{x}(0) = \mathrm{Const}($ Мама мыла раму $: -));$

или, например:

$\mathrm{x}(0) = \mathrm{str}(\mathrm{Const}($ Мама мыла раму $), 6, 4);$

То есть, неявно используются некоторые вспомогательные лучи данных для того, чтобы на место $\mathrm{Const}(\ldots)$ попало имя такого луча данных, который уже содержит нужное значение. Опять мы видим «недоделанную» часть программы, но которую легко доделать при необходимости.

11.8) $ChrEmpty$

Имя «пустого» регистра и луча данных одновременно. Значением данного луча данных является пустая строка $\ominus$.

Впрочем, пустую строку можно получить командой $\mathrm{str}(0, 0, 0)$, но удобней иметь это востребованное значение «в готовом виде» для использования в строковых функциях. Но в дальнейшем в программном тексте здесь я обычно буду писать $\ominus$ вместо $ChrEmpty$ – для краткости.

11.9) $ChrClosing$

Имя предопределённого регистра и луча данных одновременно, который содержит строку из закрывающей круглой скобки.

Это тот символ, который не может быть записан при помощи конструкции $\mathrm{Const}(\ldots)$ – когда она не в конце строки программного текста, поэтому для него предусмотрен особый луч данных.

Вообще же при начале работы программы (алгоритма) все значения необходимых «входных» аргументов уже записаны на соответствующих лучах данных. Включая случаи сколь угодно больших и даже бесконечно больших строк. Эта запись входных аргументов не делается в программе, и не считается ни одним шагом её работы.

Как видно из построения программного текста – в нём нет ни сложных композиций функций с вложенными скобками, ни каких-либо команд, отличающихся от перечисленных выше «базовых» команд. Все композиции и «производные функции» могут быть заменены (и заменяются в данном случае) операциями с промежуточными значениями, переходами в «подпрограммы» при помощи команды goto с предварительной записью на какой-либо луч данных имени ближайшей метки для возврата из «подпрограммы».

Например, нам нужна программа, которая будет сравнивать первый символ в тексте из входного аргумента Arg с латинским символом «a». И возвращать результат на луч данных с именем Out. Тогда можно выполнить данную работу при помощи следующей программы:

$\mathrm{a}(0) = \mathrm{Const}(a);$
$\mathrm{j}(0) = \mathrm{Next}(0);$
$\mathrm{b}(0) = \mathrm{str}(Arg, j, j);$
$\mathrm{i}(0) = \mathrm{Comp}(a, b);$
$\mathrm{x}(0) = \mathrm{Value}(a); \mathrm{x}() = \mathrm{Value}(i); goto\, x;$
$[a0]; Out = \mathrm{Const}(= a); .;$
$[a1]; Out = \mathrm{Const}(> a); .;$
$[a2]; Out = \mathrm{Const}(< a); .;$

Нам в любом случае придётся воспользоваться командой $\mathrm{Comp}()$, но можно обойтись только этой командой, не используя команду $\mathrm{str}()$. Данный «трюк» не удался, если бы символ «a» был последним символом в алфавите.

Но раз символ «a» не является последним в алфавите, то для его сравнения с первым символом входной строки подойдёт следующая программа:

$\mathrm{a}(0) = \mathrm{Const}(a);$
$\mathrm{b}(0) = \mathrm{Const}(b);$
$\mathrm{i}(0) = \mathrm{Comp}(a, Arg);$
$\mathrm{x}(0) = \mathrm{Value}(a); \mathrm{x}() = \mathrm{Value}(i); goto\, x;$
$[a1];$
$\mathrm{i}(0) = \mathrm{Comp}(b, Arg);$
$\mathrm{x}() = \mathrm{Value}(b); \mathrm{x}() = \mathrm{Value}(i); goto\, x;$
$[a1b0];$
$[a1b1]; Out = \mathrm{Const}(> a); .;$
$[a2]; Out = \mathrm{Const}(< a); .;$
$[a0];$

$[a1b2]; Out = \mathrm{Const}(= a);$.;

Достаточно ли перечисленных выше команд для выполнения всех возможных вычислений? На основании тезиса Чёрча можно утверждать, что да – достаточно. Потому что все команды, которые можно выполнять на Машине Тьюринга, можно выполнять и на Машине компьютерных алгоритмов (МКА). У МКА больше команд и функций – но это не ограничивает её возможности, а делает более удобной для исследования в теории компьютерных строк. Ведь каждая функция в МКА соответствует одной из функций данной теории.

Повторюсь, что числа будем считать записанными в десятичном виде, но в обратном порядке на лучах данных. Когда в первой ячейке луча данных записана цифра для единиц, во втором – для десятков и т.д.

Такая конкретизация происходит без ограничения общности – потому что всегда можно переписать модель для другой «ичности». Впрочем, речь только о позиционном представлении чисел. Но, используя язык с числами в десятичной форме можно написать на этом языке программирования интерпретатор для языка с любым другим представлением чисел. Поэтому нам достаточно рассмотреть только удобный нам вариант, обосновать его и после этого остальные варианты тоже можно считать обоснованными.

Между разными базовыми командами могут быть разные соотношения по времени в разных моделях исполнения алгоритмов – поэтому их нельзя «свести» к более узкому набору базовых команд в общем случае.

В некотором смысле данный небольшой набор «базовых команд» произволен, но он удобен для работы при совместном использовании с теорией компьютерных строк, так как построен в соответствии с функциями теории. И он достаточен для того, чтобы выполнить любые вычисления для любых вычислимых функций.

А по времени работы данный набор команд может отличаться от того, что даст какой-то иной набор базовых команд. Но время интересует нас только в пределах полиномиальной сравнимости тех моделей исполнения алгоритмов, которые полиномиально сравнимы с моделью «Машина исполнения компьютерных алгоритмов».

III. Чем не устраивает Машина Тьюринга? Какая роль в теоретических построениях у интерпретации теории?

В теории алгоритмов нам необходимо считать время работы алгоритмов, затраты памяти, сложность вычислений в зависимости от входных данных и т.д. Но алгоритмы пишутся для вычислительных систем, подобных Машине исполнения компьютерных алгоритмов, а не для аналогов Машины Тьюринга. И строковые функции программистов совпадают (кроме несущественных отличий между их реализациями) с теми строковыми функциями, что представлены в предыдущем разделе.

Пишут алгоритмы – миллионы программистов, а ищут, и находят решения для действительно сложных алгоритмических задач тысячи, если не десятки тысяч специалистов. И практически никто из них не работает с действующим образцом Машины Тьюринга или её аналогами. Для неё даже ни одного нормального языка программирования не придумано – ни программистами (им это и не нужно), ни математиками.

И я пишу теорию компьютерных строк – которая существенно опирается на накопившийся в ИТ опыт и практически заимствует язык и подходы для работы со строковыми функциям оттуда. А мог бы я опираться на практику работы с аналогами Машины Тьюринга и писать теорию на этой основе? Нет, не могу – потому что у меня и практически ни у кого нет такого опыта.

Прежде, чем пишутся аксиомы для некоторых объектов – должны быть смыслы (семантика) для этих объектов. А смыслы появляются на основе практического опыта обращения с этими объектами, притом опыта – совместного с другими людьми. Тогда и вырабатывается общий язык и общее понимание для свойств этих объектов так, чтобы люди понимали друг друга – о чём идёт речь и какие свойства объекта называются так, какие иначе и чему на практике эти слова соответствуют.

И смыслы для строк диктует практика ИТ, а других общепринятых смыслов – просто нет. Поэтому теорию необходимо строить на базе семантики из ИТ. Поэтому модель, которая является стандартной интерпретацией для данной теории, должна иметь ту же семантику и аналогичный синтаксис. Потому что именно так данная стандартная интерпретация оказывается наиболее полезной и удобной для использования вместе с данной теорий. А именно для совместного использования с теорией она и строится.

Обычно не теория, а её интерпретация позволяет сразу «увидеть» некоторое свойство у объектов теории, что позже может быть формализовано в виде теоремы с доказательством, которое бывает совсем не просто придумать.

Но модель всегда обладает своей дополнительной – по сравнению с «интерпретируемой» теорией – логикой, которая может быть сколь угодно сложной. Например, гармонические колебания мы видим и в электрических цепях, и в устройствах с пружинками, и для маятника в гравитационном поле и т.д. И эта дополнительная логика (электрических схем, упругости, гравитации) в интерпретируемой теории гармонических колебаний не аксиоматизируется, потому что мы проводим границу – где дополнительная логика модели выходит за рамки логики данного математического явления или объекта.

Например, строка должна обладать однозначно понимаемым размером (количеством символов) и, поэтому, числа арифметики не подходят на роль строк – потому что для них всегда есть стандарт-

ная интерпретация «счётных палочек», которая экспоненциально отличается по своим размерам от позиционной записи числа символами (цифрами). Это метатеорема «Об обязательном наличии стандартной интерпретации» из теории компьютерных строк.

Но при этом строка не обязана быть записана на луче данных в порядке: 1-й символ в 1-й ячейке, 2-й символ во 2-й ячейке и т.д. Строка может быть записана и «задом наперед» - и это тоже будет интерпретация для теории компьютерных строк. И логика такой модели «задом наперед» будет отличаться от логики Машины исполнения компьютерных алгоритмов, но при этом обе интерпретации будут соответствовать логике теории компьютерных строк – потому что в данной теории (в её аксиомах) нет той дополнительной логики из данных интерпретаций, которая делает их различными.

И если свойство однозначности размера строки должно быть задано в аксиомах теории для строк, то в этих аксиомах не должно быть однозначной логики для времени работы алгоритмов – потому что это уже не является свойством строк, а зависит от способов их использования. И в разных моделях применения строк мы получим разное время работы алгоритмов со строками. А теория для строк должна позволять нам выражать любые подобные модели средствами теории, чтобы иметь возможность сравнивать внутри теории характеристики этих моделей между собой и решать другие подобные задачи.

Да, теория должна позволять строить (определять) разные функции для расчёта времени работы алгоритмов – в зависимости от модели использования строк. Но при этом одна функция расчёта затрат времени для одной модели – не может исключать другую функцию для расчёта времени в другой модели. Просто теория позволяет строить разные сопоставления между строками и числами (временем работы соответствующих алгоритмов с данными). Эти сопоставления являются функциями, и мы можем определять обозначения для них в теории. Но вот что касается размеров строки, то тут есть одна функция, она прописана в логике аксиом и полностью исключает какие-либо альтернативы себе.

А где проходит граница между логикой объектов теории и дополнительной логикой их использования в соответствующей модели? Граница проходит там, где это принято соглашениями между людьми, которые проверены практикой и не содержат противоречий. И правильная теория должна соответствовать этим соглашениям. То есть – не сама теория задаёт объект, но семантика объекта имеет приоритет для построения логики (аксиоматизации) и синтаксиса теории.

И соглашения межу людьми могут уточняться, расширяться – потому что любая «достаточно выразительная» теория первого порядка не полна и не позволяет сформулировать окончательную логику для объектов. Хотя в некоторых исключительных случаях окончательную логику позволяет сформулировать логика 2-го порядка – подобно тому, как это сделано для арифметики. Но только нет, вообще говоря, способа понять, как надо расширять теорию первого порядка, чтобы это соответствовало данной теории второго порядка. А с самой теорией 2-го порядка работать можно лишь в той степени, в которой её можно «расшифровать» до уровня теории 1-го порядка.

Модель – это не теория, а некоторая схематизация свойств реальности и описание возможности решать задачи в определённой сфере человеческой деятельности на основе данных свойств.

Модель обладает значительно более богатой логикой, чем теория, потому что использует разные особенности реальности, которые всегда сопутствуют более общей (и менее богатой) логике теории. Например, логика теории гармонических колебаний не проявляется в реальности вне конкретной своей формы – электрической, упругой, гравитационной и т.д.

При описании модели мы не используем доказательств, как таковых – если понимать под ними строгий вывод из набора заранее заданных аксиом. Мы просто сверяем наше описание с известными нам свойствами реальности. И это – скорее аналог проверки теории на непротиворечивость, которая тоже не может опираться на аксиомы самой теории, как известно из второй теоремы Гёделя (я говорю везде о теориях первого порядка, если не делаю специальной оговорки о логике второго порядка).

Поэтому построение модели находится в принципиальной части вне рамок применения стандартных математических инструментов. И поэтому обычно нет смысла формулировать всю логику собственно модели в некоторую теорию – ведь не логика теории будет являться основой обоснования непротиворечивости модели, но – напротив – в основе обоснования непротиворечивости теории лежит сама модель и сверка её описания с реальностью. А интересует нас непротиворечивость гораздо менее детальной (чем полная логика модели) логика более общей (и менее богатой в логическом отношении) теории. И вот для обслуживания задач, выходящих за пределы возможностей теории, мы и должны иметь модель (или несколько), которые являются интерпретациями исследуемой теории.

Модель, наконец, является той опорой на реальность, где математики встречаются между собой, и на основе которой, они решают общие задачи. Потому что абстрактные фантазии у каждого свои, и они обычно не приходят одинаковые в голову разным людям, а вот реальность – общая. И на этой базе и решаемые задачи только и могут (отбрасывая случаи редчайших случайных совпадений одинаковых нереалистичных фантазий у разных людей) становиться общими для многих. И деньги (а без них тоже не проживёшь одной «чистой математикой») появляются где-то оттуда же – потому что задачи оказываются общими до уровня практиков и выгод от них.

Но важность теории тоже невозможно переоценить – самая общая логика для круга решаемых задач позволяет иметь ограниченный набор исходных посылок и получать результаты в бесспорном виде – на основе доказательств. Крайне облегчая передачу и сохранение знаний. Ведь практическое знание без закрепления его в строгие теории требует вовлеченности человека в постоянную деятельность по развитию сферы решаемых им задач, чтобы понимать, и «чувствовать» реальность. Как только дело тормозится, практики начинают просто «механически» воспроизводить однажды найденные «алгоритмы» действий, теряя своё понимание происходящего. И тогда практическое знание начинает исчезать – как падает велосипед, прекративший своё движение.

За последние лет сто, мы имели как раз практический бум, почти игнорировавший соответствующее развитие теорий. Он выдохся, а сама возможность подобного практического бума является редчайшим явлением в Истории. Если теперь не «переводить» накопленные практические знания в строгие теории, то наступит технологическая деградация – её предвестники уже чувствуются, «велосипед» почти остановился, и начинает падать.

И пора возвращаться к преобладающей в Истории методике сохранения и развития знаний – когда теории находятся в основе, а практика развивается преимущественно вследствие развития теорий. Возможно, это приведет к постепенному возврату устройств обществ к тем стандартам, что было (и, видимо, будут) на протяжении тысяч лет – но это уже точно не вопрос для рассмотрения в данной работе.

IV. Масштабирование вычислительной системы с центральным процессором до возможности использовать лучи данных неограниченного размера. Языки программирования для решения этой задачи

У нас нет проблем с тем, чтобы считать доступной для процессора память конечного числа лучей данных в пределах некоторого фиксированного количества начальных ячеек на каждом луче данных. Потому что это имеет практически работающие аналоги и, поэтому, может браться в качестве готового элемента математической модели.

Поэтому мы исходим из того, что для процессора доступны регистры (некоторые «короткие» лучи данных) и начальная часть всех остальных лучей данных. И в этих рамках процессор в состоянии исполнять программы на языке программирования Машины исполнения компьютерных алгоритмов, который описан выше. Язык простой и мы считаем, что интерпретатор для него имеется.

Кстати, будем называть этот язык программирования «язык программирования стандартной интерпретации», поскольку он написан для стандартной интерпретации Теории компьютерных строк – Машины исполнения компьютерных алгоритмов. Или сокращённо – «Язык СИ», «Язык SI». У нас будет далее расширение этого языка до «Аппаратный язык SI», поэтому исходный (не расширенный) язык буду называть, когда надо отличать от расширенного, «Стандартный язык SI»

Это язык программирования среднего уровня. Если же на его основе захочется составить язык высокого уровня – с операторами цикла, арифметическими операциями, вложенными скобками, подпрограммами и остальными «удобствами» современных языков программирования, то название для нового языка будет придумывать его создатель. И уже для созданного языка давать оценки времени работы используемых в его языке программирования конструкций и обоснования для этих оценок.

На базе аппаратного языка SI, имеющего доступ к регистрам и началам лучей данных нам надо добиться доступа ко всем остальным ячейкам данных. В том числе и доступ к «длинной» программе, написанной на стандартном языке SI и расположенной на луче данных $Program$. И нам нужен доступ к каждой команде этого программного текста, чтобы исполнять очередную команду, номер места первого символа текущей команды, которой написан на луче данных $i_{Program}$, переходить к следующей команде и т.д.

Номер, кстати, тоже может быть какого угодно размера. Напоминаю, что мы считаем для разбираемой реализации Машины, что числа на лучах данных записаны в десятичном представлении, но так, что цифра единиц находятся в 1-й ячейке, цифра десятков – во второй и т.д.

Таким образом, мы будем добиваться, чтобы у нас был «аппаратный интерпретатор» на аппаратном языке SI, который через свои ограниченные по доступу к памяти возможности был способен исполнить программу (записанную на стандартном языке SI) на луче данных $Program$, используя «входные данные» с других лучей данных. При этом размер программы $Program$ и размер входных данных могут выходить за рамки возможностей «прямого» использования памяти процессором.

Интерпретатор написан на аппаратном языке SI, а интерпретирует он программу с луча данных Program, написанную на стандартном языке SI. Переход исполнения от команды к команде на

луче данных $Program$ соответствует позиции начала текущей команды на луче $Program$, номер которой записан на луче данных $i_{Program}$ и меняется при смене текущей команды на следующую в соответствии с порядком их исполнения.

Мы не разбираем, каким образом происходит процесс исполнения программы аппаратного интерпретатора – просто ссылаемся на практическую возможность написания и исполнения программ, оперирующих ограниченной памятью, и будем опираться на эту исходную посылку.

Ясно, что у аппаратного интерпретатора есть какой-то свой внутренний аналог луча данных $Program$ (в ПЗУ – постоянном запоминающем устройстве) и счётчика $i_{Program}$ – но только ограниченных размеров, в отличии от «настоящих» $Program$ и $i_{Program}$.

Каким же образом наш процессор со своей ограниченной разрядностью и ограниченной возможностью по доступу к памяти сможет «добраться» до произвольной ячейки с номером N, например? В том числе и тогда, когда разрядность нашего процессора не позволяет ему обратиться к этой ячейке напрямую?

«Эстафета» - вот решение. Каждая ячейка на луче данных у нас будет маленьким компьютером с ограниченными возможностями. Но каждая ячейка (если речь не про первую ячейку луча данных) способна обратиться к двум соседним с собой ячейкам. А первая ячейка способна обратиться ко 2-й ячейке и к процессору.

И каким образом тогда процессор сможет получить данные из ячейки N?

Если нужны все данные с луча данных, то можно использовать луч данных как очередь, отправляя по «эстафете» сообщение, которое передаёт команду в узел Машины отправить символ в предыдущий узел (процессор для первой ячейки и предыдущая ячейка иначе) и передать это же сообщение дальше. Следом за ним (почти «вплотную») отправлять такое же сообщение и так далее.

А для получения символа из ячейки N процессор может отправить все цифры с луча данных, где хранится число N (используя метод очереди из предыдущего абзаца) «по эстафете». Отправить эти цифры по тому лучу данных, где находится нужная ячейка с номером N. И в каждой ячейке, где будет проходить этот «караван» цифр, надо будет отнимать 1 от всего каравана – начиная с первой (самой младшей цифры единиц) цифры перед тем, как отправить каждое сообщение из «каравана» к следующей ячейке. Когда от «каравана» останется единица (вместо исходного N), тогда «караван» достиг нужной ячейки, и она отправит назад к процессору (тоже по эстафете) своё содержимое.

Таким образом, и процессор, и ячейки способны отправлять и получать сообщения, обрабатывать их или игнорировать, пересылать сообщения (те же или изменённые) дальше или останавливать. Эти сообщения могут идти «вплотную» друг к другу, образуя передаваемые «пакеты» данных. Такая возможность передачи сообщений от устройства к ближайшим устройствам давно реализована в ИТ и, поэтому, мы можем включать её как элемент в свою модель.

Разумеется, нам нужно будет предусмотреть 2 режима работы процессора – режим «обычной» работы с аппаратным интерпретатором SI и режим работы с сообщениями. Причём для работы с сообщениями нам придётся придумать язык описания протоколов передачи данных в рамках нашей

модели.

Это означает, что нам придётся расширить стандартный язык SI до аппаратного языка SI для режима интерпретатора и до ещё какого-то языка в режиме работы с сообщениями. Аппаратный язык SI должен будет иметь, например, возможность отправлять сообщения и переходить в режим приёма/передачи сообщений. А после того, как каскад сообщений, порождённый указанным первым сообщением, завершит свою работу – процессор из режима приёма/передачи должен иметь возможность вернуться в режим работы интерпретатора (режим SI).

Хотя бы для этого нам потребуется дополнительные команды отправки сообщения и смены режима работы процессора. Назовём такой расширенный необходимыми командами язык SI «аппаратный язык SI». Позже мы рассмотрим его подробно.

Каким образом мы решим вопрос предотвращения/разрешения конфликтов между сообщениями, поступающими одновременно в одно устройство? Мы будем указывать для сообщения (ближайшее) время отправки и отправлять все сообщения с данным временем отправки из всех устройств одновременно. Нам останется только продумать и указать моменты отправки сообщений так, чтобы сообщения нигде и никогда не встречались. То есть, чтобы сообщения не приходили два или больше в одно время в одно устройство. И, таким образом, мы предотвратим конфликты между сообщениями.

Выберем единицу измерения – 1 момент. И у нас будет «сетка синхронизации» для отправки всех сообщений, которые уже готовы к отправке – в ближайший 1-момент. Следующий за ним 1-момент будет через один момент и т.д. И к этой основной сетке синхронизации у нас будут дополнительные «степенные» сетки синхронизации для 1/2 – моментов, 1/4 – моментов и т.д. А так же для 2-моментов, 4-моментов и т.д. При этом для простоты считаем, что у сеток синхронизации есть общие моменты синхронизации. Так, все 1-моменты присутствуют в сетке синхронизации 1/2 – моментов (каждый 2-й из 1/2 – моментов), в сетке синхронизации 1/4 моментов (каждый 4-й из 1/4 -моментов) и т.д.

Та «вселенная», в которой я рассматриваю построение Машины исполнения компьютерных алгоритмов, идеальна – в ней нет сбоев, в ней всегда исполняется та команда, которая должна быть исполнена, в ней нет ограничений на энергию для работы алгоритмов. В ней время на одну и ту же команду всегда тратится одно и то же, в ней передаваемая информация приходит к получающему устройству в точности та же, которая отправлена из передающего устройства. Поэтому в этой «вселенной» нет необходимости в контрольных суммах, механизмах замены сбойных пакетов данных на правильные и т.п. идеальные свойства, о которых в реальном мире мы можем только мечтать.

«Героями», которые обеспечивает бесконфликтную координацию работы разных сообщений и регистров в разных устройствах между собой, в этой вселенной являются сетки синхронизаций. Это резко упрощает построение архитектуры Машины исполнения компьютерных алгоритмов, исключая из неё возможность «опозданий» или «опережений» работы данной команды в сравнении с «единственно верной». Поэтому удалось избежать построения механизмов разрешения конфликтов между одновременно поступающими сообщениями, разбора вопросов о приоритете между событи-

ями и запросами данных, возникающими одновременно для одного устройства, не потребовалось использовать таймеры сверх сеток синхронизации. Не возникло необходимости разбирать случаи оптимистической и пессимистической блокировок данных при работе с ними в обработчиках событий. Удалось обойтись без понятия «сеанса» работы соседних узлов Машины друг с другом и т.д.

Мы будем рассматривать архитектуру, ориентированную на теоретическое, а не практическое использование. Поэтому удобство выбранной «вселенной» среди возможных альтернатив для построения Машины я оценивал с точки зрения теоретического, а не практического использования Машины. Если бы целью было построение действующего образца, то проще могла бы оказаться другая архитектура, потому что очень не просто модифицировать архитектуру, построенную для «идеальной» вселенной так, чтобы на её базе получился корректно и надёжно работающий компьютер в реальном мире. А вот если изначально закладывать способы корректировки реальных «несовершенств» нашего мира, то и действующий образец можно будет сделать значительно проще.

Поэтому не следует критиковать предлагаемую архитектуру с точки зрения практического дизайна реальных компьютеров. Разбираемая архитектура максимально абстрагирована для исследования свойств алгоритмов, которые есть помимо разнообразных возможных «вредящих нюансов». В какой степени данную теоретическую модель можно использовать для реализации на практике – слишком многофакторный в практическом отношении вопрос, чтобы обсуждать его тут.

Тем не менее, создаваемая тут архитектура для передачи и обработки сообщений вполне практична – если повторить её кратно и сравнивать результаты дублирующих друг друга устройств. И «принудительно» заставлять устройство с не типичным результатом принять результат «большинства». Это известный прием и он практически полностью гарантирует систему от сбоев. Вопрос «только» в дополнительном кратном расходе ресурсов. Но с точки зрения принципиальных рассуждений о бесконечной математической модели – мы достаточно богаты, чтобы позволить себе это (шучу).

Язык для аппаратного уровня отправки, приёма и обработки сообщений можно назвать по сути того, для чего он используется. А в основе предлагаемого далее языка аппаратного программирования взаимодействия устройств (узлов) Машины лежит передача сообщений между соседними устройствами (узлами). Притом предотвращение конфликтов между сообщениями достигается сетками синхронизаций, когда сообщения отправляются, получаются одновременно – в определённые моменты. Язык программирования одновременных сообщений соседним узлам (устройствам), както так. Или коротко по-русски «Язык ОССУ».

По-английски, вроде, язык «simultaneous messages to adjacent units». Или коротко «Язык SMAU», но я сильно сомневаюсь в своём английском.

V. Детерминированы ли алгоритмы передачи пакетов

В этом и следующих подразделах данного раздела будет дан обзор «странных» особенностей аппаратной архитектуры Машины исполнения компьютерных алгоритмов. Я сам далеко не сразу соглашался с обнаруженными особенностями бесконечной модели, не привычной по опыту использования конечных памяти, энергии, времени. В рассматриваемой архитектуре встречаются, например, бесконечные процессы, притом, что алгоритм завершает свою работу за конечное время с точки зрения процессора и его причинно-следственных связей.

Умственные вопросы лучше разбивать на части – когда это возможно – и решать по частям. Поэтому прежде, чем давать технические детали аппаратной архитектуры, лучше сразу разобрать её «странности». Иначе, при одновременном изложении того и другого, «странности» могут заслонить собой технические подробности и заставят читателя надолго задуматься и разбираться. Так было со мной, по крайней мере. Хотя я не отношусь к числу очень сообразительных людей, но хотелось бы написать понятно и для похожих на меня читателей.

Первый вопрос возникает из-за «странного» способа доступа к сколь угодно удалённым от процессора ячейкам памяти. И этот вопрос – детерминирован ли алгоритм передачи пакета данных?

Как уже упоминалось выше, передача данных на аппаратном уровне машины происходит от одного её узла к другому – соседнему. И передача данных зачастую происходит «пакетами».

И процессор, и ячейки способны отправлять и получать сообщения, обрабатывать их или игнорировать, пересылать сообщения (те же или изменённые) дальше или останавливать. Эти сообщения могут идти «вплотную» друг к другу, образуя передаваемые «пакеты» данных. Такая возможность передачи сообщений от устройства к ближайшим устройствам давно реализована в ИТ и, поэтому, мы можем включать её как элемент своей модели.

Но вот вопрос, который не может возникнуть у практиков (программистов), но вполне возможен для теоретиков (математиков):

А является ли детерминированной вычислительная система, использующая неограниченное количество устройств, одновременно исполняющих алгоритмы? Напоминаю, что недетерминированный алгоритм может выполнять одновременно много вычислительных действий, притом количество этих процессов на каждом шаге может нарастать даже экспоненциально. За счёт этого недетерминированная вычислительная система может перебирать экспоненциальное количество вариантов одновременно и обнаруживать «угадавший» правильный вариант процесс после завершения всех остальных.

Во-первых, наша система не может наращивать свои действия экспоненциально, потому что для ячеек луча данных есть только одно направление увеличения процессов – в направлении от процессора. Поэтому в пределе возможна только арифметическая прогрессия для увеличения количества процессов в лучшем случае. С процессором та же история, хоть у него не один луч, а конечное их количество. Что не принципиально в плане характера роста «каскада» генерирующих друг друга сообщений.

Кстати, для недетерминированного алгоритма с экспоненциальным ростом вариантов вычисле-

ния не существует модели, конечно. Есть не уменьшаемые размеры ячеек и моментов времени на шаг вычислений из-за соотношений Гейзенберга – иначе непредсказуемо велики колебания импульсов и энергий. Поэтому радиус шара данных, где «плодятся» по экспоненте процессы вычисления, будет очень скоро расти быстрее скорости света, что невозможно, разумеется.

Во-вторых, мы можем заменить одновременное движение всех элементов пакета данных на движение строго по одному элементу – если «сообщение разрешения» будет передаваться из одного конца пакета в другой, и отрабатываться будет только тот элемент, где данное сообщение находится сейчас. В итоге мы сделаем все те же действия что и при одновременном движении пакета, плюс ещё передача и обработка «разрешающего» сообщения. А в результате получим то же самое, что при одновременном движении всех элементов пакета.

Заметим, что мы не получили, и не потеряли ни одного «варианта» вычисления, когда перешли к исполнению строго единственного процесса (когда только в одном узле Машины в каждый момент времени что-то делается). Мы только получили дополнение в виде бесполезной (с точки зрения получения результата) передачи «разрешительного сообщения» туда-сюда много раз. И результат получился точно такой же, как при одновременном движении всех элементов пакета данных, только получили мы его гораздо медленнее и с гораздо большей затратой энергии.

Из сказанного следует, что нужно уточнить своё теоретическое понимание детерминированного алгоритма. Алгоритм может выполняться не в одной «точке» (а во многих) и при этом он вполне может быть детерминированным. Так устроена реальность, что она может «сжимать» некоторые последовательные вычисления во времени в последовательность в пространстве. Элементы пакета данных идут пусть одновременно во времени, но они последовательны в пространстве, а за счёт этого каждый элемент пакета данных приходит к результату обработки предыдущего элемента, преодолев вместо 2 шагов времени 1 шаг времени и 1 шаг пространства.

Да и не может вычислительный процесс идти только в одном атоме из всех атомов вычислительной системы в каждый момент времени. Вычисление – это заведомо «объёмный» процесс. И мысль в нашей голове занимает в каждый момент не один нейрон, а тоже имеет какой-то «объём». Хотя это у меня уже субъективное (шучу).

Таким образом, передача пакетов данных в Машине исполнения компьютерных алгоритмов – детерминированный процесс. Происходит лишь доставка однозначно «детерминированных» значений символов в/из ячеек. Вопрос лишь в способе этой доставке. И для доставки в реальности есть, например, велосипед и трейлер. Трейлером можно привезти больше и быстрее – если есть подходящая дорога, конечно.

Ситуация тут, скорее, противоположна недетерминированному алгоритму. Доставка данных всегда хуже, чем их наличие «сразу» у процессора. Можно говорить о «логистическом КПД» данной детерминированной вычислительной системы при решении данной задачи. Отношение собственно времени на вычисление (не включая время на доставку данных) к общему времени от начала вычисления до получения результата – включая время на решения и ожидание решения вопросов доставки детерминированных данных. А если ещё и доставку электричества иногда прекращают – то и это время простоя надо добавить.

VI. Что такое момент окончания протокола передачи относительно процессора, как аксиоматизировать время, теневой канал связи луча данных

Для предотвращения конфликта сообщений между собой условимся, что «эстафетные» сообщения передаются по лучу данных в направлении от процессора не чаще, чем через 1 момент. Из-за этого и скорость движения не может быть выше, чем 1 ячейка за 1 момент.

Дело в том, что такие сообщения могут уходить «в бесконечность» или просто на сколь угодно большое расстояние. Поэтому, если не ограничить скорость эстафетных сообщений «от процессора», то новая эстафета может догнать какую-нибудь древнюю эстафету на расстоянии 10 световых лет от процессора и всё там испортить.

А вот эстафеты «к процессору» или передачу сообщений в пределах нескольких ячеек – обычно мы сможем запускать с более высокой скоростью. Потому что расстояния в таких случаях ограничены и мы можем предвидеть, что там происходит и как нам предотвратить конфликты. Но тут требуются аккуратность и обоснования, конечно.

Когда можно считать, что данный протокол получения/передачи данных исполнен в достаточной мере, чтобы считать ячейки на луче данных готовыми к использованию? Не обязательно в тот момент, когда все ячейки на луче данных получили свои значения в соответствии с этим протоколом получения/передачи. Требуется лишь, чтобы «окончательный результат» данного протокола имелся только в 1-й ячейке, а в остальных формировался не позднее, чем со скоростью одна ячейка за один момент – удаляясь от процессора.

И ещё необходимое условие завершения протокола передачи в отношении данной ячейки – чтобы он уже не обращался к той ячейке, обработка которой считается законченной. Потому что даже если при поступлении сообщения от протокола передачи и не произойдёт никаких изменений, то всё равно это сообщение может вступить в конфликт с сообщением-запросом данных, пришедших от процессора.

То есть – момент оперативного завершения протокола зависит от места, относительно которого нас интересует это «оперативное завершение». Если протокол «оперативно» исполнен, то его результатом можно пользоваться при соблюдении принятых соглашений. А принятым соглашением является доступ к ячейкам луча данных при помощи сообщений, движущихся не быстрее, чем 1 ячейка за один момент в направлении от процессора.

У нас и в жизни момент окончания всяких процессов обычно «оперативный», а не абсолютный. Например, мы нажимаем выключатель, и говорим в темноте, что погасили лампу. Но это совсем не исключает того, что через 10 лет на недалёкой звезде рептилоидов для местных ящеров-астрономов мы всё ещё тянемся пальцем к кнопке выключателя, и для них свет нашей лампы всё ещё продолжает гореть.

Если правильно строить и использовать протоколы передачи, то практически никакого времени не тратиться на запись данных где-то на луче данных. Да, есть время для отправки процессором соответствующих сообщений по лучу данных по принципу «отправил-забыл». И сразу (с соблюдением соглашения о передаче сообщений «не быстрее 1 ячейка за 1 момент от процессора») можно считать их записанными в нужном месте. Сразу можно отправлять запрос на их чтение.

Даже если они должны быть записаны на расстоянии 10 световых лет от процессора – они придут и будут записаны там раньше, чем за ними придёт запрос на их чтение. Пусть даже запрос на чтение «летит», отставая лишь на пару ячеек от пакета с данными для записи.

Кстати, именно при рассмотрении протоколов приёма/передачи просматривается та теория, в которой можно было бы аксиоматизировать понятие времени для данной модели. Потребовалась бы аксиоматизация перехода ячеек из одного состояния в другое в зависимости от поступающих сообщений.

Есть такой нюанс, что в дальнейшем для удобства мы будем отчасти использовать аппаратный язык SI, который обеспечивает работу программы-интерпретатора стандартного языка SI. И исполнение команд аппаратного языка SI зависит не от сообщений, а от текущего номера позиции начала команды в программном тексте ПЗУ. Таков порядок организации последовательности исполняемых команд в традиционных языках программирования.

Разумеется, доступ к произвольным ячейкам бесконечных лучей данных будет осуществляться при помощи сообщений, и описывать алгоритмы в этой части будет язык SMAU. Но у нас есть много работы, которая может быть выполнена в пределах ограниченной памяти – конечных регистров. В том числе инициацию протоколов передачи сообщений для работы с лучами памяти можно делать такими средствами.

Работа с протоколами передачи данных по бесконечным лучам данных, когда процессор реагирует на сообщения, выполняя управляющую роль в протоколе – это работа процессора в режиме SMAU. Работа процессора с ограниченной программой из ПЗУ и ограниченными регистрами памяти, при «классическом» выполнении последовательности команд этой программы – это работа процессора в режиме исполнения программы из ПЗУ которую считаем записанной на аппаратном языке SI. А режим, в котором процессор выполняет программу на аппаратном языке SI – это режим SI процессора.

Процессор может встречать команду смены режима – в режиме SI он может отправить сообщение и перейти (по соответствующей команде) в режим SMAU. А после того, как протокол будет исполнен, процессор в завершающем протокол обработчике событий, написанном на языке SMAU, встретит команду перехода в режим SI. И после этого процессор вернётся к исполнению программы на языке аппаратного SI. А именно – к исполнению следующей команде за командой перехода в режим SMAU, которая и привела процессор к исполнению его роли в протоколе передачи. И эта роль оперативно закончена возвратом к продолжению исполнению программы на аппаратном языке SI.

Но аппаратный язык SI при отработке (в программе-интерпретаторе) любой команды стандартного языка SI заведомо ограничен текстом своей программы из ПЗУ, размерами команд, памятью (ограниченной!) регистров. Поэтому на интерпретацию каждой команды стандартного языка SI программой аппаратного языка SI времени тратится в пределах одной и той же константы $T_{SiMaxCommand}$ – если не учитывать в данном времени время на работу процессора в режим SMAU до его возврата из режима SMAU в режим SI.

Тогда $T_{SiMaxCommand}$ – это «постоянная часть» затрат времени на интерпретацию одной команды стандартного языка SI. Точнее – максимальная величина «постоянных затрат» на одну команду

стандартного языка SI.

Но самая интересная – переменная часть, в которой происходит работа с неограниченными лучами данных и сообщениями, обеспечивается на уровне языка SMAU. И вот как раз для этой части понятен принцип аксиоматизации состояний узлов Машины и смены этих состояний в зависимости от поступающих сообщений. Пояснения будут даны до конца данного подраздела.

Для простоты мы условились считать, что отправка сообщений и их поступление в узел-адресат происходят одновременно. И можно считать, что есть только 4 сетки синхронизации, мельче 1 момента – до 1 / 16 моментов включительно. Тогда можно считать, что для сообщения, синхронизированного по сетке 1-моментов каждая 1 / 16 момента ожидания – это тоже смена состояния, приближающая очередной 1-момент.

То есть – потребовалось бы привести 1-моменты к единицам измерения 1 / 16 моментов или ввести мелкую единицу 1 мгновение, такое, что 16 мгновений равны 1 моменту. Легко запомнить, кстати – потому что ещё одно мгновение – и мы вспоминаем про 17 мгновений Штирлица.

В принципе, интерпретацию программы стандартного языка SI, записанную на луче данных $Program$, можно было бы осуществлять только на языке SMAU. Но тогда пришлось бы писать протоколы передачи сообщений и обработчики соответствующих событий даже для регистров, для лексического разбора команды из программного текста луча данных $Program$ и т.д. Делать всё то, что заведомо реализовано в рамках ограниченной памяти на практике.

А при использовании традиционного по своему устройству аппаратного языка SI можно считать сделанным всё то, что реализовано на практике для заведомо фиксированных по максимальному размеру используемой памяти задач. Можно сослаться на сделанное другими и избавить себя от части занудной и не маленькой работы. Я обязательно использую эту общепринятую хитрость в дальнейшем.

Кроме того, окончательную «сборку» порядка исполнения команд программы гораздо проще реализовать на традиционном языке программирования – с его повторным использованием кода (циклами), наглядной последовательностью команд, доступностью значений одинаковых переменных из разных команд без хитрых протоколов передач по лучам данных и т.п. Возможно, это лишь дело привычки, но это общая привычка практически для всех людей, имеющих опыт программирования. Поэтому использование аппаратного языка SI помимо языка SMAU представляется мне уместным.

С другой стороны, если бы была задача аккуратной аксиоматизации времени, то всё, что нужно для интерпретации программы на луче данных Program, пришлось бы писать, на языке SMAU. Потому что для аксиоматизации времени необходимо видеть элементарный уровень получения и передачи данных – когда они идут от ячейки к соседней ячейке, когда смена состояний происходит под действием таймеров синхронизации в разных узлах Машины и т.д.

И тогда нам нужно строить описание значений для каждого элемента (регистров и таймера синхронизации) узла Машины в начальный момент времени и иметь таблицу изменения этих значений в следующий момент времени в зависимости от текущего состояния данного узла и соседних с ним узлов (которые могут передавать ему сообщения).

Притом таблица изменения значений вообще конечна. Потому что приходит только одно сообщение (во избежание конфликтов) в узел, отправляется не больше 10 сообщений из узла в данный момент времени. И всё это записано в конечных, по количеству возможных значений, регистрах узлов Машины. И таймер синхронизации тоже крутится по циклу – от 1 мгновения до (например) 8 моментов. Соседей у ячейки только 2, у процессора – все 1-е ячейки лучей данных (и лишь не более чем к 10 он отправляет сообщения в данный момент). То есть – имеется только 3 типа узла:

1. Процессор
2. Первая ячейка луча данных
3. Не первая ячейка луча данных

И конечная таблица изменения их состояний в зависимости от данного узла и от всего лишь одного (не персонализированного для данного узла) соседнего узла, который отправляет тебе сообщение (если отправляет) в следующий момент времени. Притом поступающее сообщение не зависит от того узла, который тебе его прислал – вся информация содержится в самом сообщении и вызывает один и тот же обработчик события – от кого бы оно ни поступило в таком виде.

У нас тогда есть конечная таблица переходов для всего лишь 3 типов узлов. Даже «общей» для всей Машины исполнения компьютерных алгоритмов программы нет – в отличие от Машины Тьюринга.

Есть только «маленькие программы» в виде набора обработчиков событий, действующих в зависимости от событий в своём узле. Да и реализовать эти обработчики можно без программ – потому что каждый обработчик оказывается небольшим набором команд, исполняемых друг за другом 1 раз каждая. Такое легко реализуется на уровне «железа».

Вместо «общей» программы у нас тогда есть только данные – включая программный текст интерпретируемой программы на луче данных $Program$. Пользователь после записи всех «входных» данных на лучах данных (включая лучи $Program$ с программным текстом и луч $i_{Program}$ с 1) отправляет соответствующее сообщение в процессор и начинается процесс интерпретации программы на стандартном языке SI и исполнение записанного на нём алгоритма.

И тут мы видим, как аксиоматизируется время в соответствующей теории 1-го порядка, если построить её для аппаратной архитектуры Машины исполнения компьютерных алгоритмов с интерпретатором стандартного языка SI, написанном на одном только языке SMAU (в виде обработчиков событий для каждого типа узла Машины). А именно, время будет тогда аргументом t «алгоритмической последовательности». Где каждое следующее значение элементов узла (регистров и таймера синхронизации) для аргумента $(t+1)$ зависит только от предыдущих значений элементов в момент t, притом только от значений в самом данном узле и не более чем в одном соседнем.

И вот уже на базе данной теории мы бы могли доказать теоремы о работе программы, записанной на стандартном языке SI на луче данных $Program$ при её интерпретации вышеописанным способом. Но я не буду делать теорию для Машины, как уже было сказано. Потому что если решать подобные задачи, то мне нужно больше жизней. Просто в рамках теории строк надо будет дать в статье (про теорию, а не в этой статье про модель) определение для функции-«исполнителя» алгоритмов. Такого вида, например:

$\mathrm{ff}(var, i_{Step}, Program, Init)$, где:

var – имя переменной (луча данных, входного аргумента), «$i_{Program}$», «x_1», например;

i_{Step} – Номер шага программы от начала её работы, один шаг означает отработку одной команды;

$Program$ – текст программы;

$Init$ – текст со значениями входных аргументов;

Текст со значениями входных аргументов может быть записан в формате присваиваний с использованием псевдокоманд $\mathrm{Const}()$ и подобных. Важно предусмотреть неопределённость некоторых «хвостов» данных – потому что не все аргументы программа читает до конца. И в рамках теории строк мы сможем доказать, что результат (заключительный шаг работы программы и значение на оговоренном луче данных) не будет зависеть от некоторых «хвостов». Придётся придумать формат для «хвоста» аргумента, когда этот «хвост» не нужен для расчёта. Например, в тексте $Init$ будут такие строки:

$\mathrm{Arg_1}(0) = \mathrm{Const}($ Это нужная начальная часть строки $)$;

$\mathrm{Arg_1}() = Unknown_1$;

А время, которое уходит на вычисление – не будет доказываться (у нас же не будет в этой работе теории для Машины, будет только сама модель), а будет видно по дальнейшему изложению. И на исполнение каждой команды будет требоваться время в соответствии с линейной зависимостью от размера используемых для исполнения команды данных. Общее время будет равно сумме времен по всем исполненным командам, разумеется.

Используемые данные для одной команды – это сумма размеров строк и подстрок из следующего списка, которые относятся к данной команде:

1. Начальная подстрока на луче данных $Program$ до текущей команды включительно;

2. Строка на луче данных $i_{Program}$;

3. Для команды $goto$ – начальная подстрока на луче данных $Program$ до найденной метки, или вся строка $Program$ и строка в $Command$ (регистр, где хранится текущая исполняемая команда), если метка не найдена;

4. Для команды присваивания типа $\mathrm{x_1}() = \ldots$ строка на луче данных x_1;

5. Для команды присваивания типа $\mathrm{x_1}(i) = \ldots$ где $i \neq 0$, подстрока на луче x_1 до i символа (или конца строки, если он раньше) и строка i;

6. Для команды $\mathrm{len}(x_1)$ – строка на луче данных x_1 и строка десятичной записи результата функции $\mathrm{len}(x_1)$;

7. $\mathrm{Chr}(x_1)$ – значимая часть строки x_1;

8. $\mathrm{Value}(x_1)$ – строка x_1;

9. $\mathrm{from}(x_1, x_2)$ – начальная подстрока на луче x_1 до x_2 символа (или конца строки, если он раньше) и строка x_2;

10. $\mathrm{str}(x_1, x_2, x_3)$ – начальная подстрока на луче x_1 до $\max(x_2 + x_3 - 1, 0)$ символа (или конца строки, если он раньше) и строки x_2, x_3;

11. $\mathrm{Comp}(x_1, x_2)$ – начальная подстрока x_1 (или x_2 – не важно), до первого отличающегося с другой строкой символа (или конца более короткой строки) и строка результата (строка десятичного

числа 2 – максимум);

12. $\mathrm{Next}(x_1)$ и $\mathrm{Previous}(x_1)$ – строка x_1.

Пункты 1 и 2 из данного списка имеют отношение к любой команде, разумеется.

Надо отметить, что можно оптимизировать протоколы передачи данных так, что на выполнение команды вида:

$\mathrm{x}_1(0) = \mathrm{from}(x_1, x_2)$;

будет уходить времени пропорционально числу x_2 или пропорционально размеру строки x_1, если конец строки раньше ячейки номер x_2, плюс размер (количество цифр) на луче данных x_2. Да, это верно даже для бесконечной строки. Потому что через один момент после того, как символ с позиции x_2 поступит в позицию 1 – можно будет считать протокол передачи сообщений о переносе символов в начало строки оперативно исполненным (если его правильно построить). Так как остальные символы будут поступать на места 2, 3, 4, 5 и т.д. с темпом 1 ячейка за 1 момент. Это вовсе не означает, что за 1 момент процессору доступны все символы – ведь чтобы получить конкретный символ, процессор должен будет его запросить, и ждать его поступления, пока сообщения из пакета запроса и сообщение ответа будут преодолевать оставшееся расстояние между процессором и данным символом.

Аналогично можно оптимизировать и команду виды:

$\mathrm{x}_1(0) = \mathrm{str}(x_1, x_2, x_3)$;

Но для простоты мы будем всегда перебрасывать запрошенные символы в начало луча данных – его «теневую часть», а уже после этого – в нужное место того луча данных, куда надо перенести («присвоить») эту последовательность символов.

Обычно луч данных «откуда символы» и луч данных «куда символы» - это разные лучи данных. Но, в принципе, это может быть один и тот же луч данных. И тогда сообщения, касающиеся «теневых символов» (теневые символы записаны в специальном регистре sv в каждой начальной ячейке луча-отправителя) могут конфликтовать с сообщениями, касающихся «реальных символов» (регистры rv в каждой ячейке луча-получателя). Конфликтовать они будут в том случае, если луч-получатель и луч-отправитель – это один и тот же луч, по которому одновременно идут сообщения разных типов.

Для предотвращения подобных конфликтов будем считать, что у каждого луча данных есть 2 канала передачи сообщений – основной и «теневой». И каждый канал обслуживается своим упрощенным процессором ячейки. А следить, чтобы обработчики сообщений не записывали разные значения в одни и те же регистры – придётся нам самим, и придётся правильно строить протоколы передачи с учётом этого условия.

Последний подраздел о «странных» особенностях Машины исполнения компьютерных алгоритмов будет в разделе про язык SMAU. Некоторые нюансы программирования обработчиков событий для бесконечной модели станут, заодно, введением в язык SMAU.

2 Аппаратный интерпретатор языка SI

I. Дополнительные функции и свойства в аппаратном языке SI

Нам потребуется расширить язык SI для того, чтобы написать на нём программу аппаратного интерпретатора стандартного языка SI. Впрочем, для аппаратного интерпретатора можно использовать какой угодно язык программирования, так как в данной части наша модель полностью соответствует практическим вычислительным системам с их ограничениями на размер используемой памяти. И в качестве инструментов доступа к ограниченной памяти мы будем использовать регистры.

Регистры – это конечные начальные «отрезки» лучей данных. У регистров есть имена в общем пространстве имён с лучами данных. Все символы регистров доступны «в один ход» для процессора, потому что разрядность процессора и пропускная способность каналов связи Машины достаточна для адресации каждой ячейки регистра и непосредственного обращения к любой ячейки регистра со стороны процессора.

Поэтому, если мы можем использовать какой-то язык программирования на практике, то мы можем его использовать и для написания программы аппаратного интерпретатора стандартного языка SI. Но для единообразия (чтобы не читать дополнительно книги по программированию) мы используем расширенный SI – аппаратный SI – для написания интерпретатора стандартного SI, который уже сможет использовать неограниченные лучи данных.

Напомню, что интерпретатором языка программирования L называется программа, которая читает некоторый программный текст на языке L, исполняет команды, которые записаны в этом программном тексте, использует входные аргументы и переменные, и меняет их в соответствии с командами из этого текста на языке L. То есть, интерпретатор языка L «оживляет» программный текст на языке L и приводит к результату работы этого «ожившего» программного текста.

К тому, что было изложено в подразделе 2 «Машина исполнения компьютерных алгоритмов (МКА)» раздела 1 «Практика вычислений и модель «Машина исполнения компьютерных алгоритмов» добавим следующие функции и методы (продолжив нумерацию):

12) $x_1.Register$ и $x_1.SRegister$

Свойство $.SRegister$ точно такое, как $.Register$, но относятся к «теневой строке» луча данных, поэтому далее речь будет только про свойство $.Register$, но совершенно аналогичное характерно и для свойства $.SRegister$

$.Register$ – это свойство, которое рассматривает луч данных как регистр, с ячейками которого можно оперировать в «один ход». То есть – у нас есть какая-то разрядность у процессора, которой достаточно для имён лучей данных, для чисел и строк, используемых процессором на «аппаратном» уровне. И вот в рамках этой разрядности начало луча данных может рассматриваться как регистр – но только если в пределах имеющейся разрядности виден конец строки (видна пустая ячейка после последовательности не пустых ячеек от 1й включительно).

Если же значение на луче данных x_1 выходит за рамки ограничений разрядности, то свойство $x_1.Register$ имеет значение числа $/\#\, Err\, out\, of\, range\, \#/$.

Обозначение вида: /# Номер #/ было рассмотрено в п. 11.4 описания языка SI.

Из сказанного понятно, что свойство $x_1.Register$ использует не все возможности разрядности для пользовательских нужд, а расходует некоторый ресурс разрядности процессора на служебные нужды. В частности, количество цифр в $/\#\, Err\ out\ of\ range\, \#/$ больше на 1 (или ещё больше), чем максимально допустимый размер строки на луче данных x_1, который можно получить через $x_1.Register$ без ошибки.

Свойство $x_1.Register$ заметно упрощает написание программы, потому что с начальной частью луча данных можно оперировать не как с бесконечным лучом данных, а как с регистром. К тому же можно использовать луч данных (x_1, например) как конечный регистр, если записывать его в программном тексте следующим образом:

$x_1.Register$

Но корректным такой метод оказывается лишь в том случае, если этот «регистр» не равен $/\#\, Err\ out\ of\ range\, \#/$.

Кстати, само число $/\#\, Err\ out\ of\ range\, \#/$ можно присваивать в регистр (или лучу данных через свойство $.Register$). Потому что это число обозначает и переполнение луча данных как регистра и само себя. Просто работать с этим числом через свойство $.Register$ луча данных обычным образом невозможно. Ведь свойство $.Register$ луча данных возвращает это число и для случая, когда числа $/\#\, Err\ out\ of\ range\, \#/$ нет на луче данных, но строка при этом превышает предельный размер для свойства $.Register$ – и для случая, когда на луче данных записано именно число $/\#\, Err\ out\ of\ range\, \#/$.

Но для настоящего регистра (а не для свойства $.Register$ луча данных) число $/\#\, Err\ out\ of\ range\, \#/$ вполне «нормальное», потому что регистр не может быть переполнен – ничего «лишнего» в него просто «не влезет». А всё что «влезает» - показывается «как есть» - никогда не подменяясь значением $/\#\, Err\ out\ of\ range\, \#/$, которое будет лишь в случае записи в регистре самого числа $/\#\, Err\ out\ of\ range\, \#/$.

Свойство $.Register$ можно использовать и для присваивания вида:

$x_1.\mathrm{Register}(0) = \mathrm{Value}(0);$ // Присваивание в языке SI

$x_1.Register = 0$ // Присваивание в языке SMAU

После чего весь луч данных (а не только $x_1.Register$) содержит присвоенное значение (ноль в данном случае).

Свойство $.Register$ работает только с очень ограниченной частью луча данных, где:

1. Помещается максимально длинная команда программы с луча данных $Program$, включая перевод строки и пустую ячейку после ячеек со значимыми символами;

2. Помещается любой номер, который выдаёт значимый символ для функции $\mathrm{Chr}()$ плюс ещё два символа – чтобы было место для первого номера i, такого что $\mathrm{Chr}(i) = \ominus$ и для пустого символа – чтобы увидеть конец строки. Разумеется, верным является равенство $\mathrm{Chr}(/\#\, Err\ out\ of\ range\, \#/) = \ominus$ и это соответствует всем строкам, которые могут скрываться за значением $/\#\, Err\ out\ of\ range\, \#/$ для свойства $.Register$.

3. Помещается максимальный результат функции $\mathrm{Comp}()$ – число 2 плюс один символ (пустой).

И да $.Register$ – медленное свойство, что обязательно необходимо учитывать при его применении в обработчиках событий. Дело в том, что значения ячеек с луча данных x_1 свойство $x_1.Register$ получает в соответствии с предположением об оперативном завершении протоколов передачи в отношении луча данных x_1. То есть, из первой ячейки значение будет получено (или записано) практически мгновенно (в пределах $1/16$ момента). А вот из/в 2-й – через 1 момент, из/в 3-й – через 2, и так до первой пустой ячейки – через $\mathrm{len}(x_1)$ моментов.

Получив значение из пустой ячейки – свойство .Register возвращает результат. И записывает за такое же время. Так же как и любой регистр – используется в командах до первой пустой ячейки включительно. Но настоящие регистры работают очень быстро (на 2 порядка быстрее примерно) в сравнении с 1 моментом – так надо для предотвращения конфликтов сообщений, что будет разобрано при разборе применения языка SMAU.

Но при этом всё равно $.Register$ чуть быстрее протоколов передачи сообщений для тех же нужд и намного проще. Да, надо применять его так и тогда, чтобы не завершившиеся оперативно протоколы передач уже не влияли на используемые свойством $.Register$ ячейки, но то же пришлось бы учитывать и при использовании протоколов передачи вместо свойства $.Register$, но их бы пришлось ещё и придумывать.

13) Оператор @

Обозначение @$Source_1$ – это луч данных или регистр, имя которого записано в регистре (не на луче данных, а именно в регистре) $Source_1$. Приоритет этой операции – наивысший. И запись:

@$Source_1.Register$ следует понимать как $(@Source_1).Register$.

И если в регистре $Source_1$ записана строка «y_1», то команда

@$Source_1.\mathrm{Register}(0) = \mathrm{Value}(ChrEnter)$; // Присваивание в языке SI

@$Source_1.Register = ChrEnter$ // Присваивание в языке SMAU

Запишет строку, состоящую только из одного символа перевода строки на луч данных y_1.

14) $SendTo_i\,Unit\,(/\#$ Тип сообщения $\#/, msgValue)$, $SendToS_i\,Unit\,(/\#$ Тип сообщения $\#/, msgVal$ $Setmode\,SMAU$, $Setmode\,SI$.

Два варианта команды отправки сообщения ($SendTo_i\ldots$ и $SendToS_i\ldots$) для основного и теневого канала связи луча данных, о чем упоминалось в 6 подразделе предыдущего раздела.

Команда отправки сообщения является ключевой в рамках аппаратного языка SI для доступа процессора к любым ячейкам на неограниченных лучах данных.

Эта команда отправляет сообщение $(/\#$ Тип сообщения $\#/, msgValue)$ в центральный процессор или прилежащий к процессору узел (обозначение тут для любого варианта $Unit$). Мы сейчас обсуждаем именно процессор и прилежащие к нему узлы, а про не первые ячейки лучей данных будут другие разделы. Отправляет это сообщение процессор в ближайший момент относительно той сетки синхронизации, которая заданна индексом i.

Но нужно учесть, что если данное сообщение отправляется в программе на аппаратном языке SI, то процессор находится в режиме SI, и не реагирует на поступающие сообщения. Чтобы реагировать на сообщения, ему надо будет прежде переключиться в режим SMAU. Переход между режимами происходит при помощи команд:

$Setmode\,SMAU$; // Переключение в другой режим из программы на аппаратном языке SI

$Setmode\,SI$ // Переключение в другой режим из обработчика событий на языке SMAU

Поэтому без переключения режима в режим SMAU возникнет ошибка, если в процессор придёт сообщение, когда процессор после отправки сообщения в режиме SI продолжил работать в режиме SI, а сообщение либо было отправлено в процессор, либо инициированный отправленным сообщением протокол передачи привёл к приходу сообщения в процессор.

Если протокол передачи не предполагает никаких сообщений в процессор, то смена режима не требуется. Но результат передачи сообщения возникает не сразу, что надо учитывать, и в некоторых случаях ждать результата при помощи команды $wait$, о которой будет сказано ниже.

Если же отправка сообщения инициирует протокол передачи, который потребует реакции процессора на сообщение(я), то у нас есть время на то, чтобы переключиться в режим SMAU после отправки сообщения. Потому что какое-то время сообщение находится в буфере отправки сообщений, дожидаясь момента отправки в ближайший момент сетки синхронизации, заданной индексом i.

Но нам надо обеспечить, чтобы между командой отправки сообщения и его уходом было время для исполнения команды

$Setmode\,SMAU$;

Достигается это при помощи команды $wait_i$ таким способом, например:

$wait_0$;

$SendTo_0Proc\,(/\#$ Увеличить число в $@Target_1$ на количество символов в $@Source_1\,\#/, \ominus)$;

$Setmode\,SMAU$;

Первая команда ($wait_0$) приостанавливает работу программы в режиме SI до ближайшего момента в сетке синхронизации 1-моментов. Вторая команда помещает сообщение в буфер отправки для отправки сообщения в ближайший 1-момент. Отправлено сообщение будет в процессор. И отправлено оно будет (и получено) уже не в тот момент, которого мы дождались после команды $wait_0$, а в следующий за ним. А последняя команда переводит процессор в режим SMAU, в котором он сможет отреагировать на отправленное сообщение.

Две последних команды выполнились быстрее, чем за 1 / 16 момента, поэтому от одного 1-момента до другого 1-момента хватит времени и на переход в режим SNAU и на ожидание 1-момента ухода сообщения, а после этого ещё и на поступление результата.

В конце того, как произойдут все необходимые события в режиме SMAU, возникнет событие для процессора из-за некоторого завершающего полученного им сообщения. В конце обработчика события, вызванного этим сообщением, будет стоять команда:

$Setmode\,SI$ // Это команда языка SMAU

И после этой команды процессор вернется в режим SI к исполнению аппаратной программы (аппаратного интерпретатора стандартного языка SI). Очередной командой будет та, которая расположена в ПЗУ за последней исполненной командой:

$Setmode\,SMAU$;

А результатом команды отправки сообщения будут некоторые изменения на лучах данных и в

регистрах. В том числе и такие изменения в некоторых случаях, когда в недоступные для аппаратного языка SI ячейки внесены необходимые изменения. И/или некоторые данные из недоступных для аппаратного языка SI ячеек попали в доступные ему регистры.

i – индекс, соответствующий логарифмической (по основанию 2) шкале сеток синхронизации. Если он равен 0 (нулю) – то это сетка синхронизации для 1-моментов (2^0-моментов). Если $i = -1$, то сетка синхронизации для 1 / 2 – моментов (2^{-1}-моментов). Если $i = 1$, то сетка синхронизации для 2-моментов (2^1-моментов). И т.д.

Команда отправки сообщения не ждёт ближайшего момента из сетки синхронизации в соответствии с индексом i, а просто оставляет готовое сообщение в буфере отправки сообщений данного узла (процессора тут) Машины. После этого может быть выполнена следующая команда и т.д. В том числе в буфер отправки могут быть добавлены другие сообщения для отправки в другие соседние устройства.

i не является аргументом, i является частью имени команды отправки сообщения, указывающая на её разновидность в плане используемой данной командой сеткой синхронизации.

$Unit$ – это может быть:

$Proc$, когда процессор отправляет сообщение самому себе для перехода в режим обработчика событий с вызовом соответствующего обработчика;

x_1, или имя любого другого луча данных – когда процессор отправляет сообщение в первую ячейку этого луча данных;

@$Source_1$, или в сочетании с любым другим (отличным от $Source_1$) регистром – когда процессор отправляет сообщение в первую ячейку того луча данных, имя которого записано в указанном регистре;

$(/\#$ Тип сообщения $\#/, msgValue)$ – собственно само сообщение. Состоит из двух аргументов – типа сообщения, представленного некоторым номером и самого содержимого. Точнее будет записать так:

$(msgType, msgValue)$

Считаем, что разрядность процессора и пропускная способность канала для передачи сообщений достаточны для одновременной передачи и типа сообщения, и значения этого сообщения. Обычно значением будет или символ (включая пустой), или число. При этом оба аргумента ограничены и находятся в пределах весьма небольшого максимального размера каждый. Типов сообщений меньше 1000, аналогично и со вторым аргументом – чисел не больше 1000, а для передачи символов нам хватит и размера в 1 символ.

15) $wait$, $wait_i$

Команда $wait_i$ приостанавливает выполнение команд устройством (процессором здесь) кроме передачи сообщений, которые сейчас находятся в буфере отправки сообщений. Смысл индекса i такой же, как в команде $SendTo_i \ldots$. И команда $wait_i$ ждёт ближайшего момента в той сетке синхронизации, которую задаёт индекс i. После этого приостановка выполнения программы прекращается, независимо от того, что какие-то сообщения могли ещё не уйти из буфера отправки сообщений.

Одна команда $wait_i$ не позволяет точно прогнозировать на какое время программа приостановила свою работу. Действительно, $wait_0$ ждёт ближайшего 1-момента, но сколько времени до этого ближайшего момента – не всегда ясно. Чтобы гарантированно приостановить работу программы на 2 момента (чтобы дождаться оперативного завершения протокола, например) потребуется повторения команды, вот так:

$wait_1$;

$wait_1$;

Время, на которое приостановится программа, будет от 4 до 2 моментов. Можно довести погрешность до 1 / 16 момента, если поставить много команд $wait_{-4}$;

Команда $wait$ без индекса ждёт, когда буфер сообщений устройства (процессора – в разбираемом сейчас случае) полностью очиститься. То есть, все ожидающие отправки сообщения дождутся момента своей отправки, и будут отправлены. И только после отправки всех уже подготовленных сообщений команда $wait$ позволяет процессору перейти к исполнению следующей команды.

Напоминаю, что в режиме SMAU надо считаться с угрозой возникновения конфликтов между блокировкой работы обработчиков событий узла, пока действует команда $wait$, и приходом сообщения в узел в это же время. Поэтому необходимо так строить протоколы передачи, чтобы подобных конфликтов не возникало.

16) $\mathrm{find}(Source_1, Source_2, Source_3)$

В качестве любого аргумента команды find стоит:

- либо регистр,

- либо свойство $.Registr$ некоторого луча данных (только строка на луче данных тогда не должна переполнять свойство $.Registr$),

- либо конструкция с оператором @ (например, $@Source_1.Register$) для работы со свойством $.Register$ луча данных (имя которого записано в регистре $Source_1$ в примере $@Source_1.Register$).

Что делает команда $find$ – было разобрано при разборе стандартного языка SI (во 2 пункте 2-го подраздела прошлого раздела) команды $goto$.

Команда $find$ в аппаратном языке SI нужна нам для лексического разбора очередной команды, полученной с луча данных $Program$ и помещённой в регистр $Command$. Мы разбираем её «на запчасти» (в другие регистры), получая:

Имя команды стандартного языка SI, или 2 имени (включая имя луча данных, которому присваивают значение), если это команда присваивания, аргументы (имена лучей данных), относящиеся к командам/функциям/лучам для присваивания.

17) Арифметические действия. Разумеется, арифметические действия для ограниченных регистров давно реализованы на практике, и мы переносим эту возможность в аппаратный язык SI и в язык SMAU.

II. Регистры процессора и вспомогательные лучи данных

Следующие регистры процессора используются не только в режиме SI процессора, но и в режиме SMAU, когда процессор обрабатывает события, возникающие из-за сообщений от соседних узлов Машины, и сам отправляет сообщения им и себе.

Что такое регистры, было сказано в начале предыдущего параграфа. Дополню лишь, что при этом с регистрами можно работать и как с лучами данных. Просто последняя ячейка регистра передаёт сообщение «в бесконечность» без ошибок, словно там есть очередная ячейка. И у нас всегда есть пустая ячейка (символ $\ominus$) в регистре, иначе генерируется ошибка. Поэтому все строки в регистрах имеют свой конец.

$Command$ – регистр для хранения текущей команды, которая на луче данных $Program$ начинается с позиции, номер которой указан в $i_{Program}$. Включает в себя все символы, начиная с позиции $i_{Program}$ и до перевода строки $ChrEnter$ включительно.

Напоминаю, что разрядность процессора позволяет «в один ход» оперировать любым именем луча данных и любой командой. Поэтому то, как происходит лексический разбор команды на «запчасти» (имена команд, имена лучей данных на месте разных переменных) – не имеет принципиальных отличий от того, как это делается в «обычных» процессорах современных вычислительных машин.

Регистры, в которых хранятся «запчасти» из регистра Command, будут перечислены ниже. Но то, как эти регистры-«парсеры» заполняются из $Command$, мы в этой статье рассматривать не будем, так как тут нам нужно решить лишь принципиальные вопросы о работе с бесконечными лучами данных. А для лексического разбора данных в конечных регистрах в аппаратном языке SI имеются все возможности – для работы со строками – включая поиск (при лексическом разборе потребуется искать пробелы, знак равенства, скобки, запятые), взятие подстроки и оставшегося «хвоста» строки из $Command$.

Читатель, знакомый с программированием, может сам написать на аппаратном языке SI части программы (не такие уж маленькие), в которых делается лексический разбор $Command$, а перегружать текст статьи этим разбором я не вижу необходимости.

Регистры-парсеры следующие:

$LeftVar$, $q_{LeftVar}$, $RightFunc$, $FirstArg$, $MidArg$, $LastArg$, $AloneArg$.

Как они «разбирают» регистр $Command$, поясню на примере.

Пусть Command содержит следующую строку, включая символ перевода строки:

$\mathrm{x}_1(i_1) = \mathrm{str}(x_2, i_2, i_3);$

Тогда после лексического разбора будут верными следующие равенства:

$LeftVar =$ «x_1»

$q_{LeftVar} =$ «i_1»

$RightFunc =$ «str»

$FirstArg =$ «x_2»

$MidArg =$ «i_2»

$LastArg =$ «i_3»

Замечу, что если бы команда была такой:

$\mathrm{x}_1() = \mathrm{str}(x_2, i_2, i_3);$

то было бы верно:

$q_{LeftVar}(\ldots) = \ominus$

А вот уже из регистров-парсеров имена лучей данных будут переноситься (присваиваться) в регистры $Source_i$ и $Target_i$ (о которых сказано ниже) – для работы с произвольными (необходимыми для выполнения команды) лучами данных через конструкции вида @$Source_i$ и @$Target_i$. Перенос имен в регистры $Source_i$ и $Target_i$ осуществляется простым присваиванием – потому что размер имен находится в пределах разрядности процессора и доступной для «обычных» операций памяти.

$Continue_0$. Регистр хранения номеров этапов соответствующей задачи для связывания отдельных команд (обработчиков событий) языка SMAU в последовательности команд нулевого уровня.

Напоминаю, что в языке SMAU нет команд перехода, поэтому их роль берут на себя сообщения, вызывающие необходимые на данном этапе обработчики событий.

А в нужную последовательность эти обработчики и отправляемые ими сообщения выстраиваются, благодаря значениям регистров $Continue_i$, которым присваиваются необходимые для очередного шага значения в обработчике события, и которые учитываются при выборе очередного обработчика при проверке условий в его заголовке.

В регистр $Continue_0$ текущий обработчик события записывает информацию для следующего обработчика событий. Чтобы, когда в данное устройство поступило сообщение с результатом нынешнего обработчика – чтобы тогда происходил выбор обработчика с учётом преемственности, которая записана в $Continue_0$. Как это работает – будет видно в обработчиках событий для режима SMAU.

Типичная последовательность команд нулевого уровня – это формирование и отправка по лучу данных пакета данных, состоящего из многих сообщений.

$Continue_1$, $Continue_2$ и т.д. Регистры для связывания последовательностей предыдущего уровня между собой в последовательности более высокого (на 1 единицу) уровня. Мы этого не будем сейчас использовать, но в принципе такая возможность есть. А использовать сейчас мы не будем потому, что наш метод связывания последовательностей команд выше нулевого уровня – это программа на аппаратном языке SI. Если же нам захотелось бы аксиоматизировать время, то для построения Машины, которая стала бы интерпретацией этой теории, можно было бы обойтись без аппаратного языка SI и сделать интерпретатор стандартного языка SI на базе обработчиков событий на языке SMAU. И тогда нам пригодятся некоторые $Continue_i$, помимо $Continue_0$.

$Source_1$, $Source_2$ и т.п. конечный набор. Регистры для хранения имён лучей данных, которые является источниками данных для текущих операций процессора.

$Target_1$, $Target_2$ и т.п. конечный набор. Регистры для хранения имен лучей данных, которые является получателями данных для текущих операций процессора.

Всё, что далее будет сказано про использование регистра $Source_1$ – в равной степени относится и к регистру $Target_1$ и наоборот. И для всех сообщений, имеющих тип, зависящий от $Source_i$, есть аналогичные типы с аналогичной обработкой для $Target_j$. Регистры $Source_i$ и $Target_j$ – полностью

«симметричны». Но пользоваться мы будем преимущественно теми возможностями, для которых регистр $Source_i$ выступает в качестве источника данных, а регистр $Target_j$ – в качестве получателя.

$msgBegType_0$. Регистр для хранения типа первого сообщения в пакете данных, который формируется на уровне соответствующей $Continue_0$ задачи.

$msgLastType_0$. Регистр для хранения типа последнего сообщения в пакете данных, который формируется на уровне $Continue_0$.

$msgBuff_1$, $msgBuff_2$ и т.п. конечный набор – регистры для временного хранения данных из входящего сообщения

$msgType_1$, $msgType_2$ и т.п. конечный набор – регистры для временного хранения типа входящего сообщения.

Информацию из входящего сообщения удобно иногда сохранять до следующего сообщения (или нескольких сообщений), чтобы понять – нет ли ошибки в пакете данных и/или для выбора дальнейших действий с учётом двух (а не одного) последних сообщений.

$Register_1$, $Register_2$ и т.п. конечный набор – регистры процессора для временного хранения (промежуточных) результатов работы программы, с которыми можно работать «в один ход» - то есть – как память в обычных компьютерах в пределах разрядности процессора;

$x_1.rv$ и т.д. У процессора есть доступ ко всем регистрам первых ячеек лучей данных как к своим собственным регистрам. Имя такого регистра для процессора имеет вид:

<Имя луча данных>.<Имя регистра>

Но, можно обращаться и с использованием оператора @, разумеется. Например, если в регистре процессора $Target_1$ записано значение x_1, то присвоить регистру p_1 луча данных x_1 значение 2 можно следующим образом:

@$Target_1.\mathrm{p}_1(0) = 2$; // Присваивание на аппаратном языке SI

@$Target_1.p_1 = 2$ // Присваивание на языке SMAU

$Result_1$, $Result_2$ и т.п. конечный набор – служебные лучи данных для временного хранения (промежуточных) результатов работы программы или для записи на них окончательного результата/результатов;

III. Программа интерпретатора

У нас программа на аппаратном языке SI, которая управляет исполнением пользовательской программы (той, что записана на луче данных $Program$) представляет собой в основном цикл. Но есть ещё инициализация $i_{Program}$ до начала цикла и какие-то команды оповещения пользователя после окончания цикла – связь с «операционной системой», которую мы тут не рассматриваем. Привожу в схематическом виде текст программы-интерпретатора, записанной в ПЗУ Машины исполнения компьютерных алгоритмов на аппаратном языке SI:

$i_{Program}.\mathrm{Register}(0) = 1;$

$[CommandExecution];$

// Извлечение $Command$ из $Program$ в соответствии с $i_{Program}$ и выбор группы команд по наличию символа равенства для перехода к меткам $[EqualSmb_0]$ (для команд, отличных от присваивания), либо $[EqualSmb_1]$ (для команд присваивания).

...

$[EqualSmb_0];$

// Группа исполнения для команд, отличных от присваивания (это команды метка, переход goto, остановка). Лексический разбор с заполнением регистров-парсеров. Переход к метке, соответствующей текущей интерпретируемой команде, внутри данной группы исполнения команд.

...

// 1. Для команды метки - увеличение числа в $i_{Program}$ на размер текущей команды и возврат в начало цикла $[CommandExecution]$

$[ToNextCommand];$

...

$goto\ \mathrm{Const}(CommandExecution);$

// 2. Для команды $goto$ – изменение $i_{Program}$ – если метка найдена и возврат в начало цикла $[CommandExecution]$. А для неуспешной (не найдено метки) команды $goto$ – исполнение как для п.1 (команды метки) – переход к $[ToNextCommand]$.

...

// 3. Для команды остановки выход из цикла к $[EndProgram]$

...

$goto\ \mathrm{Const}(EndProgram);$

$[EqualSmb_1];$

// Группа исполнения для случая команды присваивания

// Сначала – полный лексический разбор команды присваивания

...

// Выбор той обработки, которую надо исполнить для подготовки луча данных (слева от знака равенства) к присваиванию. И переход к соответствующей метке для подготовки луча данных.

...

// Подготовка (выбранная на предыдущем этапе обработка) соответствующего луча данных к присваиванию и переход к следующему этапу, метка $[EqualSmb_2]$.

...

$[EqualSmb_2]$;

// Теперь подготовка результата работы функции из правой стороны присваивания.

// В начале – выбор той функции, которую надо вычислить и переход к соответствующей метке для вычисления.

...

// Вычисление соответствующей функции (выбранной на предыдущем этапе) справа от знака равенства и переход к следующему этапу, метка $[EqualSmb_3]$.

...

$[EqualSmb_3]$;

// Перенос результата этапа $[EqualSmb_2]$ на подготовленный на этапе $[EqualSmb_1]$ луч данных из левой части присваивания. И после этого – исполнение как для п.1 (команды метки) – переход к $[ToNextCommand]$ для увеличения числа в $i_{Program}$ на длину строки в регистре $Command$ и возврат в начало цикла $[CommandExecution]$.

...

$[EndProgram]$;

// Заключительные операции (если нужны) – в зависимости от того, как мы оповещаем пользователя о завершения вычисления

Лексический разбор весьма простой. Проверка начала строки $Command$ на совпадение с «goto », «.», «[» и отработка этих особых случаев. Если начало не такое, то у нас команда присваивания. И лексический разбор тогда такой:

1. От начало строки из $Command$ до (не включая) «(» – имя луча данных, получающего результат. Отбрасываем отработанное начало, включая «(».

2. Теперь от нового начала до (не включая) «)=» – имя луча данных, содержащего номер позиции, куда надо начинать переносить строку из правой части равенства. Оно может быть и пустым, если тут требуется конкатенация. Найти достаточно только «)», так как в корректной программе вместе с «)» на данном этапе будет сразу и символ «=». Отбрасываем отработанное начало, включая «)=».

3. Теперь от нового начала до (не включая) «(»– имя функции, возвращающей нужный результат. Отбрасываем отработанное начало, включая «(».

4. Теперь обработка переходит к той части программы, которая соответствует имени функции из предыдущего пункта. И аналогично вышеизложенному, получаем имена всех лучей данных, которые выступают аргументами данной функции.

Для всех этих операций нам необходима лишь функция поиска одного символа в строке, возвращающая позицию. На уровне операций с регистрами мы можем считать, что такая функция есть – это просто право сослаться на аналогию работы компьютера с ограниченной памятью – ведь на уровне компьютеров это реализовано. Номер найденной позиции нам надо будет в некоторых случаях уменьшать или увеличивать на 1 или 2. Для этого у нас есть функции $\mathrm{Next}(i)$ и $\mathrm{Previous}(i)$.

Разумеется, после того, как будет реализован интерпретатор языка стандартной интерпретации

для бесконечной памяти, нам нужно будет придумывать программы, при помощи которых на этом языке можно реализовать ту или иную функцию, включая функцию поиска символа в строке. Но в этом нет принципиальной необходимости для языка программирования аппаратного уровня, работающего с регистрами ограниченной памяти – он вообще может не иметь никакого отношения к языку стандартной интерпретации, как упоминалось выше.

Напоминаю (см. подраздел VI предыдущего раздела):

На интерпретацию каждой команды стандартного языка SI программой аппаратного языка SI времени тратится в пределах одной и той же константы $T_{SiMaxCommand}$ – если не учитывать в данном времени время на работу процессора в режим SMAU до его возврата из режима SMAU в режим SI.

Программа ПЗУ, схематически записанная выше, обрабатывается процессором в режиме интерпретации программы ПЗУ. Но и сама эта программа является интерпретатором программы, записанной на луче данных Program. Делаю это замечание, чтобы обратить внимание на 2 разных смысла при использовании слова «интерпретатор» в отношении разных программ – из ПЗУ (ограниченная программа для работы с ограниченными регистрами) и на луче данных Program (для работы с неограниченными «лучами данных»). И не следует путать слово «интерпретатор» (что относится к программам) с «интерпретация» - что относится к модели «Машина исполнения компьютерных алгоритмов». Напоминаю, что Машина исполнения компьютерных алгоритмов является «стандартной интерпретацией» для Теории компьютерных строк.

В программе-интерпретаторе, схематически записанной выше, сообщения отправляются в следующих случаях, если:

1. Необходимо отправить сообщение в начале цикла $[CommandExecution]$, для получения текущей команды в регистр $Command$ с луча данных $Program$, начинающейся в позиции, номер которой указан числом на луче данных $i_{Program}$;

2. Необходимо отправить сообщение для увеличения числа на луче данных $i_{Program}$ на длину строки из регистра $Command$ – это требуется после отработки любой интерпретируемой команды, кроме успешной (имеется соответствующая метка в тексте луча данных $Program$) команды перехода ($goto$) и команды остановки (.).

3. Необходимо отправить сообщение для вычисления нового $i_{Program}$ при успешной команде перехода ($goto$);

4. Необходимо отправить сообщение для установки указателя на луче данных из левой части присваивания – для будущего приема символов из строки-результата правой части присваивания;

5. Необходимо отправить сообщение для получения результата функций из правой части присваивания;

6. Необходимо отправить сообщение для передачи символов в нужное место луча данных пункта 4 из «теневой строки» (это будет разъяснено при разборе событий на лучах данных) результата пункта 5.

Разберём в первую очередь пункт 1, который предваряет обработку любой интерпретируемой команды, и которым начинается цикл $[CommandExecution]$.

Разберём во вторую очередь работу с интерпретируемыми командами луча данных $Program$,

которые не относятся к присваиванию. Тем самым мы завершим разбор пункта 3, а пункт 2 разберём для команд, не относящихся к присваиванию. Хотя пункт 2 тут потребуется только для метки и неудачного оператора перехода «goto»;

Разберём в третью очередь обработку левой части присваивания. Тем самым закроем вопросы пункта 4;

Разберём в четвёртую очередь обработку правой части присваивания. Тем самым закроем вопросы пункта 5;

Разберём в пятую и последнюю очередь пункт 6 (собственно само присваивание), а заодно и пункт 2 (изменение $i_{Program}$) для всех команд присваивания. Тем самым мы завершим разбор пункта 2 и закроем вопросы пункта 6;

IV. Перенос текущей (начиная с номера позиции из $i_{Program}$) команды из $Program$ в регистр $Command$

$\mathrm{Source}_1(0) = \mathrm{Const}(i_{Program})$;
$\mathrm{Target}_1(0) = \mathrm{Const}(Program)$;
$\mathrm{msgBegType}_0(0) =$ /# Младший разряд уменьшаемого числа (p_1) #/;
$\mathrm{msgLastType}_0(0) =$ /# Разряды для p_1 исчерпаны #/;
$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима
$SendTo_0\, Proc$ (/# Сделать указатель p_1 в $@Target_1$ из $@Source_1$ или подобное #/, $\ominus$);
$Setmode\, SMAU$;
// Возврат произойдёт в момент оперативного завершения установки p_1 – так устроен этот протокол
$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима
// Следующий протокол сам очистит теневую строку и вернёт процессор в режим SI после полного (а не только оперативного) построения текущей команды в теневой строке Program:
$SendTo_0\, Program$ (/# Начать отправку символов в теневую строку от p_1 до $ChrEnter$ #/, $\ominus$);
$Setmode\, SMAU$;
$\mathrm{Command}(0) = \mathrm{Value}(Program.SRegister)$;

Вкратце программа делает следующее – устанавливает на луче данных указатель в той ячейке, номер которой указан в $i_{Program}$. Это делает первое сообщение.

Второе сообщение в этом блоке приказывает лучу данных $Program$ перенести все символы, начиная с ячейки-указателя и вплоть до ячейки с переводом строки, в теневую строку данного луча данных.

Оставшаяся часть обходится без сообщения, так как любая команды с луча данных Program имеет такой размер, что её можно обрабатывать как обычную переменную обычных языков программирования. Разрядности для этого у процессора и лучей памяти хватает. Поэтому теневую. строку луча данных $Program$ можно использовать как регистр (при помощи конструкции $Program.SRegister$) и напрямую присвоить в регистр $Command$. Этим мы достигаем цель данного этапа.

V. Обработка команд, отличающихся от присваивания

1) $[next_1]$;

Всё, что необходимо при отработке метки – это пропустить данную команду и перейти к выполнению следующей команды. Поэтому часть программы-интерпретатора для обработки метки следующая:

$[ToNextCommand]$;

$\mathrm{Source}_1(0) = \mathrm{Const}(Command)$;

$\mathrm{Target}_1(0) = \mathrm{Const}(i_{Program})$;

$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима

$SendTo_0\, Proc\, (/\#$ Увеличить число в $@Target_1$ на количество символов в $@Source_1\, \#/, \ominus)$;

$Setmode\, SMAU$;

$qoto\ \mathrm{Const}(CommandExecution)$;

Метка $[ToNextCommand]$ в начале этого блока поставлена для всех тех команд, после исполнения которых, надо будет переходить к следующей команде на луче данных $Program$. При обработке метки ничего другого и не происходит. Для большинства команд с луча данных $Program$ в конце их обработки необходимо выполнить именно то, что выполняется здесь после метки $[ToNextCommand]$. И чтобы не повторять данную часть программы-интерпретатора в нескольких других частях программы-интерпретатора – мы будем завершать эти части оператором перехода:

$goto\ \mathrm{Const}(ToNextCommand)$;

Такая возможность сокращать текст программы-интерпретатора выгодно отличает язык аппаратного программирования стандартной интерпретации от языка программирования одновременных сообщений соседним устройствам.

2) $goto\, @AloneArg$;

$\mathrm{Register}_1(0) = \mathrm{Const}([)$;

$\mathrm{Register}_1() = \mathrm{Value}(@AloneArg)$;

$\mathrm{Register}_1() = \mathrm{Const}(];)$;

$\mathrm{Register}_1() = \mathrm{Value}(ChrEnter)$;

$\mathrm{Source}_1(0) = \mathrm{Const}(Register_1)$; // Метка помещается в регистр – это же команда программы

$\mathrm{Target}_1(0) = \mathrm{Const}(Result_1)$; // Найденный номер позиции в $Program$ может быть как угодно велик, поэтому тут нам нужен луч данных

$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима

$SendTo_0\, Proc\, (/\#$ Вычислить $@Target_1 = \mathrm{Goto}(@Source_1)\, \#/, \ominus)$;

$Setmode\, SMAU$;

$\mathrm{Register}_1(0) = \mathrm{Const}(Goto)$;

$\mathrm{Register}_1() = \mathrm{Comp}(0, Result_1.Register)$; // $.Register$ может заменить слишком длинное число на число $/\#\, Err\, out\, of\, range\, \#/$, но любого не нулевого числа вместо $Result_1$ достаточно, чтобы отличить случай «найдено» $(\mathrm{Comp}(\ldots) \neq 0)$ от «не найдено» $(\mathrm{Comp}(\ldots) = 0)$

$goto\, Register_1$;

$[Goto0]$;

$goto$ $\mathrm{Const}(ToNextCommand)$; // «Не найдено». Просто пропускаем команду goto

$[Goto1]$;

// «Найдено». Теперь надо перенести значение из $Result_1$ в $i_{Program}$

// 1. Сначала готовим регистр $Continue_0$, который создаёт условия для возврата в режим SI после завершения отбрасывания тени и поступления сообщения /# Теневая строка построена #/ в процессор:

$Continue_0 = /\#$ Скопировать теневую строку из основной $\#/$;

$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима

// 2. Теперь получаем копию числа $Result_1$ в теневую строку $Result_1$:

$SendTo_0\, Result_1(/\#$ Отбросить тень и уведомить $\#/, \ominus)$;

$Setmode\, SMAU$;

// 3. Теперь присваиваем в $i_{Program}$ указателю (регистру) первой ячейки p_1 значение 2, что аналогично подготовке такой конструкции присваивания: $i_{Program}(0) = \ldots$ Что должно обеспечить замену прежнего значения новым:

$i_{Program}.\mathrm{p}_1(0) = 2$;

// 4. Переносим теневую строку $Result_1$ в $i_{Program}$, с заменой старого значения новым:

$\mathrm{Source}_1(0) = \mathrm{Const}(Result_1)$; // Откуда копировать

$\mathrm{Target}_1(0) = \mathrm{Const}(i_{Program})$; // Куда копировать

$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима

// Перенесли результат поиска метки команды $goto$ в $i_{Program}$

$SendTo_0\, Proc\, (/\#$ Начать перенос теневой строки из $@Source_1$ в позицию p_1 в $@Target_1\, \#/, \ominus)$;

$Setmode\, SMAU$;

// 5. Переход к исполнению очередной (найденной по $goto$) команды

$qoto$ $\mathrm{Const}(CommandExecution)$;

3) .;

$qoto$ $\mathrm{Const}(EndProgram)$;

VI. Обработка левой части команды присваивания

4.1) $@\,\mathrm{LeftVar}(@q_{LeftVar}) = \ldots$; // При $q_{LeftVar} = \ominus$

$\mathrm{Target}_1(0) = \mathrm{Value}(LeftVar)$;

// Отправим следующее сообщение сразу по лучу данных. Ответа в процессор не будет, поэтому режим не меняем:

$SendTo_0\,@Target_1$(/# Установить указатель p_1 в первой пустой ячейке #/, $\ominus$);

$wait$; // Отправляем сообщение

$wait_0$; // Дожидаемся, когда 1-я ячейка отработает и наступит оперативное завершение установки указателя

$goto\ \mathrm{Const}(EqualSmb_2)$; // Теперь отправляемся готовить строку, которую надо будет присвоить

4.2 и 4.3) $@\,\mathrm{LeftVar}(@q_{LeftVar}) = \ldots$; // При $q_{LeftVar} \neq \ominus$ **(любое число)**

$\mathrm{Source}_1(0) = \mathrm{Value}(q_{LeftVar})$;

$\mathrm{Target}_1(0) = \mathrm{Value}(LeftVar)$;

$\mathrm{msgBegType}_0(0) =$ /# Младший разряд уменьшаемого числа (p_1) #/;

$\mathrm{msgLastType}_0(0) =$ /# Разряды для p_1 исчерпаны #/;

$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима

$SendTo_0\,Proc$ (/# Сделать указатель p_1 в $@Target_1$ из $@Source_1$ или подобное #/, $\ominus$);

$Setmode\,SMAU$;

$goto\ \mathrm{Const}(EqualSmb_2)$; // Теперь отправляемся готовить строку, которую надо будет присвоить

VII. Обработка правой части команды присваивания

5) $\mathrm{len}(@AloneArg)$;

$\mathrm{Source}_1(0) = \mathrm{Value}(AloneArg)$;

$\mathrm{Target}_1(0) = \mathrm{Const}(Result_1)$;

$Result_1.\mathrm{Register}(0) = 0$; // Устанавливаем на целевом луче данных ноль, чтобы было к чему добавлять длину строки @AloneArg.

$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима

$SendTo_0\,Proc$ (/# Увеличить число в $@Target_1$ на количество символов в $@Source_1$ #/, $\ominus$);

$Setmode\,SMAU$;

$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима

$SendTo_0\,@Target_1$ (/# Отбросить тень и уведомить #/, $\ominus$);

$Setmode\,SMAU$;

$\mathrm{Source}_1(0) = \mathrm{Const}(Result_1)$; // Для следующего этапа тень является источником данных

$goto\ \mathrm{Const}(EqualSmb_3)$; // Теперь отправляемся делать присваивание. «Куда» и «что» - известно

6) $\mathrm{Chr}(@AloneArg)$;

$\mathrm{Register}_1(0) = \mathrm{Chr}(@AloneArg.Register)$;

$\mathrm{Target}_1(0) = \mathrm{Const}(Register_1)$;

$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима

$SendTo_0\ @Target_1\ (/\#$ Отбросить тень и уведомить $\#/, \ominus)$;

$Setmode\ SMAU$;

$\mathrm{Source}_1(0) = \mathrm{Const}(Register_1)$;

$goto\ \mathrm{Const}(EqualSmb_3)$;

7.1) $\mathrm{Value}(@AloneArg)$;

$\mathrm{Target}_1(0) = \mathrm{Value}(AloneArg)$;

$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима

$SendTo_0\ @Target_1\ (/\#$ Отбросить тень и уведомить $\#/, \ominus)$;

$Setmode\ SMAU$;

$\mathrm{Source}_1(0) = \mathrm{Value}(AloneArg)$;

$goto\ \mathrm{Const}(EqualSmb_3)$;

7.2) $\mathrm{from}(@FirstArg, @LastArg)$;

$\mathrm{Source}_1(0) = \mathrm{Value}(LastArg)$;

$\mathrm{Target}_1(0) = \mathrm{Value}(FirstArg)$;

// Задаём типы начального и заключительного сообщения в пакете данных

$\mathrm{msgBegType}_0(0) = /\#$ Младший разряд уменьшаемого числа $(p_1)\ \#/$;

$\mathrm{msgLastType}_0(0) = /\#$ Разряды для p_1 исчерпаны $\#/$;

$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима

// Устанавливаем указатель в ячейку с номером @LastArg на луче данных @FirstArg

$SendTo_0\ Proc\ (/\#$ Сделать указатель p_1 в $@Target_1$ из $@Source_1$ или подобное $\#/, \ominus)$;

$Setmode\ SMAU$;

$\mathrm{Source}_1(0) = \mathrm{Value}(FirstArg)$;

$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима

// Теперь выносим все символы от указателя включительно в теневую строку

$SendTo_0\ Proc\ (/\#$ Начать отправку символов в теневую строку $@Source_1$ от p_1 включительно $\#/, \ominus)$;

$Setmode\ SMAU$;

$goto\ \mathrm{Const}(EqualSmb_3)$;

7.3) $\mathrm{str}(@FirstArg, @MidArg, @LastArg)$;

$\mathrm{Source}_1(0) = \mathrm{Value}(MidArg)$;

$\mathrm{Target}_1(0) = \mathrm{Value}(FirstArg)$;

$\mathrm{msgBegType}_0(0) =$ /# Младший разряд уменьшаемого числа (p_1) #/;

$\mathrm{msgLastType}_0(0) =$ /# Разряды для p_1 исчерпаны #/;

$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима

// Устанавливаем указатель в ячейку с номером $@MidArg$ на луче данных $@FirstArg$

$SendTo_0\, Proc$ (/# Сделать указатель p_1 в $@Target_1$ из $@Source_1$ или подобное #/, $\ominus$);

$Setmode\, SMAU$;

$\mathrm{Source}_1(0) = \mathrm{Value}(LastArg)$;

$\mathrm{msgBegType}_0(0) =$ /# Младший разряд числа к уменьшению от p_1 #/;

$\mathrm{msgLastType}_0(0) =$ /# Разряды для количества символов к отправке от p_1 исчерпаны #/;

$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима

// Переносим символы в количестве $@LastArg$ от установленного указателя в позиции $@MidArg$ на луче данных $@FirstArg$ в теневую строку. Предварительная очистка теневой строки делается в этом же протоколе.

$SendTo_0\, Proc$ (/# Сделать указатель p_1 в $@Target_1$ из $@Source_1$ или подобное #/, $\ominus$);

$Setmode\, SMAU$;

$\mathrm{Source}_1(0) = \mathrm{Value}(FirstArg)$;

$goto\ \mathrm{Const}(EqualSmb_3)$;

8) $\mathrm{Comp}(@FirstArg, @LastArg)$;

$\mathrm{Source}_1(0) = \mathrm{Value}(FirstArg)$;

$\mathrm{Source}_2(0) = \mathrm{Value}(LastArg)$;

$\mathrm{Target}_1(0) = \mathrm{Const}(Result_1)$;

$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима

$SendTo_0\, Proc$ (/# Вычислить $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2$ #/, $\ominus$);

$Setmode\, SMAU$;

$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима

// Теперь получаем копию числа $Result_1$ в теневую строку $Result_1$:

$SendTo_0\, Result_1$ (/# Отбросить тень и уведомить #/, $\ominus$);

$Setmode\, SMAU$;

$\mathrm{Source}_1(0) = \mathrm{Const}(Result_1)$;

$goto\ \mathrm{Const}(EqualSmb_3)$;

9) $\mathrm{Next}(@AloneArg)$;

$\mathrm{Target}_1(0) = \mathrm{Value}(AloneArg)$;

// Отправим следующее сообщение сразу по лучу данных. Ответа в процессор не будет, поэтому режим не меняем:

$SendTo_0\,@Target_1\,(/\#$ Увеличить на $1\,\#/, \ominus)$;

$wait$; // Отправляем сообщение

$wait_0$; // Дожидаемся, когда 1-я ячейка отработает и наступит оперативное завершение установки указателя. И тут же происходит выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима в следующей отправке сообщения.

$SendTo_0\,@Target_1\,(/\#$ Отбросить тень и уведомить $\#/, \ominus)$;

$Setmode\,SMAU$;

$\mathrm{Source}_1(0) = \mathrm{Value}(AloneArg)$;

$goto\ \mathrm{Const}(EqualSmb_3)$;

В принципе тут возможна оптимизация для случая, вроде:

$\mathrm{i}(0) = \mathrm{Next}(i)$;

Тогда обычно не пришлось бы работать со всей строкой, а всё решалось бы на уровне младших разрядов. Но мы тут решаем вопрос в принципе. А вот на базе стандартного SI можно будет писать программы для быстрых арифметических действий, быстрых добавлений и отниманий единицы.

Имея в своём распоряжении строковые операции – можно программировать что угодно. Нам же сейчас важно показать, что строковые операции могут быть реализованы на практике линейно быстрые от используемых в операциях строк (от размеров используемых строк), что мы и делаем.

10) $\mathrm{Previous}(@AloneArg)$;

$\mathrm{Target}_1(0) = \mathrm{Value}(AloneArg)$;

// Отправим следующее сообщение сразу по лучу данных. Ответа в процессор не будет, поэтому режим не меняем:

$SendTo_0\,@Target_1\,(/\#$Уменьшить на $1\,\#/, \ominus)$;

$wait_1$; // Отправляем сообщение и выравнивает время по сетке синхронизации 2-моментов

$wait_1$; // Дожидаемся, когда пройдёт 2 момента, 1-я ячейка отработает и наступит оперативное завершение установки указателя. В данном протоколе иногда оперативное завершение наступает через 2 момента, а не через 1. И тут же происходит выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима в следующей отправке сообщения.

$SendTo_0\,@Target_1\,(/\#$ Отбросить тень и уведомить $\#/, \ominus)$;

$Setmode\,SMAU$;

$\mathrm{Source}_1(0) = \mathrm{Value}(AloneArg)$;

$goto\ \mathrm{Const}(EqualSmb_3)$;

Разумеется, в плане возможной оптимизации для команды вида:

$\mathrm{i}(0) = \mathrm{Previous}(i)$;

верным является то же самое, что было сказано для команды вида:

$\mathrm{i}(0) = \mathrm{Next}(i)$;

VIII. Собственно присваивание – перенос результата на нужный луч данных

// К данному моменту в $Source_1$ должно быть записано имя того луча данных (или регистра), теневая строка которого должна быть присвоена на луч данных с именем в $LeftVar$. Притом аргу-

мент присваивания i из $@\,\mathrm{LeftVar}(i) = \ldots$ уже задействован в качестве установленного указателя на луче данных $@LeftVar$.

$\mathrm{Target}_1(0) = \mathrm{Const}(LeftVar)$;

$wait_0$; // Выравнивание по сетке синхронизации 1-моментов, чтобы дать время для смены режима

$SendTo_0\, Proc$ (/# Начать перенос теневой строки из $@Source_1$ в позицию p_1 в $@Target_1$ #/, $\ominus$);

$Setmode\, SMAU$;

$goto\ \mathrm{Const}(ToNextCommand)$;

Теперь нам надо написать обработчики событий, которые будут обрабатывать сообщения, которые может отправлять разобранный выше интерпретатор стандартного языка SI. Для написания этих обработчиков будет использован Язык программирования одновременных сообщений соседним узлам (SMAU), рассмотрением которого мы займёмся в следующем разделе.

3 Язык программирования одновременных сообщений соседним узлам (SMAU)

I. Обработчики событий. Один момент – гораздо дольше работы с регистром. Одновременность ухода и прихода данного сообщения. Пакеты данных иногда уходят в бесконечность

В качестве первоначального знакомства с языком SMAU рассмотрим пару написанных на нём обработчиков событий, которые образуют протокол «отбрасывания тени». А заодно увидим оставшиеся «странности» бесконечной модели, о которых я обещал рассказать в контексте знакомства с данным языком.

$(/\#$ Отбросить тень $\#/, \ominus), rv = \ominus$

$sv = rv$ // Никаких сообщений дальше не передавать – это конец работы по «отбрасыванию тени»

$(/\#$ Отбросить тень $\#/, \ominus), rv \neq \ominus$

$sv = rv$

$SendTo_0\, NextCell\, (/\#$ Отбросить тень $\#/, \ominus)$

Заголовок обработчика начинается с шаблона (в скобках) обрабатываемого сообщения. В обеих обработчиках это сообщение с типом /# Отбросить тень #/, а второй параметр – это передаваемое сообщением значение, в данном шаблоне пустое. Далее в заголовке обработчика указаны условия, при которых он вызывается.

Условия в заголовке обработчика – это функции, которые все должны вернуть 1, чтобы данный обработчик был исполнен. Знак равенства в контексте обработчика – тоже функция, которая возвращает 1 при равенстве левой и правой сторон равенств и 0 в противном случае. Знак неравенства $\neq$ даёт противоположные результаты, с другими знаками неравенства тоже понятен результат при их использовании. Сразу перечислим другие «проверочные» функции, которые можно использовать в заголовке обработчика событий:

$IsFirst = 1$

$IsFirst = 0$

Где $IsFirst$ – неизменный регистр о том, является ли ячейка первой или не первой на луче данных – этот «флаг» принимает значение 1 или 0 соответственно;

Функции $\mathrm{IsNum}(\ldots)$ и $\mathrm{NotNum}(..)$ – могут применяться к регистрам и значению сообщения, поступившего в узел Машины. Возвращают 1 или 0 в зависимости от того, является аргумент числом (как вариант – цифрой) или нет.

Функции $\mathrm{IsFlag}(\ldots)$ и $\mathrm{NotFlag}(\ldots)$ – могут применяться к регистрам и значению сообщения, поступившего в узел Машины. Возвращают 1 или 0 в зависимости от того, является аргумент «флагом» (то есть, имеет значение 1, либо значение 0) или нет.

Этих функций проверки нам хватит для заголовков обработчиков событий. Но, в принципе, в заголовках можно использовать арифметические и строковые операции с регистрами и значениями сообщений.

Если условий в заголовке обработчика много, но можно записывать условия в несколько строк, но начинать новую строку тогда надо запятой. Вот так:

(/# Отбросить тень #/, $\ominus$)

, $rv \neq \ominus$

Условимся, что при приведённом формате заголовков у обработчика событий имеется аналогичный обработчик события для теневого канала связи луча данных, и в нём всё как в основном обработчике, но только вместо команд:

$SendTo_i\, NextCell\ldots$

$SendTo_i\, PreviousCell\ldots$

$SendTo_i\, Result_1\ldots$

$SendTo_i\, @Source_1\ldots$

Будут, соответственно, команды:

$SendToS_i\, NextCell\ldots$

$SendToS_i\, PreviousCell\ldots$

$SendToS_i\, Result_1\ldots$

$SendToS_i\, @Source_1\ldots$

А отправка сообщения процессору не меняется, потому что в центральном процессоре сходятся все каналы связи лучей данных.

$SendTo_i\, Proc\ldots$ не отличается от:

$SendToS_i\, Proc\ldots$

Если же нам захочется построить обработчик отдельно для основного канала связи или отдельно для теневого, то начало их заголовков будут иметь вид, соответственно:

(/# Тип сообщения #/, $\ominus)^R$

(/# Тип сообщения #/, $\ominus)^S$

Но для теневого обработчика для передачи сообщений между соседними ячейками применимы, разумеется, только команды вида $SendToS_i\ldots$ в отношении ячеек и лучей данных.

В теле обработчика идут чаще всего команды присваивания – не в стиле языка SI, а как это обычно в языках программирования – тут не планируется лексический разбор в рамках теории компьютерных строк, поэтому нет нужды использовать удобный для лексического разбора формат записи команд.

К тому же в языке SMAU можно использовать арифметические действия и строковые операции как готовые – мы же оперируем с ограниченными по своей памяти регистрами, поэтому можем просто сослаться на практику, что эти операции можно реализовать для памяти, ограниченной разрядностью процессора.

Разделяются команды простым переводом строки без точки с запятой. Разумеется, в теле обработчика событий знак равенства используется как знак присваивания, а не как в заголовке обработчика события.

В первом из приведённых обработчиков событие состоит в приходе сообщения в пустую ячейку – притом это первая пустая ячейка, если «идти» от процессора. Тип сообщения требует отбрасывания

тени – то есть, переноса основного значения ячейки (из регистра rv) в теневой регистр sv значения ячейки.

Но раз ячейка пустая и при этом первая пустая ячейка среди всех пустых ячеек на луче данных, то это признак того, что строка закончилась ещё в предыдущей ячейке. И действие обработчика состоит в копировании настоящего символа ячейки (пустого) в регистр теневого символа. После этого построение теневой строки, равной настоящей строке луча данных, закончено, и передачи сообщения по эстафете дальше не происходит.

Второй обработчик – для события не пустой ячейки, поэтому не только переносится настоящее значение символа в теневой регистр, но и происходит дальнейшая передача сообщения по эстафете. Индекс ноль ($SendTo_0 \ldots$) означает обычную для передачи от процессора сетку синхронизации по 1-моментам.

Напоминаю, что в команде отправки сообщения

$SendTo_i \ldots$

i – индекс, соответствующий логарифмической (по основанию 2) шкале сеток синхронизации. Если он равен 0 (нулю) – то это сетка синхронизации для 1-моментов (2^0-моментов). Если $i = -1$, то сетка синхронизации для 1 / 2 – моментов (2^{-1}-моментов). Если $i = 1$, то сетка синхронизации для 2-моментов (2^1-моментов). И т.д.

Сама команда $SendTo_i$ лишь помещает сообщение в буфер отправки сообщений данного узла Машины, и переходит к исполнению следующей команды в обработчике – если она есть. А из узла сообщения уходят в ближайший момент по их сетке синхронизации (заданной индексом ноль в данном примере). При завершении обработчика события он не дожидается отправки всех сообщений, которые накопились за время его работы в буфере отправки сообщений данного узла Машины, а сообщения уходят сами из буфера отправки сообщений в назначенное для каждого из них время.

Можно принудительно дождаться момента ухода всех сообщений из буфера отправки при помощи команды $wait$. Иногда это надо сделать хотя бы для того, чтобы разделять отправку сообщений в один и тот же узел Машины. И это предотвращает конфликты. Однако команда $wait$ прекращает работу следующих команд в текущем обработчике до момента ухода всех накопившихся сообщений в буфере отправки сообщений данного узла. И команда $wait$ прекращает запуск новых обработчиков.

А это значит, что если в текущий узел придёт сообщение в момент приостановки обработчиков из-за команды $wait$, то это будет означать конфликт сообщения с блокировкой обработчиков и ошибку в разработке протокола передачи. И нам надо следить за предотвращением подобных ошибок.

Напоминаю и про команду $wait_i$. Она приостанавливает выполнение команд устройством (процессором или ячейкой) кроме передачи сообщений, которые сейчас находятся в буфере отправки сообщений. Смысл индекса i такой же, как в команде $SendTo_i \ldots$. И команда $wait_i$ ждёт ближайшего момента в той сетке синхронизации, которую задаёт индекс i. После этого приостановка выполнения программы прекращается, независимо от того, что какие-то сообщения могут ещё не уйти из буфера отправки сообщений.

Отметим, что обычно никаких событий не может произойти после завершения работы обработчика события и отправки сообщения из процессора до тех пор, пока не вернутся сообщения, которые инициировал данный обработчик своими отправленными сообщениями. Поэтому для начала обработки нового события в процессоре необходимо, чтобы сообщения последнего обработчика не только ушли, но и много чего сделать обычно.

Хотя есть и исключения, когда процессор обрабатывает сообщения в процессе ожидания результата от запущенных ранее протоколов передачи. Это редкая ситуация, но забывать про неё – не стоит.

Но в остальных случаях конец обработчика событий в процессоре «работает» как $wait$.

Момент ухода сообщения из данного узла будем считать моментом его прихода в узел-получатель. То есть – сообщения, которое подготовлено для сетки синхронизации 1-моментов, исчезнет из узла-передатчика в ближайший 1-момент. То есть, его уже не будет в буфере отправки сообщений данного узла. Но это же сообщение в тот же момент уже будет в регистрах полученного сообщения узла-приёмника.

А через 1 / 16 момента мы получим уже первые результаты обработки данного сообщения в узле Машины, куда оно поступило. То есть – за 1 / 16 момента произойдёт:

1. Передача ячейки из ячейки-передатчика в соседнюю ячейку-приёмник.

2. Выбор обработчика событий из нескольких имеющихся у данного узла данных;

3. Выполнение всех действий данного обработчика до конца обработчика или до ближайшей команды $wait$ – кроме отправки готовых сообщений из буфера отправки сообщений. И кроме случая, когда обработчик события центрального процессора использует очень медленное свойство $.Register$ какого-то луча данных. Но свойство $.Register$ заменяет собой целый протокол.

Кстати, свойство $.Register$ не используется в заголовках обработчиков событий. Иначе нельзя было бы считать, что выбор обработчика укладывается в 1 / 16 момента, а пришлось бы разбираться, как суммировать время работы проверочных функций при выборе обработчика события для его исполнения в текущих условиях.

Сразу отметим, что никаких операторов перехода, цикла и т.п. способов изменять последовательность исполнения команд или повторно использовать код внутри обработчика события, в языке SMAU нет. Вся последовательность действий определяется только поступившим сообщением и сложившимися условиями в узле Машины, приводящим к вызову соответствующего обработчика событий. А внутри обработчика событий всё происходит строго в той последовательности, в которой команды записаны. Однозначная простая ситуация – однозначная простая последовательность действий.

Что касается более «мелких» единиц времени, чем «мгновение» (1 / 16 момента), то они в природе есть, но не будут тут использоваться. Потому что даже мгновения настолько мелкие, что всё равно будут ждать момента отправки сообщений, а только события, связанные с сообщениями, определяют процесс вычисления на уровне языка SMAU и его расход времени.

Да, если у нас есть последовательность обмена значениями, например, то там заведомо есть причинно-следственные связи и отдельные моменты времени на каждую из команд присваивания:

$Register_1 = Source_1$
$Source_1 = Target_1$
$Target_1 = Register_1$

Но «в сухом остатке» нас интересует то, что начальные значения в $Source_1$ и $Target_1$ поменялись местами, и произошло это в пределах 1 / 16 момента. И из подробной теории для времени (если её делать) была бы доказана соответствующая теорема для ограниченной теории с использованием времени – которая описывает смену состояний Машины, работающей только на уровне сообщений и языка SMAU. А состоянием является момент поступления сообщения, а следующее – уход очередного сообщения. А то, что между ними – это лишь формальное описание перехода от одного состояния к другому, которое описывает команду для получения результата, но не процесс этого его получения.

То есть, строчки кода выше лишь обозначают условную команду $\mathrm{Swap}(Source_1, Target_1)$, результат работы которой будет равен результату исполнения данных строчек, но детали процесса нас не интересуют с точки зрения времени. И поэтому единицы времени для этих деталей мы не используем, и не придумываем.

Возвращаясь к работе с регистрами в момент выбора обработчика события и при его выполнении, мы видим, что множество операций с ячейками регистра будут выполнены за время 1 / 16 от обычного времени на преодоление расстояния в одну ячейку луча данных!

Таким образом, стандартная скорость доступа от ячейки к ячейке луча данных при движении «от процессора» равная 1 ячейки за 1 момент (а зачастую нужно ещё и обратно сообщения получить) ниже скорости обращения к ячейке регистра примерно на 2 порядка, если не на 3. Но такова цена построения понятной теоретической модели.

1. Модель должна быть практически реализуемой в большом масштабе и понятным образом масштабируема на бесконечность. То есть, может быть расширена в 2 раза и далее в 2 и т.д. без изменения построенного ранее – в предположении, что имеются все необходимые материальные ресурсы.

2. Модель не должна ухудшать (в практическом отношении) характер зависимостей, но может ухудшать параметры. В нашем случае – зависимость остаётся линейной для времени передачи сигнала относительно номера ячейки.

3. В остальном приоритет всегда в пользу понятности. Затраты ресурсов не имеют значения, если на элемент модели они остаются конечными. Пусть их уйдёт во сколько угодно раз больше, но это необходимо сделать, если модель от этого будет проще, чем в случае экономии ресурсов.

Можно считать, что ячейки регистров расположены на одинаковом расстоянии от процессора (центрального или одного из двух упрощенных процессоров ячейки луча данных). Процессор имеет «мгновенный» доступ к любой из них. По такой схеме, например, осуществляется доступ к регистру p_1 первой ячейки луча данных x_1 (для присваивания ему значения 2 здесь):

$x_1.p_1 = 2$

И отмечу последнюю «странность» в бесконечной архитектуре Машине исполнения компьютерных алгоритмов – уход пакетов данных «в бесконечность» - как часть некоторых протоколов

передачи.

А именно, когда пакет проходит через ячейку, то невозможно заранее сказать, достиг он нужной ячейки или же ему надо идти дальше. Поэтому мы отправляем все сообщения (кроме последнего) – к следующей ячейке – вдруг эстафета должна быть продолжена. При этом мы делаем какую-то обработку пакета данных, например – отнимаем от передаваемого в пакете числа единицу.

И вот мы получаем последнее сообщение пакета, и видим – вдруг – что обработка пакета закончена (он стал равен единице после вычитания в текущей ячейке, например). Поэтому дальше ему идти не надо для каких-то полезных действий. Но все сообщения до предпоследнего сообщения включительно уже ушли. Что делать?

В этом случае мы меняем последнее сообщение пакета на сообщение с «холостым» типом. Чтобы никаких значимых изменений в ячейках этот пакет больше не делал. А все значимые изменения делаются только при поступлении последнего сообщения в пакете данных. И отправляем это последнее «холостое» сообщение в качестве конца пакета. Вот таким образом мы решаем вопрос с прекращением работы протокола передачи. Не прекращением передачи, но прекращением значимых действий, которые эта передача могла бы произвести.

При желании можно продумать «догоняющий сигнал», который движется с тактом времени величиной 1 / 4 момента, например, который постепенно «обгоняет» пакет данных с «холостым» типом по «полицейскому» каналу связи и прекращает его дальнейшее движение.

Впрочем, такой «полицейский» сигнал данных ничего не ускорит, и не изменит в результатах работы алгоритмов. Но может успокоить в плане предотвращения бесконечного расхода «электроэнергии», что едва ли важно для бесконечной математической модели, с учётом и без того бесконечного ресурса, «использованного» для бесконечных лучей данных.

Напоминаю, что из-за стандарта на движение сообщений «от процессора» (и запрос на значение, и запись значения движутся от процессора со скоростью 1 ячейка за 1 момент) нам не обязательно ждать «абсолютного» завершения протокола. К тому же, он может никогда не закончиться – например, отбрасывание тени, если строка бесконечная. Но пользоваться результатом отбрасывания тени мы можем через 1 момент после срабатывания обработчиков в первой ячейке. Словно весь луч данных уже имеет «тень».

Требуется лишь, чтобы «окончательный результат» данного протокола имелся только в 1-й ячейке, а в остальных формировался не позднее, чем со скоростью одна ячейка за один момент – удаляясь от процессора. И чтобы никакие сообщения протокола уже не возвращались в данную ячейку, создавая угрозу конфликта сообщений.

Надеюсь, что после данного подраздела я изложил все «странности» архитектуры Машины в сравнении с привычными конечными вычислительными машинами. Хотя за долгое время построения архитектуры Машины с бесконечной памятью, я мог слишком привыкнуть к каким-то её особенностям, чтобы осознать их «странность» и предупредить про них читателя.

II. Функции и регистры для обработчиков событий

В языке SMAU, помимо функций проверки в заголовке обработчиков событий (предыдущий подраздел) мы будем использовать в «теле» обработчиков событий команды присваивания вида:

$reg_1 = reg_2$

$reg_1 = \mathrm{Const}(string)$

$reg_1 = 123$

Где на месте reg_1 и reg_2 могут стоять разные регистры, а конструкцию $\mathrm{Const}(string)$ и 123 – это запись констант, с теми же значениями, что и для стандартного языка SI. Только в языке SMAU это настоящие команды, а не псевдо-команды.

Точно так же будем использовать предопределённые регистры (и лучи данных одновременно):

0, $ChrEnter$, $ChrClosing$ (закрывающая круглая скобка), $ChrEmpty$ (обычно буду записывать как $\ominus$)

Из строковых функций нам потребуется только $\mathrm{Chr}(\ldots)$ – символ алфавита по его номеру. Для «слишком» больших номеров и не-номеров выдаёт $\ominus$, даже не читая аргумент до конца – если его избыточность очевидна по началу.

Другие строковые функции не потребуются, хотя можно использовать и $\mathrm{str}(reg_1, reg_2, reg_3)$, и $\mathrm{from}(reg_1, reg_2)$, и $\mathrm{len}(reg_1)$.

Комментарии того же вида, что в языке SI:

/* Многострочный комментарий */

// Однострочный комментарий

И так же обозначается число «комментарием о смысле числа». Например, число 2:

/# Сколько весёлых гуся жило у бабуси (подсказка: серый и белый) #/

Что касается функций из аппаратного языка SI, то нам доступно свойство $.Register$ и $.SRegister$ в одном узле – центральном процессоре по отношению к любому лучу данных. Например:

$x_1.Register$ – вернёт основную строку луча данных x_1, если она очень небольшая (подробности были рассмотрены в разделе про аппаратный язык SI), либо вернёт $/\#\, Err\, out\, of\, range\, \#/$

$x_1.SRegister$ – вернёт теневую строку луча данных x_1.

Оба свойства можно использовать и для присваивания на луч данных небольших строк.

Оператор @ с наивысшим приоритетом применяется только к регистрам и только в обработчиках событий центрального процессора. Если $Source_1$ содержит строку «x_1», то команда...

$@Sourse_1.Register = 10$

...присвоит на луч данных x_1 строку, представляющую собой десятичное число 10. А команда ...

$Target_1 = @Sourse_1.Register$

...присвоит регистру $Target_1$ строку, которая записана на луче данных x_1, но если эта строка чрезмерно велика для свойства .Register, то присвоит $/\#\, Err\, out\, of\, range\, \#/$.

Следующие команды мы упомянули в предыдущем подразделе, а подробности были при рассмотрении аппаратного языка SI:

$SendTo_i\, Unit\, (/\#$ Тип сообщения $\#/, msgValue)$

$SendToS_i\,Unit$ (/# Тип сообщения #/, $msgValue$)

$wait_i$

$wait$

Разумеется, нам потребуется команда возврата процессора в режим SI:

$Setmode\,SI$ // Переключение в другой режим из обработчика событий на языке SMAU

И мы будем использовать арифметические операции (только сложение и вычитание) с регистрами. Вычитание будет арифметическим – когда вычитание большего числа от меньшего числа даёт ноль.

Теперь перечислим регистры, начиная с тех, которые были рассмотрены для процессора, и которые будут использованы в обработчиках событий:

$Command$

$Continue_0$. Регистр для связывания последовательностей обработчиков событий через посылаемые ими сообщения и значение данного регистра.

$Source_1$, $Source_2$ и т.п. конечный набор. Регистры для хранения имён лучей данных, которые является источниками данных для текущих операций процессора.

$Target_1$, $Target_2$ и т.п. конечный набор. Регистры для хранения имен лучей данных, которые является получателями данных для текущих операций процессора.

$msgBegType_0$. Регистр для хранения типа первого сообщения в пакете данных, который формируется на уровне соответствующей $Continue_0$ задачи.

$msgLastType_0$. Регистр для хранения типа последнего сообщения в пакете данных, который формируется на уровне $Continue_0$.

$msgBuff_1$, $msgBuff_2$ и т.п. конечный набор – регистры для временного хранения данных из входящего сообщения

$msgType_1$, $msgType_2$ и т.п. конечный набор – регистры для временного хранения типа входящего сообщения.

$Register_1$, $Register_2$ и т.п. конечный набор – регистры процессора для временного хранения (промежуточных) результатов работы программы

$x_1.rv$ и т.д. У процессора есть доступ ко всем регистрам первых ячеек лучей данных как к своим собственным регистрам. Имя такого регистра для процессора имеет вид:

<Имя луча данных>.<Имя регистра>

$Result_1$, $Result_2$ и т.п. конечный набор – служебные лучи данных для временного хранения (промежуточных) результатов работы программы или для записи на них окончательного результата/результатов;

В заключение – регистры произвольной ячейки луча данных:

rv: «Реальное» значение ячейки – это символ, последовательность которых от 1-й ячейки луча данных (включительно) до 1-й ячейки с $rv = \ominus$ (если она есть, и не включая её) образуют «основную» строку луча данных;

sv: «Теневое» значение ячейки – символ «теневой» строки луча данных;

p_1: первый указатель – принимает значения 1 (указатель в ячейке) или 0 (указатель дальше от ячейки). А для 1-й ячейки может принимать значение 2 (указатель перед ячейкой), что означает, что p_1 указывает на «нулевое» место, для которого нет реальной ячейки. Технически это удобно, потому что у функции $\mathrm{str}(x_1, 0, i)$ есть значение – пустая строка.

p_2: второй указатель – значения 1 или 0, а для 1-й ячейки может принимать значение 2;

ic: увеличивать на 1 (increment) – флаг 1 или 0;

dc: уменьшать на 1 (decrement) – флаг 1 или 0;

$MayBe1$: Флаг (1 или 0) возможного равенства числа из пакета данных единице – что соответствует приходу пакета данных в нужную ячейку, если MayBe1 оставалось равной 1 для каждого сообщения из пакета данных. Но! для первой ячейки луча данных возможно ещё и значение 2 – это нужно для проверки пакета данных на его равенство нулю – что коррелирует с таким же значением у p_1 для первой ячейки;

$IsFirst$: неизменяемый регистр, «флаг», принимающий значение 1 для первой ячейки луча данных и 0 для любой другой ячейки.

Теперь нам осталось написать обработчики событий для всех протоколов, которые вызываются из интерпретатора стандартного языка SI. Сама программа интерпретатора стандартного языка SI (написанная на аппаратном SI) была рассмотрена в предыдущем разделе. После написания обработчиков событий архитектура Машины исполнения компьютерных алгоритмов будет построена. Этим мы займёмся в следующем разделе.

4 Обработчики событий ячеек и процессора

I. Обработчики событий ячеек – присваивание значения ячейке, очистки луча данных, проверка луча данных на числовой тип значения

I.1. Присваивание значения ячейке или её регистру

Следующий обработчик события показывает, что сообщение о присваивании, поступившее в данную ячейку, приводит к изменению его обычного («реального») значения на значение $msgAbc$ из сообщения:

(/# Присвоить rv значение #/, $msgAbc$)

$rv = msgAbc$

В частности, очистку «обычного» значения ячейки вызывает сообщение:

(/# Присвоить rv значение #/, $\ominus$)

Это сообщение необходимо отправлять в следующую ячейку в том случае, если мы меняем в текущей ячейке значение rv с $\ominus$ на какой-то значимый символ, чтобы конечная строка осталась конечной:

$SendTo_0\, NextCell$ (/# Присвоить rv значение #/, $\ominus$)

Потому что нам надо установить в строке новый конец, который находится в ячейке, предшествующий первой пустой ячейке на луче данных. И поэтому мы делаем пустым значение у ячейки, следующей за данной. Ведь пустое значение данной ячейки мы «испортили» и нам надо сдвинуть конец строки на 1 ячейку от процессора.

Для всех остальных регистров ячеек имеются аналогичные сообщения. Например, для p_1:

(/# Присвоить p_1 значение #/, 1)

Отправлять данные сообщения можно не только в следующую ячейку, но и в предыдущую (если исходная ячейка не является первой на луче данных):

$SendTo_{-3}\, PreviousCell$ (Присвоить sv значение, sv)

Обратим внимание, что отправка сообщения «назад» здесь происходит очень быстро – в ближайший (1/8)-момент. И в устройстве-получателе будет отработано (сделано присваивание) к ближайшему (1/4)-моменту.

И из процессора в первую ячейку луча данных (с именем x_1, например):

$SendTo_0\, x_1$ (/# Присвоить sv значение #/, «x»)

Движение данного сообщения «вперед» (в направлении от процессора) происходит в стандартный для передачи таких сообщений ближайший 1-момент.

Аналогичный обработчик и для сообщения (/# Присвоить sv значение #/, $msgAbc$).

Для p_1 обработчиков сразу 4, в том числе 2 для ошибок:

(/# Присвоить p_1 значение #/, $msgNum$), $\mathrm{IsFlag}(msgNum)$

$p_1 = msgNum$

(/# Присвоить p_1 значение #/, $msgNum$), $msgNum = 2, IsFirst = 1$

$p_1 = msgNum$

(/# Присвоить p_1 значение #/, $msgAbc$), $\mathrm{NotFlag}(msgAbc), msgNum \neq 2$

// Аварийное завершение работы

$(/\#$ Присвоить p_1 значение $\#/, msgNum), \mathrm{NotFlag}(msgAbc), msgNum = 2, IsFirst = 0$

// Аварийное завершение работы

Но мы построим обработчики и для случая, когда нам надо защититься от удаления конца строки при присваивании нового символа вместо пустого:

$(/\#$ Присвоить rv значение с защитой конца строки $\#/, msgAbc), msgAbc = \ominus$

$rv = msgAbc$

$(/\#$ Присвоить rv значение с защитой конца строки $\#/, msgAbc), msgAbc \neq \ominus, rv \neq \ominus$

$rv = msgAbc$

$(/\#$ Присвоить rv значение с защитой конца строки $\#/, msgAbc), msgAbc \neq \ominus, rv = \ominus$

$rv = msgAbc$

$SendTo_0\, NextCell\, (/\#$ Присвоить rv значение $\#/, \ominus)$

Аналогичные обработчики для sv.

Но нам нужно ещё иметь сообщения для присваивания каких-нибудь стандартных значений (например – результатов работы функции $\mathrm{Comp}(x_1, x_2)$) лучу данных. А это может потребовать доступа из процессора не только к первой ячейке. На этот случай предусмотрим следующие сообщения и соответствующие обработчики:

$(/\#$ В ячейке с номером на 1 больше присвоить rv значение $\#/, msgAbc)$

$SendTo_0\, NextCell\, (/\#$ Присвоить rv значение $\#/, msgAbc)$

$(/\#$ В ячейке с номером на 2 больше присвоить rv значение $\#/, msgAbc)$

$SendTo_0\, NextCell\, (/\#$ В ячейке с номером на 1 больше присвоить rv значение $\#/, msgAbc)$

Этого нам хватит, наверно, но легко добавить по аналогии несколько более «дальнобойных» присваиваний. Аналогичные сообщения есть для присваивания в регистр sv.

I.2. Очистка всех «обычных» значений ячеек и очистка произвольного регистра ячеек

Разумеется, для записи пустой строки на луче данных достаточно просто очистить первую ячейку. Потому что строка заканчивается тем символом, который предшествует первой пустой ячейке. Такое наше «соглашение о способе кодирования на ленте» (Дж. Булос, Р Джеффри - Вычислимость и логика Глава 3. Машины Тьюринга – см. решения для упражнения 3.2).

Я вспомнил про «соглашение о способе кодирования на ленте в момент остановки», потому что однажды у меня была очень странная полемика о необходимости (по мнению оппонента), очищать ленту Машины Тьюринга от всего «лишнего», чтобы записать на неё результат. И спор завершился тем, что я сослался на «соглашение о способе кодирования».

Но интересно, что для модели Машина исполнения компьютерных алгоритмов есть возможность очистить весь (бесконечный!) луч данных. Практически – мгновенно. Это удобно для случая, когда мы ищем конец строки при движении не от процессора, а к процессору. Без этого можно обойтись, но проще воспользоваться имеющейся возможностью очистки всего луча данных.

$(/\#$ Очистить rv здесь и дальше $\#/, \ominus)$

$rv = \ominus$

$SendTo_0\, NextCell\, (/\#$ Очистить rv здесь и дальше $\#/, \ominus)$

Заметим, что мы отправляем «эстафетные» сообщения в сторону от процессора в 1-моменты, но в данном случае, вроде бы, могли отправить и со скоростью в 4 раза (например) быстрее – в 1/4-моенты. Но у нас ведь могут возникать какие-то аварийные сообщения. То, что мы здесь не расписываем, полагая корректными алгоритмы и входные данные. Но всё же сохраним возможность для корректной обработки состояний ошибок и отправим сообщение от процессора с «обычной» для таких передач скоростью.

Аналогичные обработчики событий имеются для сообщений:

(/# Очистить sv здесь и дальше #/, $\ominus$)

(/# Очистить p_1 здесь и дальше #/, $\ominus$)

(/# Очистить p_2 здесь и дальше #/, $\ominus$)

Только для p_1 и p_2 присваивается 0 (Ноль), а не $\ominus$.

I.3. Установка указателя p_1 и очистка rv от данной ячейки и далее

Нам потребуется сообщение для решения комплексной задачи по одновременной установке регистра p_1 в ячейке-получателе с очисткой «реальных» значений в данной ячейке и последующих.

(/# Установить указатель p_1 и очистить rv здесь и далее #/, $\ominus$)

$p_1 = 1$

$rv = \ominus$

$SendTo_0\, NextCell$ (/# Очистить rv здесь и дальше #/, $\ominus$)

I.4. Проверка на число. Может начинаться с любой ячейки луча данных

Проверка числа на корректность записи включает в себя и проверку того, чтобы самый старший разряд (цифра) у числа не была равна «0». Поэтому ниже обработчики для двух типов событий.

(/# Проверка на число #/, $\ominus$), $\mathrm{IsNum}(rv), rv \neq 0$

$SendTo\, NextCell$ (/# Проверка на число #/, $\ominus$)

(/# Проверка на число #/, $\ominus$), $\mathrm{IsNum}(rv), IsFirst = 1, rv = 0$

$SendTo\, NextCell$ (/# Проверка на число #/, $\ominus$)

(/# Проверка на число #/, $\ominus$), $\mathrm{IsNum}(rv), IsFirst = 0, rv = 0$

$SendTo\, NextCell$ (/# Проверка на число и ведущие нули #/, $\ominus$)

(/# Проверка на число и ведущие нули #/, $\ominus$), $\mathrm{IsNum}(rv), rv \neq 0$

$SendTo\, NextCell$ (/# Проверка на число #/, $\ominus$)

(/# Проверка на число и ведущие нули #/, $\ominus$), $\mathrm{IsNum}(rv), rv = 0$

$SendTo\, NextCell$ (/# Проверка на число и ведущие нули #/, $\ominus$)

(/# Проверка на число #/, $\ominus$), $\mathrm{NotNum}(rv), rv \neq \ominus$

// Это ошибка, аварийное завершение работы

(/# Проверка на число и ведущие нули #/, $\ominus$), $\mathrm{NotNum}(rv), rv \neq \ominus$

// Это ошибка, аварийное завершение работы

(/# Проверка на число #/, $\ominus$), $IsFirst = 1, \mathrm{NotNum}(rv), rv = \ominus$

// Это ошибка, аварийное завершение работы

(/# Проверка на число #/, $\ominus$), $IsFirst = 0, \mathrm{NotNum}(rv), rv = \ominus$

// Ничего не делать – это завершение проверки на цифру

$(/\#$ Проверка на число и ведущие нули $\#/, \ominus), IsFirst = 0, \mathrm{NotNum}(rv), rv = \ominus$

// Это ошибка, аварийное завершение работы

Интервал синхронизации для любого данного обработчика событий, который отправляет сообщение, считаем равным 1 моменту, так как он передаёт «эстафетное» сообщения в направлении от процессора.

II. Обработчики событий ячеек – увеличение/уменьшение числа на луче данных на 1 и работа с теневой очередью

II.1. Увеличение на 1 числа на луче данных.

Изначально сообщение приходит в 1ю ячейку луча данных.

В принципе, для записанных программой чисел проверка на число не нужна, если программа работает корректно. И для входного числового аргумента достаточно провести одну проверку – при первом использовании данного аргумента как числа. Но рассмотрим более сложный вариант увеличения числа на 1 с проверкой, потому что этот вариант универсален, хоть и чуть сложнее. Как ни странно, но проверка не замедляет работу программы (при успешной проверке), потому что никакого сообщения в процессор при успешной проверке не возвращается. Можно считать, что увеличение на 1 числа на луче данных происходит мгновенно, как только обработана 1-я ячейка.

Далее мы будем рассматривать протокол передачи сообщения /# Увеличить на 1 #/, но совершенно аналогичные обработчики у нас имеются для сообщения /# Увеличить тень на 1 #/. При этом сообщения /# Присвоить rv значение #/ и /# Проверка на число #/ заменяются на /# Присвоить sv значение #/ и /# Проверка на теневое число #/ соответственно. И необходимо заранее позаботиться, чтобы на луче данных имелась необходимая теневая строка (точнее, теневое число, в данном случае).

Теперь вернёмся к рассмотрению обработки сообщений ячейками памяти:

1) Обработчики события /# Увеличить на 1 #/**, прекращающие передачу сообщений дальше (сделано всё, что надо, для увеличения значения луча данных на 1 – либо ошибка):**

$(/\#$ Увеличить на $1\ \#/, \ominus), IsFirst = 1, \mathrm{NotNum}(rv)$

// Это ошибка, аварийное завершение работы

$(/\#$ Увеличить на $1\ \#/, \ominus), IsFirst = 0, \mathrm{NotNum}(rv), rv \neq \ominus$

// Это ошибка, аварийное завершение работы

$(/\#$ Увеличить на $1\ \#/, \ominus), IsFirst = 0, rv = \ominus$

$rv = 1$

$SendTo_0\, NextCell\, (/\#$ Присвоить rv значение $\#/, \ominus)$

Комментарий: Работа по увеличению луча данных на 1 завершается, когда встречается не цифра. Если это пустой символ, то необходимо после замены его на 1 восстановить конец строки – в следующей ячейке.

2) Обработчики события /# Увеличить на 1 #/**, переключающиеся на передачу сообщения «Проверка на число»**

$(/\#$ Увеличить на $1\ \#/, \ominus), \mathrm{IsNum}(rv), rv \neq 9$

$rv = rv + 1$

$SendTo_0\, NextCell\, (/\#$ Проверка на число $\#/, \ominus)$

3) Обработчики события «Увеличение на 1 луча данных», передающие то же сообщение дальше:

$(/\#$ Увеличить на $1\ \#/, \ominus), rv = 9$

$rv = 0$

$SendTo_0\, NextCell\, (/\#$ Увеличить на $1\, \#/, \ominus)$

II.2. Отбросить тень – скопировать обычные символы луча данных в теневые.

Сначала следующее сообщение поступает в 1ю ячейку от процессора.

$(/\#$ Отбросить тень $\#/, \ominus), rv = \ominus$

$sv = rv$ // Никаких сообщений дальше не передавать – это конец работы по «отбрасыванию тени»

$(/\#$ Отбросить тень $\#/, \ominus), rv \neq \ominus$

$sv = rv$

$SendTo_0\, NextCell\, (/\#$ Отбросить тень $\#/, \ominus)$

Рассмотренное сообщение не возвращает никакого ответа процессору. Хотя оперативное завершение данного протокола передачи наступает через один момент после получения первого сообщения 1-й ячейкой луча данных – даже если строка бесконечная.

Но при работе с очередью, оперативное завершение этого протокола создаст ситуацию, когда сокращать очередь можно будет только на 1 символ за 2 момента (рассмотрим в следующем параграфе). А это в 2 раза замедлит, в сравнении с обычной скоростью движения от процессора, движение пакетов данных и их отправку.

Поэтому мы построим ещё один протокол отбрасывания тени, который сообщает процессору о полном (а не только оперативном) завершении своей работы. Такое сообщение будет отправлено заметно позже, чем наступило оперативное завершение рассмотренного протокола. В следующем протоколе завершение придётся отсчитывать не от готовности первой ячейки луча данных, а лишь добравшись до первой пустой ячейки луча данных, отправить уведомление процессору.

Зато теневая строка будет полностью построена, а это позволит процессору после такого завершения теневой строки отправлять сообщения по теневой строке с любой скоростью – но не дальше её конца. Бесконечная строка не годится для такого протокола, разумеется, потому что её конца невозможно достигнуть, а уведомление процессору о завершении построения теневой строки не может быть отправлено поэтому.

$(/\#$ Отбросить тень и уведомить $\#/, \ominus), IsFirst = 1, rv = \ominus$

$sv = rv$

// Теперь отправляем уведомление процессору о завершении построения теневой строки:

$SendTo_{-2}\, Proc\, (/\#$ Теневая строка построена $\#/, \ominus)$

$(/\#$ Отбросить тень и уведомить $\#/, \ominus), IsFirst = 0, rv = \ominus$

$sv = rv$

// Теперь отправляем уведомление к процессору о завершении построения теневой строки:

$SendTo_{-2}\, PreviousCell\, (/\#$ Теневая строка построена $\#/, \ominus)$

$(/\#$ Отбросить тень и уведомить $\#/, \ominus), rv \neq \ominus$

$sv = rv$

$SendTo_0\, NextCell\, (/\#$ Отбросить тень и уведомить $\#/, \ominus)$

$(/\#$ Теневая строка построена $\#/, \ominus), IsFirst = 1$

$SendTo_{-2}\, Proc\, (/\#$ Теневая строка построена $\#/, \ominus)$

$(/\#$ Теневая строка построена $\#/, \ominus), IsFirst = 0$

$SendTo_{-2}\, PreviousCell\, (/\#$ Теневая строка построена $\#/, \ominus)$

Обработчик сообщения типа /# Теневая строка построена #/ для процессора будет зависеть от значения регистра $Continue_0$. Благодаря этому данное сообщение сможет использоваться в разных протоколах передач. Дадим тут обработчик для отдельной команды отбрасывания тени основной строки в теневую строку с уведомлением:

$(/\#$ Теневая строка построена $\#/, \ominus)$

$, Continue_0 = /\#$ Скопировать теневую строку из основной $\#/$

$Setmode\, SI$

II.3. Сократить теневую очередь на 1 символ

Для решения многих задач нам поможет работа с теневой строкой луча данных как с очередью. Мы отправляем сообщения (друг за другом!) по «эстафете», которая отправляет назад символ из той ячейки, до которой «эстафета» добралась. Так растёт «хвост» очереди, дальний от процессора. А «голова» очереди постепенно отдаёт процессору поступившие «издалека» символы.

Пример активной части теневой строки (очереди) по моментам времени, когда работа ведётся с исходной строкой 1234567890 где слева – процессор, а справа – «передовое» сообщение вовлекает новые ячейки в передачу символов:

1; 22; 233; 3344; 34455; 445566; 4556677; 55667788; 566778899; 6677889900;

677889900; 77889900; 7889900; 889900; 89900; 9900; 900; 00; 0.

Если «передовое» сообщение продвигается со стандартной скоростью 1 ячейка за 1 момент, то, как видим, процессор получает 1 символ за 2 момента. Поэтому и отправлять сообщения он должен только в 2-моменты. Значит, в каждый момент меняется только половина ячеек (только чётные или только нечётные). Но при этом сдвиг происходит целиком по всей очереди каждый момент. Почему? Вот пояснение на примере:

001111111111222

Нам надо сдвинуть весь блок единиц на 1 позицию каждую единицу:

011111111112222

И это достигается тем, что вместо последнего нуля ставится 1, в вместо последней единицы ставится 2.

Однако мы можем отправлять сообщения со скоростью 2 ячейки за 1 момент на луче данных, если построили теневую строку в соответствии с протоколом сообщения /# Отбросить тень и уведомить #/. И тогда процессор сможет отправлять запрос на очередной символ из очереди (сокращая очередь) каждый 1-момент. Из этого и будем исходить.

После сделанных пояснений перейдём к сообщениям и их обработчикам.

Для начала рассмотрим работу с «готовой» очередью, созданной при помощи протокола с сообщением типа /# Отбросить тень и уведомить #/.

Процессор отправляет, например, сообщение на луч данных $@Source_1$

$SendTo_0\, @Source_1\, (/\#$ В процессор из теневой очереди $\#/, \ominus)$

Вот протокол передачи сообщения по эстафете по лучу данных:

$(/\#$ В процессор из теневой очереди $\#/, \ominus), IsFirst = 1, sv = \ominus$

$SendTo_{-2}\, Proc\, (/\#$ В процессор из теневой очереди $\#/, sv)$

$(/\#$ В процессор из теневой очереди $\#/, \ominus), IsFirst = 1, sv \neq \ominus$

$SendTo_{-2}\, Proc\, (/\#$ В процессор из теневой очереди $\#/, sv)$

$SendTo_{-1}\, NextCell\, (/\#$ В процессор из теневой очереди $\#/, \ominus)$

$(/\#$ В процессор из теневой очереди $\#/, \ominus), IsFirst = 0, sv = \ominus$

$SendTo_{-2}\, PreviousCell\, (/\#$ Присвоить sv значение $\#/, sv)$

$(/\#$ В процессор из теневой очереди $\#/, \ominus), IsFirst = 0, sv \neq \ominus$

$SendTo_{-2}\, PreviousCell\, (/\#$ Присвоить sv значение $\#/, sv)$

$SendTo_{-1}\, NextCell\, (/\#$ В процессор из теневой очереди $\#/, \ominus)$

И ещё рассмотрим работу с очередью, созданной при помощи протокола с сообщением типа /# Отбросить тень #/. Напоминаю, что та очередь готова «сразу», даже если речь о строках бесконечного размера. Но использовать её можно только со скоростью 1 символ за 2 момента.

Процессор отправляет, например, сообщение на луч данных $@Source_1$

$SendTo_1\, @Source_1\, (/\#$ В процессор из произвольной теневой очереди $\#/, \ominus)$

А протокол передачи для сообщения типа /# В процессор из произвольной теневой очереди #/ точно такой же, как для /# В процессор из теневой очереди #/, то только с заменой всех:

$SendTo_{-1}\, NextCell\, (/\#$ В процессор из теневой очереди $\#/, \ominus)$

На:

$SendTo_0\, NextCell\, (/\#$ В процессор из произвольной теневой очереди $\#/, \ominus)$

Таким образом, протоколы передачи для сообщений типа /# В процессор из произвольной теневой очер и /# В процессор из теневой очереди #/ построены.

Не важно, что некоторые обработчики выше отправляет 2 сообщения в процессе обработки поступившего ему сообщения. Дело в том, что обработчик отправляет их в разные от себя стороны – поэтому они не встретятся, и не будут конфликтовать в одной ячейке.

Другое дело, что сообщения типа /# В процессор из теневой очереди #/ движутся от ячейки к ячейки в направлении от процессора по эстафете по 1/2-моментам, а сообщения типа /# Присвоить sv значе - против, и по 1/4-моментам.

Но сообщение /# Присвоить sv значение #/ движется в направлении процессора только на одну ячейку назад и всегда через 1/4-момента от момента, когда в данную ячейку поступило сообщение /# В процессор из теневой очереди #/ и через 1/4-момента от момента, когда это сообщение ушло из предыдущей ячейки. Поэтому события, вызываемые сообщениями с типами /# В процессор из теневой очереди #/ и /# Присвоить sv значение #/ всегда разнесены во времени на 1/4-момента минимум и не конфликтуют друг с другом.

Что касается сообщения /# В процессор из произвольной теневой очереди #/, то там разрыв во времени не меньше, потому что сообщения идут реже.

Разумеется, мы рассмотрели передачу по «эстафете» только одного сообщения, а для последовательной работы с очередью процессор должен отправлять их друг за другом. Это мы рассмотрим далее.

И ещё одно существенное дополнение. Помимо типа сообщения

/# В процессор из теневой очереди #/

У нас есть точно такие же обработчики для сообщений типа

/# В процессор из теневой очереди $@Source_1$ для увеличения числа в $@Target_1$#/

/# В процессор из теневой очереди $@Source_1$ для уменьшения числа в $@Target_1$#/

То есть, надо взять вышенаписанные обработчики событий для сообщений типа /# В процессор из тенев и заменить в них этот тип (в заголовках и телах обработчиков) на один из названных типа. Это и будут те обработчики, которые тоже имеются у каждой ячейки.

И у нас есть соответствующие типы сообщений для любых регистров или любых служебных лучей данных вместо $@Source_1$ и $@Target_1$. Например:

/# В процессор из теневой очереди $Command$ для увеличения числа в $i_{Program}$ #/

Приведённый тип сообщения может использоваться для перехода к исполнению следующей команды, для чего надо увеличить число на служебном луче данных $i_{Program}$ на количество символов текущей команды, которая находится в регистре $Command$. И для этого надо отправлять данные сообщения в регистр $Command$, увеличивая при этом и число в $i_{Program}$ на 1 до тех пор, пока из очереди $Command$ не будет получен пустой символ. Соответствующие обработчики событий будут даны в следующем разделе.

II.4. Уменьшение на 1 числа на луче данных.

Теперь нам надо написать обработчики событий при арифметическом вычитании 1 от числа на данном луче данных. «Арифметическое вычитание» не даёт отрицательных чисел, вычитание единицы от нуля снова даёт ноль. И при арифметическом вычитании большего числа от меньшего числа получается ноль. В остальных случаях – это обычное вычитание. Далее в этом пункте под «уменьшением числа на 1» будем понимать арифметическое вычитание 1 от числа.

Уменьшение на 1 требует возврата к уже отработанной ячейке (ближе к процессору) только в 2 случаях:

1. Мы имели дело с самым младшим разрядом, он был равен 0 (до вычитания 1), а старше разрядов не оказалось;

2. Мы имели дело с самым старшим разрядом числа, равным 1 (до вычитания 1), и это не младший разряд числа.

В первом случае нам надо вернуться в первую ячейку из второй и восстановить там прежнюю цифру – ноль (арифметическое вычитание не изменяет число, равное 0). А во втором случае необходимо вернуться в предыдущую ячейку и очистить её, потому что цифра «0» не пишется, если нет значимых (отличных от нуля) разрядов старше, когда это – не самый младший разряд.

Если уменьшение на 1 продолжится, то новое сообщение не пройдёт дальше данных ячеек. Потому что в 1-м случае там будет не 0, а младший разряд можно уменьшить хоть до 0, не переходя дальше. А во втором случае даже в старший разряд следующее сообщение об уменьшении на 1 не придёт, потому что есть предыдущий по старшинству разряд, равный теперь максимальной цифре. Равный максимальной цифре потому, что был 0, именно поэтому и пришлось менять ноль на максимальную цифру и переходить к старшему разряду для вычитания 1 в пункте 2.

Поэтому, при последовательных вычитаниях 1 из числа у нас всегда есть время (между 1-моментами) на правильное переписывание самой старшей уменьшаемой цифры. И никаких конфликтов с другими сообщениями об уменьшении на 1 не возникнет. Так как эти сообщения просто не успеют использовать цифру до того, как нужное исправление будет сделано.

Поэтому команды на уменьшение числа на 1 мы можем отправлять каждый 1-момент. Но если нам потребуется отправить другое «эстафетное» сообщение после нескольких вычитаний 1, то надо пропустить ближайший 1-момент, так как процесс исправления последней уменьшенной цифры укладывается в интервал одного момента, и может конфликтовать с сообщениями, отличными от вычитания 1.

$(/\#$ Уменьшить на $1\,\#/, \ominus), rv = 0$

$rv = 9$

$SendTo_0\, NextCell\, (/\#$ Уменьшить на $1\,\#/, \ominus)$

$(/\#$ Уменьшить на $1\,\#/, \ominus), \mathrm{IsNum}(rv), rv \neq 0, IsFirst = 1$

$rv = rv - 1$

$SendTo_0\, NextCell\, (/\#$ Проверка на число $\#/, \ominus)$

$(/\#$ Уменьшить на $1\,\#/, \ominus), \mathrm{IsNum}(rv), rv > 1, IsFirst = 0$

$rv = rv - 1$

$SendTo_0\, NextCell\, (/\#$ Проверка на число $\#/, \ominus)$

$(/\#$ Уменьшить на $1\,\#/, \ominus), \mathrm{IsNum}(rv), rv = 1, IsFirst = 0$

$rv = 0$

$SendTo_0\, NextCell\, (/\#$ Проверка на ведущий $0\,\#/, \ominus)$

$(/\#$ Проверка на ведущий $0\,\#/, \ominus), \mathrm{NotNum}(rv), rv \neq \ominus$

// Аварийное завершение работы

$(/\#$ Проверка на ведущий $0\,\#/, \ominus), \mathrm{NotNum}(rv), rv = \ominus$

$SendTo_{-2}\, PreviousCell\, (/\#$ Присвоить rv значение $\#/, \ominus)$

$(/\#$ Проверка на ведущий $0\,\#/, \ominus), \mathrm{IsNum}(rv)$

$SendTo_0\, NextCell\, (/\#$ Проверка на число $\#/, \ominus)$

$(/\#$ Уменьшить на $1\,\#/, \ominus), IsFirst = 1, \mathrm{NotNum}(rv)$

// Аварийное завершение работы

$(/\#$ Уменьшить на $1\,\#/, \ominus), IsFirst = 0, \mathrm{NotNum}(rv), rv \neq \ominus$

// Аварийное завершение работы

$(/\#$ Уменьшить на $1\,\#/, \ominus), IsFirst = 0, \mathrm{NotNum}(rv), rv = \ominus$

$SendTo_{-2}\, PreviousCell\, (/\#$Восстановить цифру $0\,\#/, \ominus)$

$(/\#$Восстановить цифру $0\,\#/, \ominus), IsFirst = 0$

// Аварийное завершение работы – старший разряд не может быть нулём, если это не разряд единиц

$(/\#$Восстановить цифру $0\,\#/, \ominus), IsFirst = 1$

$rv = 0$

Комментарий: Работа по уменьшению на 1 луча данных завершается, когда встречается не цифра.

III. Обработчики событий процессора – вычисления $@Target_1{+}\mathrm{len}(@Source_1)$**,** $@Target_1{-}\mathrm{len}(@Source_1)$**,** $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2)$**,** $@Target_1 = \mathrm{GoTo}(@Source_1)$

Можно написать такие обработчики событий процессора, запуск которых приводит к увеличению/уменьшению числа в @Target на число, равное длине строки @Source.

III.1. Прибавить к числу в $@Target_1$ длину строки из $@Source_1$

(/# Увеличить число в $@Target_1$ на количество символов в $@Source_1$ #/, $\ominus$)

$Continue_0 =$ /# Начать увеличивать число в $@Target_1$ на количество символов в $@Source_1$ #/

$SendTo_0\ @Source_1$ (/# Отбросить тень с уведомлением #/, $\ominus$)

(/# Теневая строка построена #/, $\ominus$)

, $Continue_0 =$ /# Начать увеличивать число в $@Target_1$ на количество символов в $@Source_1$ #/

$Continue_0 = \ominus$

$SendTo_0\ @Source_1$ (/# В процессор из теневой очереди $@Source_1$ для увеличения числа в $@Target_1$#/

(/# В процессор из теневой очереди $@Source_1$ для увеличения числа в $@Target_1$ #/, $msgAbc$)

, $msgAbc = \ominus$

// Этот обработчик является завершающим при увеличении числа в $@Target_1$. После чего происходит возврат процессора в режим исполнения программы аппаратного языка стандартной интерпретации.

$Setmode\ SI$

(/# В процессор из теневой очереди $@Source_1$ для увеличения числа в $@Target_1$#/, $msgAbc$)

, $msgAbc \neq \ominus$

$SendTo_0\ @Target_1$ (/# Увеличить на 1 #/, $\ominus$)

$SendTo_0\ @Source_1$ (/# В процессор из теневой очереди $@Source_1$ для увеличения числа в $@Target_1$#/

III.2. Отнять от числа в $@Target_1$ длину строки из $@Source_1$

Напоминаю, что речь идёт об арифметическом вычитании, которое при вычитании большего числа от меньшего даёт ноль.

(/#Уменьшить число в $@Target_1$ на количество символов в $@Source_1$ #/, 0)

$Continue_0 =$ /# Начать уменьшать число в $@Target_1$ на количество символов в $@Source_1$ #/

$SendTo_0\ @Source_1$ (/# Отбросить тень с уведомлением #/, $\ominus$)

(/# Теневая строка построена #/, $\ominus$)

, $Continue_0 =$ /# Начать уменьшать число в $@Target_1$ на количество символов в $@Source_1$ #/

$Continue_0 = \ominus$

$SendTo_1\ @Source_1$ (/# В процессор из теневой очереди $@Source_1$ для уменьшения числа в $@Target_1$#/

(/# В процессор из теневой очереди $@Source_1$ для уменьшения числа в $@Target_1$#/, $msgAbc$)

, $msgAbc = \ominus$

$wait_0$ // Выравниваем по сетке синхронизации 1-моментов, чтобы гарантированно пропустить следующим шагом 1 момент

$wait_0$ // Не забываем, что один момент может уйти на завершение правки самого старшего разряда уменьшаемого числа. Ждём, чтобы новые сообщения, если они будут отправлены по $@Target_1$, не вступили в конфликт со старыми, еще не обработанными, сообщениями.

// Этот обработчик является завершающим при уменьшении числа в $@Target_1$. После чего происходит возврат процессора в режим исполнения программы аппаратного языка стандартной интерпретации.

$Setmode\ SI$

(/# В процессор из теневой очереди $@Source_1$ для уменьшения числа в $@Target_1$#/, $msgAbc$)

, $msgAbc \neq \ominus$

$SendTo_0\ @Target_1$ (/# Уменьшить на 1 #/, $\ominus$)

$SendTo_0\ @Source_1$ (/# В процессор из теневой очереди $@Source_1$ для уменьшения числа в $@Target_1$#/

III.3. Вычисление $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2)$

Для расчёта $\mathrm{Comp}(@Source_1, @Source_2)$ будем использовать очереди, создаваемы сообщением /# Отбросить тень #/. Потому что одна или обе строки могут быть бесконечными, что исключает применение сообщения /# Отбросить тень с уведомлением #/. Кроме того, нам желательно получить сразу 2 очереди и при этом – практически одновременно и без уведомлений, потому что результат функции может оказаться найденным на первых символах. Есть и другие довольно очевидные резоны для выбора в пользу очередей, «сделанных» через сообщение /# Отбросить тень #/ в данном случае.

Процессор сам себе отправляет сообщение:

(/# Вычислить $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2$ #/, $\ominus$)

Вот начнём с обработчика этого события:

(/# Вычислить $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2$ #/, $\ominus$)

$SendTo_0\ @Source_1$ (/# Отбросить тень #/, $\ominus$)

$SendTo_0\ @Source_2$ (/# Отбросить тень #/, $\ominus$)

$Continue_0 =$ /# Аргумент$_1$ для $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2)$ #/

$wait$

$SendTo_0\ @Source_1$ (/# В процессор из произвольной теневой очереди #/, $\ominus$)

// Команда $wait$ тут (и в аналогичных обработчиках) не требуется, потому что процессор всё равно не будет ничего делать, пока ему не придёт сообщение (/# В процессор из произвольной теневой очер Сейчас оно должно будет прийти из $@Source_1$.

Для работы с очередью, нам надо отправлять соответствующие сообщения на луч данных с интервалом в 2 момента. Но так и получается, потому что сообщения отправляются попеременно в 2 разных луча данных в 1-моменты.

(/# В процессор из произвольной теневой очереди #/, $msgAbc$)

, $Continue_0 =$ /# Аргумент$_1$ для $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2)$ #/

$msgBuff_1 = msgAbc$

$Continue_0 =$ /# Аргумент$_2$ для $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2)$ #/

$SendTo_0\ @Source_2$ (В процессор из произвольной теневой очереди, $\ominus$)

$wait$

(/# В процессор из теневой очереди #/, $msgAbc$)

, $Continue_0 =$ /# Аргумент$_2$ для $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2)$ #/

$, msgAbc = \ominus$

$, msgBuff_1 = \ominus$

$SendTo_0\, @Target_1\, (/\#$ Очистить rv здесь и дальше $\#/, \ominus)$

$wait$

$SendTo_0\, @Target_1\, (/\#$ Присвоить rv значение $\#/, 0)$

// Этот обработчик является одним из вариантов завершения при вычислении $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2)$. Строки совпали, дойдя до конца, результат 0. Теперь происходит возврат процессора в режим исполнения программы аппаратного языка стандартной интерпретации.

$wait$

$wait_0$

$Setmode\, SI$

$(/\#$ В процессор из произвольной теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ Аргумент$_2$ для $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2)\, \#/$

$, msgBuff_1 < msgAbc$

// Здесь обрабатывается и случай $msgBuff_1 = \ominus, msgAbc \neq \ominus$

$SendTo_0\, @Target_1\, (/\#$ Очистить rv здесь и дальше $\#/, \ominus)$

$SendTo_0\, @Target_1\, (/\#$ Присвоить rv значение $\#/, 1)$

// Этот обработчик является одним из вариантов завершения при вычислении $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2)$. Строка в $@Source_1$ оказалась меньше строки в $@Source_2$, результат 1. Теперь происходит возврат процессора в режим исполнения программы аппаратного языка стандартной интерпретации.

$Setmode\, SI$

$(/\#$ В процессор из произвольной теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ Аргумент$_2$ для $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2)\, \#/$

$, msgBuff_1 > msgAbc$

// Здесь обрабатывается и случай $msgBuff_1 \neq \ominus, msgAbc = \ominus$

$SendTo_0\, @Target_1\, (/\#$ Очистить rv здесь и дальше $\#/, \ominus)$

$SendTo_0\, @Target_1\, (/\#$ Очистить rv здесь и дальше $\#/, 2)$ // Мы считаем, что для записи числа 2 нам достаточно одного символа на «аппаратном» уровне. В общем случае можно вместо данной команды использовать присваивание $@Target_1.Register = 2$, хоть это и медленно, но не дольше работы протокола передачи, если придумать и запускать протокол передачи для этого.

// Этот обработчик является одним из вариантов завершения при вычислении $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2)$. Строка в $@Source_1$ оказалась больше строки в $@Source_2$, результат 2. Теперь происходит возврат процессора в режим исполнения программы аппаратного языка стандартной интерпретации.

$Setmode\, SI$

$(/\#$ В процессор из произвольной теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ Аргумент$_2$ для $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2)\, \#/$

$, msgAbc \neq \ominus, msgBuff_1 \neq \ominus, msgAbc = msgBuff_1$

$Continue_0$ = /# Аргумент$_1$ для $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2)$ #/
$SendTo_0\ @Source_1$ (/# В процессор из произвольной теневой очереди #/, $\ominus$)

III.4. Вычисление $@Target_1 = \mathrm{Goto}(@Source_1)$

Допустим, у нас на луче данных $@Source_1$ записана полная программная строка для метки – включая обрамляющие квадратные скобки и перевод строки $ChrEnter$. Например:

$[bookmark];$

Напоминаю, что программная строка для метки не содержит ни одной квадратной скобки, кроме открывающей скобки в начале строки текста и закрывающей скобки перед двумя последними символами «;» и $ChrEnter$ строки текста. И это сильно упрощает поиск этой строки в тексте программы, находящейся на луче данных $Program$.

Облегчает поиск тем, что если в какой-то момент поиска в позиции i строки $Program$ обнаружено несовпадение с соответствующим (не первым) символом из $@Source_1$, то можно начинать новый поиск с самого начала строки $@Source_1$ и той же позиции i в строке $Program$.

И никакие уже просмотренные подстроки в строке Program, включающие в себя символ на месте i, уже не могут совпасть с началом строки $@Source_1$.

Ведь в противном случае у нас имелась бы открывающая квадратная скобка в названии метки, которая не совпадает с квадратной скобкой в начале строки $@Source_1$.

При этом начало нового поиска (с позиции i) обеспечивает продвижение поиска (и новой позиции для i) в строке $Program$ на 1 шаг при первом же сравнении (успешном или неудачном). А это значит, что затраты времени на поиск метки в строке $Program$ у нас получаются в линейных пределах относительно $\mathrm{len}(Program)$. И алгоритм поиска получается значительно проще, чем для поиска произвольной строки, так как произвольная строка может включать в себя «самоповторы» своего начала, например:

ля-ля-ля-жу-жу-ЖУ-ля-ля-ля-жу-жу-жу

Подстрока «ля-ля-ля-жу-жу» встречается как в начале, так и почти в конце данной строки. И если в «обыскиваемой» строке мы нашли подстроку «ля-ля-ля-ля-», то не исключено, что конец «ля-ля-ля-» в ней окажется нужным нам началом «ля-ля-ля-жу». И поиск надо продолжать с части уже просмотренной «обыскиваемой» строки. То же самое, если мы нашли подстроку «ля-ля-ля-жу-жу-ЖУ-ля-ля-ля-жу-жу-ЖУ», то последняя часть этой подстроки может оказаться началом нашей искомой строки.

И нам надо записать на луч данных $@Target_1$ позицию (число) в программном тексте $Program$, которая расположена сразу после программной строки метки, равной строке $@Source_1$. Притом, что строка $@Source_1$ не имеет самоповторов своего начала.

Нам нужна позиция в $Program$ именно после подстроки $@Source_1$, так как именно там расположена команда, которую необходимо исполнить после перехода к данной метке.

Если же метка не будет найдена, то на луч данных $@Target_1$ необходимо будет записать ноль.

1) Начало расчёта $Goto$

Вычисление начинается с получения процессором сообщения (которое он сам себе отправляет):
(/# Вычислить $@Target_1 = \mathrm{Goto}(@Source_1)$ #/, $\ominus$)

Вот обработчики соответствующих событий, происходящих после этого:

(/# Вычислить $@Target_1 = \mathrm{Goto}(@Source_1)$ #/, $\ominus$)

$Continue_0 =$ /# Начать вычислять $@Target_1 = \mathrm{Goto}(@Source_1)$ #/

$SendTo_0\, Program$ (/# Отбросить тень с уведомлением #/, $\ominus$) // Тут нам желательна «быстрая» в использовании тень, потому что при поиске будет происходить последовательный перебор символов программного текста. Кстати, луч данных Program – особенный, на нём не изменяются символы программного текста, а из протоколов используются только передача команды в теневое начало луча данных и поиск метки.

Для отбрасывания тени можно было бы отправлять сообщение подобное /# Отбросить тень #/ - но с удвоенной скоростью. И тогда сразу получался бы нужный результат для быстрой работы с тенью. Но для простоты воспользуемся стандартным получением «быстрой» в использовании тени – через сообщение /# Отбросить тень с уведомлением #/.

$SendTo_0\, @Target_1$ (/# Очистить rv здесь и дальше #/, $\ominus$) // Очищаем для записи нуля в следующем обработчике событий

$SendTo_0\, @Source_1$ (/# Отбросить тень #/, $\ominus$) // При построении теневой строки метки достаточно оперативного завершения. Всё равно в программном тексте будем искать в основном первую квадратную скобку из текста метки, поэтому тратить время на ожидание полного завершения построения тени – нет смысла. А вообще в данном случае у нас метка и она записана в регистре, поэтому можно оперировать её символами почти мгновенно при помощи строковых функций, но применим более общий подход, пригодный и для лучей данных

(/# Теневая строка построена #/, $\ominus$)

, $Continue_0 =$ /# Начать вычислять $@Target_1 = \mathrm{Goto}(@Source_1)$ #/

$Continue_0 =$ /# 1й символ $Program$ для $@Target_1 = \mathrm{Goto}(@Source_1)$ #/

$SendTo_0\, @Target_1$ (/# Присвоить rv значение) #/, 0) // Лишнее присваивание, но пока символа метки ещё не поступило, пусть будет 0. Уже следующий шаг сделает этот счётчик равным 1.

$SendTo_0\, Program$ (/# В процессор из теневой очереди #/, $\ominus$) // Теневая очередь программного текста готова полностью и за символом можно обращаться хоть через 1 момент после получения предыдущего

(/# 1й символ $Program$ для $@Target_1 = \mathrm{Goto}(@Source_1)$ #/, $msgAbc$)

, $msgAbc = \ominus$

// Ошибка. Аварийное завершение – программа не может быть совсем пустой.

(/# 1й символ $Program$ для $@Target_1 = \mathrm{Goto}(@Source_1)$ #/, $msgAbc$)

, $msgAbc \neq \ominus$

$Continue_0 =$ /# 1й символ $@Source_1$ для $@Target_1 = \mathrm{Goto}(@Source_1)$ #/

$msgBuff_2 = msgAbc$ // Текущий символ из $Program$ в цикле попадает в свой регистр после того, как нужный для сравнения символ из $@Source_1$ попал в свой регистр. Поэтому тут $msgBuff_2$. Но в самом начале программы порядок заполнения регистров – другой, чем при возобновлении циклов.

$SendTo_0\,@Target_1\,(/\#$ Присвоить rv значение$)\,\#/, 1)$

$SendTo_1\,@Source_1\,(/\#$ В процессор из теневой очереди $\#/, \ominus)$

2) Поиск в $Program$ первого символа метки **(открывающей квадратной скобки)** из $@Source_1$

В обработчики при $Continue_0 = /\#$ 1й символ $@Source_1$ для $@Target_1 = \mathrm{Goto}(@Source_1)\,\#/$ мы попадаем, когда поиск метки зашёл дальше 1-го символа метки, но – сорвался. Либо же попадаем сюда после трёх начальных обработчиков поиска метки, написанных выше. Здесь нам надо «снова» (если это возобновление цикла и первый раз, если это начало программы) сравнить текущий символ из Program – теперь уже с первым символом метки.

То есть, два следующих обработчика – начало «большого» цикла, когда сравнение (снова) начинается с первого символа луча данных $@Source_1$. В этом «большом цикле» есть 2 подцикла.

1-й подцикл – когда идёт перебор символов $Program$, но ни один не совпадает с 1-м символом $@Source_1$. Этот подцикл представлен единственным обработчиком в конце пункта 2.

2-й подцикл – когда идёт попарный перебор символов из $Program$ и $@Source_1$, и в каждой такой паре символы оказываются равны. Этот подцикл представлен 2-м и последним обработчикам в пункте 3.

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ 1й символ $@Source_1$ для $@Target_1 = \mathrm{Goto}(@Source_1)\,\#/$

$, msgAbc = msgBuff_2$

// Текущий символ программы совпал с первым символом метки и следующий обработчик окажется в следующем пункте – где символы из Program сравниваются с дальнейшими (не первым) символоми метки

$Continue_0 = /\#$ След. символы из $@Source_1$ и $(Program)$ для $@Target_1 = \mathrm{Goto}(@Source_1)\,\#/$

$msgBuff_1 = msgAbc$ // Считаем, что строка метки у нас всегда не пустая – там есть хотя бы скобки, «;» и перевод строки

// Число в $@Target_1$ здесь не меняем – потому что в $@Target_1$ мы отсчитываем текущее место в $Program$, а здесь мы символ из $Program$ не меняли.

$SendTo_1\,@Source_1\,(/\#$ В процессор из теневой очереди $\#/, \ominus)$

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ 1й символ $@Source_1$ для $@Target_1 = \mathrm{Goto}(@Source_1)\,\#/$

$, msgAbc \neq msgBuff_2$

// Текущий символ программы НЕ совпал с первым символом метки и мы остаёмся в текущем пункте – в подцикле сравнения очередных символов программы с первым символом метки.

$Continue_0 = /\#$ Продолжить поиск в $Program$ начала для $@Target_1 = \mathrm{Goto}(@Source_1)\,\#/$

$msgBuff_1 = msgAbc$ // Считаем, что строка метки у нас всегда не пустая – там есть хотя бы скобки, «;» и перевод строки

// Число в $@Target_1$ здесь не меняем – потому что в $@Target_1$ мы отсчитываем текущее место в $Program$, а здесь мы символ из $Program$ на следующий не меняли.

$SendTo_0\,Program\,(/\#$ В процессор из теневой очереди $\#/, \ominus)$ // Напоминаю, что у нас «быстрая теневая очередь» программного текста, к которой можно обращаться за символом хоть каждый

1-момент

Следующие обработчики активируются из-за прихода очередного символа из $Program$. При этом в регистре $msgBuff_1$ уже находится первый символ из $@Source_1$. Не рассматриваем случай конца строки метки ($msgBuff_1 = \ominus$), так как тут у нас начало программной строки метки – а это символ открывающей квадратной скобки. И именно с ним надо сравнить (равен ли) поступивший от Program текущий символ:

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ Продолжить поиск в $Program$ начала для $@Target_1 = \mathrm{Goto}(@Source_1)\, \#/$

$, msgAbc = \ominus$

// Процесс сравнения завершён с отрицательным результатом – строка в $Program$ закончилась, а начало $@Source_1$ не обнаружено

$SendTo_0\, @Target_1\, (/\#$ Очистить rv здесь и дальше $\#/, \ominus)$

$wait$ // Делаем минимальную паузу, которая гарантировано отделяет момент отправки следующих сообщений от сообщений выше – чтобы они все не ушли одновременно

$SendTo_0\, @Target_1\, (/\#$ Присвоить rv значение$)\, \#/, 0)$

// Этот обработчик является одним из вариантов завершения при вычислении $@Target_1 = \mathrm{Goto}(@Source_1)$. Теперь происходит возврат процессора в режим исполнения программы аппаратного языка стандартной интерпретации.

$Setmode\, SI$

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ Продолжить поиск в $Program$ начала для $@Target_1 = \mathrm{Goto}(@Source_1)\, \#/$

$, msgAbc \neq \ominus$

$, msgAbc = msgBuff_1$

// Процесс поиска начала $@Source_1$ удался, дальше надо сравнивать следующие символы из $@Source_1$ и $Program$ попарно. Это рассмотрено в следующем пункте

$Continue_0 = /\#$ След. символы из $@Source_1$ и $(Program)$ для $@Target_1 = \mathrm{Goto}(@Source_1)\, \#/$

$msgBuff_2 = msgAbc$ // Сохраняем очередной символ из программного текста

$SendTo_0\, @Target_1\, (/\#$ Увеличить на $1\, \#/, \ominus)$ // Совпало или нет – но номер позиции символа в программе надо увеличивать, раз рассматриваем очередной символ из $Program$. А когда (и если) метка совпадёт с подстрокой в программе – тогда и зафиксируем результат. В противном случае очистим накопившееся число, и поставим вместо него ноль.

$SendTo_0\, @Source_1\, (/\#$ В процессор из теневой очереди $\#/, \ominus)$ // Это «медленная очередь», но 1 момент прошёл при получении символа их тени Program, а сейчас будет 2-й момент, поэтому можем отправить сообщение в ближайший 1-момент.

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ Продолжить поиск в $Program$ начала для $@Target_1 = \mathrm{Goto}(@Source_1)\, \#/$

$, msgAbc \neq \ominus$

$, msgAbc \neq msgBuff_1$

// Процесс поиска начала $@Source_1$ будет продолжен с тем же $Continue_0$ с нового символа в $Program$, так как текущий символ не подошёл. Данный обработчик представляет собой 1-й подцикл из 2-х. В этом подцикле все последовательные символы Program сравниваются, но не совпадают с 1-м символом метки.

$msgBuff_2 = msgAbc$ // Сохраняем очередной символ из программного текста

$SendTo_0\, @Target_1\, (/\#$ Увеличить на $1\, \#/, \ominus)$ // Совпало или нет – но номер позиции символа в программе надо увеличивать, раз рассматриваем очередной символ из Program. А когда (и если) метка совпадёт с подстрокой в программе – тогда и зафиксируем результат. В противном случае очистим накопившееся число, и поставим вместо него ноль.

$SendTo_0\, Program\, (/\#$ В процессор из теневой очереди $\#/, \ominus)$ // Напоминаю, что у нас «быстрая тень» программного текста, к которой можно обращаться за символом хоть каждый 1-момент

3) Проверка очередных символов из $Program$ и $@Source_1$ **на равенство**

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ След. символы из $@Source_1$ и $(Program)$ для $@Target_1 = \mathrm{Goto}(@Source_1)\, \#/$

$, msgAbc = \ominus$

// Процесс сравнения успешно завершён – строка в $@Source_1$ закончилась.

$SendTo_0\, @Target_1\, (/\#$ Увеличить на $1\, \#/, \ominus)$ // Не забываем, что нам нужна позиция после метки, поэтому сдвигаем позицию с позиции последнего совпавшего символа на следующий за ним.

$wait$

$wait_0$ // Ждём, чтобы увеличение на 1 было оперативно завершено

// Этот обработчик является одним из вариантов завершения при вычислении $@Target_1 = \mathrm{Goto}(@Source_1)$. Теперь происходит возврат процессора в режим исполнения программы аппаратного языка стандартной интерпретации.

$Setmode\, SI$

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ След. символы из $@Source_1$ и $(Program)$ для $@Target_1 = \mathrm{Goto}(@Source_1)\, \#/$

$, msgAbc \neq \ominus$

// Строка метки ещё не закончилась – надо получить для сравнения с её текущим символом очередной символ из $Program$

$msgBuff_1 = msgAbc$ // Здесь будет храниться очередной символ из строки метки $@Source_1$

$Continue_0 = /\#$ След. символы из $(@Source_1)$ и $Program$ для $@Target_1 = \mathrm{Goto}(@Source_1)\, \#/$

$SendTo_0\, Program\, (/\#$ В процессор из теневой очереди $\#/, \ominus)$

Следующие обработчики активируются из-за прихода очередного символа из Program. При этом в регистре $msgBuff_1$ уже находится текущий (не пустой) символ луча данных $@Source_1$. Поступивший из $Program$ символ надо сравнивать (равен ли) с символом из $@Source_1$ (он в $msgBuff_1$ сейчас):

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ След. символы из $(@Source_1)$ и $Program$ для $@Target_1 = \mathrm{Goto}(@Source_1)\, \#/$

$, msgAbc = \ominus$

// Процесс сравнения завершён с отрицательным результатом – строка в $Program$ закончилась, а строка метки $@Source_1$ целиком в $Program$ не нашлась.

$SendTo_0\, @Target_1\, (/\#$ Очистить rv здесь и дальше $\#/, \ominus)$

$wait$ // Делаем минимальную паузу, которая гарантировано отделяет момент отправки следующих сообщений от сообщений выше – чтобы они все не ушли одновременно

$SendTo_0\, @Target_1\, (/\#$ Присвоить rv значение$)\, \#/, 0)$

$wait$ // Ждём, чтобы сообщение попало на луч данных

$wait_0$ // Дожидаемся оперативного завершения присваивания нуля в $@Target_1$

// Этот обработчик является одним из вариантов завершения при вычислении $@Target_1 = \mathrm{Goto}(@Source_1)$. Теперь происходит возврат процессора в режим исполнения программы аппаратного языка стандартной интерпретации.

$Setmode\, SI$

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ След. символы из $(@Source_1)$ и $Program$ для $@Target_1 = \mathrm{Goto}(@Source_1)\, \#/$

$, msgAbc \neq \ominus$

$, msgBuff_1 \neq msgAbc$

// Необходимо вернуться к поиску начала $@Source_1$ в $Program$ с текущей позиции, так как текущий символ $@Source_1$ (не первый) не равен ему.

$Continue_0 = /\#$ 1й символ $@Source_1$ для $@Target_1 = \mathrm{Goto}(@Source_1)\, \#/$

$msgBuff_2 = msgAbc$ // Сохраняем очередной символ из программного текста

$SendTo_0\, @Target_1\, (/\#$ Увеличить на $1\, \#/, \ominus)$ // Совпали символы или нет, но по лучу данных Program мы все равно движемся и считаем номер символа от его начала

$SendTo_0\, @Source_1\, (/\#$ Отбросить тень $\#/, \ominus)$

$wait$ // Делаем минимальную паузу, которая гарантировано отделяет момент отправки следующих сообщений от сообщений выше – чтобы они все не ушли одновременно

$SendTo_0\, @Source_1\, (/\#$ В процессор из теневой очереди $\#/, \ominus)$ // Вообще-то для корректного получения символов из очереди надо отправлять сообщения в 2-моменты, но сразу после отбрасывания тени имеем ситуацию оперативного завершения и первый запрос на продвижение очереди можно делать через 1 момент после ухода сообщения об отбрасывания тени.

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ След. символы из $(@Source_1)$ и $Program$ для $@Target_1 = \mathrm{Goto}(@Source_1)\, \#/$

$, msgAbc \neq \ominus$

$, msgBuff_1 = msgAbc$

// Процесс сравнения очередной пары символов из $@Source_1$ и $Program$ будет продолжен, так как в текущей паре символы оказались равными. Данный обработчик вместе с 2-м обработчиком данного пункта (3) представляет собой 2-й подцикл из 2-х.

$Continue_0 = /\#$ След. символы из $@Source_1$ и $(Program)$ для $@Target_1 = \mathrm{Goto}(@Source_1)\, \#/$

$msgBuff_2 = msgAbc$ // Сохраняем очередной символ из программного текста

$SendTo_0\,@Target_1\,(/\#$ Увеличить на $1\,\#/, \ominus)$

$SendTo_0\,@Source_1\,(/\#$ В процессор из теневой очереди $\#/, \ominus)$ // Вообще-то для корректного получения символов из очереди надо отправлять сообщения в 2-моменты, но один момент уже прошёл, пока получали очередной символ из программного текста, поэтому можем отправлять сообщение в следующий ближайший 1-момент.

IV. Обработчики событий процессора – установка указателя p_1 и аналогичные операции на луче данных $@Target_1$ на основании числа, записанного на луче данных $@Source_1$

IV.1. Типы сообщений в пакете данных для установки p_1 и аналогичных операций

Сначала разберём, как готовятся и отправляются пакеты данных из процессора, а в следующем подразделе будут протоколы передачи этих пакетов по лучам данных.

Ещё до начала работы разбираемого ниже протокола необходимо, чтобы регистрам процессора $msgBegType_0$ и $msgLastType_0$ были присвоены типы первого и последнего сообщений пакета. Все сообщения между крайними (не включая крайних) в данном протоколе имеют всегда один и тот же тип /# Очередной разряд уменьшаемого числа (p_1) #/,поэтому для него регистр не требуется.

В качестве типов первого и последнего сообщения в пакете данных могут быть следующие числа:

1. /# Младший разряд уменьшаемого числа (p_1) #/, /# Разряды для p_1 исчерпаны #/;
2. /# Младший разряд уменьшаемого числа (p_1) #/, /# Разряды для количества символов к отправке
3. /# Младший разряд уменьшаемого числа (p_1) #/, /# Разряды для уменьшаемого числа исчерпаны
4. /# Младший разряд числа к уменьшению от p_1 #/, /# Разряды для количества символов к отправке

Для напоминания – типом остальных сообщений в разбираемых пакетах данных является число:

/# Очередной разряд уменьшаемого числа (p_1) #/

К пункту 1. В первом случае пакет данных по мере своего движения от процессора передаёт от ячейки к ячейке уменьшаемое на 1 (в процессе передачи) число для того, чтобы установить указатель p_1 в той ячейке, в которой уменьшаемое число сократиться до 1, либо в первой пустой ячейке – если она встретиться раньше.

К пункту 2. Во втором случае пакет данных по мере своего движения от ячейки-указателя (p_1) в направлении прочь от процессора, отправляет символы к началу луча данных. И эти символы выстраиваются в начале луча данных в теневую строку. А число, которое передаётся в пакете данных, уменьшается на 1 при каждой передачи символа в начало строки. До тех пор, пока число не станет равным 1 (тогда отправляется последний символ), либо не будет достигнут конец строки. Тогда передача в начало луча данных прекращается, а в процессор отправляется сообщение, что нужная теневая строка построена.

К пункту 3. Чтобы не усложнять протокол передачи данных, уменьшение числа делается «почти» независимо ни от чего. А именно – происходит уменьшение цифры на 1 (для цифр от 1 до 9) или замена цифры 0 на 9 с дальнейшим уменьшением следующего (старшего) разряда. Если же старшего разряда не оказывается, а вместо него – ноль или поступает сообщение об окончании блока цифр (последний тип сообщения из пп.1 и 2), то тип этого сообщения меняется на последний тип из п.3 и отправляется дальше – «в бесконечность».

Но отработка с вычитанием от передаваемого числа продолжается и дальше. И тогда пакет данных с таким заключительным сообщением ничего не меняет и можно считать, что весь луч данных «мгновенно» остался неподверженным изменениям от дальнейшего движения пакета данных с таким заключительным сообщением.

Тут мы используем тот же прием, что и при «мгновенной» очистке всех символов на луче данных в 1-м блоке обработчиков событий ячеек. При желании можно продумать «догоняющий сигнал», который движется с тактом времени величиной 1/4-момента, например, который постепенно «обгоняет» пакет данных с «перебором» и прекращает его дальнейшее движение.

Впрочем, такой «полицейский» сигнал данных ничего не ускорит, и не изменит в результатах работы алгоритмов. Но может успокоить в плане предотвращения бесконечного расхода «электроэнергии», что едва ли важно для бесконечной математической модели, с учётом и без того бесконечного ресурса, «использованного» для бесконечных лучей данных.

Но, кстати, если строка на луче данных не является бесконечной (то есть – найдётся ячейка с регистром $rv = \ominus$), то пакет данных прекратит своё движение в первой же «пустой» ячейке, как мы скоро увидим.

К пункту 4. В четвертом случае пакет данных движется от процессора к ячейке-указателю (где $p_1 = 1$) в неизменном виде (пусть на луче данных x_1). И только на ячейке-указателе пакет данных преобразуется в пакет данных из пункта 2, и начинает отправлять нужное количество символов (пусть j), начиная с ячейки-указателя (пусть её номер i). Этим в итоге достигается построение теневой строки $\mathrm{str}(x_1, i, j)$ в начале луча данных x_1.

Впрочем, если до ячейки с $p_1 = 1$ сообщению с типом /# Младший разряд числа к уменьшению от p_1 #/ встретиться ячейка с пустым символом $\ominus$, то в качестве результата работы пакета будет построена пустая теневая строка.

Разумеется, процессор может отправлять только пакеты данных, соответствующие 1 и 4 пунктам. А пакеты данных пунктов 3 и 4 являются производными, и возникают из первых двух вариантов в процессе движения пакета по лучу данных.

IV.2. Начать установку p_1 (первого указателя) или чего-то аналогичного на луче данных $@Target_1$ по номеру из луча $@Source_1$.

$(/\#$ Сделать указатель p_1 в $@Target_1$ из $@Source_1$ или подобное $\#/, \ominus)$

// Это сообщение процессор отправляет сам себе, и реагирует на его получение, а затем использует заранее подготовленные регистры $Source_1$ и $Target_1$. Но по факту пакет данных, который начинает формироваться в этот момент, может быть не только пакетом установки p_1. Свойства пакета определяются значениями регистров $msgBegType_0$ и $msgLastType_0$.

$Continue_0 = /\#$ Начать делать указатель p_1 в $@Target_1$ из $@Source_1$ или подобное $\#/$

$SendTo_0\, @Source_1\, (/\#$ Отбросить тень с уведомлением $\#/, \ominus)$

Дальше идут 2 немного разных обработчика, которые в итоге генерируют одно и то же сообщение, но для случая построения теневой строки есть одно дополнение в сравнении со случаем установки указателя. А именно – очистка теневой строки перед тем, как её строить.

Первый обработчик – проще. Случай установки указателя:

$(/\#$ Теневая строка построена $\#/, \ominus)$

$, Continue_0 = /\#$ Начать делать указатель p_1 в $@Target_1$ из $@Source_1$ или подобное $\#/$

$, msgLastType_0 = /\#$ Разряды для p_1 исчерпаны $\#/$

// Получение младшего разряда из теневой очереди с именем из $Source_1$ для установки указателя p_1 на луче с именем из $Target_1$:

$Continue_0 = /\#$ Младший разряд уменьшаемого числа в $@Target_1$ из $@Source_1\,(p_1)\,\#/$

$SendTo_0\,@Source_1\,(/\#$ В процессор из теневой очереди $\#/, \ominus)$

Второй обработчик (случай построения теневой строки) – включает ещё предварительную очистку теневой строки:

$(/\#$ Теневая строка построена $\#/, \ominus)$

$, Continue_0 = /\#$ Начать делать указатель p_1 в $@Target_1$ из $@Source_1$ или подобное $\#/$

$, msgLastType_0 = /\#$ Разряды для количества символов к отправке от p_1 исчерпаны $\#/$

// Получение младшего разряда из теневой очереди с именем из $Source_1$ для передачи в теневую строку на луче данных с именем из $Target_1$ нужного количества символов от указателя включительно:

$SendTo_0\,@Target_1\,(/\#$ Очистить sv здесь и дальше $\#/, \ominus)$

$Continue_0 = /\#$ Младший разряд уменьшаемого числа в $@Target_1$ из $@Source_1\,(p_1)\,\#/$

$SendTo_0\,@Source_1\,(/\#$ В процессор из теневой очереди $\#/, \ominus)$

IV.3.Отправить младший разряд p_1 или что-то аналогичного для луча данных $@Target_1$ из луча данных $@Source_1$.

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ Младший разряд уменьшаемого числа в $@Target_1$ из $@Source_1\,(p_1)\,\#/$

$, \mathrm{NotNum}(msgAbc)$

// Аварийное завершение программы. Не цифра вместо младшего разряда

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ Младший разряд уменьшаемого числа в $@Target_1$ из $@Source_1\,(p_1)\,\#/$

$, \mathrm{IsNum}(msgAbc)$

// Мы допускаем, что отправляемое в $@Target_1$ число окажется нулем. Потому что в теории компьютерных строк истинно $\mathrm{str}(x, 0, i) = \mathrm{str}(x, i, 0) = \ominus$. И для модели исполнения алгоритмов мы хотим добиться того же. Поэтому нет проверки, предотвращающей отправку нуля.

$Continue_0 = /\#$ Очередной разряд уменьшаемого числа в $@Target_1$ из $@Source_1\,(p_1)\,\#/$

$msgBuff_1 = 1$ // Тут мы будем помнить последнюю цифру. Это на случай, если цифра окажется ведущим нулём. Тогда строка в $@Source_1$ – не число, так как ведущие нули запрещены. А младший разряд допускает любое число, поэтому присвоим пока 1 (не ноль).

$SendTo_0\,@Target_1\,(msgBegType_0, msgAbc)$

$SendTo_0\,@Source_1\,(/\#$ В процессор из теневой очереди $\#/, \ominus)$

IV.4. Отправить очередной или последний разряд для установки p_1 или чего-то аналогичного на луче данных $@Target_1$ из луча данных $@Source_1$.

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ Очередной разряд уменьшаемого числа в $@Target_1$ из $@Source_1\,(p_1)\,\#/$

$, \mathrm{NotNum}(msgAbc)$

$, msgAbc \neq \ominus$

// Ошибка, аварийное завершение программы. В качестве очередного разряда – не цифра и не пустая ячейка

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ Очередной разряд уменьшаемого числа в $@Target_1$ из $@Source_1\,(p_1)\,\#/$

$, msgAbc = \ominus$

$, msgBuff_1 = 0$

// Аварийное завершение – у числа не может быть ведущих нулей

Теперь осталось рассмотреть только 3 обработчика. Последним рассмотрим тот, который отправляет «внутренние» сообщения пакета (не 1-е и не последнее). А вот что касается 2 обработчиков для отправки последних сообщений, то между ними есть разница:

Первый вариант последнего сообщения – завершает пакет данных для установки указателя p_1 и это видно по условию заголовка в обработчике:

$, msgLastType_0 = /\#$ Разряды для p_1 исчерпаны $\#/$

Такой пакет данных относится к числу «отправил и забыл», потому что мы можем сразу отправлять со стандартной скоростью 1 ячейка за 1 момент сообщения, использующие установленный указатель p_1. Потому что эти «следующие сообщения» всё равно придут позже того момента, когда наш пакет данных установит указатель p_1 в нужной ячейке. Поэтому сразу (почти сразу) после отправки последнего сообщения пакета по лучу данных происходит возврат процессора в режим SI командой

$Setmode\,SI$

Второй вариант последнего сообщения – завершает пакет данных для получения подстроки указанного размера начиная от ячейки с указателем. Притом подстрока будет получена – в качестве теневой строки в начале луча данных. И это видно по условию заголовка в обработчике:

$, msgLastType_0 = /\#$ Разряды для количества символов к отправке от p_1 исчерпаны $\#/$

Теневая строка – это лишь полуфабрикат для дальнейшего использования, а использовать теневую строку для её отправки в нужное место мы будем в качестве очереди. И, как мы уже разбирали в протоколах типа /# Отбросить тень и уведомить #/ и /# Отбросить тень #/, для обеспечения стандартной скорости работы с очередью нам потребуется не оперативное завершение при её построении, а полное завершение построения.

И для уведомления о полном завершении построения теневой подстроки в дальнейшем протоколе для луча данных мы используем сообщение к процессору:

$(/\#$ Теневая строка построена $\#/, \ominus)$

А в процессоре мы заранее установим регистр так же, как для завершения протокола сообщения /# Отбросить тень и уведомить #/:

$Continue_0 = /\#$ Скопировать теневую строку из основной $\#/$

То есть, процессор будет ждать во втором варианте прихода уведомления о завершении построения теневой строки. И это уведомление будет ему отправлено при исполнении протокола передачи сообщений с типом

/# Дополнить теневую строку #/

Такова общая картина, подробности которой мы увидим ниже. А пока возвращаемся к оставшимся трём обработчикам отправки пакета данных:

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ Очередной разряд уменьшаемого числа в $@Target_1$ из $@Source_1\,(p_1)\,\#/$

$, msgAbc = \ominus$

$, msgBuff_1 \neq 0$

$, msgLastType_0 = /\#$ Разряды для p_1 исчерпаны $\#/$

// Это первый вариант последнего сообщения пакета данных. Смысл его работы изложен выше, а вот программный текст:

// Мы ждём лишь оперативного завершения установки указателя:

$SendTo_0\,@Target_1\,(msgLastType_0, \ominus)$

$wait$

$wait_0$

// Теперь первая ячейка на луче данных отработана (последнее сообщение ушло и один момент на обработку первой ячейки подождали). Поэтому протокол оперативно завершён относительно процессора и можно возвращаться в режим SI

$Setmode\,SI$

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ Очередной разряд уменьшаемого числа в $@Target_1$ из $@Source_1\,(p_1)\,\#/$

$, msgAbc = \ominus$

$, msgBuff_1 \neq 0$

$, msgLastType_0 = /\#$ Разряды для количества символов к отправке от p_1 исчерпаны $\#/$

// Это второй вариант последнего сообщения пакета данных. Смысл его работы изложен выше, а вот программный текст:

// Процессор будет ждать сообщения $(/\#$ Теневая строка построена $\#/, \ominus)$ для возврата в режим SI. Поэтому процессор «становится в засаду» с нужным значением $Continue_0$:

$Continue_0 = /\#$ Скопировать теневую строку из основной $\#/$

$SendTo_0\,@Target_1\,(msgLastType_0, \ominus)$

// Напоминаю, что конец обработчика процессора зачастую работает как wait. Именно так работает и конец данного обработчика, так как в данном случае нет ничего, кроме инициированного данным обработчиком протокола передачи, а ответ в процессор придёт только в результате сообщения /# Теневая строка построена #/ из-за последнего отправленного сообщения.

Для обработки «ответного» сообщения /# Теневая строка построена #/ мы воспользуемся тем же обработчиком, который позволяет завершить построение тени и вернуться в режим SI процессора (напоминание):

$(/\#$ Теневая строка построена $\#/, \ominus)$

$, Continue_0 = /\#$ Скопировать теневую строку из основной $\#/$

$Setmode\,SI$

Остался последний обработчик – когда отправлять надо очередную цифру, а не завершать отправку пакета данных:

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ Очередной разряд уменьшаемого числа в $@Target_1$ из $@Source_1\,(p_1)\,\#/$

$, \mathrm{IsNum}(msgAbc)$

// Этот обработчик формирует и отправляет самое «типичное» сообщение пакета данных – его внутреннее сообщение (не первое и не последнее):

$msgBuff_1 = msgAbc$

$SendTo_0\,@Target_1\,(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

$SendTo_1\,@Source_1\,(/\#$ В процессор из теневой очереди $\#/, \ominus)$

V. Обработчики событий ячеек – установка указателя p_1 (или его аналогов) на луче данных, где расположены эти ячейки

V.1.Сообщения с символами для построения теневой подстроки данного луча данных

Сначала рассмотрим обработку вспомогательных сообщений, которые будут генерироваться пакетами данных, передающих уменьшаемое число.

(/# Дополнить теневую строку #/, $msgAbc$)

(/# Теневая строка построена #/, $\ominus$)

Данные второго сообщения мы разбирали в протоколах с сообщением /# Отбросить тень и уведомить #
и в начале протоколов увеличения/уменьшения числа на длину строки и расчёта результата команды goto.

Здесь мы воспользуемся тем же обработчиком, который позволяет завершить построение тени и вернуться в режим SI процессора:

(/# Теневая строка построена #/, $\ominus$)

, $Continue_0$ = /# Скопировать теневую строку из основной #/

$Setmode\, SI$

Как только сообщение об извлечении подстроки достигает ячейки с $p_1 = 1$ (если этот поиск не прервётся выходом на пустую ячейку), начинается отправка назад (если текущая ячейка дальше 1-й) символов из очередной достигнутой ячейки в начало луча данных, на уровень теневых значений ячеек – в результате чего в начале луча данных должна выстроиться нужная теневая подстрока.

Мы считаем, что все теневые символы до начала построения теневой строки очищены (пустые). Это должно быть сделано до отправки пакета данных, инициирующих возврат нужной теневой подстроки в начало луча данных.

Начинается переброска очередного символа в начало луча данных тем, что в ячейку приходит соответствующее сообщение

(/# Дополнить теневую строку #/, $msgAbc$)

из предыдущей ячейки, и данное сообщение передаётся по «эстафете» до нужного места в начало луча данных.

Подобные сообщения не добавляют теневые символы (кроме как в случае первой ячейки), а только ищут текущий конец теневой строки. И доходят в своём поиске до самой далекой от процессора ячейки со значимым (не пустым) теневым символом. А добавление нового теневого символа осуществляется уже отправкой сообщения в следующую от неё (с пустым теневым символом) ячейку:

$SendTo_{-2}\, NextCell$ (/# Присвоить sv значение #/, $msgAbc$)

Вот описание соответствующего протокола передачи.

(/# Дополнить теневую строку #/, $msgAbc$), $IsFirst = 1, sv = \ominus, msgAbc = \ominus$

// Записывать нечего – пустая строка видна по первому присланному символу. Пустая строка и стоит в теневом регистре. Но строка – построена и сообщение об этом пора отправлять:

$SendTo_{-2}\, Proc$ (/# Теневая строка построена #/, $\ominus$)

(/# Дополнить теневую строку #/, $msgAbc$), $IsFirst = 1, sv = \ominus, msgAbc \neq \ominus$

$sv = msgAbc$

(/# Дополнить теневую строку #/, $msgAbc$), $IsFirst = 0, sv = \ominus$

$SendTo_{-2}\, PreviousCell$ (/# Дополнить теневую строку #/, $msgAbc$)

(/# Дополнить теневую строку #/, $msgAbc$), $IsFirst = 0, sv \neq \ominus, msgAbc = \ominus$

// Сообщение прошло (двигаясь к процессору) ячейки с пустыми теневыми символами, и вышло в самую дальнюю от процессора ячейку с не пустым теневым символом. Протокол исполнен почти полностью – построенную теневую строку он больше не «потревожит» новыми сообщения, осталось лишь пройти теневую строку сообщением о завершении её построения. Относительное завершение для процессора было бы при $IsFirst = 1, sv \neq \ominus, msgAbc \neq \ominus$, если бы мы не отправляли уведомлений процессору о завершении. А мы – отправляем такое уведомление:

$SendTo_{-2}\, PreviosCell$ (/# Теневая строка построена #/, $\ominus$)

(/# Дополнить теневую строку #/, $msgAbc$), $IsFirst = 1, sv \neq \ominus, msgAbc = \ominus$

// То же, что предыдущий обработчик, но только сообщение отправляется непосредственно процессору из первой ячейки:

$SendTo_{-2}\, Proc$ (/# Теневая строка построена #/, $\ominus$)

(/# Дополнить теневую строку #/, $msgAbc$), $sv \neq \ominus, msgAbc \neq \ominus$

$SendTo_{-2}\, NextCell$ (/# Присвоить sv значение #/, $msgAbc$)

Не возникнет ли конфликтов между данными сообщениями? Сообщения с типом /# Дополнить теневую движутся к процессору, сообщения с типом /# Присвоить sv значение #/ движутся от процессора (хоть только на 1 ячейку) – могут ли они встретиться или сработать в ошибочном порядке?

Эти сообщения изначально отправляются ячейками, которые обрабатывают пакет данных по отправке символов в теневую строку. Эти пакеты движутся от процессора со скоростью 1 ячейка за 1 момент. Поэтому время между отправкой сообщений /# Дополнить теневую строку #/ от соседних ячеек тоже равно 1 моменту. Более того, следующее сообщение стартует с более далекой от процессора ячейки. Обозначим более раннее сообщение буквой A, а следующее за ним буквой B.

Сообщение A стартует из ячейки i в момент 1/4 – если считать за ноль момент поступления в ячейку сообщения, порождающего сообщение A («предок» сообщения A). В момент 1 «предок» сообщения B передаётся в ячейку $i + 1$. В момент 5/4 из этой ячейки стартует сообщение B, и сразу оказывается (так мы условились считать) в ячейке i. И оно отправляется «по эстафете» из ячейки i в момент 6/4. Разница во времени с сообщением А равна:

$6/4 - 1/4 = 5/4$

Но возьмём даже разницу в 1 момент для надёжности – иногда нам будет ясно, что в текущей ячейке находится последний не пустой символ для теневой строки, и мы будем отправлять его, а за ним – пустой символ. И оба сообщения будут уходить из одной и той же ячейки, но с промежутком в 1 момент.

Пусть сообщение A с типом /# Дополнить теневую строку #/ приходит в момент времени 0 (условимся взять этот момент за начало отсчёта) в самую близкую к процессору пустую по регистру sv ячейку $j + 1$ (куда надо будет записать соответствующий символ из сообщения A). Когда в эту

ячейку придёт сообщение типа /# Присвоить sv значение #/ с тем значением, которое принесло сообщение A?

Вот расчёт:

В момент 1/4 сообщение оказывается в самой дальней от процессора не пустой ячейке j. В момент 2/4 в ячейку $j+1$ приходит из ячейки j сообщение с типом /# Присвоить sv значение #/. В момент ранее 3/4 в ячейке $j+1$ завершается обработка данного сообщения.

Сообщение B поступит в ячейку $j+1$ только в момент 1 в самом «быстром» случае. Поэтому никаких конфликтов не возникает. И ещё больше времени будет между записью в ячейку значения из сообщения A и «заглядыванием» в ту же ячейку сообщения B, если эта ячейка – первая на луче данных.

Единственный нюанс, если ячейка i, из которой происходит отправка символа в теневую строку, совпадает с ячейкой, в которую в итоге надо будет записать этот теневой символ. Конфликта тоже нет – потому что в случае самой «близкой» отправки сообщения B после A их разделяет 1 момент. И сообщение В ждет этого момента в буфере отправки ячейки i до момента 1. А когда приходит этот момент – символ из сообщения A уже записан в качестве теневого символа ячейки i в момент 3/4. И конфликта сообщений не возникает – из-за разделяющих их 1/4 момента.

Протокол передачи сообщений, передающих символы для построения теневой строки, завершается отправкой сообщения с типом /# Теневая строка построена #/. Протокол передачи «по эстафете» к процессору сообщений данного типа был приведён в конце параграфа 2.2 текущего раздела.

V.2. Пустая ячейка для пакета данных установки p_1 и аналогичных.

Понятно, что первая пустая ячейка – она же и последняя, потому что дальше сообщения установки указателя или получения символов не идут.

1. Первая ячейка с пустым символом – обработчики начала и середины пакета данных

Единственная особенность пакета, передающего уменьшаемое число, связана с первой ячейкой – для неё может быть ещё вариант равенства передаваемого числа нулю. В другие ячейки такое число просто не пройдёт. А пока что – в первой ячейке – нам нужны обработчики, отличающие ноль от другого числа.

$(/\#$ Младший разряд уменьшаемого числа $(p_1)\ \#/, msgAbc), rv = \ominus, IsFirst = 1, msgAbc = 0$

$MayBe1 = 2$ // Только если число не равно 0, тогда 1-я ячейка с пустым символом укажет на себя (при установке p_1). Пока есть подозрение, что число равно 0 и укажет оно на «виртуальную» ячейку с номером ноль.

$(/\#$ Младший разряд уменьшаемого числа $(p_1)\ \#/, msgAbc), rv = \ominus, IsFirst = 1, msgAbc \neq 0$

$MayBe1 = 1$ // Число заведомо не ноль и если этот пакет устанавливает p_1, то будет $p_1 = 1$, так как это конец строки (пустой символ)

$(/\#$ Младший разряд числа к уменьшению от $p_1\ \#/, msgAbc), rv = \ominus, IsFirst = 1$

// Для передаваемого в неизменном виде числа, сообщения начала и середины пакета ничего не делают, так как дальше конца строки никакие сообщения не идут, а передавать в теневую строку

тут заведомо нечего.

$MayBe1 = 1$ // Но метку о возможной единице ставим – чтобы не обрабатывать вариант $MayBe1 = 0$ для середины пакета

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc), rv = \ominus, IsFirst = 1, MayBe1 = 0$

// Такая ситуация невозможна, потому что после младшего разряда у нас MayBe1 заведомо не ноль. Но ставлю тут заголовок этого обработчика для полноты перебора вариантов.

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc), rv = \ominus, IsFirst = 1, MayBe1 = 1$

// Не обрабатываем это сообщение и ждём следующее, потому что уже известно, что пакет данных передаёт не 0

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc), rv = \ominus, IsFirst = 1, MayBe1 = 2, msgAbc = 0$

// Не обрабатываем это сообщение, потому что пока сохраняется подозрение, что пакет данных передаёт ноль.

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc), rv = \ominus, IsFirst = 1, MayBe1 = 2, msgAbc \neq 0$

$MayBe1 = 1$ // Число заведомо не ноль и если этот пакет устанавливает p_1, то будет $p_1 = 1$

2. Не первая ячейка с пустым символом – обработчики начала и середина пакета данных

Но для не-первой ячейки игнорируются все сообщения, кроме последних сообщений в пакете данных при таком контексте:

$(/\#$ Младший разряд уменьшаемого числа $\#/, msgAbc), rv = \ominus, IsFirst = 0$

$(/\#$ Младший разряд числа к уменьшению от p_1 $\#/, msgAbc), rv = \ominus, IsFirst = 0$

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc), rv = \ominus, IsFirst = 0$

Игнорируются вышеназванные сообщения потому, что они все передают не нулевое число – иначе число не прошло бы первую ячейку, не став «холостым» – что будет видно в дальнейшем. Поэтому все числа в данном контексте аналогичны единице – если они не «холостые».

3. Первая ячейка с пустым символом – обработчики последнего сообщения пакета данных

$(/\#$ Разряды для p_1 исчерпаны $\#/, \ominus), rv = \ominus, IsFirst = 1, MayBe1 = 2$

$p_1 = 2$ // Число в пакете данных равно 0, а значит, указатель соответствует нулевому положению в строке и виртуальной «ячейке» с номером 0 «перед» текущей ячейкой с номером 1.

$MayBe1 = 0$

$(/\#$ Разряды для p_1 исчерпаны $\#/, \ominus), rv = \ominus, IsFirst = 1, MayBe1 = 1$

$p_1 = 1$ // Число в пакете данных не равно 0, а значит, указатель p_1 должен указать на текущую ячейку

$MayBe1 = 0$

$(/\#$ Разряды для p_1 исчерпаны $\#/, \ominus), rv = \ominus, IsFirst = 1, MayBe1 = 0$

// Это никак не обрабатывается, так как условия этого обработчика никогда не возникают – число признаётся или единицей ($p_1 = 1$), или нулём ($p_1 = 2$). Но дальше пустой ячейки сообщение

не идёт ($p_1 = 0$).

(/# Разряды для количества символов к отправке от p_1 исчерпаны $\#/, \ominus), rv = \ominus, IsFirst = 1$

// Нет смысла дальше искать указатель – строка закончилась, не начавшись. Осталось только сообщить о завершении построения теневой строки.

$MayBe1 = 0$

$SendTo_{-2}\, Proc$ (/# Теневая строка построена $\#/, \ominus$)

(/# Разряды для количества символов к отправке исчерпаны (отправлять) $\#/, \ominus$)

$, rv = \ominus, IsFirst = 1$

// Этот случай никогда не обрабатывается, и не возникает. Потому что в 1ю ячейку может прийти вместо данного сообщения только сообщение с типом /# Разряды для количества символов к отправке о
А уже это сообщение в дальнейшем может породить и тип /# Разряды для количества символов к отправке

4. Не первая ячейка с пустым символом – обработчики окончания пакета данных

(/# Разряды для p_1 исчерпаны $\#/, \ominus), rv = \ominus, IsFirst = 0$

// Для любого числа больше нуля данная ячейка становится указателем – дальше идти уже некуда. А передающий ноль пакет с сообщением такого типа в конце – не может придти в непервую ячейку луча данных, так как его не пропустят предыдущие ячейки.

$p_1 = 1$

(/# Разряды для количества символов к отправке от p_1 исчерпаны $\#/, \ominus), rv = \ominus, IsFirst = 0$

$MayBe1 = 0$

$SendTo_{-2}\, PreviousCell$ (/# Теневая строка построена $\#/, \ominus$)

(/# Разряды для количества символов к отправке исчерпаны (отправлять) $\#/, \ominus$)

$, rv = \ominus, IsFirst = 0$

$MayBe1 = 0$

$SendTo_{-2}\, PreviousCell$ (/# Теневая строка построена $\#/, \ominus$)

Но есть случай, когда будет проигнорировано даже последнее сообщение в пакете данных:

(/# Разряды для уменьшаемого числа исчерпаны с перебором $\#/, \ominus), rv = \ominus$

// Ничего не делать. Холостой «полёт» пакета данных с перебором – закончен.

Теперь перейдём к рассмотрению обработчиков событий при поступлении рассматриваемых сейчас сообщений в не пустые ячейки луча данных.

V.3. Сообщение поступает в первую ячейку с не пустым символом

Первая ячейка луча данных имеет свои особенности – в ней может быть установлен указатель для «виртуальной» нулевой ячейки ($p_1 = 2$) – если число в пакете данных для установки p_1 равно нулю. Поэтому именно для первой ячейки будет организована проверка передаваемого числа на равенство 0 и – если равенство нулю выполнено – то произойдёт ещё и смена статуса пакета, передающего число, на «холостой». Дальше он будет двигаться от процессора, но ничего менять уже не будет.

1. Обработчики начала пакета данных (1-я ячейка с не пустым символом)

На этапе получения младшего разряда ячейка начинает отнимать единицу от всего передаваемого числа – если тип сообщения требует именно этого:

$(/\#$ Младший разряд уменьшаемого числа $(p_1)\ \#/, msgAbc)$

$, rv \neq \ominus, IsFirst = 1, msgAbc = 0$

$dc = 1$ // Тип сообщения и младший разряд требуют продолжить отнимать 1 от след. разряда

$MayBe1 = 2$ // Пакет данных, возможно, передаёт ноль

$SendTo_0\, NextCell\, (/\#$ Младший разряд уменьшаемого числа $(p_1)\ \#/, 9)$

$(/\#$ Младший разряд уменьшаемого числа $(p_1)\ \#/, msgAbc), rv \neq \ominus, IsFirst = 1, msgAbc = 1$

$dc = 0$

$MayBe1 = 1$ // Не исключено, что пакет данных передаёт единицу

$SendTo_0\, NextCell\, (/\#$ Младший разряд уменьшаемого числа $(p_1)\ \#/, 0)$

$(/\#$ Младший разряд уменьшаемого числа $(p_1)\ \#/, msgAbc), rv \neq \ominus, IsFirst = 1, msgAbc > 1$

$dc = 0$

$MayBe1 = 0$ // Пакет данных передаёт число заведомо больше 1

$SendTo_0\, NextCell\, (/\#$ Младший разряд уменьшаемого числа $(p_1)\ \#/, msgAbc - 1)$

$(/\#$ Младший разряд числа к уменьшению от $p_1\ \#/, msgAbc)$

$, rv \neq \ominus, IsFirst = 1, p_1 = 2$

// Самый простой случай, когда подстрока берется как $\mathrm{str}(x, 0, i)$ – с нулевой позиции. А это всегда $\ominus$. Но нам надо отправить пакет данных дальше, сменив его статус на «холостой» в последнем сообщении.

$dc = 0$ // Пакеты с таким типом начального сообщением передают числа без вычитания, кроме случая $p_1 = 1$

$MayBe1 = 0$ // Длину подстроки можно считать какой угодно, так как будет холостой пакет

$SendTo_0\, NextCell\, (/\#$ Младший разряд числа к уменьшению от $p_1\ \#/, msgAbc)$

$(/\#$ Младший разряд числа к уменьшению от $p_1\ \#/, msgAbc)$

$, rv \neq \ominus, IsFirst = 1, p_1 = 0, msgAbc = 0$

// Простой случай начального сообщения в пакете запроса подстроки. Нюанс – когда размер подстроки задан как ноль.

$dc = 0$

$MayBe1 = 2$ // Возможно, пакет данных передаёт ноль

$SendTo_0\, NextCell\, (/\#$ Младший разряд числа к уменьшению от $p_1\ \#/, msgAbc)$

$(/\#$ Младший разряд числа к уменьшению от $p_1\ \#/, msgAbc)$

$, rv \neq \ominus, IsFirst = 1, p_1 = 0, msgAbc \neq 0$

// Простой случай начального сообщения в пакете запроса подстроки. К тому же размер подстроки здесь явно не ноль.

$dc = 0$

$MayBe1 = 0$ // Длина подстроки – больше 0, а вопрос про равенство 1 пока $(p_1 = 0)$ не важен

$SendTo_0\, NextCell\, (/\#$ Младший разряд числа к уменьшению от $p_1\ \#/, msgAbc)$

$(/\#$ Младший разряд числа к уменьшению от $p_1\ \#/, msgAbc)$

$, rv \neq \ominus, IsFirst = 1, p_1 = 1, msgAbc = 0$

// Начальное сообщения в пакете запроса подстроки достигло начала подстроки. Пора начинать отправлять символы для построения теневой подстроки в начале луча данных. Если размер подстроки – не ноль. Это и надо проверить.

$dc = 1$

$MayBe1 = 2$ // Возможно, пакет данных передаёт ноль

$SendTo_0\, NextCell\, (/\#$ Младший разряд уменьшаемого числа $(p_1)\, \#/, 9)$

$(/\#$ Младший разряд числа к уменьшению от $p_1\, \#/, msgAbc)$

$, rv \neq \ominus, IsFirst = 1, p_1 = 1, msgAbc = 1$

// Начальное сообщения в пакете запроса подстроки достигло начала подстроки. Пора начинать отправлять символы для построения теневой подстроки в начале луча данных.

$dc = 0$

$MayBe1 = 1$ // Возможно, пакет данных передаёт единицу

$SendTo_0\, NextCell\, (/\#$ Младший разряд уменьшаемого числа $(p_1)\, \#/, 0)$

$(/\#$ Младший разряд числа к уменьшению от $p_1\, \#/, msgAbc)$

$, rv \neq \ominus, IsFirst = 1, p_1 = 1, msgAbc > 1$

// Начальное сообщения в пакете запроса подстроки достигло начала подстроки. Пора начинать отправлять символы для построения теневой подстроки в начале луча данных.

$dc = 0$

$MayBe1 = 0$ // Пакет данных передаёт число, большее 1

$SendTo_0\, NextCell\, (/\#$ Младший разряд уменьшаемого числа $\#/, msgAbc - 1)$

2. Обработчики середины пакета данных (1-я ячейка с не пустым символом)

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

$, rv \neq \ominus, IsFirst = 1, MayBe1 = 2, msgAbc = 0, dc = 0$

// Хоть $MayBe1 = 2$, но может быть $dc = 0$, если передаётся запрос на получение подстроки.

$SendTo_0\, NextCell\, (/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

$, rv \neq \ominus, IsFirst = 1, MayBe1 = 2, msgAbc = 0, dc = 1$

$SendTo_0\, NextCell\, (/\#$ Очередной разряд уменьшаемого числа $\#/, 9)$

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

$, rv \neq \ominus, IsFirst = 1, MayBe1 = 2, msgAbc \neq 0, dc = 0$

// Хоть $MayBe1 = 2$, но может быть $dc = 0$, если передаётся запрос на получение подстроки

$MayBe1 = 0$

$SendTo_0\, NextCell\, (/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

$, rv \neq \ominus, IsFirst = 1, MayBe1 = 2, msgAbc \neq 0, dc = 1$

$dc = 0$

$MayBe1 = 0$

$SendTo_0\, NextCell\, (/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc - 1)$

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

$, rv \neq \ominus, IsFirst = 1, MayBe1 = 1, msgAbc = 0$

// Раз $MayBe1 = 1$, то $dc = 0$ в любом случае. И это пока сохраняется, так как очередная цифра – ноль.

$SendTo_0\, NextCell\, (/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

$, rv \neq \ominus, IsFirst = 1, MayBe1 = 1, msgAbc \neq 0$

// Раз $MayBe1 = 1$, то $dc = 0$ в любом случае. Но число в пакете данных – не 1 из-за не нулевого и не самого младшего разряда

$MayBe1 = 0$

$SendTo_0\, NextCell\, (/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

$, rv \neq \ominus, IsFirst = 1, MayBe1 = 0$

// Раз $MayBe1 = 0$, то $dc = 0$ в любом случае. Просто передаём сообщение дальше

$SendTo_0\, NextCell\, (/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

3. Обработчики последнего сообщения пакета данных (1-я ячейка с не пустым символом)

$(/\#$ Разряды для p_1 исчерпаны $\#/, \ominus), rv \neq \ominus, IsFirst = 1, MayBe1 = 1$

$p_1 = 1$ // Пакет данных передаёт 1, значит – устанавливаем указатель для 1-й ячейки.

$MayBe1 = 0$

// Указатель в 1-й ячейке установили, дальше пакет данных не должен ничего делать, но он ушёл почти весь. Последним сообщением пакета данных делаем его статус «холостым». Дальше пакет данных ничего не изменит:

$SendTo_0\, NextCell\, (/\#$ Разряды для уменьшаемого числа исчерпаны с перебором $(p_1)\, \#/, \ominus)$

$(/\#$ Разряды для p_1 исчерпаны $\#/, \ominus), rv \neq \ominus, IsFirst = 1, MayBe1 = 2$

$p_1 = 2$ // Пакет данных передаёт 0, значит – устанавливаем указатель для 0-й ячейки, но – в 1ой ячейке, так как 0-ая ячейка «виртуальная».

$MayBe1 = 0$

$dc = 0$

// Указатель в 1-й ячейке установили, дальше пакет данных не должен ничего делать, но он ушёл почти весь. Последним сообщением пакета данных делаем его статус «холостым». Дальше пакет данных ничего не изменит:

$SendTo_0\, NextCell\, (/\#$ Разряды для уменьшаемого числа исчерпаны с перебором $(p_1)\, \#/, \ominus)$

$(/\#$ Разряды для p_1 исчерпаны $\#/, \ominus), rv \neq \ominus, IsFirst = 1, MayBe1 = 0$

$p_1 = 0$ // Пакет данных передаёт число, большее 1. Поэтому указатель будет не в первой и не для 0-й ячейке. Отправляем пакет дальше:

$SendTo_0\, NextCell\, (/\#$ Разряды для p_1 исчерпаны $\#/, \ominus)$

$(/\#$ Разряды для количества символов к отправке от p_1 исчерпаны $\#/, \ominus)$

$, rv \neq \ominus, IsFirst = 1, p_1 = 2$

// Отправить в теневую очередь символы с нулевой позиции ($p_1 = 2$) – это значит, сразу сообщить о построении пустой теневой очереди, так как $\mathrm{str}(x, 0, i) = \ominus$

$MayBe1 = 0$

$dc = 0$

$SendTo_{-2}\, Proc\, (/\#$ Теневая строка построена $\#/, \ominus)$

// Всё сделали, дальше пакет данных не должен ничего делать, но он ушёл почти весь. Последним сообщением пакета данных делаем его статус «холостым». Дальше пакет данных ничего не изменит:

$SendTo_0\, NextCell\, (/\#$ Разряды для уменьшаемого числа исчерпаны с перебором $(p_1)\, \#/, \ominus)$

$(/\#$ Разряды для количества символов к отправке от p_1 исчерпаны $\#/, \ominus)$

$, rv \neq \ominus, IsFirst = 1, MayBe1 = 2$

// Отправить в теневую очередь символы с какой-то позиции в количестве ноль штук ($MayBe1 = 2$) – это значит, сразу сообщить о построении пустой теневой очереди, так как $\mathrm{str}(x, i, 0) = \ominus$

$MayBe1 = 0$

$dc = 0$

$SendTo_{-2}\, Proc\, (/\#$ Теневая строка построена $\#/, \ominus)$

// Всё сделали, дальше пакет данных не должен ничего делать, но он ушёл почти весь. Последним сообщением пакета данных делаем его статус «холостым». Дальше пакет данных ничего не изменит:

$SendTo_0\, NextCell\, (/\#$ Разряды для уменьшаемого числа исчерпаны с перебором $(p_1)\, \#/, \ominus)$

$(/\#$ Разряды для количества символов к отправке от p_1 исчерпаны $\#/, \ominus)$

$, rv \neq \ominus, IsFirst = 1, p_1 = 1, MayBe1 = 1$

// Отправить в теневую очередь символы с 1 позиции ($p_1 = 1$) в количестве 1 штуки ($MayBe1 = 1$) – это значит, отправить символ только с 1-й позиции и сообщить после этого о построении всей теневой очереди.

$MayBe1 = 0$

$dc = 0$

$sv = rv$ // Построили всю теневую строку – её первый и единственный символ совпадает с первым символом «реальной» строки луча данных

$SendTo_{-2}\, Proc\, (/\#$ Теневая строка построена $\#/, \ominus)$

// Всё сделали, дальше пакет данных не должен ничего делать, но он уже ушёл «по эстафете» почти весь. Последним сообщением пакета данных делаем его статус «холостым». Дальше пакет данных ничего не изменит:

$SendTo_0\, NextCell\, (/\#$ Разряды для уменьшаемого числа исчерпаны с перебором $(p_1)\, \#/, \ominus)$

$(/\#$ Разряды для количества символов к отправке от p_1 исчерпаны $\#/, \ominus)$

$, rv \neq \ominus, IsFirst = 1, p_1 = 1, MayBe1 = 0$

// Отправляем в теневую очередь символы, начиная (прямо сейчас) с 1 позиции ($p_1 = 1$). А пакет пойдёт дальше «по эстафете», отправляя дальнейшие запрошенные символы.

$MayBe1 = 0$

$dc = 0$

$sv = rv$ // Записали 1й символ теневой строки – он совпал с первым символом «реальной» строки луча данных

// Передаём по эстафете пакет данных для построения теневой очереди далее:

$SendTo_0\, NextCell\, (/\#$ Разряды для количества символов к отправке исчерпаны (отправлять) $\#/, \ominus)$

$(/\#$ Разряды для количества символов к отправке от p_1 исчерпаны $\#/, \ominus)$

$, rv \neq \ominus, IsFirst = 1, p_1 = 0, MayBe1 \neq 2$

// Просто передаём запрос на не-нулевое количество символов дальше

$MayBe1 = 0$

$dc = 0$

$SendTo_0\, NextCell\, (/\#$ Разряды для количества символов к отправке от p_1 исчерпаны $\#/, \ominus)$

$(/\#$ Разряды для количества символов к отправке исчерпаны (отправлять) $\#/, \ominus), rv \neq \ominus, IsFirst = 1$

// Этот случай никогда не обрабатывается, и не возникает. Потому что в 1ю ячейку может прийти только сообщение с типом /# Разряды для количества символов к отправке от p_1 исчерпаны $\#/$. А уже это сообщение в дальнейшем может породить и тип /# Разряды для количества символов к отправке

$(/\#$ Разряды для уменьшаемого числа исчерпаны с перебором $(p_1)\, \#/, \ominus), rv \neq \ominus, IsFirst = 1$

// Этот случай никогда не обрабатывается, и не возникает. По аналогичным причинам, что и для предыдущего «обработчика»

V.4. Сообщение поступает в не первую ячейку с не пустым символом

1. Обработчики начала пакета данных (не 1-я ячейка с не пустым символом)

На этапе получения младшего разряда ячейка начинает отнимать единицу от всего передаваемого числа – если тип сообщения требует именно этого:

$(/\#$ Младший разряд уменьшаемого числа $(p_1)\, \#/, msgAbc)$

$, rv \neq \ominus, IsFirst \neq 1, msgAbc = 0$

$dc = 1$ // Тип сообщения и младший разряд требуют продолжить отнимать 1 от след. разряда.

$MayBe1 = 0$ // Число заведомо не 1 (msgAbc = 0) и больше нуля, иначе не прошло бы 1-ю ячейку.

$SendTo_0\, NextCell\, (/\#$ Младший разряд уменьшаемого числа $(p_1)\, \#/, 9)$

$(/\#$ Младший разряд уменьшаемого числа $(p_1)\, \#/, msgAbc)$

$, rv \neq \ominus, IsFirst \neq 1, msgAbc = 1$

$dc = 0$

$MayBe1 = 1$ // Не исключено, что пакет данных передаёт единицу

$SendTo_0\, NextCell\, (/\#$ Младший разряд уменьшаемого числа $(p_1)\, \#/, 0)$

$(/\#$ Младший разряд уменьшаемого числа $(p_1)\, \#/, msgAbc)$

$, rv \neq \ominus, IsFirst \neq 1, msgAbc > 1$

$dc = 0$

$MayBe1 = 0$ // Пакет данных передаёт число заведомо больше 1

$SendTo_0\, NextCell\, (/\#$ Младший разряд уменьшаемого числа $(p_1)\, \#/, msgAbc - 1)$

$(/\#$ Младший разряд числа к уменьшению от $p_1\ \#/, msgAbc), rv \neq \ominus, IsFirst \neq 1, p_1 = 2$

// Невозможный случай для не первой ячейки, чтобы $p_1 = 2$

$(/\#$ Младший разряд числа к уменьшению от $p_1\ \#/, msgAbc), rv \neq \ominus, IsFirst \neq 1, p_1 = 0$

$dc = 0$

$MayBe1 = 0$

// Указатель в данной ячейке не установлен, поэтому просто продолжаем эстафетную передачу.

$SendTo_0\ NextCell\ (/\#$ Младший разряд числа к уменьшению от $p_1\ \#/, msgAbc)$

$(/\#$ Младший разряд числа к уменьшению от $p_1\ \#/, msgAbc), rv \neq \ominus, IsFirst \neq 1, p_1 = 1, msgAbc = 0$

// Начальное сообщения в пакете запроса подстроки достигло начала подстроки. Пора начинать отправлять символы для построения теневой подстроки в начале луча данных. Размер подстроки – заведомо не ноль из-за $msgAbc = 0$ и того, что первая ячейка не пропускает ноль, не обозначив пакет как холостой.

$dc = 1$

$MayBe1 = 0$

$SendTo_0\ NextCell\ (/\#$ Младший разряд уменьшаемого числа $(p_1)\ \#/, 9)$

$(/\#$ Младший разряд числа к уменьшению от $p_1\ \#/, msgAbc), rv \neq \ominus, IsFirst \neq 1, p_1 = 1, msgAbc = 1$

// Начальное сообщения в пакете запроса подстроки достигло начала подстроки. Пора начинать отправлять символы для построения теневой подстроки в начале луча данных.

$dc = 0$

$MayBe1 = 1$ // Возможно, пакет данных передаёт единицу

$SendTo_0\ NextCell\ (/\#$ Младший разряд уменьшаемого числа $(p_1)\ \#/, 0)$

$(/\#$ Младший разряд числа к уменьшению от $p_1\ \#/, msgAbc), rv \neq \ominus, IsFirst \neq 1, p_1 = 1, msgAbc > 1$

// Начальное сообщения в пакете запроса подстроки достигло начала подстроки. Пора начинать отправлять символы для построения теневой подстроки в начале луча данных.

$dc = 0$

$MayBe1 = 0$ // Пакет данных передаёт число, большее 1

$SendTo_0\ NextCell\ (/\#$ Младший разряд уменьшаемого числа $(p_1)\ \#/, msgAbc - 1)$

2. Обработчики середина пакета данных (не 1-я ячейка с не пустым символом)

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc), rv \neq \ominus, IsFirst \neq 1, MayBe1 = 2$

// $MayBe1 = 2$ - невозможная ситуация для $IsFirst \neq 1$, так как число равное нулю в пакете данных не пропускает 1-я ячейка, не обозначив пакет как холостой

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc), rv \neq \ominus, IsFirst \neq 1, MayBe1 = 1, msgAbc = 0$

// Раз $MayBe1 = 1$, то $dc = 0$ в любом случае. И это пока сохраняется, так как очередная цифра – ноль.

$SendTo_0\ NextCell\ (/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc), rv \neq \ominus, IsFirst \neq 1, MayBe1 = 1, msgAbc \neq 0$

// Раз $MayBe1 = 1$, то $dc = 0$ в любом случае. Но число в пакете данных – не 1 из-за не нулевого и не самого младшего разряда

$MayBe1 = 0$

$SendTo_0\, NextCell\, (/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

$, rv \neq \ominus, IsFirst \neq 1, MayBe1 = 0, dc = 1, msgAbc = 0$

// Пока dc остаётся прежним и надо отнимать 1 от следующего разряда

$SendTo_0\, NextCell\, (/\#$ Очередной разряд уменьшаемого числа $\#/, 9)$

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

$, rv \neq \ominus, IsFirst \neq 1, MayBe1 = 0, dc = 1, msgAbc \neq 0$

$dc = 0$ // Дальше отнимать 1 не потребуется

$SendTo_0\, NextCell\, (/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc - 1)$

$(/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

$, rv \neq \ominus, IsFirst \neq 1, MayBe1 = 0, dc = 0$

$SendTo_0\, NextCell\, (/\#$ Очередной разряд уменьшаемого числа $\#/, msgAbc)$

3. Обработчики последнего сообщения пакета данных (не 1-я ячейка с не пустым символом)

$(/\#$ Разряды для p_1 исчерпаны $\#/, \ominus), rv \neq \ominus, IsFirst \neq 1, MayBe1 = 2$

// $MayBe1 = 2$ - невозможная ситуация для не первой ячейки, так как пакет данных с числом, равным нулю не пропустит 1-я ячейка, не сделав его холостым

$(/\#$ Разряды для p_1 исчерпаны $\#/, \ominus), rv \neq \ominus, IsFirst \neq 1, MayBe1 = 1$

$p_1 = 1$ // Пакет данных передаёт 1, значит – устанавливаем указатель для данной ячейки.

$MayBe1 = 0$

// Указатель в 1-й ячейке установили, дальше пакет данных не должен ничего делать, но он ушёл почти весь. Последним сообщением пакета данных делаем его статус «холостым». Дальше пакет данных ничего не изменит:

$SendTo_0\, NextCell\, (/\#$ Разряды для уменьшаемого числа исчерпаны с перебором $(p_1)\, \#/, \ominus)$

$(/\#$ Разряды для p_1 исчерпаны $\#/, \ominus), rv \neq \ominus, IsFirst \neq 1, MayBe1 = 0$

$p_1 = 0$ // Пакет данных передаёт число, большее 1. Поэтому указатель надо будет расположить дальше. И обязательно очистить все указатели до него – что и сделано в текущей ячейке. Отправляем пакет дальше:

$SendTo_0\, NextCell\, (/\#$ Разряды для p_1 исчерпаны $\#/, \ominus)$

$(/\#$ Разряды для количества символов к отправке от p_1 исчерпаны $\#/, \ominus)$

$, rv \neq \ominus, IsFirst \neq 1, MayBe1 = 2$

// $MayBe1 = 2$ - невозможная ситуация для не первой ячейки, так как пакет данных с числом, равным нулю, не пропустит 1-я ячейка, не сделав пакет холостым

$(/\#$ Разряды для количества символов к отправке от p_1 исчерпаны $\#/, \ominus)$

$, rv \neq \ominus, IsFirst \neq 1, p_1 = 2$

// $p_1 = 2$ - невозможная ситуация для не первой ячейки, так как пакет данных для установки p_1 с числом, равным нулю, не пропустит 1-я ячейка, не сделав пакет холостым

$(/\#$ Разряды для количества символов к отправке от p_1 исчерпаны $\#/, \ominus)$

$, rv \neq \ominus, IsFirst \neq 1, p_1 = 1, MayBe1 = 1$

// Отправить в теневую очередь символы с 1 позиции ($p_1 = 1$) в количестве 1 штуки ($MayBe1 = 1$) – это значит, отправить символ только с текущей позиции и сообщить после этого о построении всей теневой очереди.

$MayBe1 = 0$

$dc = 0$

$SendTo_{-2}\, PreviousCell\, (/\#$ Дополнить теневую строку $\#/, rv)$ // Это сообщение установит теневой символ в первой ячейке, притом двигаться будет только в направлении к процессору, не генерируя никаких сообщений от процессора

$wait$ // Задержка для предотвращения конфликта между сообщениями

$SendTo_{-2}\, PreviousCell\, (/\#$ Теневая строка построена $\#/, \ominus)$

// Всё сделали, дальше пакет данных не должен ничего делать, но он уже ушёл «по эстафете» почти весь. Последним сообщением пакета данных делаем его статус «холостым». Дальше пакет данных ничего не изменит:

$SendTo_0\, NextCell\, (/\#$ Разряды для уменьшаемого числа исчерпаны с перебором $(p_1)\, \#/, \ominus)$

$(/\#$ Разряды для количества символов к отправке от p_1 исчерпаны $\#/, \ominus)$

$, rv \neq \ominus, IsFirst \neq 1, p_1 = 1, MayBe1 = 0$

// Отправляем в теневую очередь символы, начиная (прямо сейчас) с 1 позиции ($p_1 = 1$). А пакет пойдёт дальше «по эстафете», отправляя дальнейшие запрошенные символы.

$MayBe1 = 0$

$dc = 0$

$SendTo_{-2}\, PreviousCell\, (/\#$ Дополнить теневую строку $\#/, rv)$

// Передаём по эстафете пакет данных для построения теневой очереди далее:

$SendTo_0\, NextCell\, (/\#$ Разряды для количества символов к отправке исчерпаны (отправлять) $\#/, \ominus)$

$(/\#$ Разряды для количества символов к отправке от p_1 исчерпаны $\#/, \ominus)$

$, rv \neq \ominus, IsFirst \neq 1, p_1 = 0$

// Просто передаём запрос на не-нулевое количество символов дальше

$MayBe1 = 0$

$dc = 0$

$SendTo_0\, NextCell\, (/\#$ Разряды для количества символов к отправке от p_1 исчерпаны $\#/, \ominus)$

$(/\#$ Разряды для количества символов к отправке исчерпаны (отправлять) $\#/, \ominus)$

$, rv \neq \ominus, IsFirst \neq 1, MayBe1 = 1$

// Заканчиваем отправлять в теневую очередь символы в данной ячейке, так как остался 1. Значит, надо отправить символ только с текущей позиции и сообщить после этого о построении всей теневой очереди.

$MayBe1 = 0$

$dc = 0$

$wait_0$ // Выравниваем момент отправки по сетке синхронизации 1-моментов

$SendTo_{-2}\, PreviousCell\, (/\#$ Дополнить теневую строку $\#/, rv)$

$wait_0$ // Задержка на 1 момент (после предыдущего выравнивания), для предотвращения конфликта между сообщениями

$SendTo_{-2}\, PreviousCell\, (/\#$ Теневая строка построена $\#/, \ominus)$

// Всё сделали, дальше пакет данных не должен ничего делать, но он уже ушёл «по эстафете» почти весь. Последним сообщением пакета данных делаем его статус «холостым». Дальше пакет данных ничего не изменит:

$SendTo_0\, NextCell\, (/\#$ Разряды для уменьшаемого числа исчерпаны с перебором $(p_1)\, \#/, \ominus)$

$(/\#$ Разряды для количества символов к отправке исчерпаны (отправлять) $\#/, \ominus)$

$, rv \neq \ominus, IsFirst \neq 1, MayBe1 = 0$

// Продолжаем отправлять в теневую очередь символы и из текущей ячейки. А пакет пойдёт дальше «по эстафете», отправляя дальнейшие запрошенные символы.

$MayBe1 = 0$

$dc = 0$

$SendTo_{-2}\, PreviousCell\, (/\#$ Дополнить теневую строку $\#/, rv)$

// Передаём по эстафете пакет данных для построения теневой очереди далее:

$SendTo_0\, NextCell\, (/\#$ Разряды для количества символов к отправке исчерпаны (отправлять) $\#/, \ominus)$

$(/\#$ Разряды для уменьшаемого числа исчерпаны с перебором $(p_1)\, \#/, \ominus).rv \neq \ominus, IsFirst \neq 1$

// Сообщение даёт тип «холостого» всему пакету. Просто отправляем сообщение дальше.

$SendTo_0\, NextCell\, (/\#$ Разряды для уменьшаемого числа исчерпаны с перебором $(p_1)\, \#/, \ominus)$

VI. Обработчики остальных простых событий – передача символов в ячейки и из ячеек от указателя включительно, установка указателя в первой пустой ячейке

VI.1. Запись символа в ячейку с указателем и смещение указателя на 1 ячейку дальше от процессора

Условимся, что при отсутствии указателя (ячейка с $p_1 \neq 0$) внутри луча данных, его роль выполняет ячейка с пустым символом, расположенная сразу за последним символом строки. Если строка пустая, то первую ячейку считаем ячейкой с указателем.

1) Запись пустого символа в ячейке с указателем

Сначала разберем запись пустого символа. Это особый случай, который нельзя реализовать посредством сообщений, записывающих остальные символы в ячейку с указателем. Пустой символ в ячейке делает эту ячейку позицией после окончания строки – если ячейки перед ней не пустые. Поэтому и указатель сдвигать дальше – нельзя. Ведь невозможны такие строки, где не на последнем месте находится пустой символ. Да и на последнем месте пустой символ может быть только в пустой строке, да и то условно, оставляя длину строки нулевой. Вот протокол передачи соответствующих сообщений:

(/# Очистить строку, начиная от p_1 #/, $\ominus$), $p_1 \neq 0$

$rv = \ominus$

$SendTo_0\, NextCell$ (/# Очистить rv здесь и дальше #/, $\ominus$)

(/# Очистить строку, начиная от p_1 #/, $\ominus$), $p_1 = 0, rv = \ominus$

$p_1 = 1$ // $p_1 \neq 0$ не был найден в пределах строки плюс один символ – это ошибка. Ставим его хотя бы сразу после последнего не пустого символа – это как если бы строка, начиная отсюда, была очищена из-за указателя.

$SendTo_0\, NextCell$ (/# Очистить rv здесь и дальше #/, $\ominus$)

(/# Очистить строку, начиная от p_1 #/, $\ominus$), $p_1 = 0, rv \neq \ominus$

$SendTo_0\, NextCell$ (/# Очистить строку, начиная от p_1 #/, $\ominus$)

Теперь – запись обычных (не пустых) символов:

2) Сообщения с символом, предназначенным для записи в ячейку с указателем и сдвига указателя.

(/# Записать символ в ячейке с указателем p_1 и сдвинуть указатель #/, $msgAbc$), $msgAbc = \ominus$

// Аварийное сообщение – пустой символ не может никуда записываться так, чтобы сдвинуть указатель

(/# Записать символ в ячейке с указателем p_1 и сдвинуть указатель #/, $msgAbc$)

, $msgAbc \neq \ominus, p_1 = 2$

// Мы в первой ячейке, но указатель указывает на нулевую виртуальную ячейку. Это случай в языке стандартной интерпретации записывается присваиванием вида $\mathrm{x}_1(0) = \ldots$ При этом присваивании очищается вся первоначальная строка, а новая записывается «начисто».

$p_1 = 0$

$rv = msgAbc$

$SendTo_0\, NextCell\, (/\#$ Установить указатель p_1 и очистить rv здесь и далее $\#/, \ominus)$

$(/\#$ Записать символ в ячейке с указателем p_1 и сдвинуть указатель $\#/, msgAbc)$

$, msgAbc \neq \ominus, p_1 \neq 2, rv = \ominus$

// Мы заменяем пустой символ новым не пустым символом. В таком случае сама такая ячейка является указателем, вне зависимости от значения p_1. При такой замене размер строки увеличивается на один символ и надо обеспечить, чтобы этот символ был последним в строке – сразу за этим символом должен быть пустой символ и новое положение указателя.

$p_1 = 0$

$rv = msgAbc$

$SendTo_0\, NextCell\, (/\#$Установить указатель p_1 и очистить rv здесь и далее $\#/, \ominus)$

$(/\#$ Записать символ в ячейке с указателем p_1 и сдвинуть указатель $\#/, msgAbc)$

$, msgAbc \neq \ominus, p_1 = 1, rv \neq \ominus$

// Мы заменяем не пустой символ на новый не пустой символ. При такой замене размер строки не меняется, и мы просто сдвигаем указатель на одну ячейку прочь от процессора

$p_1 = 0$

$rv = msgAbc$

$SendTo_0\, NextCell\, (/\#$ Присвоить p_1 значение $\#/, 1)$

$(/\#$ Записать символ в ячейке с указателем p_1 и сдвинуть указатель $\#/, msgAbc)$

$, msgAbc \neq \ominus, p_1 = 0, rv \neq \ominus$

// Мы в «транзитной» ячейке – она без указателя и находится внутри строки. Движемся дальше

$SendTo_0\, NextCell\, (/\#$ Записать символ в ячейке с указателем p_1 и сдвинуть указатель $\#/, msgAbc)$

VI.2. Запись символов из теневой очереди в ячейки от указателя включительно и далее

Теперь пусть луч данных $@Source_1$ содержит теневую строку, построение которой завершено полностью (а не оперативно), которую надо перенести на луч данных $@Target_1$, начиная от ячейки (пусть номер i) с указателем $p_1 \neq 0$. То есть, перенести строку по правилам команды присваивания языка программирования стандартной интерпретации:

$\mathrm{x}_1(i) = \ldots$ \\ Присваивание, если указатель указывает на нулевую ячейку $(i = 0)$ или ячейку внутри строки $(0 < i \leq \mathrm{len}(x_1))$

$\mathrm{x}_1() = \ldots$ \\ Присваивание, если указатель расположен в первой ячейке после окончания строки

Но есть важный нюанс – для работы с очередью (откуда перебрасываем символы) мы будем использовать теневой канал связи. Это предотвратит конфликт сообщений, идущих из источника с сообщениями, идущими по приёмнику, если $Source_1 = Target_1$. Напоминаю, что у нас всегда есть обработчики для теневого канала связи, аналогичные обычным обработчикам (для основных каналов связи лучей данных) – если не оговорено противное. Только в «теневом» обработчике используются команды отправки сообщений из ячеек вида:

$SendToS_i \ldots$

Вместо

$SendTo_i \ldots$

Вот протокол для необходимой нам передачи символов:

$(/\#$ Начать перенос теневой строки из $@Source_1$ в позицию p_1 в $@Target_1\ \#/, \ominus)$

// Это сообщение процессор отправляет сам себе, и реагирует на его получение, а затем использует заранее подготовленные регистры $Source_1$ и $Target_1$.

$Continue_0 = /\#$ Переносить теневые символы из $@Source_1$ в позицию p_1 в $@Target_1\ \#/$

$SendToS_0\ @Source_1\ (/\#$ В процессор из теневой очереди $\#/, \ominus)$

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ Переносить теневые символы из $@Source_1$ в позицию p_1 в $@Target_1\ \#/$

$, msgAbc \neq \ominus$

$SendTo_0\ @Target_1\ (/\#$ Записать символ в ячейке с указателем p_1 и сдвинуть указатель $\#/, msgAbc)$

$SendToS_0\ @Source_1\ (/\#$ В процессор из теневой очереди $\#/, \ominus)$

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ Переносить теневые символы из $@Source_1$ в позицию p_1 в $@Target_1\ \#/$

$, msgAbc = \ominus, @Target_1.p_1 = 2$

// Особый случай, когда теневая строка – пустая. Тут только первый символ поступил из очереди, потому что регистру $@Target_1.p_1$ ещё не присвоен ноль. Но раз указатель указывает на нулевую ячейку, то надо очистить весь луч $@Target_1$ и завершить работу в режиме SMAU

$SendTo_0\ @Target_1\ (/\#$ Очистить строку, начиная от $p_1\ \#/, \ominus)$

$wait$ // Дождались отправки сообщения

$wait_0$ // Дождались (с запасом) установки в первой ячейке пустого символа

// Этот обработчик является одним из вариантов завершения при переносе теневой строки $@Source_1$ в позицию p_1 в $@Target_1$. После чего происходит возврат процессора в режим исполнения программы аппаратного языка стандартной интерпретации.

$Setmode\ SI$

$(/\#$ В процессор из теневой очереди $\#/, msgAbc)$

$, Continue_0 = /\#$ Переносить теневые символы из $@Source_1$ в позицию p_1 в $@Target_1\ \#/$

$, msgAbc = \ominus, @Target_1.p_1 \neq 2$

// Все теневые символы исчерпаны, очищать ничего от ячейки с $p_1 \neq 0$ в $@Target_1$ не требуется, так как очищать требовалось только для случая $p_1 = 2$. И это было бы сделано для $p_1 = 2$ на 1-м же передаваемом символе

// Этот обработчик является одним из вариантов завершения при переносе теневой строки $@Source_1$ в позицию p_1 в $@Target_1$. После чего происходит возврат процессора в режим исполнения программы аппаратного языка стандартной интерпретации.

$Setmode\ SI$

VI.3. Передача всех символов от указателя включительно в теневую строку

Нам надо сформировать результат функции $\mathrm{from}(x, i)$ в качестве теневой строки из строки луча данных x. Луч данных x – это используемый далее для передачи сообщений луч данных. А в качестве i выступает номер первой ячейки, задаваемый значением её регистра $p_1 \neq 0$. Если $p_1 = 1$, то i совпадает с номером ячейки, если $p_1 = 2$, то это 1-я ячейка, но $i = 0$.

Мы будем рассматривать сообщения 2 видов:

(/# Отправить символы в теневую строку от p_1 включительно #/, $\ominus$)

(/# Отправлять символы в теневую строку из данной ячейки и далее #/, $\ominus$)

Первое сообщение двигается «по эстафете» от первой ячейки до ячейки с указателем $p_1 \neq 0$ или первой пустой ячейки, а, когда достигает своей цели – начинается (если это не пустая ячейка) этап движения второго сообщения. Второе сообщение отправляет в теневую строку все символы (если они есть) от целевой ячейки первого сообщения и до конца строки.

Отметим, что обработка данных сообщений не предусматривает в процессе этой обработки первоначальную очистку теневой строки. Она должна быть выполнена до отправки сообщения (/# Отправить символы в теневую строку от p_1 включительно #/, $\ominus$).

Данный протокол возвращает процессор в режим SI когда он полностью (а не только оперативно) завершён и все символы теневой строки находятся на своих местах. Команда смены режима для непустой строки возникает при отработке возникающего в процессе работы протокола сообщения /# Дополнить теневую строку #/.

Для такой реакции – которая приведёт к возврату процессора в режим SI – необходимо заранее задать значение его регистру. Мы уже построили соответствующий обработчик при рассмотрении протокола работы для сообщения /# Отбросить тень и уведомить #/:

$Continue_0 =$ /# Скопировать теневую строку из основной #/

Поэтому начнём описание протокола с работы процессора – отправке сообщения для построения теневой строки и реакции на сообщение о завершении построения.

1) Отправка запроса процессора и его реакция на сообщение о готовности теневой строки (копия подстроки от ячейки с $p_1 = 1$ **включительно и до конца), следующие:**

(/# Начать отправку символов в теневую строку $@Source_1$ от p_1 включительно #/, $\ominus$)

// Это сообщение процессор отправляет сам себе, и реагирует на его получение, а затем использует заранее подготовленные регистры $Source_1$ и $Target_1$.

$Continue_0 =$ /# Скопировать теневую строку из основной #/

$SendTo_0\, @Source_1$ (/# Очистить sv здесь и дальше #/, $\ominus$)

$wait$

$SendTo_0\, @Source_1$ (/# Отправить символы в теневую строку от p_1 включительно #/, $\ominus$)

Напоминание про ранее написанный обработчик:

(/# Теневая строка построена #/, $\ominus$)

, $Continue_0 =$ /# Скопировать теневую строку из основной #/

$Setmode\, SI$

2) Протокол передачи сообщения (/#Отправить символы в теневую строку от p_1 включительно #/, $\ominus$) **следующий:**

(/# Отправить символы в теневую строку от p_1 включительно #/, $\ominus$), $p_1 = 2$

// Мы находимся в первой ячейке, но указатель указывает на «виртуальную» нулевую ячейку. В силу равенства $\mathrm{from}(x, 0) = \ominus$ мы завершаем построение теневой строки, и это – пустая строка.

$SendTo_{-2}\, Proc$ (/# Теневая строка построена #/, $\ominus$)

(/# Отправить символы в теневую строку от p_1 включительно $\#/, \ominus), p_1 = 1, IsFirst = 1, rv = \ominus$

// Мы находимся в первой ячейке, но строка – пустая. Поэтому мы завершаем построение теневой строки, и это – пустая строка.

$SendTo_{-2}\, Proc$ (/# Теневая строка построена $\#/, \ominus$)

(/# Отправить символы в теневую строку от p_1 включительно $\#/, \ominus), p_1 = 1, IsFirst = 1, rv \neq \ominus$

// Мы находимся в первой ячейке, и у нас есть не пустой символ для построения теневой строки. Поэтому мы переносим его в теневой регистр и начинаем эстафету дополнения теневой строки всеми следующими символами.

$sv = rv$

$SendTo_0\, NextCell$ (/# Отправлять символы в теневую строку из данной ячейки и далее $\#/, \ominus$)

(/# Отправить символы в теневую строку от p_1 включительно $\#/, \ominus), p_1 = 1, IsFirst = 0, rv = \ominus$

// Мы находимся в не первой ячейке, где должен быть первый символ для построения теневой строки. Но этот символ – пустой Поэтому мы завершаем построение теневой строки, и это – пустая строка.

$SendTo_{-2}\, PreviousCell$ (/# Теневая строка построена $\#/, \ominus$)

(/# Отправить символы в теневую строку от p_1 включительно $\#/, \ominus), p_1 = 1, IsFirst = 0, rv \neq \ominus$

// Мы находимся в не первой ячейке, где имеется первый символ для построения теневой строки. Поэтому мы отправляем его – для начала построения теневой строки, и начинаем эстафету дополнения теневой строки всеми следующими символами.

$SendTo_{-2}\, PreviousCell$ (/# Дополнить теневую строку $\#/, rv$)

$SendTo_0\, NextCell$ (/# Отправлять символы в теневую строку из данной ячейки и далее $\#/, \ominus$)

(/# Отправить символы в теневую строку от p_1 включительно $\#/, \ominus), p_1 = 0, IsFirst = 1, rv = \ominus$

// Мы в первой же ячейке дошли до конца строки, а указателя так и не обнаружили. Поэтому мы сообщаем прямо в процессор о построении теневой строки, и это – пустая строка.

$SendTo_{-2}\, Proc$ (/# Теневая строка построена $\#/, \ominus$)

(/# Отправить символы в теневую строку от p_1 включительно $\#/, \ominus), p_1 = 0, IsFirst = 0, rv = \ominus$

// Мы в не первой ячейке дошли до конца строки, а указателя так и не обнаружили. Поэтому мы отправляем по «эстафете» сообщение к процессору о построении теневой строки, и это – пустая строка.

$SendTo_{-2}\, PreviousCell$ (/# Теневая строка построена $\#/, \ominus$)

(/# Отправить символы в теневую строку от p_1 включительно $\#/, \ominus), p_1 = 0, rv \neq \ominus$

// Указатель пока не обнаружен, но и строка ещё не кончилась. Поэтому продолжаем поиск.

$SendTo_0\, NextCell$ (/# Отправить символы в теневую строку от p_1 включительно $\#/, \ominus$)

3) Протокол передачи сообщения (/# Отправлять символы в теневую строку из данной ячейки и далее **следующий:**

(/# Отправлять символы в теневую строку из данной ячейки и далее #/, $\ominus$), $IsFirst = 1$

// Аварийное завершение. При правильных протоколах передачи и обработки событий – невозможная ситуация. В первую ячейку могут поступать сообщения (/# Отправить символы в теневую строку от но не сообщения (/# Отправить символы в теневую строку от p_1 включительно #/, $\ominus$).

(/# Отправлять символы в теневую строку из данной ячейки и далее #/, $\ominus$), $IsFirst \neq 1, rv = \ominus$

// Мы дошли до конца строки, все необходимые символы в теневую строку отправлены. Поэтому мы отправляем по «эстафете» последнее сообщение «достройки», которое ничего уже не достроит, зато отправит процессору сообщение, что /# Теневая строка построена #/.

$SendTo_{-2}\, PreviousCell$ (/# Дополнить теневую строку #/, rv)

(/# Отправлять символы в теневую строку из данной ячейки и далее #/, $\ominus$), $IsFirst \neq 1, rv \neq \ominus$

// Мы продолжаем двигаться внутри строки, поэтому отправляем текущий символ в теневую строку, и продолжаем передачу сообщения в следующую ячейку о продолжении построения теневой строки.

$SendTo_{-2}\, PreviousCell$ (/# Дополнить теневую строку #/, rv)

$SendTo_0\, NextCell$ (/# Отправлять символы в теневую строку из данной ячейки и далее #/, $\ominus$)

VI.4. Передача всех символов от указателя до перевода строки включительно в теневую строку

Нам нужно получать команды с луча данных Program. Начало команды указано числом i на луче данных $i_{Program}$. Но мы рассмотрим протокол передачи в предположении, что указатель $p_1 = 1$ уже установлен на луче данных $Program$ в ячейке с номером i.

Всякая команд (в корректном программном тексте) языка стандартной интерпретации заканчивается точкой с запятой и переводом строки:

;

Притом «невидимый» перевод строки в предыдущем абзаце тоже считается.

Протокол передачи сообщений для получения текущей команды в теневую строку не сильно отличается от предыдущего протокола для передачи всех символов от указателя включительно. Но только вместо пустого символа тут выступает ещё и символ $ChrEnter$ (с нюансами), и его надо передавать так же, как символы до него.

Мы будем рассматривать сообщения 2 видов:

(/# Отправить команду программы от p_1 до $ChrEnter$ в теневую строку #/, $\ominus$)

(/# Отправлять символы до $ChrEnter$ включительно в теневую строку #/, $\ominus$)

Первое сообщение двигается «по эстафете» от первой ячейки до ячейки с указателем $p_1 \neq 0$ или первой ячейки с пустым символом, а, когда достигает своей цели – то при корректной целевой ячейке ($p_1 = 1$) начинается этап движения второго сообщения. Второе сообщение отправляет в теневую строку все символы (если они есть) от целевой ячейки первого сообщения и до ячейки с символом $ChrEnter$ включительно.

Отметим, что обработка данных сообщений не предусматривает в процессе этой обработки первоначальную очистку теневой строки. Она должна быть выполнена до отправки сообщения (/# Отправить команду программы от p_1 до $ChrEnter$ в теневую строку #/, $\ominus$).

Данный протокол возвращает процессор в режим SI когда он полностью (а не только оперативно) завершён и все символы теневой строки находятся на своих местах. Команда смены режима для непустой строки возникает при отработке возникающего в процессе работы протокола сообщения /# Дополнить теневую строку #/.

Для такой реакции – которая приведёт к возврату процессора в режим SI – необходимо заранее задать значение его регистру. Мы уже построили соответствующий обработчик при рассмотрении протокола работы для сообщения /# Отбросить тень и уведомить #/:

$Continue_0 =$ /# Скопировать теневую строку из основной #/

Поэтому начнём описание протокола с работы процессора – отправке сообщения для построения теневой строки и реакции на сообщение о завершении построения.

1) Отправка запроса процессора и его реакция на сообщение о готовности теневой строки (копия подстроки от ячейки с $p_1 = 1$ и до до $ChrEnter$ **включительно), следующие:**

(/# Начать отправку символов в теневую строку от p_1 до $ChrEnter$ #/, $\ominus$)

// Это сообщение процессор отправляет сам себе, и реагирует на его получение, а затем использует заранее подготовленные регистры $Source_1$ и $Target_1$.

$Continue_0 =$ /# Скопировать теневую строку из основной #/

$SendTo_0\, Program$ (/# Очистить sv здесь и дальше #/, $\ominus$)

$wait$

$SendTo_0\, Program$ (/# Отправить команду программы от p_1 до $ChrEnter$ в теневую строку #/, $\ominus$)

Напоминание про ранее написанный обработчик:

(/# Теневая строка построена #/, $\ominus$)

, $Continue_0 =$ /# Скопировать теневую строку из основной #/

$Setmode\, SI$

2) Протокол передачи сообщения (/#Отправить команду программы от p_1 до $ChrEnter$ в теневую строк
следующий:

(/# Отправить команду программы от p_1 до $ChrEnter$ в теневую строку #/, $\ominus$), $p_1 = 2$

// Аварийное завершение работы, так как при поиске команды на луче данных не может быть указателя на виртуальную нулевую ячейку

(/# Отправить команду программы от p_1 до $ChrEnter$ в теневую строку #/, $\ominus$), $rv = \ominus$

// Аварийное завершение работы, так как при поиске команды на луче данных текст программы не может кончиться раньше, чем номер позиции начала текущей команды

(/# Отправить команду программы от p_1 до $ChrEnter$ в теневую строку #/, $\ominus$)

, $p_1 = 1, rv = ChrEnter$

// Аварийное завершение работы, так как команда не может состоять из одного символа перевода строки

(/# Отправить команду программы от p_1 до $ChrEnter$ в теневую строку #/, $\ominus$)

$, p_1 = 1, IsFirst = 1, rv \neq ChrEnter, rv \neq \ominus$

// Мы находимся в первой ячейке, и у нас есть не пустой символ и не перевод строки для построения теневой строки. Поэтому мы переносим его в теневой регистр и начинаем эстафету дополнения теневой строки всеми следующими символами.

$sv = rv$

$SendTo_0\, NextCell\, (/\#$ Отправлять символы до $ChrEnter$ включительно в теневую строку $\#/, \ominus)$

$(/\#$ Отправить команду программы от p_1 до $ChrEnter$ в теневую строку $\#/, \ominus)$

$, p_1 = 1, IsFirst = 0, rv \neq ChrEnter, rv \neq \ominus$

// Мы находимся в не первой ячейке, где имеется первый символ для построения теневой строки. Поэтому мы отправляем его – для начала построения теневой строки, и начинаем эстафету дополнения теневой строки всеми следующими символами до символа перевода строки аключительно.

$SendTo_{-2}\, PreviousCell\, (/\#$ Дополнить теневую строку $\#/, rv)$

$SendTo_0\, NextCell\, (/\#$ Отправлять символы до $ChrEnter$ включительно в теневую строку $\#/, \ominus)$

$(/\#$ Отправить команду программы от p_1 до $ChrEnter$ в теневую строку $\#/, \ominus), p_1 = 0, rv \neq \ominus$

// Указатель пока не обнаружен, но и строка ещё не кончилась. Поэтому продолжаем поиск.

$SendTo_0\, NextCell\, (/\#$ Отправить команду программы от p_1 до $ChrEnter$ в теневую строку $\#/, \ominus)$

2) Протокол передачи сообщения $(/\#$ Отправлять символы до $ChrEnter$ включительно в теневую строк **следующий:**

$(/\#$ Отправлять символы до $ChrEnter$ включительно в теневую строку $\#/, \ominus), IsFirst = 1$

// Аварийное завершение. При правильных протоколах передачи и обработки событий – невозможная ситуация. В первую ячейку могут поступать сообщения $(/\#$ Отправить команду программы от p_1 до но не сообщения $(/\#$ Отправить команду программы от p_1 до $ChrEnter$ в теневую строку $\#/, \ominus)$.

$(/\#$ Отправлять символы до $ChrEnter$ включительно в теневую строку $\#/, \ominus), IsFirst \neq 1, rv = \ominus$

// Аварийное завершение. Команда не может кончаться ничем, кроме перевода строки

$(/\#$ Отправлять символы до $ChrEnter$ включительно в теневую строку $\#/, \ominus)$

$, IsFirst \neq 1, rv = ChrEnter$

// Мы добрались до конца команды. Отправляем символ $ChrEnter$ и, затем, завершаем построение теневой строки, «дополняя» её пустым символом.

$SendTo_{-2}\, PreviousCell\, (/\#$ Дополнить теневую строку $\#/, rv)$

$wait_0$

$wait_0$ // Нам нужно гарантированно пропустить 1 момент после отправки предыдущего сообщения, поэтому $wait_0$ повторяем 2 раза.

$SendTo_{-2}\, PreviousCell\, (/\#$ Дополнить теневую строку $\#/, \ominus)$

$(/\#$ Отправлять символы до $ChrEnter$ включительно в теневую строку $\#/, \ominus)$

$, IsFirst \neq 1, rv \neq ChrEnter, rv \neq \ominus$

// Мы продолжаем двигаться внутри строки, поэтому отправляем текущий символ в теневую строку, и продолжаем передачу сообщения в следующую ячейку о продолжении построения теневой

строки.

$SendTo_{-2}\, PreviousCell\, (/\#$ Дополнить теневую строку $\#/, rv)$

$SendTo_0\, NextCell\, (/\#$ Отправлять символы до $ChrEnter$ включительно в теневую строку $\#/, \ominus)$

VI.5. Установка указателя в первой пустой ячейке

Для выполнения операций конкатенации в языке программирования стандартной интерпретации используется присваивание вида

$\mathrm{x}_1() = \ldots;$

А в создаваемом аппаратном языке программирования SI для дописывания в конец строки новых символов необходимо установить в первую пустую ячейку указатель. Ранее мы рассмотрели, как символы поступают в ячейку с указателем, и записываются в данной ячейке, сдвигая указатель дальше. А теперь построим протокол передачи сообщения для установки указателя сразу после последнего символа строки:

$(/\#$ Установить указатель p_1 в первой пустой ячейке $\#/, \ominus), rv = \ominus$

$p_1 = 1$

$(/\#$ Установить указатель p_1 в первой пустой ячейке $\#/, \ominus), rv \neq \ominus$

$p_1 = 0$

$SendTo_0\, NextCell\, (/\#$ Установить указатель p_1 в первой пустой ячейке $\#/, \ominus)$

Вот и весть протокол для передачи данного сообщения. После отправки данного сообщения можно после команд:

$wait$ // Дожидаемся 1-момента и отправки сообщения

$wait_0$ // Дожидаемся, чтобы 1-я ячейка гарантированно была отработана

Можно считать, что указатель установлен, так как протокол оперативно завершён. Никаких сообщений в процессор об установке указателя не возвращается.

VII. Свод результатов

Теперь проверим, что все случаи отправки сообщений из программы-интерпретатора стандартного языка SI рассмотрены нами. В качестве нумерации соответствующих частей программы-интерпретатора используем нумерацию подразделов в соответствующем разделе. Нумерация подразделов с разбором частей программы-интерпретатора в разделе «2. Аппаратный интерпретатор языка SI» начинается с 4:

4. Перенос текущей (начиная с номера позиции из $i_{Program}$) команды из $Program$ в регистр $Command$

Использовались 2 команды отправки сообщений:

$SendTo_0\,Proc\,(/\#$ Сделать указатель p_1 в $@Target_1$ из $@Source_1$ или подобное $\#/, \ominus)$;

$SendTo_0\,Program\,(/\#$ Начать отправку символов в теневую строку от p_1 до $ChrEnter\,\#/, \ominus)$;

Обработчик для первой команды был разобран в текущем разделе в параграфе «4.2. Начать установку p_1 (первого указателя) или чего-то аналогичного на луче данных», а для второй – в параграфе «6.4. Передача всех символов от указателя до перевода строки включительно в теневую строку», пункт 1.

Вкратце эта часть программы-интерпретатора делает следующее – устанавливает на луче данных указатель в той ячейке, номер которой указан в $i_{Program}$. Это делает первое сообщение.

Второе сообщение в этом блоке приказывает лучу данных $Program$ перенести все символы, начиная с ячейки-указателя и вплоть до ячейки с переводом строки, в теневую строку данного луча данных.

Оставшаяся часть обходится без сообщения, так как любая команды с луча данных Program имеет такой размер, что её можно обрабатывать как обычную переменную обычных языков программирования.

5. Обработка команд, отличающихся от присваивания

1) $[next_1]$;

Использовалась 1 команда отправки сообщения:

$SendTo_0\,Proc\,(/\#$ Увеличить число в $@Target_1$ на количество символов в $@Source_1\,\#/, \ominus)$;

Обработчик для данной команды был разобран в текущем разделе в параграфе «3.1. Прибавить к числу в $@Target_1$ длину строки из $@Source_1$».

После исполнения большинства команд надо переходить к следующей команде на луче данных $Program$. При обработке метки ничего другого и не происходит. И сообщение используется для увеличения числа на луче данных $i_{Program}$ на размер команды в регистре $Command$.

2) $goto\,@AloneArg$;

Использовались 3 команды отправки сообщений:

$SendTo_0\,Proc\,(/\#$ Вычислить $@Target_1 = \mathrm{Goto}(@Source_1)\,\#/, \ominus)$;

$SendTo_0\,Result_1\,(/\#$ Отбросить тень и уведомить $\#/, \ominus)$;

$SendTo_0\,Proc\,(/\#$ Начать перенос теневой строки из $@Source_1$ в позицию p_1 в $@Target_1\,\#/, \ominus)$;

Обработчик для первой команды был разобран в текущем разделе в параграфе «3.4. Вычисление $@Target_1 = \mathrm{Goto}(@Source_1)$», пункт 1.

Для второй – в параграфе «2.2. Отбросить тень – скопировать обычные символы луча данных в теневые».

Для третьей – в параграфе «6.2. Запись символов из теневой очереди в ячейки от указателя включительно и далее».

По типу-комментарию первого сообщения видно, что почти всё делает обработчик данного сообщения и обработчики вызванных им событий – находя на луче данных $Program$ номер позиции сразу за меткой из луча данных $@AloneArg$. Если соответствующая метка на луче $Program$ найдётся, разумеется – иначе будет ноль.

Два других сообщения переносят результат поиска (если он не ноль) на луч данных $i_{Program}$. Либо же (если результат ноль) то интерпретатор переходит делать то же, что и для метки – менять текущую команду на следующую команду.

3) .;

В этой части программы-интерпретатора обошлось без отправки сообщений

6. Обработка левой части команды присваивания

4.1) $@\,\mathrm{LeftVar}(@q_{LeftVar}) = \ldots$; // При $q_{LeftVar} = \ominus$

Использовалась 1 команда отправки сообщения:

$SendTo_0\,@Target_1\,(/\#$ Установить указатель p_1 в первой пустой ячейке $\#/, \ominus)$;

Обработчик для данной команды был разобран в текущем разделе в параграфе «6.5. Установка указателя в первой пустой ячейке».

4.2 и 4.3) $@\,\mathrm{LeftVar}(@q_{LeftVar}) = \ldots$; // При $q_{LeftVar} \neq \ominus$ **(любое число)**

Использовалась 1 команда отправки сообщения:

$SendTo_0\,Proc\,(/\#$ Сделать указатель p_1 в $@Target_1$ из $@Source_1$ или подобное $\#/, \ominus)$;

Обработчик для данной команды был разобран в текущем разделе в параграфе «4.2. Начать установку p_1 (первого указателя) или чего-то аналогичного на луче данных».

7. Обработка правой части команды присваивания

Забегая вперёд – в этой части программы-интерпретатора данные для присваивания (правая часть команды присваивания) только готовятся. Поэтому в конце разбора разных правых частей получается та или иная теневая строка. А собственно присваиванием этой теневой строки в нужное место нужного луча данных программа-интерпретатор будет заниматься после разбираемых тут вычислений для правых частей присваивания.

5) $\mathrm{len}(@AloneArg)$;

Использовались 2 команды отправки сообщений:

$SendTo_0\,Proc\,(/\#$ Увеличить число в $@Target_1$ на количество символов в $@Source_1\,\#/, \ominus)$;

$SendTo_0\,@Target_1\,(/\#$ Отбросить тень и уведомить $\#/, \ominus)$;

Обработчик для первой команды был разобран в текущем разделе в параграфе «3.1. Прибавить к числу в $@Target_1$ длину строки из $@Source_1$», а для второй – в параграфе «2.2. Отбросить тень – скопировать обычные символы луча данных в теневые».

6) $\mathrm{Chr}(@AloneArg)$;

Использовалась 1 команда отправки сообщения:

$SendTo_0\,@Target_1\,(/\#$ Отбросить тень и уведомить $\#/, \ominus)$;

Обработчик для команды был разобран в текущем разделе в параграфе «2.2. Отбросить тень – скопировать обычные символы луча данных в теневые».

Для вычисления Chr() сообщений не потребовалось, потому что значимая часть аргумента (от которой зависит результат) рассматривалась как регистр при помощи свойства $.Register$. Сообщение потребовалось лишь для того, чтобы подготовить из результата вычисления теневую строку для дальнейшего присваивания.

7.1) $\mathrm{Value}(@AloneArg)$;

Использовалась 1 команда отправки сообщения:

$SendTo_0\,@Target_1\,(/\#$ Отбросить тень и уведомить $\#/, \ominus)$;

Обработчик для команды был разобран в текущем разделе в параграфе «2.2. Отбросить тень – скопировать обычные символы луча данных в теневые».

Результатом в данном случае является сам аргумент. Поэтому использовалось лишь сообщение для представления аргумента в виде теневой строки.

7.2) $\mathrm{from}(@FirstArg, @LastArg)$;

Использовались 2 команды отправки сообщений:

$SendTo_0\,Proc\,(/\#$ Сделать указатель p_1 в $@Target_1$ из $@Source_1$ или подобное $\#/, \ominus)$;

$SendTo_0\,Proc\,(/\#$ Начать отправку символов в теневую строку $@Source_1$ от p_1 включительно $\#/, \ominus)$;

Обработчик для первой команды был разобран в текущем разделе в параграфе «4.2. Начать установку p_1 (первого указателя) или чего-то аналогичного на луче данных», а для второй – в параграфе «6.3. Передача всех символов от указателя включительно в теневую строку», пункт 1.

По типам-комментариям сообщений понятно, как формируется теневая строка, равная результату данной функции. Указатель устанавливается в позицию $@LastArg$ луча данных $@FirstArg$, разумеется.

7.3) $\mathrm{str}(@FirstArg, @MidArg, @LastArg)$;

Использовалась 1 команда отправки сообщения, но 2 раза:

$SendTo_0\,Proc\,(/\#$ Сделать указатель p_1 в $@Target_1$ из $@Source_1$ или подобное $\#/, \ominus)$; Обработчик для команды был разобран в текущем разделе в параграфе «4.2. Начать установку p_1 (первого указателя) или чего-то аналогичного на луче данных»,

Первый раз данная команда использовалась для установки указателя в позицию номер $@MidArg$ на луче данных $@FirstArg$. А второй раз – для передачи в теневую строку символов от указателя включительно в количестве $@LastArg$ штук. С учётом всех нюансов вроде слишком короткой строки (или значения ноль) для $@MidArg$ или нехватки символов (или значения ноль) для $@LastArg$, конечно.

Разница между 2-мя данными случаями отправки одинаковых сообщений – в значениях соответствующих регистров. В первом случае значения такие:

$\mathrm{msgBegType}_0(0) = /\#$ Младший разряд уменьшаемого числа $(p_1)\,\#/$;

$\mathrm{msgLastType}_0(0) = /\#$ Разряды для p_1 исчерпаны $\#/$;

Во втором случае значения следующие:

$\mathrm{msgBegType}_0(0) =$ /# Младший разряд числа к уменьшению от p_1 #/;

$\mathrm{msgLastType}_0(0) =$ /# Разряды для количества символов к отправке от p_1 исчерпаны #/;

8) $\mathrm{Comp}(@FirstArg, @LastArg)$;

Использовались 2 команды отправки сообщений:

$SendTo_0\, Proc$ (/# Вычислить $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2$ #/, $\ominus$);

$SendTo_0\, Result_1$ (/# Отбросить тень и уведомить #/, $\ominus$);

Обработчик для первой команды был разобран в текущем разделе в параграфе «3.3. Вычисление $@Target_1 = \mathrm{Comp}(@Source_1, @Source_2)$», а для второй – в параграфе «2.2. Отбросить тень – скопировать обычные символы луча данных в теневые».

По типу-комментарию первого сообщения понятно, что практически всё вычисление инициируются данным сообщением.

9) $\mathrm{Next}(@AloneArg)$;

Использовались 2 команды отправки сообщений:

$SendTo_0\, @Target_1$ (/# Увеличить на 1 #/, $\ominus$);

$SendTo_0\, @Target_1$ (/# Отбросить тень и уведомить #/, $\ominus$);

Обработчик для первой команды был разобран в текущем разделе в параграфе «2.1. Увеличение на 1 числа на луче данных», а для второй – в параграфе «2.2. Отбросить тень – скопировать обычные символы луча данных в теневые».

По типу-комментарию первого сообщения понятно, что практически всё вычисление инициируются данным сообщением.

10) $\mathrm{Previous}(@AloneArg)$;

Использовались 2 команды отправки сообщений:

$SendTo_0\, @Target_1$ (/#Уменьшить на 1 #/, $\ominus$);

$SendTo_0\, @Target_1$ (/# Отбросить тень и уведомить #/, $\ominus$);

Обработчик для первой команды был разобран в текущем разделе в параграфе «2.4. Уменьшение на 1 числа на луче данных», а для второй – в параграфе «2.2. Отбросить тень – скопировать обычные символы луча данных в теневые».

По типу-комментарию первого сообщения понятно, что практически всё вычисление инициируются данным сообщением.

8. Собственно присваивание – перенос результата на нужный луч данных

К данному моменту в $Source_1$ должно быть записано имя того луча данных (или регистра), теневая строка которого должна быть присвоена на луч данных с именем в $LeftVar$. Притом аргумент присваивания i из $@\,\mathrm{LeftVar}(i) = \ldots$ уже задействован в качестве установленного указателя на луче данных $@LeftVar$.

Использовалась 1 команда отправки сообщения:

$SendTo_0\, Proc$ (/# Начать перенос теневой строки из $@Source_1$ в позицию p_1 в $@Target_1$ #/, $\ominus$);

Обработчик для команды был разобран в текущем разделе в параграфе «6.2. Запись символов из теневой очереди в ячейки от указателя включительно и далее».

В то место (луч данных и номер позиции), которое было задано в части программы под условным номером 6 (как мы тут нумеруем процесс разбора программы) переносятся данные, которые были сформированы в части программы под условным номером 7.

На этом перечисление всех команд отправки сообщений, необходимых для работы программы-интерпретатора стандартного языка SI закончено.

5 Культурное влияние на данное исследование

Хотелось бы сказать о той «подсказке», которая не видна по результату, но очень помогла процессу исследования:

Я пытался разобраться с тем, как можно было бы аксиоматизировать время, и, не следует ли предусмотреть в теории строк какие-то аксиомы для этого. Одним из усложняющих факторов был вопрос о соизмеримости «эталонов», которыми можно отсчитывать промежутки времени. И тут мне снова попался на глаза мультфильм «38 попугаев» и ироничная завершающая фраза Удава «А в попугаях-то я гораздо длиннее».

И тут прояснилось, что если есть конечный набор «эталонных» друзей, то совершенно не важно, каким другом ты измеряешь то, что хочешь измерить. Конечные эталоны из конечного списка эталонов всегда полиномиально соразмерны друг с другом. И не важно – с точки зрения характера зависимостей – чего именно 38 в измеряемом объекте – попугаев или слоненков. Потому что если характер зависимости соответствует полиному 2-й степени, например, то зависимость будет квадратичной и в попугаях, и в слоненках. И то же верно для экспоненты.

Поэтому в дальнейшем я следил только за конечностью количества «эталонов», не задумываясь над вопросом их соразмерности. В итоге я пришёл к сеткам синхронизации, которые не просто имеют понятную соразмерность, но даже кратны относительно «эталонов» друг друга. Но я не стремился к этому специально, просто процесс исследования значительно упростился, что помогло в итоге рассмотреть значительно легче множество подходов, и лучше оптимизировать результат.

И ещё одна подсказка существенно повлияла не только на процесс исследования, но и на архитектуру Машины:

Я искал протокол передачи уменьшаемого от ячейки к ячейке числа по «эстафете» так, чтобы при достижении этим числом значения 1 дальнейшее движение пакета данных прекращалось. И это получалось довольно сложно, с учётом того, что никакой «конец» у луча данных или строки на луче данных не гарантирован для бесконечной математической модели.

Но бесконечность имеет свои преимущества – можно просто продолжить движение пакета данных, лишив его силы изменять что-либо, просто заменив последнее сообщение пакета данных на «холостое» сообщение.

На эту мысль меня навела американская комедия о космическом пассажирском рейсе «Аэроплан-2» в русском переводе А.М. Михалёва (в оригинале там другой диалог с игрой слов over-under-dunno в именах персонажей Over-Unger-Dunn):

- Господа, познакомьтесь, это ваш капитан - Капитан Прием.

- Это Ваш штурман мистер Доза и Ваш старший помощник мистер Перебор.

Забавно, что они там тоже вылетели за рамки курса («перебор») и тоже боролись с компьютером – а мы должны его заранее правильно наладить, чтоб избежать неприятностей. В фильме «мальчики в совете директоров» не дали времени на такую наладку, а зря.

Ячейка — это Приём («капитан Приём») сообщений (цифр) пакета данных (числа). Приём отнимает 1 от числа. Пока вычитание по 1 от числа в каждой ячейке укладывается в рамки передаваемого числа — это ячейки и число в состоянии Доза. Но когда сообщение доходит до слишком

дальних ячеек — начинается состояние Перебор, что отмечается в последнем сообщении пакета, но «полёт корабля» - продолжается.

При «переборе» движение пакета данных становится «холостым» и продолжается в «бесконечность». За счет этого удаётся избавиться от необходимости думать над способом остановки дальнейшей передачи пакета данных.

6 Перспективы использования модели «Машина исполнения компьютерных алгоритмов», рассмотренной в данной работе

Я не думаю, что предложенная архитектура поспособствует увеличению оперативной памяти у компьютеров. Разрядность процессоров увеличивалась не для того, чтобы добавить компьютеру доступной оперативной памяти, а потому, что память становилась экономически более доступной.

Нынешней разрядности (64) – достаточно для адресации 16 эксбибайт (это более, чем 18,44 эксабайт). Этого вполне хватит, чтобы превзойти возможности производства оперативной памяти на столетия вперёд, если не навсегда в рамках Земли. Если смотреть спрос, то на рынке оперативной памяти работает наибольшее количество производителей среди всех компьютерных комплектующих, насколько я читал, поэтому проблема (теперь и раньше) в памяти, а не в разрядности.

Другое дело, что как раз для простых дешёвых цифровых устройств описанная архитектура снимает ограничение на размер доступной оперативной памяти. Достаточным для простой работы с большой памятью становится 32-разрядный (и даже с меньшей разрядностью) процессор. Для него потребуется не очень большая интегральная схема, не лучший техпроцесс («нанометры»). А это может сильно удешевить производство из-за меньших требований к точности оборудования, к чистоте кристалла и т.п.

Удобна и возможность «плавающего» размера для адреса ячейки – потому что сейчас программы для 64-разрядных процессоров ощутимо больше совершенно аналогичных программ, но предназначенных для 32-разрядных процессоров.

Разумеется, «ячейки памяти» в рассмотренной архитектуре несколько сложнее, чем используемая на практике «пассивная» оперативная память, но не сильно. К тому же на практике никто не будет передавать сообщения от ячейки к ячейке, а будут передавать от блока в 4 Гб к соседнему такому же, например. И на 1 Гб памяти будет приходиться только четверть (меньше 1) «сложной» ячейки памяти, условно говоря.

Но самое интересное в практическом плане даже не это. А то, что появляется возможность «универсального» отношения к любой памяти, как к оперативной памяти – притом без всяких ограничений на размер и делений «это для 32-разрядного процессора, а это – для 64-разрядного». И потенциально это очень упрощает (и ощутимо удешевляет) аппаратную разработку и низкоуровневое программирование.

В качестве примера, подтверждающего важность универсализации, можно привести перенесение методов из Интернета на настольные компьютеры, возрастающее доминирование кроссплатформенных языков и замена компиляторов интерпретаторами (кроме системного уровня, конечно). Потому что людям (не только пользователям, но и программистам) гораздо удобнее использовать что-то универсальное везде и всегда, чем учить и держать в голове много разнородных методов для разных ситуаций.

То, что сказано выше – касается практики. Но меня больше интересует теория, и написана эта работа для теоретиков в первую очередь. И она укладывается в русло крайне необходимого сейчас создания теоретической основы для рассмотрения всяких практических вычислительных систем.

Потому что – я об этом писал выше – период технологического скачка (в 100 и даже 200 лет) почти закончен, поэтому без перевода практической компетентности на уровень теоретического понимания произойдёт быстрая (по меркам Истории) технологическая деградация.

И, возможно, данная работа как раз является примером того, что теория теперь снова будет выходить – и выйдет – на первый план, а новые достижения практики будут, как правило, лишь следовать за успехами в теории – как это было (и – думаю – будет снова) тысячи лет до периода технологического скачка.

Период технологического скачка, когда практика шла впереди теории – редчайшее исключение в Истории. К тому же практика и в этот исключительный период опиралась на созданную до этого периода (и – в не очень большой степени – во время этого периода) теоретическую базу. Когда прежняя теоретическая база была почти полностью использована для практического скачка – закончился и сам этот скачок.

Теперь предстоят новые тысячи лет осмысления произошедших практических изменений в мире, выстраивание теорий на базе новой реальности и т.п. жизнь науки, сохранения и передачи знаний следующим поколениям. Есть и другой вариант будущего – технологическая деградация, но надо постараться избежать подобного развития событий – хотя в Истории много примеров цивилизационных падений.

Если же говорить конкретно, то данное исследование позволило нам получить такую интерпретацию для Теории компьютерных строк (Машина исполнения компьютерных алгоритмов), которая практически буквально соответствует используемым в Теории компьютерных строк функциям и логике. Поэтому все выгоды использования модели (наглядность, «предчувствие» теорем, основа непротиворечивости и т.д.) теперь обоснованы и находятся в нашем распоряжении.

К тому же, рассмотренная Машина исполнения компьютерных алгоритмов построена в соответствии с современной практикой построения вычислительных машин, её «эксплуатация» в этом смысле понятна и привычна с точки зрения стандартов ИТ.

Кто использовал Машину Тьюринга для оценки времени работы алгоритмов? В лучшем случае – очень небольшое число теоретиков, далёкое от практического программирования. А вот МКА вполне годится для оценок времени, потому что ключевая часть вычислений вписана в «сетки синхронизаций» с однозначными интервалами времени между состояниями узлов Машины.

Теперь есть модель, в которой алгоритм является таким же объектом теории, что и данные. Программный текст, использование этого текста в качестве алгоритма, и время на его исполнение в качестве алгоритма, доступно теперь исследованию в рамках Теории компьютерных строк в той же мере, что и используемые программой данные.

Теоретическая картина является завершённой в отношении Машины исполнения компьютерных алгоритмов, что даёт возможность для «чисто» теоретических исследований на базе данной интерпретации, без привлечения каких-либо дополнительных «практических» сведений и без потери – при этом – существенных для теории алгоритмов сведений о работе Машины.

Список литературы

[1] *Гай Харт-Девис.* Word 2000. Руководство разработчика. Издательская группа BHV, Киев. 2000

[2] *Кен Гетц, Пол Литвин, Майк Гилберт.* Access 2000. Руководство разработчика. Том 1. Настольные приложения. Издательская группа BHV, Киев. 2000

[3] *Кен Гетц, Пол Литвин, Майк Гилберт.* Access 2000. Руководство разработчика. Том 2. Корпоративные приложения. Издательская группа BHV, Киев. 2001

[4] *А. В. Попов.* Введение в Windows PowerShell. БХВ-Петербург, СПб. 2009

[5] *Марк Дж. Прайс.*. *C#* 7 и Net Core. Кросс-платформенная разработка для профессионалов. Питер, СПб. 2018

[6] *А.Н. Васильев.* Программирование на Java Script в примерах и задачах. Эксмо, М. 2019

[7] *А. Сорокин.* 8, 16, 32, 64, 128, ... или размышление о плодах прогресса. http://www.electrosad.ru/Processor/64_128.htm Интернет 2011-2013

[8] *Г. Остер (сценарист), И. Уфимцев (режиссёр).* 38 попугаев (мультфильм). Союзмультфильм, М. 1976

[9] *Э. Джин (сценарист), М. Рейсс (сценарист), К. Финклмен (режиссёр, сценарист), А.М. Михалёв (переводчик).* . Аэроплан 2: Продолжение (кинокомедия, фантастика) Paramount Pictures, США 1982